MODELLING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Oliver Graham Evans

May, 2018

MODELLING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

ABSTRACT

A probabilistic model for the spatial distribution of kelp fronds is developed based on a kite-shaped geometry and simple assumptions about the motion of fronds due to water velocity. Radiative transfer theory is then applied to determine the radiation field by using the kelp model to determine optical properties of the medium. Finite difference and asymptotic solutions are explored, and behavior of the results over the parameter space is investigated. Numerical simulations to predict the lifetime biomass production of kelp plants are performed to compare our light model to the previous exponential decay model.

ACKNOWLEDGEMENTS

Acknowledgments: This project was supported in part by the National Science Foundation under Grant No. EEC-1359256, and by the Norwegian National Research Council, Project number 254883/E40.

Mentors: Shane Rogers, Department of Environmental Engineering, Clarkson University; Ole Jacob Broch, and Aleksander Handå, SINTEF Fisheries and Aquaculture, Trondheim, Norway.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Motivation

Given the global rise in population, efficient and innovative resource utilization is increasingly important. In particular, food and fuel are clearly in high demand. Meanwhile, growing concern for the negative environmental impacts of petroleum-based fuel is generating a market for biofuel, especially corn-based ethanol. However, corn-based ethanol has been heavily criticized for diverting land usage away from food production. At the same time, a great deal of unutilized saltwater coastline is available for both food and fuel production through seaweed cultivation. Specifically, the sugar kelp *Saccharina Latissima* is known to be a viable source of food, both for direct human consumption and for fish cultivation, as well as for biofuel production.

Furthermore, nitrogen leakage into water bodies is a significant ecological danger, and is especially relevant near large conventional agriculture facilities due to run-off from nitrogen-based fertilizers, as well as near wastewater treatment plants. As a specific example, there is a wastewater treatment plant in Boothbay Harbor, Maine which is facing increasingly demanding EPA regulations limiting the concentration of certain nutrients permissible to be released into the ocean via wastewater treatment outfalls. In order to adhere to these stricter requirements using conventional nutrient remediation, a significant quantity of specialized equipment would be necessary, which is not currently present in the Boothbay Harbor plant. Being surrounded on all sides by water and private property, the treatment plant lacks the necessary space for the additional equipment, and would therefore need to move their entire facility to a new location in order to conform to these new nutrient regulations. As an alternative to conventional nutrient remediation techniques, the cultivation of the macroalgae *Saccharina Latissima* (sugar kelp) near the outfall site has been proposed. The purpose of such an undertaking would be twofold: to prevent eutrophication of the surrounding ecosystem by sequestering the nutrients in question, and to reduce the need for nutrient input, which is one of the largest costs in macroalgae cultivation.

Once grown, a variety of products can be derived from macroalgae, including biofuel, fish/cattle feed stock, and high value chemical materials such as alginate and agar. Food for human consumption is also a common product of kelp aquaculture, though it may not be ideal for a wastewater treatment application. Thus, there is an ongoing effort to investigate the feasibility, and optimal implementation of kelp farming in wastewater treatment operations.

Industrial scale macroalgae cultivation has long existed in Eastern Asia due to the popularity of seaweed in Asian cuisine. More recently, kelp aquaculture has been developing in Scandinavia and in the Northeastern United States. For example, the MACROSEA project is a four year international research collaboration funded by the Research Council of Norway targeting "successful and predictable production of high quality biomass thereby making significant steps towards industrial macroalgae cultivation in Norway." The project includes both cultivators and scientists, working to develop a precise understanding of the full life cycle of kelp and its interaction with its environment. A fundamental aspect of this endeavor is the development of mathematical models to describe the growth of kelp. Work is underway at SINTEF, a private Norwegian research institution, to develop such models. Ole Jacob Broch is a mathematician at SINTEF, a research organization in Trondheim, Norway, who has been working to model the growth of *Saccharina Latissima* using SINMOD, a large-scale 3D hydrodynamical ocean model developed at SINTEF which generates data on water temperature, water velocity, light intensity, and phytoplankton concentrations among other valuable quantities [13].

One aspect of the model which has yet to be fully developed is the availability of light, considering factors such as absorption and scattering by the aquatic medium, as well as by the kelp itself. In this thesis, we contribute to this effort by developing a first-principles model of the light field in a kelp farming environment. As a first step, a model for the spatial distribution of kelp is developed. Radiative transfer theory is then applied to determine the effects of the kelp and water on the availability of light throughout the medium. We pursue a numerical finite difference solution to the Radiative Transfer Equation, and subsequently discuss asymptotic approximations which prove to be sufficiently accurate and less computationally intensive. We also provide a detailed description of the numerical solution of this model, accompanied by source code for a FORTRAN implementation of the solution. This model can be used independently, or in conjunction with a life cycle kelp model to determine the amount of light available for photosynthesis at a single time step.

Figure 1.1: *Saccharina Latissima* being harvested

1.2 Background on Kelp Models

Mathematical modeling of macroalgae growth is not a new topic, although it is a reemerging one. Several authors in the second half of the twentieth century were interested in describing the growth and composition of the macroalgae *Macrocystis pyrifera*, commonly known as “giant kelp,” which grows prolifically off the coast of southern California. The first such mathematical model was developed by W.J. North for the Kelp Habitat Improvement Project at the California Institute of Technology in 1968 using seven variables. By 1974, Nick Anderson greatly expanded on North's work, and created the first comprehensive model of kelp growth which he programmed using FORTRAN [1]. In his model, he accounts for solar radiation intensity as a function of time of year and time of day, and refraction on the surface of the water. He uses a simple model for shading, simply specifying a single parameter which determines the percentage of light which is allowed to pass through the kelp canopy floating on the surface of the water. He also accounts for attenuation due to turbidity using Beer's Law. Using this data on the availability of light, he calculates the photosynthesis rates and the growth experienced by the kelp.

Over a decade later in 1987, G.A. Jackson expanded on Anderson's model for *Macrocystis pyrifera*, with an emphasis on including more environmental parameters and a more complete description of the growth and decay of the kelp [7]. He takes into account respiration, frond decay, and most importantly for my work, sub-canopy light attenuation due to self-shading. He simply adds a coefficient to the exponential decay of light as a function of depth to represent shading from kelp fronds. He doesn't consider and radial nor angular dependence on shading. Jackson also expands Anderson's definition of canopy shading, treating the canopy not as a single layer, but as 0, 1, or 2 discrete layers, each composed of individual fronds. While this is a significant improvement over Anderson's light model, it is still rather simplistic.

Both Anderson's and Jackson's model were carried out by numerically solving a system of differential equations over small time intervals. In 1990, M.A. Burgman and V.A. Gerard developed a stochastic population model [3]. This approach is quite different, and functions by dividing kelp plants into groups based on size and age, and generating random numbers to determine how the population distribution over these groups changes over time, based on measured rates of growth, death, decay, light availability, etc. That same year, Nyman et. al. tested a similar model in New Zealand, as well as a Markov chain model, and compared the results with experimental data [9].

In 1996 and 1998 respectively, P. Duarte and J.G. Ferreira used the size-class approach to create a more general model of macroalgae growth, and Yoshimori et. al. created a differential equation model of *Laminaria religiosa* with specific emphasis on temperature dependence of growth rate [5, 14]. These were the some of the first models of kelp growth that did not specifically relate to *Macrocystis pyrifera* ("giant kelp"). Initially, there was a great deal of excitement about this species due to it's incredible size and growth rate, but difficulties in harvesting and negative environmental impacts have caused scientists to investigate other kelp species.

1.3 Background on Radiative Transfer

In terms of optical quantities, our primary interest is in the radiance at each point from all directions, which affects the photosynthetic rate of the kelp, and therefore the total amount of biomass producible in a given area as well as the total nutrient remediation potential. The equation governing the radiance throughout the system is known as the Radiative Transfer Equation (RTE), which has been largely unutilized in the fields of oceanography and aquaculture. Meanwhile, it has been studied extensively in two fields: stellar astrophysics and computer graphics. In its full form, radiance is a function of 3 spatial dimensions, 2 angular dimensions, and frequency, making for an incredibly complex problem. In this work, frequency is ignored, only monochromatic radiation is considered. The RTE states that along a given path, radiance is decreased by absorption and scattering out of the path, while it is increased by emission and scattering into the path. In our situation, emission is negligible, owing only perhaps to some small luminescent phytoplankton or some such anomaly, and can therefore be safely ignored.

We use monochromatic radiative transfer in order to model the light field in an aqueous environment populated by vegetation. The vegetation (kelp) is modeled by a spatial probability distribution, which we assume to be given. The two quantities we seek to compute are *radiance* and *irradiance*. Radiance is the intensity of light in at a particular point in a particular direction, while irradiance is the total light intensity at a point in space, integrated over all angles. The Radiative Transfer

Equation is an integro-partial differential equation for radiance, which has been used primarily in stellar astrophysics; it's application to marine biology is fairly recent [8].

Understanding the growth rate and nutrient recovery by kelp cultures has important marine biological implications. For example, recent work by our research group at Clarkson University, the University of Maine, and SINTEF Fisheries and Aquaculture is investigating kelp aquaculture as a means to recover nutrients from wastewater effluent plumes in coastal environments into a valuable biomass feedstock for many products. Current models for kelp growth place little emphasis on the way in which nearby plants shade one another. Self-shading may be a significant model feature, though, as light availability may impact the growth and composition of the kelp biomass, and thus the mixture of goods that may be derived.

1.4 Overview of Thesis

The remainder of this document is organized as follows. In Chapter 2, we develop a probabilistic model to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. We begin Chapter 3 with a survey of fundamental radiometric quantities and optical properties of matter. We use the spatial kelp distribution from Chapter 2 to determine optical properties of the combined water-kelp medium. We then present the Radiative Transfer Equation, an integro-partial differential equation which describes the the light field as a function of position and angle. An asymptotic expansion is explored for cases where absorption dominates scattering in the medium, such as in very clear water or high kelp density. In Chapter 4, details are given for the numerical solution of the equations from Chapters 2 and 3. Both the full finite difference solution and the asymptotic approximation are described. Next, in Chapter 5, we discuss the availability of necessary parameters in the literature. For those which are not readily available, we give rough estimates and briefly describe experimental methods for their determination. Then, in Chapter 6, we investigate the necessary grid resolution for adequate accuracy in the full finite difference solution and compare to the asymptotic approximation for a few parameter sets. Further, we determine the effect of varying a few key parameters on the light field predicted by the asymptotic approximation. Afterwards, we use the light model developed here in a full lifecycle simulation of kelp growth and compare the light field and biomass production to those predicted by a simpler 1D exponential decay light model. Finally, we conclude with Chapter 7 by giving a brief summary of the model, discuss and its performance, and suggest improvements and avenues for future work.

CHAPTER II

KELP MODEL

In order to properly model the spatial distribution of light around the kelp, it is first necessary to formulate a spatial description of the kelp, which we do in this chapter. We take a probabilistic approach to describing the kelp. We begin by describing the distribution of kelp fronds, and through algebraic manipulation, we are able to assign to each point in space a probability that kelp occupies the location.

2.1 Physical Setup

Being a salt water species, macroalgae cultivation occurs primarily in the ocean, with the exception of the initial stage of growth, where microscopic kelp spores are inoculated onto a thread in a small laboratory pool. This thread is then wrapped around a large rope, which is placed in the ocean and generally suspended by buoys in one of two configurations: horizontal or vertical. Thus far, I am primarily concerned with modeling the vertical rope case, in which the kelp plants extend radially outward from the rope in all directions, which are made up of a single frond (leaf), stipe (stem) and holdfast (root structure). We consider a rectangular grid of such vertical ropes. Plants extending from each rope will shade both themselves and their neighbors to varying degrees based on the depth of the kelp, the rope spacing, the angle of incident light on the surface and the nature of scattering in the water. In addition, light will be naturally absorbed by the water to varying degrees as determined by the clarity of the water.

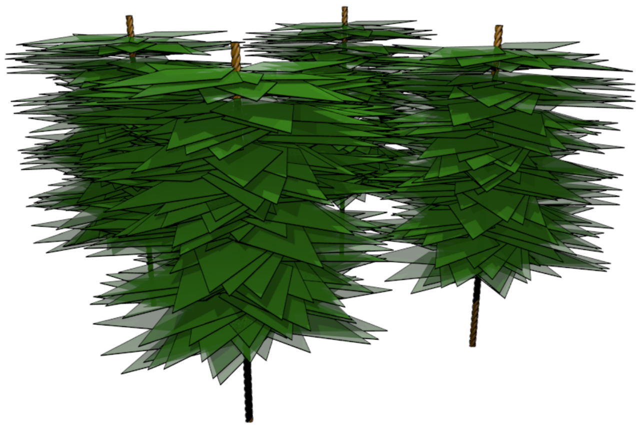

Figure 2.1: 4×4 array of vertical kelp ropes

2.2 Coordinate System

Consider the rectangular domain

$$x_{\text{min}} \leq x \leq x_{\text{max}},$$

$$y_{\text{min}} \leq y \leq y_{\text{max}},$$

$$z_{\text{min}} \leq z \leq z_{\text{max}}.$$

For all three dimensional analysis, we use the absolute coordinate system defined in figure 2.2. In the following sections, it is necessary to convert between Cartesian and spherical coordinates, which we do using the relations

$$\begin{aligned} x &= r\sin\phi\cos\theta, \\ y &= r\sin\phi\sin\theta, \\ z &= r\cos\phi. \end{aligned} \tag{2.1}$$

Therefore, for some function $f(x, y, z)$, we can write its derivative along a path in spherical coordinates in terms of Cartesian coordinates using the chain rule.

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial r} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial r} \tag{2.2}$$

Then, calculating derivatives from (2.1) yields

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\sin\phi\cos\theta + \frac{\partial f}{\partial y}\sin\phi\sin\theta + \frac{\partial f}{\partial z}\cos\phi. \tag{2.3}$$

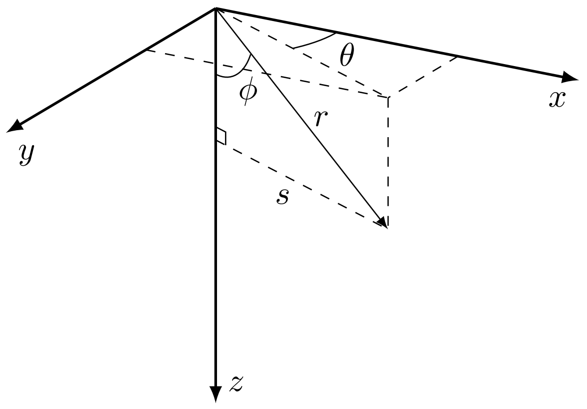

Figure 2.2: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ

2.3 Population Distributions

2.3.1 Frond Shape

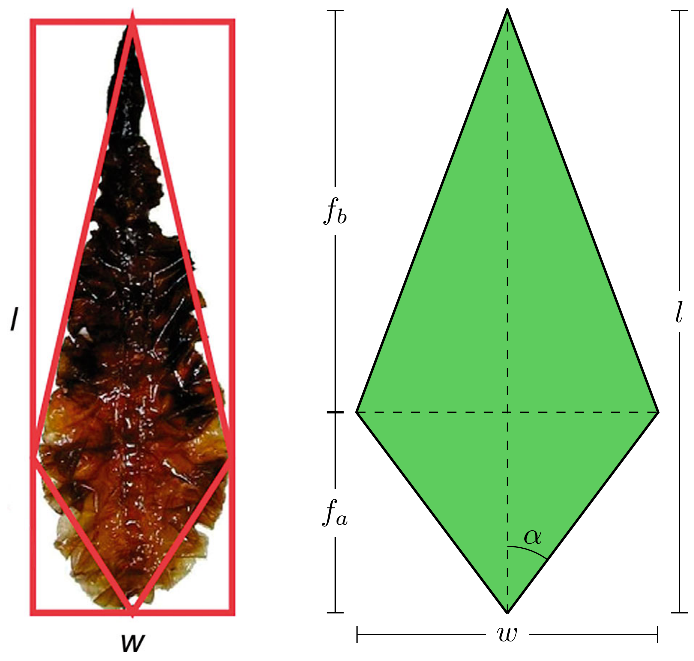

Figure 2.3: Simplified kite-shaped frond

We assume the frond is a kite with length l from base to tip, and width w from left to right. The shortest distance from the base to the diagonal connecting the left and right corners is called f_a, and the shortest distance from that diagonal to the tip is called f_b. We have

$$f_a + f_b = l \tag{2.4}$$

When considering a whole population with varying sizes, it is more convenient to specify ratios than absolute lengths. Let the following ratios be defined.

$$f_r = \frac{l}{w} \tag{2.5}$$

$$f_s = \frac{f_a}{f_b} \tag{2.6}$$

These ratios are assumed to be consistent among the entire population, making all fronds geometrically similar. With these definitions, the shape of the frond can be fully specified by l, f_r, and f_s. It is possible, then, to redefine w, f_a and f_b as follows from the preceding formulas.

$$w = \frac{l}{f_r} \tag{2.7}$$

$$f_a = \frac{l f_s}{1 + f_s} \tag{2.8}$$

$$f_b = \frac{l}{1 + f_s} \tag{2.9}$$

The angle α, half of the angle at the base corner, is also important in our analysis. Using the above equations,

$$\alpha = \tan^{-1}\left(\frac{2 f_r f_s}{1 + f_s}\right) \tag{2.10}$$

The area of the frond is given by

$$A = \frac{lw}{2} = \frac{l^2}{2 f_r}. \tag{2.11}$$

Likewise, if the area is known, then the length is

$$l = \sqrt{2 A f_r} \tag{2.12}$$

2.3.2 Length and Angle Distributions

We assume that frond lengths are normally distributed with mean μ_l and standard deviation σ_l. We assume the frond angle varies according to the von Mises distribution, which is the periodic analogue of the normal distribution, defined on $[-\pi, \pi]$ rather than $(-\infty, \infty)$. The von Mises distribution has two parameters, μ and κ, which shift and sharpen its peak respectively, as shown in Figure 2.4. κ can be considered analogous to $1/\sigma$ in the normal distribution. Here, we use $\mu = \theta_w$ and $\kappa = v_w$. That is, in the case of zero current velocity, the frond angles are be distributed uniformly, while as current velocity increases, they become increasingly likely to be pointing in the direction of the current. Note that θ_w and v_w vary over depth.

The PDF for this distribution is

$$P_{\theta_f}(\theta_f) = \frac{\exp\left(v_w \cos(\theta_f - v_w)\right)}{2\pi I_0(v_w)} \tag{2.13}$$

where $I_0(x)$ is the modified Bessel function of the first kind of order 0. Notice that unlike the normal distribution, the von Mises distribution approaches a *non-zero* uniform distribution as κ approaches 0.

$$\lim_{v_w \to 0} P_{\theta_f}(\theta_f) = \frac{1}{2\pi} \, \forall \, \theta_f \in [-\pi, \pi] \tag{2.14}$$

2.3.3 Joint Length-Orientation Distribution

The previous two distributions can reasonably be assumed to be independent of one another. That is, the angle of the frond does not depend on the length, or vice versa. Therefore, the probability of a frond simultaneously having a given frond length and angle is the product of their individual probabilities.

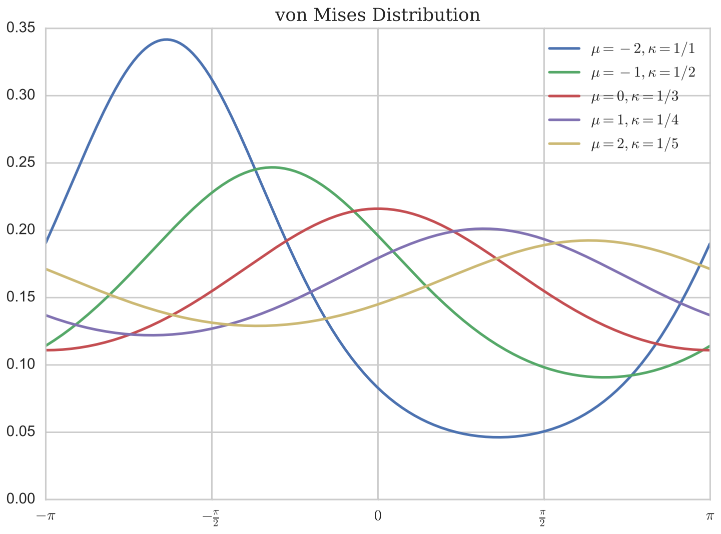

Figure 2.4: von Mises distribution for a variety of parameters

Given independent events A and B,

$$P(A \cap B) = P(A)P(B) \tag{2.15}$$

Then the probability of frond length l and frond angle θ_f coinciding is

$$P_{2D}(\theta_f, l) = P_{\theta_f}(\theta_f) \cdot L(l) \tag{2.16}$$

A contour plot of this 2D distribution for a specific set of parameters is shown in figure 2.5, where probability is represented by color in the 2D plane. Darker green represents higher probability, while lighter beige represents lower probability. In figure 2.6, 50 samples are drawn from this distribution and plotted.

It is important to note that if P_{θ_f} were dependent on l, the above definition of P_{2D} would no longer be valid. For example, it might be more realistic to say that larger fronds are less likely to bend towards the direction of the current. In this case, (2.15) would no longer hold, and it would be necessary to use the following more general relation.

$$P(A \cap B) = P(A)P(B|A) = P(B)P(B|A) \tag{2.17}$$

This is currently not taken into consideration in this model.

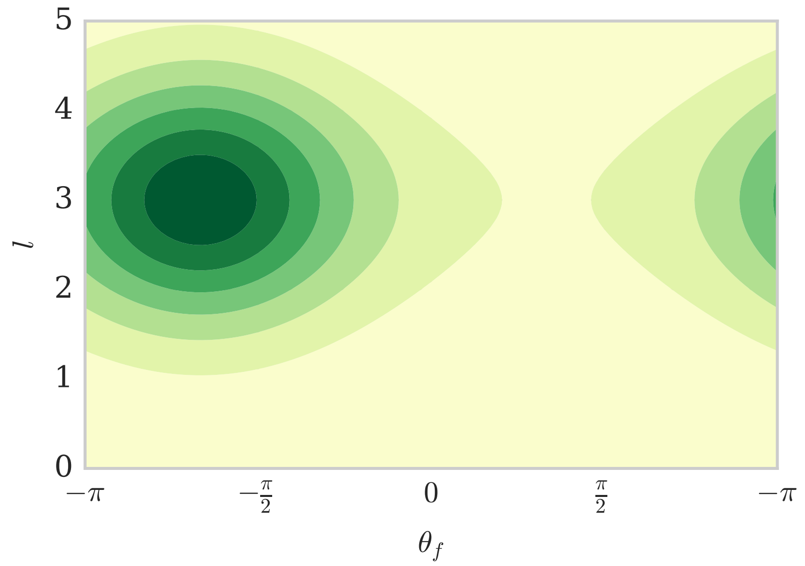

Figure 2.5: 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$

2.4 Spatial Distribution

2.4.1 Rotated Coordinate System

To determine under what conditions a frond will occupy a given point, we begin by describing the shape of the frond in Cartesian and then converting to polar coordinates. Of primary interest are the edges connected to the frond tip. For convenience,

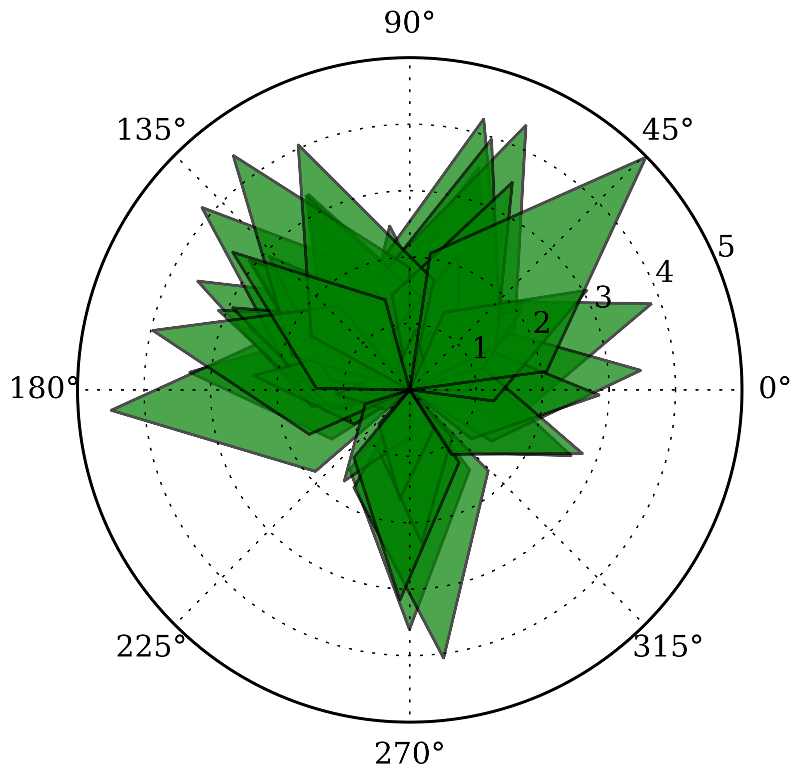

Figure 2.6: A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$.

we will use a rotated coordinate system (θ', s) such that the line connecting the base to the tip is vertical, with the base at $(0,0)$. The Cartesian analogue of this coordinate system, (x', y'), has the following properties.

$$x' = s\cos\theta' \tag{2.18}$$

$$y' = s\sin\theta' \tag{2.19}$$

and

$$s = \sqrt{x'^2 + y'^2} \tag{2.20}$$

$$\theta' = \operatorname{atan2}(y, x) \tag{2.21}$$

2.4.2 Functional Description of Frond Edge

With this coordinate system established, we can describe the outer two edges of the frond in Cartesian coordinates as a piecewise linear function connecting the left corner: $(-w/2, f_a)$, the tip: $(0, l)$, and the right corner: $(w/2, f_a)$. This function has the form

$$y'_f(x') = l - \operatorname{sign}(x')\frac{f_b}{w/2}x'. \tag{2.22}$$

Using the equations in Section 2.4.1, this can be written in polar coordinates after some rearrangement as

$$s'_f(\theta') = \frac{l}{\sin\theta' + S(\theta')\frac{2f_b}{w}\cos\theta'} \tag{2.23}$$

where

$$S(\theta') = \operatorname{sign}(\theta' - \pi/2) \tag{2.24}$$

Then, using the relationships in Section 2.3.1, we can rewrite the above equation in terms of our frond ratios f_s and f_r.

$$s'_f(\theta') = \frac{l}{\sin\theta' + S(\theta')\frac{2f_r}{1+f_s}\cos\theta'} \tag{2.25}$$

2.4.3 Absolute Coordinates

To generalize to a frond pointed at an angle θ_f, we will use the coordinate system (θ, s) such that

$$\theta = \theta' + \theta_f - \frac{\pi}{2} \tag{2.26}$$

Then, for a frond pointed at the arbitrary angle θ_f, the function for the outer edges can be written as

$$s_f(\theta) = s'_f\left(\theta - \theta_f + \frac{\pi}{2}\right) \tag{2.27}$$

2.4.4 Conditions for Occupancy

Consider a fixed frond of length l at an angle θ_f. The point (θ, s) is occupied by the frond if

$$|\theta_f - \theta| < \alpha \tag{2.28}$$

and

$$s < s_f(\theta) \tag{2.29}$$

Equivalently, letting the point (θ, s) be fixed, a frond occupies the point if the following conditions are satisfied.

$$\theta - \alpha < \theta_f < \theta + \alpha \tag{2.30}$$

and

$$l > l_{min}(\theta, s) \tag{2.31}$$

where

$$l_{min}(\theta, s) = s \cdot \frac{l}{s_f(\theta)} \tag{2.32}$$

Then, considering the point to be fixed, (2.30) and (2.31) define the spacial region $R_s(\theta, s)$ called the "occupancy region for (θ, s)" with the property that if the tip of a frond lies within this region (i.e. $(\theta_f, l) \in R_s(\theta, s)$), then it occupies the point. $R_s(3\pi/4, 3/2)$ is shown in blue in figure 2.7 and the smallest possible occupying fronds for several values of θ_f are shown in various colors. Any frond longer than these at the same angle will also occupy the point.

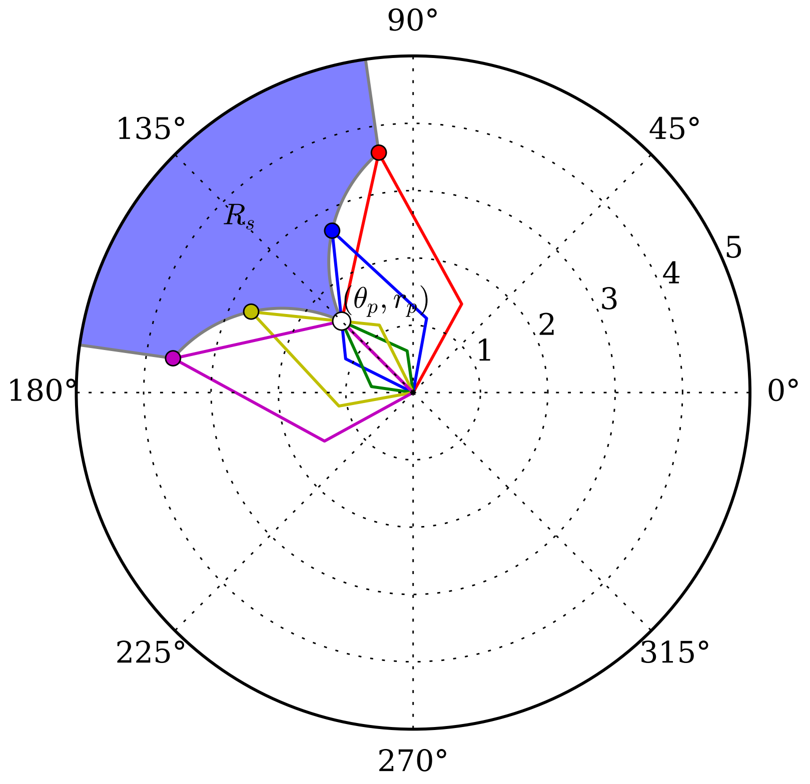

Figure 2.7: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$

2.4.5 Probability of Occupancy

We are interested in the probability that, given a fixed point (θ, s), values of l and θ_f chosen from the distributions described in Section 2.3.2 will fall in the occupancy region. This is found by integrating P_{2D} over the occupancy region for (θ, s), as depicted in figure 2.8.

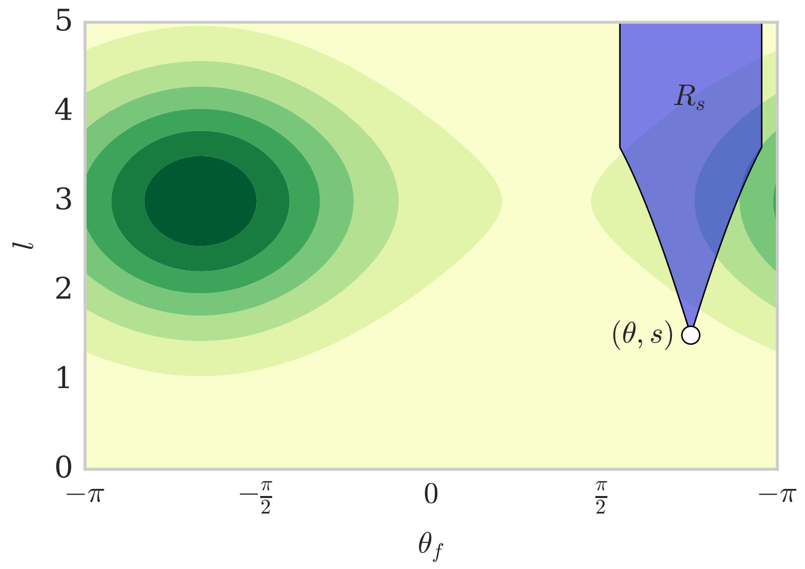

Figure 2.8: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the $\theta_f - l$ plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$

Now, integrating $P_{2D}(\theta_f, l)$ over $R_s(\theta, s)$ yields the proportion of the population occupying the point (θ, s).

$$\begin{aligned}\tilde{P}_k(\theta, s, z) &= \iint_{R_s(\theta,s)} P_{2D}(\theta_f, l)\, dl\, d\theta_f \\ &= \int_{\theta-\alpha}^{\theta+\alpha} \int_{l_{min}(\theta_f)}^{\infty} P_{2D}(\theta_f, l)\, dl\, d\theta_f \end{aligned} \tag{2.33}$$

Then, multiplying $\tilde{P}_k$ by the number of fronds in the population n of the depth layer gives the expected number of fronds occupying the point. Now, assuming a uniform thickness t for all fronds, and a thickness dz of the depth layer, we find the proportion of the grid cell occupied by kelp to be

$$P_k = \frac{nt}{dz}\tilde{P}. \tag{2.34}$$

Then, the effective absorption coefficient can be calculated at any point in space as

$$a(\vec{x}) = P_k(\vec{x})a_k + (1 - P_k(\vec{x}))a_w \tag{2.35}$$

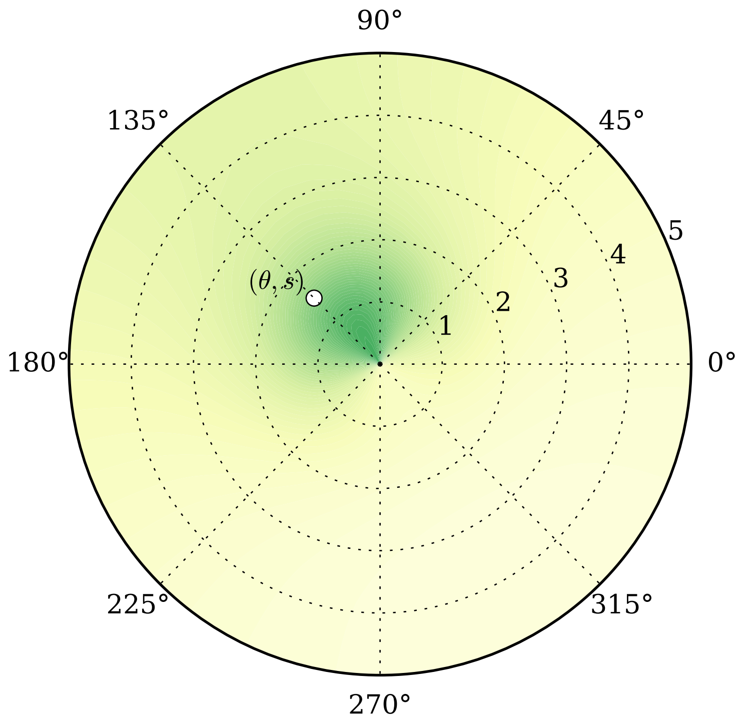

Figure 2.9: Contour plot of the probability of occupying sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$

CHAPTER III

LIGHT MODEL

Now that we have formulated the distribution of kelp throughout the medium, we introduce the radiative transfer equation, which is used to calculate the light field.

3.1 Optical Definitions

3.1.1 Radiometric Quantities

One of the most fundamental quantities in optics is radiant flux Φ, which is the has units of energy per time. The quantity of primary interest in modeling the light field is radiance L, which is defined as the radiant flux per steradian per projected surface area perpendicular to the direction of propagation of the beam. That is,

$$L = \frac{d^2\Phi}{dA d\omega} \tag{3.1}$$

Once the radiance L is calculated everywhere, the irradiance is

$$I(\vec{x}) = \int_{4\pi} L(\vec{x}, \vec{\omega}) \, d\omega. \tag{3.2}$$

Integrating $I(\vec{x})$, which has units W/m^2, over the surface of a frond, produces the power (with units W) transmitted to the frond. This is discussed further in Section 4.4.1

Irradiance can be converted to moles of photons (also called Einsteins) per second as

$$1\,\mathrm{W/m^2} = 4.2\,\mu\mathrm{mol\,photons/s}. \tag{3.3}$$

3.1.2 Inherent Optical Properties

We must now define a few inherent optical properties (IOPs) which depend only on the medium of propagation. These phenomena are governed by three inherent optical properties (IOPs) of the medium. The absorption coefficient $a(\vec{x})$ (units m^{-1}) defines the proportional loss of radiance per unit length. The scattering coefficient b (units m^{-1}), defines the proportional loss of radiance per unit length, and is assumed to be constant over space.

The volume scattering function (VSF) $\beta(\Delta) : [-1, 1] \to \mathbb{R}^+$ (units sr^{-1}) defines the probability of light scattering at any given angle from its source. Formally, given two directions $\vec{\omega}$ and $\vec{\omega}'$, $\beta(\vec{\omega} \cdot \vec{\omega}')$ is the probability density of light scattering from $\vec{\omega}$ into $\vec{\omega}'$ (or vice-versa). Of course, since a single direction subtends no solid angle, the probability of scattering occurring exactly from $\vec{\omega}$ to $\vec{\omega}'$ is 0. Rather, we say that the probability of radiance being scattered from a direction ω into an element of solid angle Ω is $\int_\Omega \beta(\vec{\omega} \cdot \vec{\omega}')\, d\vec{\omega}'$.

The VSF is normalized such that

$$\int_{-1}^{1} \beta(\Delta)\, d\Delta = \frac{1}{2\pi}, \tag{3.4}$$

so that for any ω,

$$\int_{4\pi} \beta(\vec{\omega} \cdot \vec{\omega}')\, d\vec{\omega}' = 1. \tag{3.5}$$

i.e., the probability of light being scattered to some direction on the unit sphere is 1.

3.2 The Radiative Transfer Equation

3.2.1 Ray Notation

Consider a fixed position $\vec{x}$ and direction $\vec{\omega}$ such that $\vec{\omega} \cdot \hat{z} \neq 0$.

Let $\vec{l}(\vec{x}, \vec{\omega}, s)$ denote the linear path containing $\vec{x}$ with initial z coordinate given by

$$z_0 = \begin{cases} 0, & \vec{\omega} \cdot \hat{z} < 0 \\ z_{\text{max}}, & \vec{\omega} \cdot \hat{z} > 0 \end{cases} \tag{3.6}$$

Then,

$$\vec{l}(\vec{x}, \vec{\omega}, s) = \frac{1}{\tilde{s}}(s\vec{x} + (\tilde{s} - s)\vec{x_0}(\vec{x}, \vec{\omega})) \tag{3.7}$$

where

$$\vec{x_0}(\vec{x}, \vec{\omega}) = \vec{x} - \tilde{s}\vec{\omega} \tag{3.8}$$

is the origin of the ray, and

$$\tilde{s} = \frac{\vec{x} \cdot \hat{z} - z_0}{\vec{\omega} \cdot \hat{z}} \tag{3.9}$$

is the path length from $\vec{x_0}(\vec{x}, \vec{\omega})$ to $\vec{x}$.

3.2.2 Colloquial Description

Denote the radiance at $\vec{x}$ in the direction $\vec{\omega}$ by $L(\vec{x}, \vec{\omega})$. As light travels along $\vec{l}(\vec{x}, \vec{\omega}, s)$, interaction with the medium produces three phenomena of interest:

1. Radiance is decreased due to absorption.

2. Radiance is decreased due to scattering out of the path to other directions.

3. Radiance is increased due to scattering into the path from other directions.

3.2.3 Equation of Transfer

Then, combining these phenomena, the Radiative Transfer equation along $\vec{l}(\vec{x},\vec{\omega})$ becomes

$$\frac{dL}{ds}(\vec{l}(\vec{x},\vec{\omega},s),\vec{\omega}) = -(a(\vec{x})+b)L(\vec{x},\vec{\omega}) + b\int_{4\pi}\beta(\vec{\omega}\cdot\vec{\omega}')L(\vec{x})\,d\omega', \tag{3.10}$$

where $\int_{4\pi}$ denotes integration over the unit sphere.

Now, we have

$$\begin{aligned}\frac{dL}{ds}(\vec{l}(\vec{x},\vec{\omega},s),\vec{\omega}) &= \frac{d\vec{l}}{ds}(\vec{x},\vec{\omega},s)\cdot\nabla L(\vec{x},\vec{\omega}',\vec{\omega})\\ &= \vec{\omega}\cdot\nabla L(\vec{x},\vec{\omega})\end{aligned}$$

Then, the general form of the Radiative Transfer Equation is

$$\vec{\omega}\cdot\nabla L(\vec{x},\vec{\omega}) = -(a(\vec{x})+b)L(\vec{x},\vec{\omega}) + b\int_{4\pi}\beta(\vec{\omega}\cdot\vec{\omega}')L(\vec{x},\vec{\omega}')\,d\omega' \tag{3.11}$$

or, equivalently,

$$\vec{\omega}\cdot\nabla L(\vec{x},\vec{\omega}) + a(\vec{x})L(\vec{x},\vec{\omega}) = b\left(\int_{4\pi}\beta(\vec{\omega}\cdot\vec{\omega}')L(\vec{x},\vec{\omega}')\,d\omega' - L(\vec{x},\vec{\omega})\right) \tag{3.12}$$

3.2.4 Boundary Conditions

We use periodic boundary conditions in the x and y directions.

$$L\left((x_{\text{min}},y,z),\vec{\omega}\right) = L\left((x_{\text{max}},y,z),\vec{\omega}\right) \tag{3.13}$$

$$L\left((x,y_{\text{min}},z),\vec{\omega}\right) = L\left((x,y_{\text{max}},z),\vec{\omega}\right) \tag{3.14}$$

In the z direction, we specify a spatially uniform downwelling light just under the surface of the water by a function $f(\vec{\omega})$. Or if $z_{\text{min}} > 0$, then the radiance at $z = z_{\text{min}}$ should be specified instead (as opposed to the radiance at the first grid cell center).

Further, we assume that no upwelling light enters the domain from the bottom.

$$L(\vec{x_s}, \vec{\omega}) = f(\omega) \text{ if } \vec{\omega} \cdot \hat{z} > 0 \tag{3.15}$$

$$L(\vec{x_b}, \vec{\omega}) = 0 \text{ if } \vec{\omega} \cdot \hat{z} < 0 \tag{3.16}$$

3.3 Low-Scattering Approximation

In clear waters where absorption is more important than scattering, an asymptotic expansion can be used whereby the light field is generated through a sequence of discrete scattering events.

3.3.1 Asymptotic Expansion

Taking b to be small, we introduce the asymptotic series

$$L(\vec{x}, \vec{\omega}) = L_0(\vec{x}, \vec{\omega}) + bL_1(\vec{x}, \vec{\omega}) + b^2 L_2(\vec{x}, \vec{\omega}) + \cdots . \tag{3.17}$$

Then, substituting the above into the RTE,

$$\begin{aligned}
&\vec{\omega}\cdot\nabla\left[L_0(\vec{x},\vec{\omega})+bL_1(\vec{x},\vec{\omega})+b^2L_2(\vec{x},\vec{\omega})+\cdots\right]\\
&+a(\vec{x})\left[L_0(\vec{x},\vec{\omega})+bL_1(\vec{x},\vec{\omega})+b^2L_2(\vec{x},\vec{\omega})+\cdots\right]\\
&=b\Bigg(\int_{4\pi}\beta(|\vec{\omega}-\vec{\omega}'|)\left[L_0(\vec{x},\vec{\omega}')+bL_1(\vec{x},\vec{\omega}')+b^2L_2(\vec{x},\vec{\omega}')+\cdots\right]\,d\vec{\omega}'\\
&-\left[L_0(\vec{x},\vec{\omega})+bL_1(\vec{x},\vec{\omega})+b^2L_2(\vec{x},\vec{\omega})+\cdots\right]\Bigg)
\end{aligned}\tag{3.18}$$

Then, grouping like powers of b, we have the decoupled set of equations

$$\vec{\omega}\cdot\nabla L_0(\vec{x},\vec{\omega})+a(\vec{x})L_0(\vec{x})=0\tag{3.19}$$

$$\vec{\omega}\cdot\nabla L_1(\vec{x},\vec{\omega})+a(\vec{x})L_1(\vec{x})=\int_{4\pi}\beta(|\vec{\omega}-\vec{\omega}'|)L_0(\vec{x},\vec{\omega}')\,d\vec{\omega}'-L_0(\vec{x},\vec{\omega})\tag{3.20}$$

$$\vec{\omega}\cdot\nabla L_2(\vec{x},\vec{\omega})+a(\vec{x})L_2(\vec{x})=\int_{4\pi}\beta(|\vec{\omega}-\vec{\omega}'|)L_1(\vec{x},\vec{\omega}')\,d\vec{\omega}'-L_1(\vec{x},\vec{\omega})\tag{3.21}$$

$$\vdots$$

For boundary conditions, let x_s be a point on the surface of the domain. Then,

$$L_0(\vec{x_s},\vec{\omega})+bL_1(\vec{x_s},\vec{\omega})+b^2L_2(\vec{x_s},\vec{\omega})+\cdots=\begin{cases}f(\omega), & \hat{z}\cdot\omega>0\\ 0, & \text{otherwise,}\end{cases}\tag{3.22}$$

which becomes

$$L_0(\vec{x},\vec{\omega}) = \begin{cases} f(\omega), & \hat{z}\cdot\omega > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{3.23}$$

$$L_1(\vec{x},\vec{\omega}) = 0 \tag{3.24}$$

$$L_2(\vec{x},\vec{\omega}) = 0. \tag{3.25}$$

$$\vdots$$

3.3.2 Analytical Solution

For all $\vec{x},\vec{\omega}$, let

$$\tilde{a}(s) = a(\vec{l}(\vec{x},\vec{\omega}),s), \tag{3.26}$$

$$\frac{du_0}{ds}(s) + \tilde{a}(s)u_0(s) = 0, u_0(0) = f(\vec{\omega}) \tag{3.27}$$

Then,

$$u_0(s) = f(\omega)\exp\left(-\int_0^s \tilde{a}(s)\,ds\right), \tag{3.28}$$

$$L_0(\vec{l}(\vec{x},\vec{\omega},s),\vec{\omega}) = u_0(s) \tag{3.29}$$

$$g_n(s) = \int_{4\pi} \beta(|\vec{\omega}-\vec{\omega}'|)L_{n-1}(\vec{l}(\vec{x},\vec{\omega'},s),\vec{\omega}')\,d\vec{\omega}' - L_{n-1}(\vec{l}(\vec{x},\vec{\omega},s),\vec{\omega}) \tag{3.30}$$

$$\frac{du_n}{ds}(s) + \tilde{a}(s)u_n(s) = g_n(s), u_n(0) = 0 \tag{3.31}$$

Then,

$$u_n(s) = \int_0^s g_n(s') \exp\left(- \int_{s''}^{s'} \tilde{a}(s'')\, ds'' \right) ds' \tag{3.32}$$

$$L_n(\vec{l}(\vec{x}, \vec{\omega}, s), \vec{\omega}) = u_n(s) \tag{3.33}$$

CHAPTER IV

NUMERICAL SOLUTION

In this chapter, the mathematical details involved in the numerical solution of the previously described equations are presented. It is assumed that this model is run in conjunction with a model describing the growth of kelp over its life cycle, which calls this light model periodically to update the light field.

4.1 Super-Individuals

The algorithm described in this chapter has two components. First, a probabilistic description of the kelp is generated at each point in a discrete spatial grid. Second, optical properties of the resulting kelp-water medium are derived, and the light field is calculated. The first component is described here.

4.1.1 Frond Length Distribution

Rather than model each kelp frond, a subset of the population, called super-individuals, are modelled explicitly, and are considered to represent many identical individuals, as in [11]. Specifically, at each depth k, there are n super-individuals, indexed by i. Super-individual i has a frond area A_{ki} and represents n_{ki} individual fronds.

From (2.12), the frond length of the super-individual is $l_{ki} = \sqrt{2A_{ki}f_r}$. Given the super-individual data, we calculate the mean μ and standard deviation σ frond lengths using the formulas:

$$\mu_k = \frac{\sum_{i=1}^{N} l_{ki}}{\sum_{i=1}^{N} n_{ki}}, \tag{4.1}$$

$$\sigma_k = \frac{\sum_{i=1}^{N} (l_{ki} - \mu_k)^2}{\sum_{i=1}^{N} n_{ki}}. \tag{4.2}$$

We then assume that frond lengths are normally distributed in each depth layer with mean μ_k and standard deviation σ_k.

4.2 Discrete Grid

The following is a description of the uniform, rectangular spatial-angular grid used in the numerical implementation of this model. It is assumed that all simulated quantities are constant over the interior of a grid cell.

The number of grid cells in each dimension are denoted by n_x, n_y, n_z, n_θ, and n_ϕ, with uniform spacings dx, dy, dz, $d\theta$, and $d\phi$ between adjacent grid points.

The following indices are assigned to each dimension:

$$x \to i \tag{4.3}$$

$$y \to j \tag{4.4}$$

$$z \to k \tag{4.5}$$

$$\theta \to l \tag{4.6}$$

$$\phi \to m \tag{4.7}$$

It is convenient, however, to use a single index p to refer to directions $\vec{\omega}$ rather than referring to θ and ϕ separately. Then, the center of a generic grid cell will be denoted as $(x_i, y_j, z_k, \vec{\omega}_p)$, and the boundaries between adjacent grid cells will be referred to as *edges*. One-indexing is employed throughout this document.

4.2.1 Spatial Grid

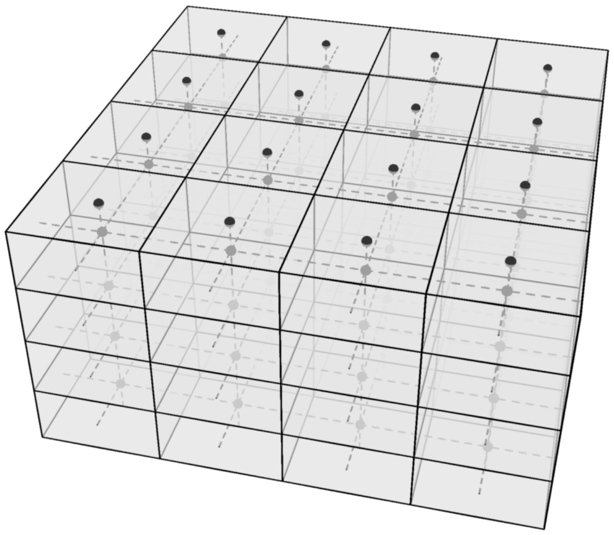

Figure 4.1: Spatial grid

$$dx = \frac{x_{\max} - x_{\min}}{n_x} \tag{4.8}$$

$$dy = \frac{y_{\max} - y_{\min}}{n_y} \tag{4.9}$$

$$dz = \frac{z_{\max} - z_{\min}}{n_z} \tag{4.10}$$

Denote the edges as

$$x_i^e = (i-1)x \text{ for } i = 1, \ldots, n_x \tag{4.11}$$

$$y_j^e = (j-1)y \text{ for } j = 1, \ldots, n_y \tag{4.12}$$

$$z_k^e = (k-1)z \text{ for } k = 1, \ldots, n_z \tag{4.13}$$

and the cell centers as

$$x_i = (i-1/2)dx \text{ for } i = 1, \ldots, n_x \tag{4.14}$$

$$y_j = (j-1/2)dy \text{ for } j = 1, \ldots, n_y \tag{4.15}$$

$$z_k = (k-1/2)dz \text{ for } k = 1, \ldots, n_z \tag{4.16}$$

Note that in this convention, there are the same number of edges and cells, and edges preceed centers.

Also, note that no grid center is located on the plane $z = 0$. The surface radiance boundary condition is treated separately.

4.2.2 Angular Grid

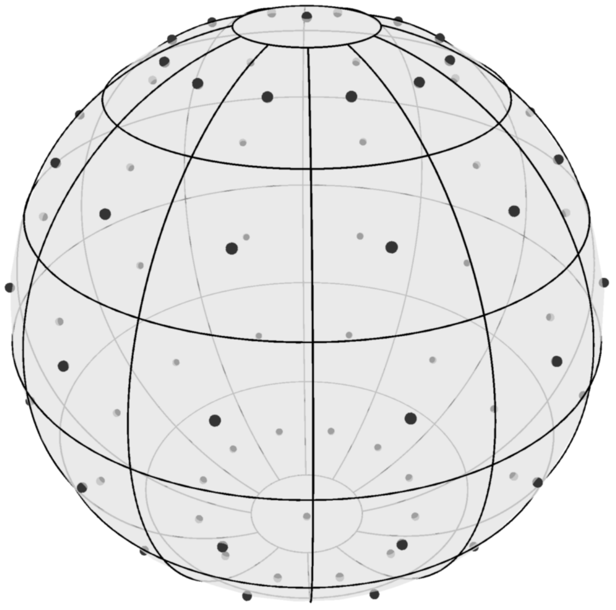

Figure 4.2: Angular grid at each point in space

Now, we define the azimuthal angle such that

$$\theta_l = (l-1)d\theta. \tag{4.17}$$

For the sake of periodicity, we need

$$\theta_1 = 0, \tag{4.18}$$

$$\theta_{n_\theta} = 2\pi - d\theta, \tag{4.19}$$

which requires

$$d\theta = \frac{2\pi}{n_\theta}. \tag{4.20}$$

For the polar angle, we similarly let

$$\phi_m = (m-1)d\phi \tag{4.21}$$

Since the polar azimuthal is not periodic, we also store the endpoint, so

$$\phi_1 = 0, \tag{4.22}$$

$$\phi_{n_\phi} = \pi. \tag{4.23}$$

This gives us

$$d\phi = \frac{\pi}{n_\phi - 1}. \tag{4.24}$$

It is also useful to define the edges between angular grid cells as

$$\theta_l^e = (l - 1/2)d\theta, \qquad l = 1, \ldots, n_\theta \tag{4.25}$$

$$\phi_m^e = (m - 1/2)d\phi, \quad m = 1, \ldots, n_\phi - 1. \tag{4.26}$$

Note that while θ has its final edge following its final center, this is not the case for ϕ.

As shown in Figure 4.2, $\phi = 0$ and $\phi = \pi$, called the north ($+z$) and south ($-z$) poles respectively, are treated separately. The total number of angles considered is $n_{\vec{\omega}} = n_\phi n_\theta - 2(n_\theta - 1)$. Since the poles create a non-rectangular angular grid in the sense that $n_{\vec{\omega}}$ is not the product of two integers, it is advantageous to use a single variable $p = 1, \ldots, n_{\vec{\omega}}$ to index angles $\vec{\omega} = (\theta, \phi)$ such that $p \in \{2, \ldots, n_{\vec{\omega}} - 1\}$ refers to the interior of the angular grid, and $p = 1$ and $p = n_{\vec{\omega}}$ refer to the north and south

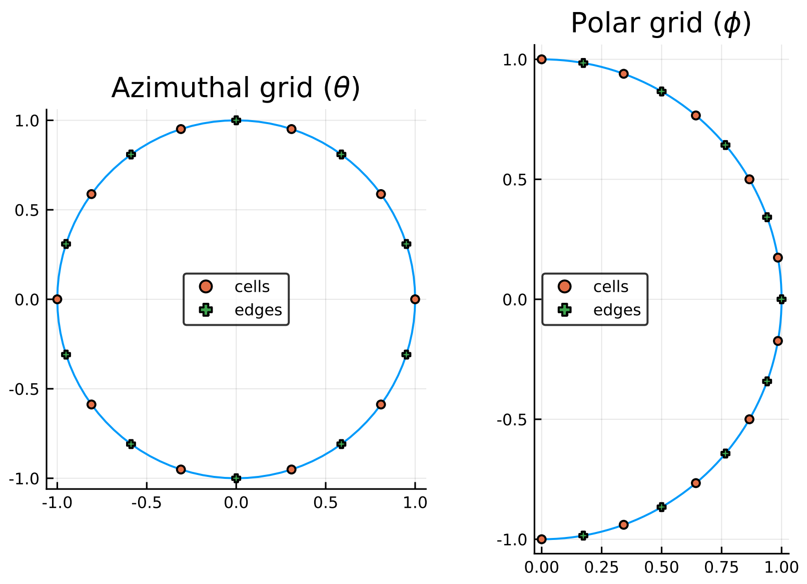

Figure 4.3: Angular grid

poles respectively. The following notation is used.

$$\hat{l}(p) = \text{mod1}(p, n_\theta) \tag{4.27}$$

$$\hat{m}(p) = \text{ceil}(p/n_\theta) + 1 \tag{4.28}$$

$$\hat{\theta}_p = \theta_{\hat{l}(p)} \tag{4.29}$$

$$\hat{\phi}_p = \phi_{\hat{m}(p)} \tag{4.30}$$

Thus, it follows that

$$p = (\hat{m}(p) - 2)\, n_\theta + \hat{l}(p). \tag{4.31}$$

Accordingly, define

$$\hat{p}(l, m) = (m - 1)n_\theta + l. \tag{4.32}$$

Further, we refer to the angular grid cell centered at $\vec{\omega}_p$ as Ω_p, and the solid angle subtended by Ω_p is denoted $|\Omega_p|$. The areas of the grid cells are calculated as follows. Note that there is a temporary abuse of notation in that the same symbols ($d\theta$ and $d\phi$) are being used for infinitessimal differential and for finite grid spacing.

For the poles, we have

$$|\Omega_1| = |\Omega_{n_{\vec{\omega}}}| = \int_{\Omega_1} d\vec{\omega} \tag{4.33}$$

$$= \int_0^{2\pi} \int_0^{d\phi/2} \sin\phi \, d\phi \, d\theta \tag{4.34}$$

$$= 2\pi \cos\phi \Big|_{d\phi/2}^{0} \tag{4.35}$$

$$= 2\pi(1 - \cos(d\phi/2)) \tag{4.36}$$

And for all other angular grid cells,

$$|\Omega_p| = \int_{\Omega_p} d\vec{\omega} \tag{4.37}$$

$$= \int_{\theta_l^e}^{\theta_{l+1}^e} \int_{\phi_m^e}^{\phi_{m+1}^e} \sin(\phi) \, d\phi \, d\theta \tag{4.38}$$

$$= d\theta \int_{\phi_m^e}^{\phi_{m+1}^e} \sin(\phi) \, d\phi \tag{4.39}$$

$$= d\theta \left(\cos(\phi_m^e) - \cos(\phi_{m+1}^e) \right). \tag{4.40}$$

4.2.3 Angular Quadrature

We assume that all quantities are constant within a spatial-angular grid cell. We therefore employ the midpoint rule for both spatial and angular integration.

Define the *angular characteristic function*

$$\mathcal{X}_p^{\Omega}(\vec{\omega}) = \begin{cases} 1, & \vec{\omega} \in \Omega_p \\ \\ 0, & \text{otherwise} \end{cases} \tag{4.41}$$

$$\begin{aligned} \int_{4\pi} f(\vec{\omega})\, d\vec{\omega} &= \int_{4\pi} \sum_{p=1}^{n_{\vec{\omega}}} f_p \mathcal{X}_p^{\Omega}(\vec{\omega})\, d\vec{\omega} && (4.42) \\ &= \sum_{p=1}^{n_{\vec{\omega}}} f_p \int_{4\pi} \mathcal{X}_p^{\Omega}(\vec{\omega})\, d\vec{\omega} && (4.43) \\ &= \sum_{p=1}^{n_{\vec{\omega}}} f_p \int_{\Omega_p} d\vec{\omega} && (4.44) \\ &= \sum_{p=1}^{n_{\vec{\omega}}} f_p \left|\Omega_p\right| && (4.45) \end{aligned}$$

4.2.4 Scattering Integral

Specifically, we integrate β to determine the amount of light scattered between angular grid cells.

Consider two angular grid cells, Ω and Ω'. The average probability density of scattering from $\vec{\omega} \in \Omega$ to $\vec{\omega}' \in \Omega'$ (or vice versa) is

$$\beta_{pp'} = \frac{1}{|\Omega|\,|\Omega'|} \int_{\Omega} \int_{\Omega'} \beta(\vec{\omega} \cdot \vec{\omega}')\, d\vec{\omega'}\, d\vec{\omega} \tag{4.46}$$

Denote the radiance at $(x_i, y_j, z_k, \vec{\omega}_p)$ by L_{ijkp}. Then, the total radiance scattered into Ω_p from $\Omega_{p'}$ is

$$\begin{aligned} \int_{\Omega} \int_{\Omega'} \beta(\vec{\omega} \cdot \vec{\omega}') L(\vec{x}, \vec{\omega}')\, d\vec{\omega}'\, d\vec{\omega} &= L_{ijkp'} \int_{\Omega} \int_{\Omega_{p'}} \beta(\vec{\omega} \cdot \vec{\omega}')\, d\vec{\omega}'\, d\vec{\omega} && (4.47) \\ &= \beta_{pp'}\, |\Omega|\, |\Omega'|\, L_{ijkp'}. && (4.48) \end{aligned}$$

Hence, the average radiance scattered is $\beta_{pp'}\, |\Omega'|\, L_{ijkp'}$.

4.3 Finite Difference

We now discuss the discretization of derivatives on the spatial grid.

4.3.1 Discretization

For the spatial interior of the domain, we use the 2nd order central difference formula (CD2) to approximate the derivatives, which is

$$f'(x) = \frac{f(x+dx) - f(x-dx)}{2dx} + \mathcal{O}(dx^3). \tag{CD2}$$

When applying the PDE on the upper or lower boundary, we use the forward and backward difference (FD2 and BD2) formulas respectively. Omitting $\mathcal{O}(dx^3)$, we have

$$f'(x) = \frac{-3f(x) + 4f(x+dx) - f(x+2dx)}{2dx} \tag{FD2}$$

$$f'(x) = \frac{3f(x) - 4f(x-dx) + f(x-2dx)}{2dx} \tag{BD2}$$

For the upper and lower boundaries, we need an asymmetric finite difference method. In general, the Taylor Series of a function f about x is

$$f'(x+\varepsilon) = \sum_{n=1}^{\infty} \frac{f^{(n)}(x)}{n!}\varepsilon^n \tag{4.49}$$

Truncating after the first few terms, we have

$$f'(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \mathcal{O}(\varepsilon^3) \tag{4.50}$$

Similarly, replacing ε with $-\varepsilon/2$ we have

$$f'(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \tag{4.51}$$

Rearranging (4.50) produces

$$f''(x)\varepsilon^2 = 2f(x+\varepsilon) - 2f(x) - 2f'(x)\varepsilon + \mathcal{O}(\varepsilon^3) \tag{4.52}$$

Combining (4.51) with (4.52) gives

$$\begin{aligned}
\varepsilon f'(x) &= 2f(x) - 2f(x-\frac{\varepsilon}{2}) + f''(x)\frac{\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3) \\
&= 2f(x) - 2f(x-\frac{\varepsilon}{2}) + \frac{f(x+\varepsilon)}{4} - \frac{f(x)}{4} - \frac{f'(x)\varepsilon}{4} + \mathcal{O}(\varepsilon^3) \\
&= \frac{4}{5}\left(2f(x) - 2f(x-\frac{\varepsilon}{2}) + \frac{f(x+\varepsilon)}{4} - \frac{f(x)}{4}\right) + \mathcal{O}(\varepsilon^3)
\end{aligned}$$

Then, dividing by ε gives

$$f'(x) = \frac{-8f(x-\frac{\varepsilon}{2}) + 7f(x) + f(x+\varepsilon)}{5\varepsilon} + \mathcal{O}(\varepsilon^2) \tag{4.53}$$

Similarly, substituting $\varepsilon \to -\varepsilon$, we have

$$f'(x) = \frac{-f(x-\varepsilon) - 7f(x) + 8f(x+\frac{\varepsilon}{2})}{5\varepsilon} + \mathcal{O}(\varepsilon^2) \tag{4.54}$$

4.3.2 Difference Equation

In general, we have

$$\vec{\omega}\cdot\nabla L_p = -(a+b)L_p + \sum_{p'=1}^{n_{\vec{\omega}}} \beta_{pp'} L_{p'}. \tag{4.55}$$

Then,

$$\vec{\omega}\cdot\nabla L_p + (a + b(1-\beta_{pp'}))L_p - \sum_{p'=1}^{n_{\vec{\omega}}} \beta_{pp'} L_{p'} = 0 \tag{4.56}$$

Interior:

$$\begin{aligned}
0 = {} & \frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
& + \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
& + \frac{L_{ij,k+1,p} - L_{ij,k-1,p}}{2dz} \cos\hat{\phi}_p \\
& + (a_{ijk} + b(1-\beta_{pp'}))L_{ijkp} - \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'}
\end{aligned} \tag{4.57}$$

Surface downwelling (BC):

$$\begin{aligned}
0 = {} & \frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
& + \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
& + \frac{-8f_p + 7L_{ijkp} + L_{ij,k+1,p}}{5dz} \cos\hat{\phi}_p \\
& + (a_{ijk} + b(1-\beta_{pp'}))L_{ijkp} \\
& - \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'}.
\end{aligned}$$

Combining L_{ijkp} terms on the left and moving the boundary condition to the right gives

$$\begin{aligned}
& \frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
& + \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
& + \frac{L_{ij,k+1,p}}{5dz} \cos\hat{\phi}_p \\
& + (a_{ijk} + b(1-\beta_{pp'}) + \frac{7}{5dz}\cos\hat{\phi}_p)L_{ijkp} \\
& - \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'} = \frac{8f_p}{5dz}\cos\hat{\phi}_p.
\end{aligned} \tag{4.58}$$

Likewise for the bottom boundary condition, we have

$$\begin{aligned}
0 = &\frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
&+ \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
&- \frac{L_{ij,k-1,p}}{5dz} \cos\hat{\phi}_p \\
&+ (a_{ijk} + b(1 - \beta_{pp'}) - \frac{7}{5dz}\cos\hat{\phi}_p) L_{ijkp} \\
&- \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'}.
\end{aligned} \tag{4.59}$$

Now, for upwelling light at the first depth layer (non-BC), we apply FD2.

$$\begin{aligned}
0 = &\frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
&+ \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
&+ \frac{-3L_{ijkp} + 4L_{ij,k+1,p} - L_{ij,k+2,p}}{2dz} \cos\hat{\phi}_p \\
&+ (a_{ijk} + b(1 - \beta_{pp'})) L_{ijkp} \\
&- \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'}.
\end{aligned} \tag{4.60}$$

Grouping L_{ijkp} terms gives

$$\begin{aligned}
0 = &\frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\
&+ \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\
&+ \frac{4L_{ij,k+1,p} - L_{ij,k+2,p}}{2dz} \cos\hat{\phi}_p \\
&+ \left(a_{ijk} + b(1 - \beta_{pp'}) - 3\frac{\cos\hat{\phi}_p}{2dz} \right) L_{ijkp} \\
&- \sum_{p'=1}^{n_{\varpi}} \beta_{pp'} L_{ijkp'}.
\end{aligned} \tag{4.61}$$

Similarly, for downwelling light at the lowest depth layer, we have

$$\begin{aligned}0 = &\frac{L_{i+1,jkp} - L_{i-1,jkp}}{2dx} \sin\hat{\phi}_p \cos\hat{\theta}_p \\ &+ \frac{L_{i,j+1,kp} - L_{i,j-1,kp}}{2dy} \sin\hat{\phi}_p \sin\hat{\theta}_p \\ &+ \frac{-4L_{ij,k-1,p} + L_{ij,k-2,p}}{2dz} \cos\hat{\phi}_p \\ &+ \left(a_{ijk} + b(1-\beta_{pp'}) + 3\frac{\cos\hat{\phi}_p}{2dz} \right) L_{ijkp} \\ &- \sum_{p'=1}^{n_{\vec{\omega}}} \beta_{pp'} L_{ijkp'}\end{aligned} \tag{4.62}$$

4.3.3 Structure of Linear System

Describe layout of matrix.

Derivative case	**# nonzero/row**	**# of rows**
interior	$n_{\vec{\omega}} + 6$	$n_x n_y (n_z - 2) n_{\vec{\omega}}$
surface downwelling	$n_{\vec{\omega}} + 5$	$n_x n_y n_{\vec{\omega}}/2$
bottom upwelling	$n_{\vec{\omega}} + 5$	$n_x n_y n_{\vec{\omega}}/2$
surface upwelling	$n_{\vec{\omega}} + 6$	$n_x n_y n_{\vec{\omega}}/2$
bottom downwelling	$n_{\vec{\omega}} + 6$	$n_x n_y n_{\vec{\omega}}/2$

Table 4.1: Breakdown of nonzero matrix elements by derivative case

Number of rows/columns: $n_x n_y n_z n_{\vec{\omega}}$

Number of nonzero RHS entries: $n_x n_y n_z/2$

Total number of nonzero matrix entries: $n_x n_y n_{\vec{\omega}} \left[n_z(n_{\vec{\omega}} + 6) - 1 \right]$

4.3.4 GMRES

GMRES is a Krylov Subspace method. These work like this. Here's what's special about GMRES. Advantages. Drawbacks. Not practical for running in SINMOD.

4.4 Numerical Asymptotics

Given a position $\vec{x}$ and direction $\vec{\omega}$, a path through the discrete grid can be constructed as described in Appendix A, from which we can extract piecewise constant variations of the path absorption coefficient, $\tilde{a}(s)$ and the effective source, $g_n(s)$ from 3.3.2. Then, we proceed as follows.

* Here are the equations for calculating the double integral over ray paths required for the asymptotics. It will hopefully make more sense once I add words to accompany the symbols.

Let

$$g_n(s) = \sum_{i=1}^{N-1} g_{ni}\mathcal{X}_i(s) \tag{4.63}$$

$$\tilde{a}(s) = \sum_{i=1}^{N-1} \tilde{a}_i\mathcal{X}_i(s) \tag{4.64}$$

$$\tag{4.65}$$

and

$$\mathcal{X}_i(s) = \begin{cases} 1, & a_I \leq s < s_{i+1} \\ 0, & \text{otherwise} \end{cases} \tag{4.66}$$

and $\{s_i\}_{i=1}^{N}$ is increasing.

Let $ds_i = s_{i+1} - s_i$.

Let $\hat{i}(s) = \min\{i \in \{1, \dots, N\} : s_i > s\}$. Let $\tilde{d}(s) = s_{\hat{i}(s)} - s$.

We have $s_1 = 0$ and $s_N = \tilde{s}$.

$$\begin{aligned}
u_n(\tilde{s}) &= \int_0^{\tilde{s}} g_n(s') \exp\left(-\int_{s''}^{s'} \tilde{a}(s'')\, ds''\right) ds' && (4.67)\\
&= \int_0^{s_N} \sum_{i=1}^{N-1} g_{ni}\mathcal{X}_i(s') \exp\left(-\int_{s''}^{s'} \sum_{j=1}^{N-1} \tilde{a}_j \mathcal{X}_j(s'')\, ds''\right) ds' && (4.68)\\
&= \sum_{i=1}^{N-1} g_{ni} \int_0^{s_N} \mathcal{X}_i(s') \exp\left(-\sum_{j=1}^{N-1} \tilde{a}_j \int_{s''}^{s'} \mathcal{X}_j(s'')\, ds''\right) ds' && (4.69)\\
&= \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp\left(-\tilde{a}_{\hat{i}(s')-1}\tilde{d}(s') - \sum_{j=\hat{i}(s')}^{N-1} \tilde{a}_j ds_j\right) ds' && (4.70)\\
&= \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp\left(-\tilde{a}_i(s_{i+1} - s') - \sum_{j=i+1}^{N-1} \tilde{a}_j ds_j\right) ds' && (4.71)
\end{aligned}$$

Let

$$b_i = -\tilde{a}_i s_{i+1} - \sum_{j=i+1}^{N-1} \tilde{a}_j ds_j. \qquad (4.72)$$

Then,

$$\begin{aligned}
u_n(\tilde{s}) &= \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp\left(\tilde{a}_i s' + b_i\right) ds' && (4.73)\\
&= \sum_{i=1}^{N-1} g_{ni} e^{b_i} \int_{s_i}^{s_{i+1}} \exp\left(\tilde{a}_i s'\right) ds' && (4.74)
\end{aligned}$$

Let

$$d_i = \int_{s_i}^{s_{i+1}} \exp\left(\tilde{a}_i s'\right)\, ds' \tag{4.75}$$

$$= \begin{cases} ds_i, & \tilde{a} = 0 \\ \\ \left(\exp(\tilde{a}_i s_{i+1}) - \exp(\tilde{a}_i s_i)\right)/\tilde{a}_i, & \text{otherwise} \end{cases} \tag{4.76}$$

Then,

$$u_n(\tilde{s}) = \sum_{i=1}^{N-1} g_{ni} d_i e^{b_i} \tag{4.77}$$

4.4.1 Perceived Irradiance

The average irradiance experienced by a kelp frond in depth layer k is

$$\tilde{I}_k = \frac{\sum_{ij} P_{ijk} I_{ijk}}{\sum_{ij} P_{ijk}} \tag{4.78}$$

The irradiance perceived by a the kelp is expected to be slightly lower than the average irradiance,

$$\bar{I}_k = \frac{\sum_{ij} I_{ijk}}{n_x n_y} \tag{4.79}$$

since the kelp is more densely located at the center of the domain where the light field is reduced, whereas the simple average is influenced by regions of higher irradiance at the edges of the domain where kelp is not present.

CHAPTER V

PARAMETER VALUES

I'll describe what one would do in order to determine "frond bending coefficients", as well as optical properties of water and kelp, citing literature and reporting values obtained by others.

5.1 Parameters from Literature

* More to come

5.2 Frond Distribution Parameters

5.2.1 Rotation

5.2.2 Lift

Parameter Name	Symbol	Value(s)	Citation	Notes
Kelp Absorptance	A_k	0.8	[4]	Actually for *Macrocystis Pyrifera*
Water absorption coefficient	a_w	?	?	?
Scattering coefficient	b	0.366	[12]	Table 2, $b_{\lambda 0}$, mean
VSF	β	tabulated	[10, 12],	Currently using Petzold
Frond thickness	t	0.4 mm	Ole Jacob	Carina? ***
Water absorption coefficient	a_w	0.03 1 1/m	[6]	Fig. 6, dense cluster. Samnanger Fjord, Western Norway.
Water scattering coefficient	a_w	0.5 1 1/m	[6]	Fig. 7, dense cluster. Samnanger Fjord, Western Norway.
Surface solar irradiance	I_0	$50\,\mathrm{W\,m^{-2}}$	[2]	Irradiance for maximal photosynthesis, converted from photons

Site	$\boldsymbol{a}(\mathbf{m}^{-1})$	$\boldsymbol{b}(\mathbf{m}^{-1})$	$\boldsymbol{c}(\mathbf{m}^{-1})$	$\boldsymbol{a/c}$	$\boldsymbol{b/c}$
AUTEC 7	0.082	0.117	0.199	0.412	0.588
AUTEC 8	0.114	0.037	0.151	0.753	0.247
AUTEC 9	0.122	0.043	0.165	0.742	0.258
HAOCE 5	0.195	0.275	0.47	0.415	0.585
HAOCE 11	0.179	0.219	0.398	0.449	0.551
NUC 2200	0.337	1.583	1.92	0.176	0.824
NUC 2040	0.366	1.824	2.19	0.167	0.833
NUC 2240	0.125	1.205	1.33	0.094	0.906
Filtered Fresh	0.093	0.009	0.102	0.907	0.093
Filtered Fresh + Scat.	0.138	0.547	0.685	0.202	0.798
Fresh + Scat. + Abs.	0.764	0.576	1.34	0.57	0.43
As Delivered	0.196	1.284	1.48	0.133	0.867
Filtered 40 min	0.188	0.407	0.595	0.315	0.685
Filtered 1hr 40 min	0.093	0.081	0.174	0.537	0.463
Filtered 18hr	0.085	0.008	0.093	0.909	0.091

Table 5.2: Petzold IOP summary [10]. I'll pull a few cases from here and point out when the asymptotic approximation will work.

CHAPTER VI

MODEL ANALYSIS

6.1 Grid Study

Run many grid sizes with GMRES, using asymptotic solution as initial guess. Compare CPU times and accuracy, assuming largest grid is “true” solution. Determine necessary grid size to achieve reasonable accuracy.

6.2 Asymptotic Convergence

Compare asymptotic solutions to GMRES with reasonable grid size as determined above. Compare CPU time and accuracy. Determine ideal number of scatters to include (number of terms in asymptotic series). Repeat for a few values of scattering coefficient.

6.3 Sensitivity Analysis

Vary parameters and measure average differences in radiance for full grid, as well as average irradance over depth.

- absorption coefficient
- scattering coefficient

- VSF

- frond bending coefficient

6.4 Kelp Cultivation Simulation

Run Ole Jacob's model with my new light model, compare:

- irrad over time for several depths

- computation time

- harvestable biomass

CHAPTER VII

CONCLUSION

We present a probabilistic model for the spatial distribution of kelp, and develop a first-principles model for the light field, considering absorption and scattering due to the water and kelp. A full finite difference solution is presented, and an asymptotic approximation based on discrete scattering events is subsequently developed.

Future work:

- Frond bending
- Horizontal kelp ropes (long lines)
- etc.

BIBLIOGRAPHY

[1] N. Anderson. A mathematical model for the growth of giant kelp. *Simulation*, 22(4):97–105, 1974.

[2] O. J. Broch and D. Slagstad. Modelling seasonal growth and composition of the kelp Saccharina latissima. *Journal of Applied Phycology*, 24(4):759–776, Aug. 2012.

[3] M. A. Burgman and V. A. Gerard. A stage-structured, stochastic population model for the giant kelpMacrocystis pyrifera. *Marine Biology*, 105(1):15–23, 1990.

[4] M. F. Colombo-Pallotta, E. Garca-Mendoza, and L. B. Ladah. Photosynthetic Performance, Light Absorption, and Pigment Composition of Macrocystis Pyrifera (laminariales, Phaeophyceae) Blades from Different Depths1. *Journal of Phycology*, 42(6):1225–1234, Dec. 2006.

[5] P. Duarte and J. G. Ferreira. A model for the simulation of macroalgal population dynamics and productivity. *Ecological modelling*, 98(2-3):199–214, 1997.

[6] B. Hamre, . Frette, S. R. Erga, J. J. Stamnes, and K. Stamnes. Parameterization and analysis of the optical absorption and scattering coefficients in a western

Norwegian fjord: a case II water study. *Applied Optics*, 42(6):883, Feb. 2003.

[7] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp Macrocystis pyrifera. *Marine Biology*, 95(4):611–624, 1987.

[8] C. Mobley. Radiative Transfer in the Ocean. In *Encyclopedia of Ocean Sciences*, pages 2321–2330. Elsevier, 2001.

[9] M. Nyman, M. Brown, M. Neushul, and J. A. Keogh. *Macrocystis pyrifera in New Zealand: testing two mathematical models for whole plant growth*, volume 2. Sept. 1990.

[10] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.

[11] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Ecological Modelling*, 80:161–170, Mar. 1994.

[12] A. Sokolov, M. Chami, E. Dmitriev, and G. Khomenko. Parameterization of volume scattering function of coastal waters based on the statistical approach. *Optics express*, 18(5):4615–4636, 2010.

[13] P. Wassmann, D. Slagstad, C. W. Riser, and M. Reigstad. Modelling the ecosystem dynamics of the Barents Sea including the marginal ice zone. *Journal of Marine Systems*, 59(1-2):1–24, Jan. 2006.

[14] A. Yoshimori, T. Kono, and H. Iizumi. Mathematical models of population dynamics of the kelp Laminaria religiosa, with emphasis on temperature dependence. *Fisheries Oceanography*, 7(2):136146, 1998.

APPENDICES

APPENDIX A

RAY TRACING ALGORITHM

In order to evaluate a path integral through the previously described grid, it is first necessary to construct a one-dimensional piecewise constant integrand which is discontinuous at unevenly spaced points corresponding to the intersections between the path and edges in the spatial grid.

Consider a grid center $\vec{p_1} = (p_{1x}, p_{1y}, p_{1z})$ and a corresponding path $\vec{l}(\vec{x_1}, \vec{\omega}, s)$. To find the location of discontinuities in the itegrand, we first calculate the distance from its origin, $\vec{p_0} = \vec{x_0}(\vec{p_1}, \vec{\omega}) = (p_{0x}, p_{0y}, p_{0z})$ to grid edges in each dimension separately.

Given

$$x_i = p_{0x} + \frac{s_i^x}{\tilde{s}}(p_{1x} - p_{0x}) \tag{A.1}$$

$$y_j = p_{0y} + \frac{s_j^y}{\tilde{s}}(p_{1y} - p_{0y}) \tag{A.2}$$

$$z_k = p_{0z} + \frac{s_k^z}{\tilde{s}}(p_{1z} - p_{0z}) \tag{A.3}$$

we have

$$s_i^x = \tilde{s}\frac{x_i - p_{0x}}{p_{1x} - p_{0x}} \tag{A.4}$$

$$s_i^y = \tilde{s}\frac{y_i - p_{0y}}{p_{1y} - p_{0y}} \tag{A.5}$$

$$s_i^z = \tilde{s}\frac{z_i - p_{0z}}{p_{1z} - p_{0z}} \tag{A.6}$$

$$\tag{A.7}$$

We also keep a record for each dimension specifying whether the ray increases or decreases in the dimension. Let

$$\delta_x = \text{sign}(p_{0x} - p_{1x}) \tag{A.8}$$

$$\delta_y = \text{sign}(p_{0y} - p_{1y}) \tag{A.9}$$

$$\delta_z = \text{sign}(p_{0z} - p_{1z}) \tag{A.10}$$

For convenience, we also store a closely related quantity, σ with a value 1 for increasing rays and 0 for decreasing rays in each dimension

$$\sigma_x = (\delta_x + 1)/2 \tag{A.11}$$

$$\sigma_y = (\delta_y + 1)/2 \tag{A.12}$$

$$\sigma_z = (\delta_z + 1)/2 \tag{A.13}$$

For this algorithm, we keep two sets of indices. (i, j, k) indexes the grid cell, and will be used for extracting physical quantities from each cell along the path. Meanwhile, (i^e, j^e, k^e) will index the edges between grid cells, beginning after the first cell. i.e., $i^e = 1$ refers not to the plane $x = x_{\text{min}}$, but to $x = x_{\text{min}} + dx$.

Let (i_0, j_0, k_0) be the indices of the grid cell containing $\vec{p_0}$.

That is,

$$i_0 = \text{ceil}\left(\frac{p_{0x} - x_{\text{min}}}{dx}\right) \tag{A.14}$$

$$j_0 = \text{ceil}\left(\frac{p_{0y} - y_{\text{min}}}{dy}\right) \tag{A.15}$$

$$k_0 = \text{ceil}\left(\frac{p_{0z} - z_{\text{min}}}{dz}\right) \tag{A.16}$$

Then,

$$i_0^e = i_0 + \sigma_x \tag{A.17}$$

$$j_0^e = j_0 + \sigma_y \tag{A.18}$$

$$k_0^e = k_0 + \sigma_z \tag{A.19}$$

Now, we calculate the distance from p_0 along the path to edges in each dimension.

$$s_i^x = \hat{s}\frac{x_i^e - p_{0x}}{p_{1x} - p_{0x}} \tag{A.20}$$

$$s_j^y = \hat{s}\frac{y_j^e - p_{0y}}{p_{1y} - p_{0y}} \tag{A.21}$$

$$s_k^z = \hat{s}\frac{z_k^e - p_{0z}}{p_{1z} - p_{0z}} \tag{A.22}$$

For each grid cell, we check the path lengths required to cross the next x, y, and z edge-planes. Then, we move to the next grid cell in that dimension. That is,

* We also track s, the path length.

Consider i, j, k fixed (denoting the current grid cell).

$$d = \text{argmin}_{x,y,z} \left\{ s_i^x - s, s_j^y - s, s_k^z \right\} \tag{A.23}$$

* This doesn't quite make sense yet.

$$\begin{cases} i = i + \delta_x, & \text{if } d = x \\ j = j + \delta_y, & \text{if } d = y \\ z = k + \delta_z, & \text{if } d = z \end{cases} \tag{A.24}$$

and

$$\begin{cases} i^e = i^e + \delta_x, & \text{if } d = x \\ j^e = j^e + \delta_y, & \text{if } d = y \\ z^e = k^e + \delta_z, & \text{if } d = z \end{cases} \tag{A.25}$$

Then, move to the adjacent grid cell in the dimension which requires the shortest step to reach an edge. Save ds of the path through this cell. Also save abs. coef. and source.

APPENDIX B

FORTRAN CODE

The full FORTRAN implementation of the model described in this thesis. This code can be found online at:

`https://github.com/OliverEvans96/kelp`

`https://gitlab.com/OliverEvans96/kelp`

utils.f90

```fortran
! General utilities which might be useful in other settings
module utils
implicit none

! Constants
double precision, parameter :: pi = 4.D0 * datan(1.D0)

contains

! Determine base directory relative to current directory
! by looking for Makefile, which is in the base dir
! Assuming that this is executed from within the git repo.
function getbasedir()
    implicit none

    ! INPUTS:
    ! Number of paths to check
    integer, parameter :: numpaths = 3
    ! Maximum length of path names
    integer, parameter :: maxlength = numpaths * 2 - 1
    ! Paths to check for Makefile
    character(len=maxlength), parameter, dimension(numpaths) :: check_paths &
```

```fortran
            = (/ '.    ', '..   ', '../..' /)
    ! Temporary path string
    character(len=maxlength) tmp_path
    ! Whether Makefile has been found yet
    logical found
    ! Path counter
    integer ii
    ! Lengths of paths
    integer, dimension(numpaths) ::  pathlengths

    ! OUTPUT:
    ! getbasedir - relative path to base
       directory
    ! Will either return '.', '..', or '../..'
    character(len=maxlength) getbasedir


    ! Determine length of each path
    pathlengths(1) = 1
    do ii = 2, numpaths
        pathlengths(ii) = 2 + 3 * (ii - 2)
    end do

    ! Loop through paths
    do ii = 1, numpaths
        ! Determine this path
        tmp_path = check_paths(ii)

        ! Check whether Makefile is in this
           directory
        !write(*,*) 'Checking "', tmp_path(1:
           pathlengths(ii)), '"'
        inquire(file=tmp_path(1:pathlengths(ii))
            // '/Makefile', exist=found)
        ! If so, stop. Otherwise, keep looking.
        if(found) then
            getbasedir = tmp_path(1:pathlengths(
               ii))
            exit
        end if
    end do

    ! If it hasn't been found, then this script
       was probably called
    ! from outside of the repository.
    if(.not. found) then
        write(*,*) 'BASE DIR NOT FOUND.'
    end if

end function

! Determine array size from min, max and step
! If alignment is off, array will overstep the
   maximum
function bnd2max(xmin,xmax,dx)
    implicit none

    ! INPUTS:
    ! xmin - minimum x value in array
    ! xmax - maximum x value in array (inclusive
       )
    ! dx - step size
    double precision, intent(in) :: xmin, xmax,
       dx

    ! OUTPUT:
    ! step2max - maximum index of array
    integer bnd2max

    ! Calculate array size
    bnd2max = int(ceiling((xmax-xmin)/dx))
end function

! Create array from bounds and number of
   elements
! xmax is not included in array
function bnd2arr(xmin,xmax,imax)
    implicit none

    ! INPUTS:
    ! xmin - minimum x value in array
    ! xmax - maximum x value in array (exclusive
       )
    double precision, intent(in) :: xmin, xmax
    ! imax - number of elements in array
    integer imax

    ! OUTPUT:
    ! bnd2arr - array to generate
    double precision, dimension(imax) :: bnd2arr

    ! BODY:

    ! Counter
    integer ii
    ! Step size
    double precision dx

    ! Calculate step size
    dx = (xmax - xmin) / imax

    ! Generate array
    do ii = 1, imax
        bnd2arr(ii) = xmin + (ii-1) * dx
    end do

end function

function mod1(i, n)
  implicit none
  integer i, n, m
  integer mod1

  m = modulo(i, n)

  if(m .eq. 0) then
     mod1 = n
  else
     mod1 = m
  end if

end function mod1

function sgn_int(x)
  integer x, sgn_int
  ! Standard signum function
  sgn_int = sign(1,x)
  if(x .eq. 0.) sgn_int = 0
end function sgn_int

function sgn(x)
  double precision x, sgn
  ! Standard signum function
  sgn = sign(1.d0,x)
  if(x .eq. 0.) sgn = 0
end function sgn

! Interpolate single point from 1D data
function interp(x0,xx,yy,nn)
    implicit none

    ! INPUTS:
    ! x0 - x value at which to interpolate
    double precision, intent(in) :: x0
    ! xx - ordered x values at which y data is
       sampled
    ! yy - corresponding y values to interpolate
    double precision, dimension (nn), intent(in)
        :: xx,yy
    ! nn - length of data
    integer, intent(in) :: nn

    ! OUTPUT:
    ! interp - interpolated y value
    double precision interp

    ! BODY:

    ! Index of lower-adjacent data (xx(i) < x0 < xx(i+1))
    integer ii
    ! Slope of liine between (xx(ii),yy(ii)) and (xx(ii+1),yy(ii+1))
    double precision mm

    ! If out of bounds, then return endpoint value
    if (x0 < xx(1)) then
       interp = yy(1)
    else if (x0 > xx(nn)) then
       interp = yy(nn)
    else

      ! Determine ii
      do ii = 1, nn
          if (xx(ii) > x0) then
              ! We've now gone one index too far.
              exit
          end if
      end do

      ! Determine whether we're on the right endpoint
      if(ii-1 < nn) then
          ! If this is a legitimate interpolation, then
          ! subtract since we went one index too far
          ii = ii - 1

          ! Calculate slope
          mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) - xx(ii))

          ! Return interpolated value
          interp = yy(ii) + mm * (x0 - xx(ii))
      else
          ! If we're actually interpolating the right endpoint,
          ! then just return it.
          interp = yy(nn)
      end if

   end if

end function

```

```
! Calculate unshifted position of periodic image
! Assuming xmin, xmax are extreme attainable
   values of x
function shift_mod(x, xmin, xmax)
  double precision x, xmin, xmax
  double precision mod_part, shift_mod
  mod_part = mod(x-xmin, xmax-xmin)
  if(mod_part .ge. 0) then
     ! In this case, mod_part is distance
        between image & lower bound
     shift_mod = xmin + mod_part
  else
     ! In this case, mod_part is distance
        between image & upper bound
     shift_mod = xmax + mod_part
  endif
end function shift_mod

! Bilinear interpolation on evenly spaced 2D
   grid
! Assume upper endpoint is not included and is
   identical
! to the lower endpoint, which is included.
function bilinear_array_periodic(x, y, nx, ny,
   x_vals, y_vals, fun_vals)
  implicit none
  double precision x, y
  integer nx, ny
  double precision, dimension(:) :: x_vals,
     y_vals
  double precision, dimension(:,:) :: fun_vals

  double precision dx, dy, xmin, ymin
  integer i0, j0, i1, j1
  double precision x0, x1, y0, y1
  double precision z00, z10, z01, z11

  double precision bilinear_array_periodic

  xmin = x_vals(1)
  ymin = y_vals(1)
  dx = x_vals(2) - x_vals(1)
  dy = y_vals(2) - y_vals(1)

  ! Add 1 for one-indexing
  i0 = int(floor((x-xmin)/dx))+1
  j0 = int(floor((y-ymin)/dy))+1

  x0 = x_vals(i0)
  y0 = y_vals(j0)

  ! Periodic wrap
```

```fortran
  if(i0 .lt. nx) then
     i1 = i0 + 1
     x1 = x_vals(i1)
  else
     i1 = 1
     x1 = x_vals(nx) + dx
  endif

  if(j0 .lt. ny) then
     j1 = j0 + 1
     y1 = y_vals(j1)
  else
     j1 = 1
     y1 = y_vals(ny) + dy
  endif

  z00 = fun_vals(i0,j0)
  z10 = fun_vals(i1,j0)
  z01 = fun_vals(i0,j1)
  z11 = fun_vals(i1,j1)

  bilinear_array_periodic = bilinear(x, y, x0,
     y0, x1, y1, z00, z01, z10, z11)
end function bilinear_array_periodic

! Bilinear interpolation on evenly spaced 2D
  grid
! Assume upper and lower endpoints are included
function bilinear_array(x, y, x_vals, y_vals,
   fun_vals)
  implicit none
  double precision x, y
  double precision, dimension(:) :: x_vals,
     y_vals
  double precision, dimension(:,:) :: fun_vals

  double precision dx, dy, xmin, ymin
  integer i0, j0, i1, j1
  double precision x0, x1, y0, y1
  double precision z00, z10, z01, z11

  double precision bilinear_array

  xmin = x_vals(1)
  ymin = y_vals(1)
  dx = x_vals(2) - x_vals(1)
  dy = y_vals(2) - y_vals(1)

  ! Add 1 for one-indexing
  i0 = int(floor((x-xmin)/dx))+1
  j0 = int(floor((y-ymin)/dy))+1
  i1 = i0 + 1
  j1 = j0 + 1

  ! Bounds checking
  ! if(i0 .lt. 1) then
  !     i0 = 1
  !     i1 = 1
  ! else if(i1 .gt. nx) then
  !     i0 = nx
  !     i1 = nx
  ! endif
  ! if(j0 .lt. 1) then
  !     j0 = 1
  !     j1 = 1
  ! else if(j1 .gt. ny) then
  !     j0 = ny
  !     j1 = ny
  ! endif

  x0 = x_vals(i0)
  x1 = x_vals(i1)
  y0 = y_vals(j0)
  y1 = y_vals(j1)

  z00 = fun_vals(i0,j0)
  z10 = fun_vals(i1,j0)
  z01 = fun_vals(i0,j1)
  z11 = fun_vals(i1,j1)

  bilinear_array = bilinear(x, y, x0, y0, x1, y1, z00, z01, z10, z11)
end function bilinear_array

! ilinear interpolation of a function of two variables
! over a rectangle of points.
! Weight each point by the area of the sub-rectangle involving
! the point (x,y) and the point diagonally across the rectangle
function bilinear(x, y, x0, y0, x1, y1, z00, z01, z10, z11)
  implicit none
  double precision x, y
  double precision x0, y0, x1, y1, z00, z01, z10, z11
  double precision a, b, c, d
  double precision bilinear

  a = (x-x0)*(y-y0)
  b = (x1-x)*(y-y0)
  c = (x-x0)*(y1-y)
  d = (x1-x)*(y1-y)

  bilinear = (a*z11 + b*z01 + c*z10 + d*z00) / (
     a + b + c + d)
end function bilinear

! Integrate using left endpoint rule
! Assuming the right endpoint is not included in
    arr
function lep_rule(arr,dx,nn)
    implicit none

    ! INPUTS:
    ! arr - array to integrate
    double precision, dimension(nn) :: arr
    ! dx - array spacing (mesh size)
    double precision dx
    ! nn - length of arr
    integer, intent(in) :: nn

    ! OUTPUT:
    ! lep_rule - integral w/ left endpoint rule
    double precision lep_rule

    ! BODY:

    ! Counter
    integer ii

    ! Set output to zero
    lep_rule = 0.0d0

    ! Accumulate integral
    do ii = 1, nn
        lep_rule = lep_rule + arr(ii) * dx
    end do

end function

! Integrate using trapezoid rule
! Assuming both endpoints are included in arr
function trap_rule_dx(arr, dx, nn)
  implicit none
  double precision, dimension(nn) :: arr
  double precision dx
  integer ii, nn
  double precision trap_rule_dx

  trap_rule_dx = 0.0d0

  do ii=1, nn-1
```

```
      trap_rule_dx = trap_rule_dx + 0.5d0 * dx * (arr(ii) + arr(ii+1))
   end do

end function trap_rule_dx

! Integrate using trapezoid rule
! Assuming both endpoints are included in arr
function trap_rule_uneven(xx, yy, nn)
   implicit none
   double precision, dimension(nn) :: xx
   double precision, dimension(nn) :: yy
   integer ii, nn
   double precision trap_rule_uneven

   trap_rule_uneven = 0.0d0

   do ii=1, nn-1
      trap_rule_uneven = trap_rule_uneven + 0.5d0 * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii+1))
   end do
end function trap_rule_uneven

function trap_rule_dx_uneven(dx, yy, nn)
   implicit none
   double precision, dimension(nn-1) :: dx
   double precision, dimension(nn) :: yy
   integer ii, nn
   double precision trap_rule_dx_uneven

   trap_rule_dx_uneven = 0.0d0

   do ii=1, nn-1
      trap_rule_dx_uneven = trap_rule_dx_uneven + 0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
   end do
end function trap_rule_dx_uneven

! Integrate using midpoint rule
! First and last bins, only use inner half
function midpoint_rule_halfends(dx, yy, nn) result(integral)
   implicit none
   integer ii, nn
   double precision, dimension(nn) :: dx, yy
   double precision integral

   if(nn > 1) then
     integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(nn))

```

```
    do ii=2, nn-1
      integral = integral + dx(ii)*yy(ii)
    end do
  else
    integral = 0.d0
  end if
end function midpoint_rule_halfends

! Normalize 1D array and return integral w/ left
    endpoint rule
function normalize_dx(arr,dx,nn)
    implicit none

    ! INPUTS:
    ! arr - array to normalize
    double precision, dimension(nn) :: arr
    ! dx - array spacing (mesh size)
    double precision dx
    ! nn - length of arr
    integer, intent(in) :: nn

    ! OUTPUT:
    ! normalize - integral before normalization
       (left endpoint rule)
    double precision normalize_dx

    ! BODY:

    ! Calculate integral
    normalize_dx = lep_rule(arr,dx,nn)

    ! Normalize array
    arr = arr / normalize_dx

  end function normalize_dx

! Normalize 1D unevenly-spaced array and
! return integral w/ trapezoid rule
! Will not be quite accurate if rightmost
   endpoint is not included
! (Very small for VSF, so not a big deal there)
! Modifies yy in place
function normalize_uneven(xx, yy, nn) result(
   norm)
  implicit none

  ! INPUTS:
  ! xx, yy - array values of data to normalize
  double precision, dimension(nn) :: xx, yy
  ! nn - length of arr
  integer, intent(in) :: nn

  ! OUTPUT:
  ! normalize - integral before normalization (
     left endpoint rule)
  double precision norm

  ! BODY:

  ! Calculate integral
  ! PERHAPS WE SHOULD USE TRAPEZOID RULE
  norm = trap_rule_uneven(xx, yy, nn)

  ! Normalize array
  yy(:) = yy(:) / norm

end function normalize_uneven

! Read 2D array from file
function read_array(filename,fmtstr,nn,mm,
   skiplines_in)
    implicit none

    ! INPUTS:
    ! filename - path to file to be read
    ! fmtstr - input format (no parentheses, don
       't specify columns)
    ! e.g. 'E10.2', not '(2E10.2)'
    character(len=*), intent(in) :: filename,
       fmtstr
    ! nn - Number of data rows in file
    ! mm - number of data columns in file
    integer, intent(in) :: nn, mm
    ! skiplines - optional - number of lines to
       skip from header
    integer, optional :: skiplines_in
    integer skiplines

    ! OUTPUT:
    double precision, dimension(nn,mm) ::
       read_array

    ! BODY:

    ! Row counter
    integer ii
    ! File unit number
    integer, parameter :: un = 10
    ! Final format to use
    character(len=256) finfmt

    ! Generate final format string
    write(finfmt,'(A,I1,A,A)') '(', mm, fmtstr,
       ')'

    ! Print message
    !write(*,*) 'Reading data from "', trim(
       filename), '"'
    !write(*,*) 'using format "', trim(finfmt),
       '"'

    ! Open file
    open(unit=un, file=trim(filename), status='
       old', form='formatted')

    ! Skip lines if desired
    if(present(skiplines_in)) then
        skiplines = skiplines_in
        do ii = 1, skiplines
            ! Read without variable ignores the
               line
            read(un,*)
        end do
    else
        skiplines = 0
    end if

    ! Loop through lines
    do ii = 1, nn
        ! Read one row at a time
        read(unit=un, fmt=trim(finfmt))
           read_array(ii,:)
    end do

    ! Close file
    close(unit=un)

end function

! Print 2D array to stdout
subroutine print_int_array(arr,nn,mm,fmtstr_in)
  implicit none

  ! INPUTS:
  ! arr - array to print
  integer, dimension (nn,mm), intent(in) :: arr
  ! nn - number of data rows in file
  ! nn - number of data columns in file
  integer, intent(in) :: nn, mm
  ! fmtstr - output format (no parentheses, don'
     t specify columns)
  ! e.g. 'E10.2', not '(2E10.2)'
  character(len=*), optional :: fmtstr_in
  character(len=256) fmtstr

  ! NO OUTPUTS
```

```fortran

   ! BODY

   ! Row counter
   integer ii
   ! Final format to use
   character(len=256) finfmt

   ! Determine string format
   if(present(fmtstr_in)) then
      fmtstr = fmtstr_in
   else
      fmtstr = 'I10'
   end if

   ! Generate final format string
   write(finfmt,'(A,I4,A,A)') '(', mm, trim(
      fmtstr), ')'

   ! Loop through rows
   do ii = 1, nn
      ! Print one row at a time
      write(*,finfmt) arr(ii,:)
   end do

   ! Print blank line after
   write(*,*) ' '

end subroutine print_int_array

subroutine print_array(arr,nn,mm,fmtstr_in)
    implicit none

    ! INPUTS:
    ! arr - array to print
    double precision, dimension (nn,mm), intent(
       in) :: arr
    ! nn - number of data rows in file
    ! nn - number of data columns in file
    integer, intent(in) :: nn, mm
    ! fmtstr - output format (no parentheses,
       don't specify columns)
    ! e.g. 'E10.2', not '(2E10.2)'
    character(len=*), optional :: fmtstr_in
    character(len=256) fmtstr

    ! NO OUTPUTS

    ! BODY

    ! Row counter
    integer ii
    ! Final format to use
    character(len=256) finfmt

    ! Determine string format
    if(present(fmtstr_in)) then
        fmtstr = fmtstr_in
    else
        fmtstr = 'ES10.2'
    end if

    ! Generate final format string
    write(finfmt,'(A,I4,A,A)') '(', mm, trim(
       fmtstr), ')'

    ! Loop through rows
    do ii = 1, nn
        ! Include row number
        !write(*,'(I10)', advance='no') ii
        ! Print one row at a time
        write(*,finfmt) arr(ii,:)
    end do

    ! Print blank line after
    write(*,*) ' '

end subroutine

! Write 2D array to file
subroutine write_array(arr,nn,mm,filename,
   fmtstr_in)
    implicit none

    ! INPUTS:
    ! arr - array to print
    double precision, dimension (nn,mm), intent(
       in) :: arr
    ! nn - number of data rows in file
    ! nn - number of data columns in file
    integer, intent(in) :: nn, mm
    ! filename - file to write to
    character(len=*) filename
    ! fmtstr - output format (no parentheses,
       don't specify columns)
    ! e.g. 'E10.2', not '(2E10.2)'
    character(len=*), optional :: fmtstr_in
    character(len=256) fmtstr

    ! NO OUTPUTS

    ! BODY

    ! Row counter
    integer ii
    ! Final format to use
    character(len=256) finfmt
    ! Dummy file unit to use
    integer, parameter :: un = 20

    ! Open file for writing
    open(unit=un, file=trim(filename), status='
       replace', form='formatted')

    ! Determine string format
    if(present(fmtstr_in)) then
        fmtstr = fmtstr_in
    else
        fmtstr = 'E10.2'
    end if

    ! Generate final format string
    write(finfmt,'(A,I4,A,A)') '(', mm, trim(
       fmtstr), ')'

    ! Loop through rows
    do ii = 1, nn
        ! Print one row at a time
        write(un,finfmt) arr(ii,:)
    end do

    ! Close file
    close(unit=un)

end subroutine

subroutine zeros(x, n)
  implicit none
  integer n, i
  double precision, dimension(n) :: x

  do i=1, n
     x(i) = 0
  end do
end subroutine zeros

end module
```

sag.f90

```
module sag
use utils
use fastgl

implicit none

```

```
! Spatial grids do not include upper endpoints.
! Angular grids do include upper endpoints.
! Both include lower endpoints.

! To use:
! call grid%set_bounds(...)
! call grid%set_num(...) (or set_uniform_spacing
   )
! call grid%init()
! ...
! call grid%deinit()

!integer, parameter :: pi = 3.141592653589793D
   +00

type index_list
   integer i, j, k, p
 contains
   procedure :: init => index_list_init
   procedure :: print => index_list_print
end type index_list

type angle2d
   integer ntheta, nphi, nomega
   double precision dtheta, dphi
   double precision, dimension(:), allocatable
      :: theta, phi, theta_edge, phi_edge
   double precision, dimension(:), allocatable
      ::  theta_p, phi_p, theta_edge_p,
      phi_edge_p
   double precision, dimension(:), allocatable
      :: cos_theta, sin_theta, cos_phi, sin_phi
   double precision, dimension(:), allocatable
      :: cos_theta_edge, sin_theta_edge,
      cos_phi_edge, sin_phi_edge
   double precision, dimension(:), allocatable
      :: cos_theta_p, sin_theta_p, cos_phi_p,
      sin_phi_p
   double precision, dimension(:), allocatable
      :: cos_theta_edge_p, sin_theta_edge_p,
      cos_phi_edge_p, sin_phi_edge_p
   double precision, dimension(:), allocatable
      :: area_p
 contains
   procedure :: set_num => angle_set_num
   procedure :: phat, lhat, mhat
   procedure :: init => angle_init ! Call after
      set_num
   procedure :: integrate_points =>
      angle_integrate_points
   procedure :: integrate_func =>
      angle_integrate_func
   procedure :: deinit => angle_deinit
end type angle2d

type angle_dim
   integer num
   double precision minval, maxval, prefactor
   double precision, dimension(:), allocatable
      :: vals, weights, sin, cos
 contains
   procedure :: set_bounds => angle_set_bounds
   procedure :: set_num => angle1d_set_num
   procedure :: deinit => angle1d_deinit
   procedure :: integrate_points =>
      angle1d_integrate_points
   procedure :: integrate_func =>
      angle1d_integrate_func
   procedure :: assign_linspace =>
      angle1d_assign_linspace
   procedure :: assign_legendre
end type angle_dim

type space_dim
   integer num
   double precision minval, maxval
   double precision, dimension(:), allocatable
      :: vals, edges, spacing
 contains
   procedure :: integrate_points =>
      space_integrate_points
   procedure :: trapezoid_rule
   procedure :: set_bounds => space_set_bounds
   procedure :: set_num => space_set_num
   procedure :: set_uniform_spacing =>
      space_set_uniform_spacing
   !procedure :: set_num_from_spacing
   procedure :: set_uniform_spacing_from_num
   procedure :: set_spacing_array =>
      space_set_spacing_array
   procedure :: deinit => space_deinit
   procedure :: assign_linspace
end type space_dim

type space_angle_grid !(sag)
  type(space_dim) :: x, y, z
  type(angle2d) :: angles
  double precision, dimension(:), allocatable ::
      x_factor, y_factor
contains
  procedure :: set_bounds => sag_set_bounds
  procedure :: set_num => sag_set_num
  procedure :: init => sag_init
  procedure :: deinit => sag_deinit
  !procedure :: set_num_from_spacing => sag_set_num_from_spacing
  procedure :: set_uniform_spacing_from_num => sag_set_uniform_spacing_from_num
  procedure :: calculate_factors => sag_calculate_factors
end type space_angle_grid

contains

  subroutine index_list_init(indices)
    class(index_list) indices
    indices%i = 1
    indices%j = 1
    indices%k = 1
    indices%p = 1
  end subroutine

  subroutine index_list_print(indices)
    class(index_list) indices

    write(*,*) 'i, j, k, p =', indices%i, indices%j, indices%k, indices%p
  end subroutine index_list_print

  subroutine angle_set_num(angles, ntheta, nphi)
    class(angle2d) :: angles
    integer ntheta, nphi
    angles%ntheta = ntheta
    angles%nphi = nphi
    angles%nomega = ntheta*(nphi-2) + 2
  end subroutine angle_set_num

  function lhat(angles, p) result(l)
    class(angle2d) :: angles
    integer l, p
    if(p .eq. 1) then
       l = 1
    else if(p .eq. angles%nomega) then
       l = 1
    else
       l = mod1(p-1, angles%ntheta)
   end if
  end function lhat

  function mhat(angles, p) result(m)
    class(angle2d) :: angles
    integer m, p
    if(p .eq. 1) then
       m = 1
    else if(p .eq. angles%nomega) then
```

```
      m = angles%nphi
    else
      m = ceiling(dble(p-1)/dble(angles%ntheta)
         ) + 1
    end if
  end function mhat

  function phat(angles, l, m) result(p)
    class(angle2d) :: angles
    integer l, m, p

    if(m .eq. 1) then
      p = 1
    else if(m .eq. angles%nphi) then
      p = angles%nomega
    else
      p = (m-2)*angles%ntheta + l + 1
    end if
  end function phat

  subroutine angle_init(angles)
    class(angle2d) :: angles
    integer l, m, p
    double precision area


    ! TODO: CONSIDER REMOVING non-p
    allocate(angles%theta(angles%ntheta))
    allocate(angles%phi(angles%nphi))
    allocate(angles%theta_edge(angles%ntheta))
    allocate(angles%phi_edge(angles%nphi-1))
    allocate(angles%theta_p(angles%nomega))
    allocate(angles%phi_p(angles%nomega))
    allocate(angles%theta_edge_p(angles%nomega))
    allocate(angles%phi_edge_p(angles%nomega))
    allocate(angles%cos_theta_p(angles%nomega))
    allocate(angles%sin_theta_p(angles%nomega))
    allocate(angles%cos_phi_p(angles%nomega))
    allocate(angles%sin_phi_p(angles%nomega))
    allocate(angles%cos_theta(angles%nomega))
    allocate(angles%sin_theta(angles%nomega))
    allocate(angles%cos_phi(angles%nomega))
    allocate(angles%sin_phi(angles%nomega))
    allocate(angles%cos_theta_edge(angles%ntheta
       ))
    allocate(angles%sin_theta_edge(angles%ntheta
       ))
    allocate(angles%cos_phi_edge(angles%nphi-1))
    allocate(angles%sin_phi_edge(angles%nphi-1))
```

```
   allocate(angles%cos_theta_edge_p(angles%nomega))
   allocate(angles%sin_theta_edge_p(angles%nomega))
   allocate(angles%cos_phi_edge_p(angles%nomega-1))
   allocate(angles%sin_phi_edge_p(angles%nomega-1))
   allocate(angles%area_p(angles%nomega))

   ! Calculate spacing
   angles%dtheta = 2.d0*pi/dble(angles%ntheta)
   angles%dphi = pi/dble(angles%nphi-1)

   ! Create grids
   do l=1, angles%ntheta
      angles%theta(l) = dble(l-1)*angles%dtheta
      angles%cos_theta(l) = cos(angles%theta(l))
      angles%sin_theta(l) = sin(angles%theta(l))
      angles%theta_edge(l) = dble(l-0.5d0)*angles%dtheta
      angles%cos_theta_edge(l) = cos(angles%theta_edge(l))
      angles%sin_theta_edge(l) = sin(angles%theta_edge(l))
   end do

   do m=1, angles%nphi
      angles%phi(m) = dble(m-1.d0)*angles%dphi
      angles%cos_phi(m) = cos(angles%phi(m))
      angles%sin_phi(m) = sin(angles%phi(m))
      if(m<angles%nphi) then
         angles%phi_edge(m) = dble(m-0.5d0)*angles%dphi
         angles%cos_phi_edge(m) = cos(angles%phi_edge(m))
         angles%sin_phi_edge(m) = sin(angles%phi_edge(m))
      end if
   end do

   ! Create p arrays
   do m=2, angles%nphi-1
      area = angles%dtheta &
           * (angles%cos_phi_edge(m-1) - angles%cos_phi_edge(m))
      do l=1, angles%ntheta
```

```
          p = angles%phat(l, m)

          angles%theta_p(p) = angles%theta(l)
          angles%phi_p(p) = angles%phi(m)
          angles%theta_edge_p(p) = angles%theta_edge(l)
          angles%phi_edge_p(p) = angles%phi_edge(m)

          angles%cos_theta_p(p) = cos(angles%theta_p(p))
          angles%sin_theta_p(p) = sin(angles%theta_p(p))
          angles%cos_phi_p(p) = cos(angles%phi_p(p))
          angles%sin_phi_p(p) = sin(angles%phi_p(p))

          angles%cos_theta_edge_p(p) = cos(angles%theta_edge_p(p))
          angles%sin_theta_edge_p(p) = sin(angles%theta_edge_p(p))
          angles%cos_phi_edge_p(p) = cos(angles%phi_edge_p(p))
          angles%sin_phi_edge_p(p) = sin(angles%phi_edge_p(p))

          angles%area_p(p) = area
       end do
    end do

    ! Poles
    l=1
    area = 2.d0*pi*(1.d0-cos(angles%dphi/2.d0))

    ! North Pole
    p = 1
    m=1
    angles%theta_p(p) = angles%theta(l)
    angles%theta_edge_p(p) = angles%theta_edge(l)
    angles%phi_p(p) = angles%phi(m)
    ! phi_edge_p only defined up to nphi-1.
    angles%phi_edge_p(p) = angles%phi_edge(m)
    angles%cos_theta_p(p) = cos(angles%theta_p(p))
    angles%sin_theta_p(p) = sin(angles%theta_p(p))
    angles%cos_phi_p(p) = cos(angles%phi_p(p))
    angles%sin_phi_p(p) = sin(angles%phi_p(p))
```

```fortran
    angles%cos_theta_edge_p(p) = cos(angles%theta_edge_p(p))
    angles%sin_theta_edge_p(p) = sin(angles%theta_edge_p(p))
    angles%cos_phi_edge_p(p) = cos(angles%phi_edge_p(p))
    angles%sin_phi_edge_p(p) = sin(angles%phi_edge_p(p))
    angles%area_p(p) = area

    ! South Pole
    p = angles%nomega
    m = angles%nphi
    angles%theta_p(p) = angles%theta(1)
    angles%theta_edge_p(p) = angles%theta_edge(1)
    angles%phi_p(p) = angles%phi(m)
    angles%cos_theta_p(p) = cos(angles%theta_p(p))
    angles%sin_theta_p(p) = sin(angles%theta_p(p))
    angles%cos_phi_p(p) = cos(angles%phi_p(p))
    angles%sin_phi_p(p) = sin(angles%phi_p(p))
    angles%area_p(p) = area
  end subroutine angle_init

  ! Integrate function given function values at grid cells
  function angle_integrate_points(angles, func_vals) result(integral)
    class(angle2d) :: angles
    double precision, dimension(angles%nomega) :: func_vals
    double precision integral
    integer p

    integral = 0.d0

    do p=1, angles%nomega
       integral = integral + angles%area_p(p) * func_vals(p)
    end do

  end function angle_integrate_points

  function angle_integrate_func(angles, func_callable) result(integral)
    class(angle2d) :: angles
    double precision, external :: func_callable
    double precision, dimension(:), allocatable
       :: func_vals
    double precision integral
    integer p
    double precision theta, phi

    allocate(func_vals(angles%nomega))

    do p=1, angles%nomega
       theta = angles%theta_p(p)
       phi = angles%phi_p(p)
       func_vals(p) = func_callable(theta, phi)
    end do

    integral = angles%integrate_points(func_vals
       )

    deallocate(func_vals)
  end function angle_integrate_func

  subroutine angle_deinit(angles)
    class(angle2d) :: angles
    deallocate(angles%theta)
    deallocate(angles%phi)
    deallocate(angles%theta_edge)
    deallocate(angles%phi_edge)
    deallocate(angles%theta_p)
    deallocate(angles%phi_p)
    deallocate(angles%theta_edge_p)
    deallocate(angles%phi_edge_p)
    deallocate(angles%cos_theta)
    deallocate(angles%sin_theta)
    deallocate(angles%cos_phi)
    deallocate(angles%sin_phi)
    deallocate(angles%cos_theta_p)
    deallocate(angles%sin_theta_p)
    deallocate(angles%cos_phi_p)
    deallocate(angles%sin_phi_p)
    deallocate(angles%cos_theta_edge)
    deallocate(angles%sin_theta_edge)
    deallocate(angles%cos_phi_edge)
    deallocate(angles%sin_phi_edge)
    deallocate(angles%cos_theta_edge_p)
    deallocate(angles%sin_theta_edge_p)
    deallocate(angles%cos_phi_edge_p)
    deallocate(angles%sin_phi_edge_p)
    deallocate(angles%area_p)
  end subroutine angle_deinit


```

```
!!! ANGLE 1D !!!

subroutine angle_set_bounds(angle, minval,
   maxval)
  class(angle_dim) :: angle
  double precision minval, maxval
  angle%minval = minval
  angle%maxval = maxval
end subroutine angle_set_bounds

subroutine angle1d_set_num(angle, num)
  class(angle_dim) :: angle
  integer num
  angle%num = num
end subroutine angle1d_set_num

subroutine angle1d_assign_linspace(angle)
  class(angle_dim) :: angle
  double precision spacing
  integer i

  spacing = (angle%maxval - angle%minval) /
     dble(angle%num)
  do i=1, angle%num
     angle%vals(i) = (i-1) * spacing
  end do
end subroutine angle1d_assign_linspace

! To calculate \int_{xmin}^{xmax} f(x) dx :
! int = prefactor * sum(weights * f(roots))
subroutine assign_legendre(angle)
  class(angle_dim) :: angle
  double precision root, weight, theta
  integer i
  ! glpair produces both x and theta, where x=
     cos(theta). We'll throw out theta.

  allocate(angle%vals(angle%num))
  allocate(angle%weights(angle%num))
  allocate(angle%sin(angle%num))
  allocate(angle%cos(angle%num))

  ! Prefactor for integration
  ! From change of variables
  angle%prefactor = (angle%maxval - angle%
     minval) / 2.d0

  do i = 1, angle%num
     call glpair(angle%num, i, theta, weight,
        root)
```

```fortran
       call affine_transform(root, -1.d0, 1.d0, angle%minval, angle%maxval)
       angle%vals(i) = root
       angle%weights(i) = weight
       angle%sin(i) = sin(root)
       angle%cos(i) = cos(root)
    end do

  end subroutine assign_legendre

  ! Integrate callable function over angle via Gauss-Legendre quadrature

  function angle1d_integrate_func(angle, func_callable) result(integral)
    class(angle_dim) :: angle
    double precision, external :: func_callable
    double precision, dimension(:), allocatable :: func_vals
    double precision integral
    integer i

    allocate(func_vals(angle%num))

    do i=1, angle%num
       func_vals(i) = func_callable(angle%vals(i))
    end do

    integral = angle%integrate_points(func_vals)

    deallocate(func_vals)
  end function angle1d_integrate_func

  ! Integrate function given function values sampled at legendre theta values
  function angle1d_integrate_points(angle, func_vals) result(integral)
    class(angle_dim) :: angle
    double precision, dimension(angle%num) :: func_vals
    double precision integral

    integral = angle%prefactor * sum(angle%weights * func_vals)
  end function angle1d_integrate_points

  subroutine angle1d_deinit(angle)
    class(angle_dim) :: angle
    deallocate(angle%vals)
    deallocate(angle%weights)
    deallocate(angle%sin)
    deallocate(angle%cos)
  end subroutine angle1d_deinit


  !! SPACE !!

  ! Integrate function given function values
     sampled at even grid points
  function space_integrate_points(space,
     func_vals) result(integral)
    class(space_dim) :: space
    double precision, dimension(space%num) ::
       func_vals
    double precision integral

    ! Encapsulate actual method for easy
       switching
    integral = space%trapezoid_rule(func_vals)

  end function space_integrate_points

  function trapezoid_rule(space, func_vals)
     result(integral)
    class(space_dim) :: space
    double precision, dimension(space%num) ::
       func_vals
    double precision integral

    integral = 0.5d0 * sum(func_vals * space%
       spacing)
  end function

  subroutine space_set_bounds(space, minval,
     maxval)
    class(space_dim) :: space
    double precision minval, maxval
    space%minval = minval
    space%maxval = maxval
  end subroutine space_set_bounds

  subroutine space_set_num(space, num)
    class(space_dim) :: space
    integer num
    space%num = num
  end subroutine space_set_num

  subroutine space_set_uniform_spacing(space,
     spacing)
    class(space_dim) :: space
    double precision spacing
    integer k
    do k=1, space%num
      space%spacing(k) = spacing
   end do
  end subroutine space_set_uniform_spacing

  subroutine space_set_spacing_array(space, spacing)
    class(space_dim) :: space
    double precision, dimension(space%num) :: spacing
    space%spacing = spacing
  end subroutine space_set_spacing_array

  subroutine assign_linspace(space)
    class(space_dim) :: space
    double precision spacing
    integer i

    allocate(space%vals(space%num))
    allocate(space%edges(space%num))
    allocate(space%spacing(space%num))

    spacing = spacing_from_num(space%minval, space%maxval, space%num)
    call space%set_uniform_spacing(spacing)

    do i=1, space%num
       space%edges(i) = space%minval + dble(i-1) * space%spacing(i)
       space%vals(i) = space%minval + dble(i-0.5d0) * space%spacing(i)
    end do

  end subroutine assign_linspace

  subroutine set_uniform_spacing_from_num(space)
    ! Create evenly spaced grid (linspace)
    class(space_dim) :: space
    double precision spacing

    spacing = spacing_from_num(space%minval, space%maxval, space%num)
    call space%set_uniform_spacing(spacing)

  end subroutine set_uniform_spacing_from_num

 !  subroutine set_num_from_spacing(space)
 !    class(space_dim) :: space
```

```
! 	!space%num = num_from_spacing(space%minval
    , space%maxval, space%spacing)

!  end subroutine set_num_from_spacing

  subroutine space_deinit(space)
    class(space_dim) :: space
    deallocate(space%vals)
    deallocate(space%edges)
    deallocate(space%spacing)
  end subroutine space_deinit

  !! SAG !!

  subroutine sag_set_bounds(grid, xmin, xmax,
     ymin, ymax, zmin, zmax)
    class(space_angle_grid) :: grid
    double precision xmin, xmax, ymin, ymax,
       zmin, zmax

    call grid%x%set_bounds(xmin, xmax)
    call grid%y%set_bounds(ymin, ymax)
    call grid%z%set_bounds(zmin, zmax)
  end subroutine sag_set_bounds

  subroutine sag_set_uniform_spacing(grid, dx,
     dy, dz)
    class(space_angle_grid) :: grid
    double precision dx, dy, dz
    call grid%x%set_uniform_spacing(dx)
    call grid%y%set_uniform_spacing(dy)
    call grid%z%set_uniform_spacing(dz)
  end subroutine sag_set_uniform_spacing

  subroutine sag_set_num(grid, nx, ny, nz,
     ntheta, nphi)
    class(space_angle_grid) :: grid
    integer nx, ny, nz, ntheta, nphi
    call grid%x%set_num(nx)
    call grid%y%set_num(ny)
    call grid%z%set_num(nz)
    call grid%angles%set_num(ntheta, nphi)
  end subroutine sag_set_num

  subroutine sag_init(grid)
    class(space_angle_grid) :: grid

    call grid%x%assign_linspace()
    call grid%y%assign_linspace()
    call grid%z%assign_linspace()

    call grid%angles%init()
    call grid%calculate_factors()

  end subroutine sag_init

  subroutine sag_calculate_factors(grid)
    ! Factors by which depth difference is
       multiplied
    ! in order to calculate distance traveled in
        the
    ! (x, y) direction along a ray in the (theta
       , phi)
    ! direction
    class(space_angle_grid) :: grid
    integer p, nomega
    double precision theta, phi

    nomega = grid%angles%nomega

    allocate(grid%x_factor(nomega))
    allocate(grid%y_factor(nomega))

    do p=1, nomega
       theta = grid%angles%theta_p(p)
       phi = grid%angles%phi_p(p)
       grid%x_factor(p) = tan(phi) * cos(theta)
       grid%y_factor(p) = tan(phi) * sin(theta)
    end do

  end subroutine sag_calculate_factors

  subroutine sag_set_uniform_spacing_from_num(
     grid)
    class(space_angle_grid) :: grid
    call grid%x%set_uniform_spacing_from_num()
    call grid%y%set_uniform_spacing_from_num()
    call grid%z%set_uniform_spacing_from_num()
  end subroutine
     sag_set_uniform_spacing_from_num

  ! subroutine sag_set_num_from_spacing(grid)
  !   class(space_angle_grid) :: grid
  !   call grid%x%set_num_from_spacing()
  !   call grid%y%set_num_from_spacing()
  !   call grid%z%set_num_from_spacing()

  ! end subroutine sag_set_num_from_spacing

  subroutine sag_deinit(grid)
    class(space_angle_grid) :: grid
    call grid%x%deinit()
    call grid%y%deinit()
    call grid%z%deinit()
    call grid%angles%deinit()

    deallocate(grid%x_factor)
    deallocate(grid%y_factor)
  end subroutine sag_deinit

  ! Affine shift on x from [xmin, xmax] to [ymin
     , ymax]
  subroutine affine_transform(x, xmin, xmax,
     ymin, ymax)
    double precision x, xmin, xmax, ymin, ymax
    x = ymin + (ymax-ymin)/(xmax-xmin) * (x-xmin
       )
  end subroutine affine_transform

  function num_from_spacing(xmin, xmax, dx)
     result(n)
    double precision xmin, xmax, dx
    integer n
    n = floor( (xmax - xmin) / dx )
  end function num_from_spacing

  function spacing_from_num(xmin, xmax, nx)
     result(dx)
    double precision xmin, xmax, dx
    integer nx
    dx = (xmax - xmin) / dble(nx)
  end function spacing_from_num
end module sag
```

kelp3d.f90

```
! Kelp 3D
! Oliver Evans
! 8/31/2017

! Given superindividual/water current data at
   each depth, generate kelp distribution at
   each point in 3D space

module kelp3d

use kelp_context

implicit none

contains

```

```
subroutine generate_grid(xmin, xmax, nx, ymin,
   ymax, ny, zmin, zmax, nz, ntheta, nphi, grid,
    p_kelp)
  double precision xmin, xmax, ymin, ymax, zmin,
      zmax
  integer nx, ny, nz, ntheta, nphi
  type(space_angle_grid) grid
  double precision, dimension(:,:,:),
     allocatable :: p_kelp

  call grid%set_bounds(xmin, xmax, ymin, ymax,
     zmin, zmax)
  call grid%set_num(nx, ny, nz, ntheta, nphi)

  allocate(p_kelp(nx,ny,nz))

end subroutine generate_grid

subroutine kelp3d_deinit(grid, rope, p_kelp)
  type(space_angle_grid) grid
  type(rope_state) rope
  double precision, dimension(:,:,:),
     allocatable :: p_kelp
  call rope%deinit()
  call grid%deinit()
  deallocate(p_kelp)
end subroutine kelp3d_deinit

subroutine calculate_kelp_on_grid(grid, p_kelp,
   frond, rope, quadrature_degree)
  type(space_angle_grid), intent(in) :: grid
  type(frond_shape), intent(in) :: frond
  type(rope_state), intent(in) :: rope
  type(point3d) point
  integer, intent(in) :: quadrature_degree
  double precision, dimension(grid%x%num, grid%y
     %num, grid%z%num) :: p_kelp
  type(depth_state) depth

  integer i, j, k, nx, ny, nz
  double precision x, y, z

  nx = grid%x%num
  ny = grid%y%num
  nz = grid%z%num

  do k=1, nz
    z = grid%z%vals(k)
    call depth%set_depth(rope, grid, k)
    do i=1, nx
```

```fortran
      x = grid%x%vals(i)
      do j=1, ny
        y = grid%y%vals(j)
        call point%set_cart(x, y, z)
        p_kelp(i, j, k) = kelp_proportion(point,
            frond, grid, depth,
           quadrature_degree)
        !p_kelp(i, j, k) = prob_kelp(point,
           frond, depth, quadrature_degree)
     end do
    end do
  end do
end subroutine calculate_kelp_on_grid

subroutine shading_region_limits(theta_low_lim,
   theta_high_lim, point, frond)
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  double precision, intent(out) :: theta_low_lim
     , theta_high_lim

  theta_low_lim = point%theta - frond%alpha
  theta_high_lim = point%theta + frond%alpha
end subroutine shading_region_limits

function prob_kelp(point, frond, depth,
   quadrature_degree)
! P_s(theta_p, r_p) - This is the proportion of
   the population of this depth layer which can
   be found in this Cartesian grid cell.
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  type(depth_state), intent(in) :: depth
  integer, intent(in) :: quadrature_degree
  double precision prob_kelp
  double precision theta_low_lim, theta_high_lim

  call shading_region_limits(theta_low_lim,
     theta_high_lim, point, frond)
  prob_kelp = integrate_ps(theta_low_lim,
     theta_high_lim, quadrature_degree, point,
     frond, depth)
end function prob_kelp

function kelp_proportion(point, frond, grid,
   depth, quadrature_degree)
  ! This is the proportion of the volume of the
     Cartesian grid cell occupied by kelp
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  type(depth_state), intent(in) :: depth
  type(space_angle_grid), intent(in) :: grid
  integer, intent(in) :: quadrature_degree
  double precision p_k, n, t, dz
  double precision kelp_proportion

  n = depth%num_fronds
  dz = grid%z%spacing(depth%depth_layer)
  t = frond%ft
  !write(*,*) 'KELP PROPORTION'
  !write(*,*) 'n=', n
  !write(*,*) 'dz=', dz
  !write(*,*) 't=', t
  !write(*,*) 'coef=', n*t/dz
  p_k = prob_kelp(point, frond, depth,
     quadrature_degree)
  kelp_proportion = n*t/dz * p_k
end function kelp_proportion

function integrate_ps(theta_low_lim,
    theta_high_lim, quadrature_degree, point,
    frond, depth) result(integral)
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  double precision, intent(in) :: theta_low_lim,
      theta_high_lim
  integer, intent(in) :: quadrature_degree
  type(depth_state), intent(in) :: depth
  double precision integral
  double precision, dimension(:), allocatable ::
      integrand_vals
  integer i

  type(angle_dim) :: theta_f
  call theta_f%set_bounds(theta_low_lim,
     theta_high_lim)
  call theta_f%set_num(quadrature_degree)
  call theta_f%assign_legendre()

  allocate(integrand_vals(theta_f%num))

  do i=1, theta_f%num
    integrand_vals(i) = ps_integrand(theta_f%
        vals(i), point, frond, depth)
  end do

  integral = theta_f%integrate_points(
     integrand_vals)

  deallocate(integrand_vals)
```

```fortran
  call theta_f%deinit()

end function integrate_ps

function ps_integrand(theta_f, point, frond,
   depth)
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  type(depth_state), intent(in) :: depth
  double precision theta_f, l_min
  double precision angular_part, length_part
  double precision ps_integrand

  l_min = min_shading_length(theta_f, point,
     frond)

  angular_part = depth%angle_distribution_pdf(
     theta_f)
  length_part =  1 - depth%
     length_distribution_cdf(l_min)

  ps_integrand = angular_part * length_part
end function ps_integrand


function min_shading_length(theta_f, point,
   frond) result(l_min)
! L_min(\theta)
  type(point3d), intent(in) :: point
  type(frond_shape), intent(in) :: frond
  double precision, intent(in) :: theta_f
  double precision l_min
  double precision tpp
  double precision frond_frac

  ! tpp === theta_p_prime
  tpp = point%theta - theta_f + pi / 2.d0
  frond_frac = 2.d0 * frond%fr / (1.d0 + frond%
     fs)
  l_min = point%r * (sin(tpp) + angular_sign(tpp
     ) * frond_frac * cos(tpp))
end function min_shading_length

! function frond_edge(theta, theta_f, L, fs, fr)
! ! r_f(\theta)
!   double precision, intent(in) :: theta,
   theta_f, L, fs, fr
!   double precision, intent(out) :: frond_edge
!
!   frond_edge = relative_frond_edge(theta -
   theta_f + pi/2.d0)
```

```fortran
!
! end function frond_edge
!
! function relative_frond_edge(theta_prime, L,
    fs, fr)
! ! r_f'(\theta')
!   double precision, intent(in) :: theta_prime,
     L, fs, fr
!   double precision, intent(out) ::
    relative_frond_edge
!
!   relative_frond_edge = L / (sin(theta_prime)
    + angular_sign(theta_prime * alpha(fs, fr) *
    cos(theta_prime)))
! end function relative_frond_edge

function angular_sign(theta_prime)
! S(\theta')
  double precision, intent(in) :: theta_prime
  double precision angular_sign

  ! This seems to be incorrect in summary.pdf as
      of 9/9/18
  ! In the report, it's written as sgn(
      theta_print - pi/2.d0)
  ! This results in L_min < 0 - not good!
  angular_sign = sgn(pi/2.d0 - theta_prime)
end function angular_sign

end module kelp3d
```

rte_sparse_matrices.f90

```fortran
module rte_sparse_matrices
use sag
use kelp_context
use mgmres
!use hdf5_utils
implicit none

type solver_params
   integer maxiter_inner, maxiter_outer
   double precision tol_abs, tol_rel
end type solver_params

type rte_mat
   type(space_angle_grid) grid
   type(optical_properties) iops
   type(solver_params) params
   integer nx, ny, nz, nomega
   integer i, j, k, p
```

```
    integer nonzero, n_total
    integer x_block_size, y_block_size,
       z_block_size, omega_block_size

    double precision, dimension(:), allocatable
       :: surface_vals

    ! A stored in coordinate form in row, col,
       data
    integer, dimension(:), allocatable :: row,
       col
    double precision, dimension(:), allocatable
       :: data
    ! b and x stored in rhs in full form
    double precision, dimension(:), allocatable
       :: rhs, sol

    ! Pointer to solver subroutine
    ! Set to mgmres by default
    procedure(solver_interface), pointer, nopass
       :: solver => mgmres_st

  contains
    procedure :: init => mat_init
    procedure :: deinit => mat_deinit
    procedure :: calculate_size
    procedure :: set_solver_params =>
       mat_set_solver_params
    procedure :: assign => mat_assign
    procedure :: add => mat_add
    procedure :: assign_rhs => mat_assign_rhs
    !procedure :: store_index => mat_store_index
    !procedure :: find_index => mat_find_index
    procedure :: set_bc => mat_set_bc
    procedure :: solve => mat_solve
    procedure :: ind => mat_ind
    !procedure :: to_hdf => mat_to_hdf
    procedure attenuate
    procedure angular_integral

    ! Derivative subroutines
    procedure x_cd2
    procedure x_cd2_first
    procedure x_cd2_last
    procedure y_cd2
    procedure y_cd2_first
    procedure y_cd2_last
    procedure z_cd2
    procedure z_fd2
    procedure z_bd2
    procedure z_surface_bc
    procedure z_bottom_bc

```

```fortran
end type rte_mat

interface
   ! Define interface for external procedure
   ! https://stackoverflow.com/questions/8549415/how-to-declare-the-interface-section-for-a-procedure-argument-which-in-turn-ref
   subroutine solver_interface(n_total, nonzero, row, col, data, &
        sol, rhs, maxiter_outer, maxiter_inner, &
        tol_abs, tol_rel)
     integer ::  n_total, nonzero
     integer, dimension(nonzero) :: row, col
     double precision, dimension(nonzero) :: data
     double precision, dimension(nonzero) :: sol
     double precision, dimension(n_total) :: rhs
     integer :: maxiter_outer, maxiter_inner
     double precision :: tol_abs, tol_rel
   end subroutine solver_interface
end interface

contains

  subroutine mat_init(mat, grid, iops)
    class(rte_mat) mat
    type(space_angle_grid) grid
    type(optical_properties) iops
    integer nnz, n_total

    mat%grid = grid
    mat%iops = iops

    call mat%calculate_size()

    n_total = mat%n_total
    nnz = mat%nonzero
    allocate(mat%surface_vals(grid%angles%nomega))
    allocate(mat%row(nnz))
    allocate(mat%col(nnz))
    allocate(mat%data(nnz))
    allocate(mat%rhs(n_total))
    allocate(mat%sol(n_total))

    call zeros(mat%rhs, n_total)
    call zeros(mat%sol, n_total)

  end subroutine mat_init

  subroutine mat_deinit(mat)
    class(rte_mat) mat
    deallocate(mat%row)
    deallocate(mat%col)
    deallocate(mat%data)
    deallocate(mat%rhs)
    deallocate(mat%sol)
    deallocate(mat%surface_vals)
  end subroutine mat_deinit

  subroutine calculate_size(mat)
    class(rte_mat) mat
    integer nx, ny, nz, nomega

    nx = mat%grid%x%num
    ny = mat%grid%y%num
    nz = mat%grid%z%num
    nomega = mat%grid%angles%nomega

    !mat%nonzero = nx * ny * ntheta * nphi * ( (
       nz-1) * (6 + ntheta * nphi) + 1)
    mat%nonzero = nx * ny * nomega * (nz * (
       nomega + 6) - 1)
    mat%n_total = nx * ny * nz * nomega

    !mat%theta_block_size = 1
    !mat%phi_block_size = mat%theta_block_size *
        ntheta
    mat%omega_block_size = 1
    mat%y_block_size = mat%omega_block_size *
       nomega
    mat%x_block_size = mat%y_block_size * ny
    mat%z_block_size = mat%x_block_size * nx

  end subroutine calculate_size

!  subroutine mat_to_hdf(mat,filename)
!    class(rte_mat) mat
!    character(len=*) filename
!    call write_coo(filename, mat%row, mat%col,
   mat%data, mat%nonzero)
!  end subroutine mat_to_hdf

  subroutine mat_set_bc(mat, bc)
    class(rte_mat) mat
    class(boundary_condition) bc
    integer p

    do p=1, mat%grid%angles%nomega/2
```

```fortran
      mat%surface_vals(p) = bc%bc_grid(p)
    end do
  end subroutine mat_set_bc

  subroutine mat_solve(mat)
    class(rte_mat) mat
    type(solver_params) params

    params = mat%params

    write(*,*) 'mat%n_total =', mat%n_total
    write(*,*) 'mat%nonzero =', mat%nonzero
    write(*,*) 'size(mat%row) =', size(mat%row)
    write(*,*) 'size(mat%col) =', size(mat%col)
    write(*,*) 'size(mat%data) =', size(mat%data)
    write(*,*) 'size(mat%sol) =', size(mat%sol)
    write(*,*) 'size(mat%rhs) =', size(mat%rhs)
    write(*,*) 'params%maxiter_outer =', params%maxiter_outer
    write(*,*) 'params%maxiter_inner =', params%maxiter_inner
    write(*,*) 'params%tol_rel =', params%tol_rel
    write(*,*) 'params%tol_abs =', params%tol_abs
    ! open(unit=1, file='row.txt')
    ! open(unit=2, file='col.txt')
    ! open(unit=3, file='data.txt')
    ! open(unit=4, file='rhs.txt')
    ! open(unit=5, file='sol.txt')
    ! write(1,*) mat%row
    ! write(2,*) mat%col
    ! write(3,*) mat%data
    ! write(4,*) mat%rhs

    ! close(1)
    ! close(2)
    ! close(3)
    ! close(4)

    call mat%solver(mat%n_total, mat%nonzero, &
         mat%row, mat%col, mat%data, mat%sol, mat%rhs, &
         params%maxiter_outer, params%maxiter_inner, &
         params%tol_abs, params%tol_rel)

    ! write(5,*) mat%sol
    ! close(5)

  end subroutine mat_solve

  subroutine mat_set_solver_params(mat, maxiter_outer, &
       maxiter_inner, tol_abs, tol_rel)
    class(rte_mat) mat
    integer maxiter_outer, maxiter_inner
    double precision tol_abs, tol_rel

    mat%params%maxiter_outer = maxiter_outer
    mat%params%maxiter_inner = maxiter_inner
    mat%params%tol_abs = tol_abs
    mat%params%tol_rel = tol_rel
  end subroutine mat_set_solver_params

  function mat_ind(mat, i, j, k, p) result(ind)
    ! Assuming var ordering: z, x, y, omega
    class(rte_mat) mat
    integer i, j, k, p
    integer ind

    ind = (i-1) * mat%x_block_size + (j-1) * mat%y_block_size + &
         (k-1) * mat%z_block_size + p * mat%omega_block_size
  end function mat_ind

  subroutine mat_assign(mat, row_num, ent, val, i, j, k, p)
    ! It's assumed that this is the only time this entry is defined
    class(rte_mat) mat
    double precision val
    integer i, j, k, p
    integer row_num, ent

    mat%row(ent) = row_num
    mat%col(ent) = mat%ind(i, j, k, p)
    mat%data(ent) = val

    ent = ent + 1
  end subroutine mat_assign

  subroutine mat_add(mat, repeat_ent, val)
    ! Use this when you know that this entry has already been assigned
    ! and you'd like to add this value to the existing value.

    class(rte_mat) mat
    double precision val
    integer repeat_ent

    ! Entry number where value is already stored
    mat%data(repeat_ent) = mat%data(repeat_ent)
       + val
  end subroutine mat_add

  subroutine mat_assign_rhs(mat, row_num, data)
    class(rte_mat) mat
    double precision data
    integer row_num

    mat%rhs(row_num) = data
  end subroutine mat_assign_rhs

  ! subroutine mat_store_index(mat, row_num,
     col_num)
  !   ! Remember where we stored information for
       this matrix element
  !   class(rte_mat) mat
  !   integer row_num, col_num
  !   !mat%index_map(row_num, col_num) = mat%ent
  ! end subroutine

  ! function mat_find_index(mat, row_num,
     col_num) result(index)
  !   ! Find the position in row, col, data
     where this entry
  !   ! is defined.
  !   class(rte_mat) mat
  !   integer row_num, col_num, index

  !   index = mat%index_map(row_num, col_num)

  !   ! This took up 95% of execution time.
  !   ! Only search up to most recently assigned
       index
  !   ! do index=1, mat%ent-1
  !   !    if( (mat%row(index) .eq. row_num) .
     and. (mat%col(index) .eq. col_num)) then
  !   !       exit
  !   !    end if
  !   ! end do
  ! end function mat_find_index

  subroutine attenuate(mat, indices, repeat_ent)
    ! Has to be called after angular_integral
    ! Because they both write to the same matrix
        entry
    ! And adding here is more efficient than a
       conditional
```

```fortran
    ! in the angular loop.
    class(rte_mat) mat
    double precision attenuation
    type(index_list) indices
    double precision aa, bb
    integer repeat_ent

    aa = mat%iops%abs_grid(indices%i, indices%j, indices%k)
    bb = mat%iops%scat
    attenuation = aa + bb

    call mat%add(repeat_ent, attenuation)
  end subroutine attenuate

  subroutine x_cd2(mat, indices, row_num, ent)
    class(rte_mat) mat
    double precision val, dx
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dx = mat%grid%x%spacing(1)

    val = mat%grid%angles%sin_phi_p(p) &
         * mat%grid%angles%cos_theta_p(p) / (2.d0 * dx)

    call mat%assign(row_num,ent,-val,i-1,j,k,p)
    call mat%assign(row_num,ent,val,i+1,j,k,p)
  end subroutine x_cd2

  subroutine x_cd2_first(mat, indices, row_num, ent)
    class(rte_mat) mat
    double precision val, dx
    integer nx
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dx = mat%grid%x%spacing(1)
    nx = mat%grid%x%num

    val = mat%grid%angles%sin_phi_p(p) &
         * mat%grid%angles%cos_theta_p(p) / (2.
            d0 * dx)

    call mat%assign(row_num,ent,-val,nx,j,k,p)
    call mat%assign(row_num,ent,val,i+1,j,k,p)
  end subroutine x_cd2_first

  subroutine x_cd2_last(mat, indices, row_num,
     ent)
    class(rte_mat) mat
    double precision val, dx
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dx = mat%grid%x%spacing(1)

    val = mat%grid%angles%sin_phi_p(p) &
         * mat%grid%angles%cos_theta_p(p) / (2.
            d0 * dx)

    call mat%assign(row_num,ent,-val,i-1,j,k,p)
    call mat%assign(row_num,ent,val,1,j,k,p)
  end subroutine x_cd2_last

  subroutine y_cd2(mat, indices, row_num, ent)
    class(rte_mat) mat
    double precision val, dy
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dy = mat%grid%y%spacing(1)

    val = mat%grid%angles%sin_phi_p(p) &
```

```
         * mat%grid%angles%sin_theta_p(p) / (2.
            d0 * dy)

    call mat%assign(row_num,ent,-val,i,j-1,k,p)
    call mat%assign(row_num,ent,val,i,j+1,k,p)
  end subroutine y_cd2

  subroutine y_cd2_first(mat, indices, row_num,
     ent)
    class(rte_mat) mat
    double precision val, dy
    integer ny
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dy = mat%grid%y%spacing(1)
    ny = mat%grid%y%num

    val = mat%grid%angles%sin_phi_p(p) &
         * mat%grid%angles%sin_theta_p(p) / (2.
            d0 * dy)

    call mat%assign(row_num,ent,-val,i,ny,k,p)
    call mat%assign(row_num,ent,val,i,j+1,k,p)
  end subroutine y_cd2_first

  subroutine y_cd2_last(mat, indices, row_num,
     ent)
    class(rte_mat) mat
    double precision val, dy
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dy = mat%grid%y%spacing(1)

    val = mat%grid%angles%sin_phi_p(p) &
         * mat%grid%angles%sin_theta_p(p) / (2.
            d0 * dy)

    call mat%assign(row_num,ent,-val,i,j-1,k,p)
    call mat%assign(row_num,ent,val,i,1,k,p)
  end subroutine y_cd2_last

  subroutine z_cd2(mat, indices, row_num, ent)
    class(rte_mat) mat
    double precision val, dz
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dz = mat%grid%z%spacing(indices%k)

    val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
        dz)

    call mat%assign(row_num,ent,-val,i,j,k-1,p)
    call mat%assign(row_num,ent,val,i,j,k+1,p)
  end subroutine z_cd2

  subroutine z_fd2(mat, indices, row_num, ent,
     repeat_ent)
    ! Has to be called after angular_integral
    ! Because they both write to the same matrix
        entry
    ! And adding here is more efficient than a
       conditional
    ! in the angular loop.
    class(rte_mat) mat
    double precision val, val1, val2, val3, dz
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent, repeat_ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dz = mat%grid%z%spacing(indices%k)

    val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
        dz)

    val1 = -3.d0 * val
    val2 = 4.d0 * val
    val3 = -val

    call mat%add(repeat_ent, val1)
    call mat%assign(row_num,ent,val2,i,j,k+1,p)
    call mat%assign(row_num,ent,val3,i,j,k+2,p)
  end subroutine z_fd2

  subroutine z_bd2(mat, indices, row_num, ent,
     repeat_ent)
    ! Has to be called after angular_integral
    ! Because they both write to the same matrix
        entry
    ! And adding here is more efficient than a
       conditional
    ! in the angular loop.
    class(rte_mat) mat
    double precision val, val1, val2, val3, dz
    type(index_list) indices
    integer i, j, k, p
    integer row_num, ent, repeat_ent

    i = indices%i
    j = indices%j
    k = indices%k
    p = indices%p

    dz = mat%grid%z%spacing(indices%k)

    val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
        dz)

    val1 = 3.d0 * val
    val2 = -4.d0 * val
    val3 = val

    call mat%add(repeat_ent, val1)
    call mat%assign(row_num,ent,val2,i,j,k-1,p)
    call mat%assign(row_num,ent,val3,i,j,k-2,p)
  end subroutine z_bd2

  subroutine angular_integral(mat, indices,
     row_num, ent)
    class(rte_mat) mat
    ! Primed angular integration variables
    integer pp
    double precision val
    type(index_list) indices
    integer row_num, ent

    ! Interior
```

```fortran
    do pp=1, mat%grid%angles%nomega
      ! TODO: Make sure I don't have p and pp backwards
      val = -mat%iops%scat * mat%iops%vsf_integral(indices%p, pp)
      call mat%assign(row_num, ent, val, indices%i, indices%j, indices%k, pp)
    end do
  end subroutine angular_integral

  subroutine z_surface_bc(mat, indices, row_num, ent, repeat_ent)
    class(rte_mat) mat
    double precision bc_val
    type(index_list) indices
    double precision val1, val2, dz
    integer row_num, ent, repeat_ent

    dz = mat%grid%z%spacing(1)

    val1 = mat%grid%angles%cos_phi_p(indices%p) / (5.d0 * dz)
    val2 = 7.d0 * val1
    bc_val = 8.d0 * val1 * mat%surface_vals(indices%p)

    call mat%assign(row_num,ent,val1,indices%i,indices%j,2,indices%p)
    call mat%add(repeat_ent, val2)
    call mat%assign_rhs(row_num, bc_val)

  end subroutine z_surface_bc

    subroutine z_bottom_bc(mat, indices, row_num, ent, repeat_ent)
    class(rte_mat) mat
    type(index_list) indices
    double precision val1, val2, dz
    integer nz
    integer row_num, ent, repeat_ent

    dz = mat%grid%z%spacing(1)
    nz = mat%grid%z%num

    val1 = -mat%grid%angles%cos_phi_p(indices%p) / (5.d0 * dz)
    val2 = 7.d0 * val1

    call mat%assign(row_num,ent,val1,indices%i,indices%j,nz-1,indices%p)
    call mat%add(repeat_ent, val2)

  end subroutine z_bottom_bc

  ! Finite difference wrappers

  ! subroutine wrap_x_cd2(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%x_cd2(indices)
  ! end subroutine wrap_x_cd2

  ! subroutine wrap_x_cd2_last(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%x_cd2_last(indices)
  ! end subroutine wrap_x_cd2_last

  ! subroutine wrap_x_cd2_first(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%x_cd2_first(indices)
  ! end subroutine wrap_x_cd2_first

  ! subroutine wrap_y_cd2(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%y_cd2(indices)
  ! end subroutine wrap_y_cd2

  ! subroutine wrap_y_cd2_last(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%y_cd2_last(indices)
  ! end subroutine wrap_y_cd2_last

  ! subroutine wrap_y_cd2_first(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%y_cd2_first(indices)
  ! end subroutine wrap_y_cd2_first

  ! subroutine wrap_z_cd2(mat, indices)
  !   type(rte_mat) mat
  !   type(index_list) indices
  !   call mat%z_cd2(indices)
  ! end subroutine wrap_z_cd2

end module rte_sparse_matrices
```

rte3d.f90

```
module rte3d
use kelp_context
use rte_sparse_matrices
use light_context
implicit none

interface
   subroutine deriv_interface(mat, indices,
      row_num, ent)
     use rte_sparse_matrices
     class(rte_mat) mat
     type(index_list) indices
     integer row_num, ent
   end subroutine deriv_interface
   subroutine angle_loop_interface(mat, indices,
       ddx, ddy)
     use rte_sparse_matrices
     import deriv_interface
     type(space_angle_grid) grid
     type(rte_mat) mat
     type(index_list) indices
     procedure(deriv_interface) :: ddx, ddy
   end subroutine angle_loop_interface
end interface

contains

subroutine whole_space_loop(mat, indices)
  type(rte_mat) mat
  type(index_list) indices
  integer i, j, k

  procedure(deriv_interface), pointer :: ddx,
     ddy
  procedure(angle_loop_interface), pointer ::
     angle_loop

  !$ integer omp_get_num_procs
  !$ integer num_threads_z, num_threads_x,
     num_threads_y

  ! Enable nested parallelism
  !$ call omp_set_nested(.true.)

  ! Use nz procs for outer loop,
  ! or num_procs if num_procs < nz
  ! Divide the rest of the tasks as appropriate

  !$ num_threads_z = min(omp_get_num_procs(),
     mat%grid%z%num)
  !$ num_threads_x = min( &
  !$     omp_get_num_procs()/num_threads_z, &
  !$     mat%grid%x%num)
  !$ num_threads_y = min( &
  !$     omp_get_num_procs()/(num_threads_z*
     num_threads_x), &
  !$     mat%grid%y%num)

  !$ write(*,*) 'num_procs =', omp_get_num_procs
     ()
  !$ write(*,*) 'ntz =', num_threads_z
  !$ write(*,*) 'ntx =', num_threads_x
  !$ write(*,*) 'nty =', num_threads_y

  !$omp parallel do default(none) shared(mat) &
  !$omp private(ddx,ddy,angle_loop, k, i, j)
     private(indices) &
  !$omp shared(num_threads_x,num_threads_y,
     num_threads_z) &
  !$omp num_threads(num_threads_z) if(
     num_threads_z .gt. 1)
  do k=1, mat%grid%z%num
     write(*,*) 'k =', k
     indices%k = k
     if(k .eq. 1) then
        angle_loop => surface_angle_loop
     else if(k .eq. mat%grid%z%num) then
        angle_loop => bottom_angle_loop
     else
        angle_loop => interior_angle_loop
     end if

     !$omp parallel do default(none) shared(mat)
         private(i,j) &
     !$omp firstprivate(indices,angle_loop, k)
        private(ddx,ddy) &
     !$omp shared(num_threads_x,num_threads_y,
        num_threads_z) &
     !$omp num_threads(num_threads_x) if(
        num_threads_x .gt. 1)
     do i=1, mat%grid%x%num
        indices%i = i
        if(indices%i .eq. 1) then
           ddx => x_cd2_first
        else if(indices%i .eq. mat%grid%x%num)
           then
           ddx => x_cd2_last
        else
           ddx => x_cd2
        end if
```

```
        !$omp parallel do default(none) shared(mat) private(j) &
        !$omp firstprivate(indices,ddx,ddy,angle_loop, i, k) &
        !$omp shared(num_threads_x,num_threads_y,num_threads_z) &
        !$omp num_threads(num_threads_y) if(num_threads_y .gt. 1)
        do j=1, mat%grid%y%num
           indices%j = j
           if(indices%j .eq. 1) then
              ddy => y_cd2_first
           else if(indices%j .eq. mat%grid%y%num) then
              ddy => y_cd2_last
           else
              ddy => y_cd2
           end if

           call angle_loop(mat, indices, ddx, ddy)
        end do
        !$omp end parallel do
     end do
     !$omp end parallel do
  end do
  !$omp end parallel do
end subroutine whole_space_loop

function calculate_start_ent(grid, indices) result(ent)
  type(space_angle_grid) grid
  type(index_list) indices
  integer ent
  integer boundary_nnz, interior_nnz
  integer num_boundary, num_interior
  integer num_this_x, num_this_z

  ! Nonzero matrix entries for an surface or bottom spatial grid cell
  ! Definitely an integer since nomega is even
  boundary_nnz = grid%angles%nomega * (2 * grid%angles%nomega + 11) / 2
  ! Nonzero matrix entries for an interior spatial grid cell
  interior_nnz = grid%angles%nomega * (grid%angles%nomega + 6)

  ! Order: z, x, y, omega
  ! Total number traversed so far in each spatial category
```

```fortran
  ! row
  num_this_x = indices%j - 1
  ! depth layer
  num_this_z = (indices%i - 1) * grid%y%num +
     num_this_x

  ! Calculate number of spatial grid cells of
     each type which have
  ! already been traversed up to this point
  if(indices%k .eq. 1) then
     num_boundary = num_this_z
     num_interior = 0
  else if(indices%k .eq. grid%z%num) then
     num_boundary = (grid%x%num * grid%y%num) +
        num_this_z
     num_interior = (grid%z%num-2) * grid%x%num
        * grid%y%num
  else
     num_boundary = grid%x%num * grid%y%num
     num_interior = num_this_z + (indices%k-2) *
          grid%x%num * grid%y%num
  end if

  ent = num_boundary * boundary_nnz +
     num_interior * interior_nnz + 1
end function calculate_start_ent

function calculate_repeat_ent(ent, p) result(
    repeat_ent)
  integer ent, p, repeat_ent
  ! Entry number for row=mat%ind(i,j,k,p), col=
     mat%ind(i,j,k,p),
  ! which will be modified multiple times in
     this matrix row
  repeat_ent = ent + p - 1
end function calculate_repeat_ent

subroutine interior_angle_loop(mat, indices, ddx
    , ddy)
  type(rte_mat) mat
  type(index_list) indices
  procedure(deriv_interface) :: ddx, ddy
  integer p
  integer ent, repeat_ent
  integer row_num

  ! Determine which matrix row to start at
  ent = calculate_start_ent(mat%grid, indices)
  indices%p = 1
  row_num = mat%ind(indices%i, indices%j,
     indices%k, indices%p)

  do p=1, mat%grid%angles%nomega
     indices%p = p
     repeat_ent = calculate_repeat_ent(ent, p)
     call mat%angular_integral(indices, row_num,
         ent)
     call ddx(mat, indices, row_num, ent)
     call ddy(mat, indices, row_num, ent)
     call mat%z_cd2(indices, row_num, ent)
     call mat%attenuate(indices, repeat_ent)
     row_num = row_num + 1
  end do
end subroutine

subroutine surface_angle_loop(mat, indices, ddx,
    ddy)
  type(rte_mat) mat
  type(index_list) indices
  integer p
  procedure(deriv_interface) :: ddx, ddy
  integer ent, repeat_ent
  integer row_num

  ! Determine which matrix row to start at
  ent = calculate_start_ent(mat%grid, indices)
  indices%p = 1
  row_num = mat%ind(indices%i, indices%j,
     indices%k, indices%p)

  ! Downwelling
  do p=1, mat%grid%angles%nomega / 2
     indices%p = p
     repeat_ent = calculate_repeat_ent(ent, p)
     call mat%angular_integral(indices, row_num,
         ent)
     call ddx(mat, indices, row_num, ent)
     call ddy(mat, indices, row_num, ent)
     call mat%z_surface_bc(indices, row_num, ent
        , repeat_ent)
     call mat%attenuate(indices, repeat_ent)
     row_num = row_num + 1
  end do
  ! Upwelling
  do p=mat%grid%angles%nomega/2+1, mat%grid%
     angles%nomega
     indices%p = p
     repeat_ent = calculate_repeat_ent(ent, p)
     call mat%angular_integral(indices, row_num,
         ent)
     call ddx(mat, indices, row_num, ent)
```

```fortran
      call ddy(mat, indices, row_num, ent)
      call mat%z_fd2(indices, row_num, ent,
         repeat_ent)
      call mat%attenuate(indices, repeat_ent)
      row_num = row_num + 1
   end do
end subroutine surface_angle_loop

subroutine bottom_angle_loop(mat, indices, ddx,
    ddy)
   type(rte_mat) mat
   type(index_list) indices
   integer p
   integer row_num, ent, repeat_ent
   procedure(deriv_interface) :: ddx, ddy

   ! Determine which matrix row to start at
   ent = calculate_start_ent(mat%grid, indices)
   indices%p = 1
   row_num = mat%ind(indices%i, indices%j,
      indices%k, indices%p)

   ! Downwelling
   do p=1, mat%grid%angles%nomega/2
      indices%p = p
      repeat_ent = calculate_repeat_ent(ent, p)
      call mat%angular_integral(indices, row_num,
          ent)
      call ddx(mat, indices, row_num, ent)
      call ddy(mat, indices, row_num, ent)
      call mat%z_bd2(indices, row_num, ent,
         repeat_ent)
      call mat%attenuate(indices, repeat_ent)
      row_num = row_num + 1
   end do
   ! Upwelling
   do p=mat%grid%angles%nomega/2+1, mat%grid%
      angles%nomega
      indices%p = p
      repeat_ent = calculate_repeat_ent(ent, p)
      call mat%angular_integral(indices, row_num,
          ent)
      call ddx(mat, indices, row_num, ent)
      call ddy(mat, indices, row_num, ent)
      call mat%z_bottom_bc(indices, row_num, ent,
          repeat_ent)
      call mat%attenuate(indices, repeat_ent)
      row_num = row_num + 1
   end do
end subroutine bottom_angle_loop
```

```fortran

subroutine gen_matrix(mat)
  type(rte_mat) mat
  type(index_list) indices

  call indices%init()

  call whole_space_loop(mat, indices)
  ! call surface_space_loop(mat, indices)
  ! call interior_space_loop(mat, indices)
  ! call bottom_space_loop(mat, indices)
end subroutine gen_matrix

subroutine rte3d_deinit(mat, iops, light)
  type(rte_mat) mat
  type(optical_properties) iops
  type(light_state) light

  call mat%deinit()
  call iops%deinit()
  call light%deinit()
end subroutine

end module rte3d
```

kelp_context.f90

```fortran
module kelp_context
use sag
use prob
implicit none

! Point in cylindrical coordinates
type point3d
   double precision x, y, z, theta, r
 contains
   procedure :: set_cart => point_set_cart
   procedure :: set_cyl => point_set_cyl
   procedure :: cartesian_to_polar
   procedure :: polar_to_cartesian
end type point3d

type frond_shape
  double precision fs, fr, tan_alpha, alpha, ft
contains
  procedure :: set_shape => frond_set_shape
  procedure :: calculate_angles =>
     frond_calculate_angles
end type frond_shape

type rope_state
   integer nz
   double precision, dimension(:), allocatable
      :: frond_lengths, frond_stds, num_fronds,
      water_speeds, water_angles
contains
    procedure :: init => rope_init
    procedure :: deinit => rope_deinit
end type rope_state

type depth_state
   double precision frond_length, frond_std,
      num_fronds, water_speeds, water_angles,
      depth
   integer depth_layer
contains
   procedure :: set_depth
   procedure :: length_distribution_cdf
   procedure :: angle_distribution_pdf
end type depth_state

type optical_properties
   integer num_vsf
   type(space_angle_grid) grid
   double precision, dimension(:), allocatable
      :: vsf_angles, vsf_vals, vsf_cos
   double precision, dimension(:), allocatable
      :: abs_water
   double precision abs_kelp, vsf_scat_coef,
      scat
   ! On x, y, z grid - including water & kelp.
   double precision, dimension(:,:,:),
      allocatable :: abs_grid
   ! On theta, phi, theta_prime, phi_prime grid
   double precision, dimension(:,:), allocatable
       :: vsf, vsf_integral
 contains
   procedure :: init => iop_init
   procedure :: calculate_coef_grids
   procedure :: load_vsf
   procedure :: eval_vsf
   procedure :: calc_vsf_on_grid
   procedure :: deinit => iop_deinit
   procedure :: vsf_from_function
end type optical_properties

type boundary_condition
   double precision I0, decay, theta_s, phi_s
   type(space_angle_grid) grid
   double precision, dimension(:), allocatable
      :: bc_grid
 contains
   procedure :: bc_gaussian
   procedure :: init => bc_init
   procedure :: deinit => bc_deinit
```

```fortran
end type boundary_condition

contains

  function bc_gaussian(bc, theta, phi)
    class(boundary_condition) bc
    double precision theta, phi, diff
    double precision bc_gaussian
    diff = angle_diff_3d(theta, phi, bc%theta_s,
        bc%phi_s)
    bc_gaussian = exp(-bc%decay * diff)
  end function bc_gaussian

  subroutine bc_init(bc, grid, theta_s, phi_s,
     decay, I0)
    class(boundary_condition) bc
    type(space_angle_grid) grid
    double precision theta_s, phi_s, decay, I0
    integer p
    double precision theta, phi

    allocate(bc%bc_grid(grid%angles%nomega))

    bc%theta_s = theta_s
    bc%phi_s = phi_s
    bc%decay = decay
    bc%I0 = I0

    ! Only set BC for downwelling light
    do p=1, grid%angles%nomega/2
       theta = grid%angles%theta_p(p)
       phi = grid%angles%phi_p(p)
       bc%bc_grid(p) = bc%bc_gaussian(theta, phi
         )
    end do
    ! Zero upwelling light specified at surface
    do p=grid%angles%nomega/2+1, grid%angles%
       nomega
       bc%bc_grid(p) = 0.d0
    end do

    ! Normalize
    bc%bc_grid = bc%I0 * bc%bc_grid &
         / grid%angles%integrate_points(bc%
             bc_grid)

  end subroutine bc_init

  subroutine bc_deinit(bc)
    class(boundary_condition) bc
    deallocate(bc%bc_grid)
    end subroutine

  subroutine point_set_cart(point, x, y, z)
    class(point3d) :: point
    double precision x, y, z
    point%x = x
    point%y = y
    point%z = z
    call point%cartesian_to_polar()
  end subroutine point_set_cart

  subroutine point_set_cyl(point, theta, r, z)
    class(point3d) :: point
    double precision theta, r, z
    point%theta = theta
    point%r = r
    point%z = z
    call point%polar_to_cartesian()
  end subroutine point_set_cyl

  subroutine polar_to_cartesian(point)
    class(point3d) :: point
    point%x = point%r*cos(point%theta)
    point%y = point%r*sin(point%theta)
  end subroutine polar_to_cartesian

  subroutine cartesian_to_polar(point)
    class(point3d) :: point
    point%r = sqrt(point%x**2 + point%y**2)
    point%theta = atan2(point%y, point%x)
  end subroutine cartesian_to_polar

  subroutine frond_set_shape(frond, fs, fr, ft)
    class(frond_shape) frond
    double precision fs, fr, ft
    frond%fs = fs
    frond%fr = fr
    frond%ft = ft
    call frond%calculate_angles()
  end subroutine frond_set_shape

  subroutine frond_calculate_angles(frond)
    class(frond_shape) frond
    frond%tan_alpha = 2.d0*frond%fs*frond%fr /
       (1.d0 + frond%fs)
    frond%alpha = atan(frond%tan_alpha)
  end subroutine

  subroutine iop_init(iops, grid)
    class(optical_properties) iops
    type(space_angle_grid) grid

    iops%grid = grid

    ! Assume that these are preallocated and passed to function
    ! Nevermind, don't assume this.
    allocate(iops%abs_water(grid%z%num))

    ! Assume that these must be allocated here
    allocate(iops%vsf_angles(iops%num_vsf))
    allocate(iops%vsf_vals(iops%num_vsf))
    allocate(iops%vsf_cos(iops%num_vsf))
    allocate(iops%vsf(grid%angles%nomega,grid%angles%nomega))
    allocate(iops%vsf_integral(grid%angles%nomega,grid%angles%nomega))
    allocate(iops%abs_grid(grid%x%num, grid%y%num, grid%z%num))
  end subroutine iop_init

  subroutine calculate_coef_grids(iops, p_kelp)
    class(optical_properties) iops
    double precision, dimension(:,:,:) :: p_kelp

    integer k

    ! Allow water IOPs to vary over depth
    do k=1, iops%grid%z%num
      iops%abs_grid(:,:,k) = (iops%abs_kelp - iops%abs_water(k)) * p_kelp(:,:,k) + iops%abs_water(k)
   end do

  end subroutine calculate_coef_grids


  subroutine load_vsf(iops, filename, fmtstr)
    class(optical_properties) :: iops
    character(len=*) :: filename, fmtstr
    double precision, dimension(:,:), allocatable :: tmp_2d_arr
    integer num_rows, num_cols, skiplines_in

    ! First column is the angle at which the measurement is taken
    ! Second column is the value of the VSF at that angle
    num_rows = iops%num_vsf
    num_cols = 2
    skiplines_in = 1 ! Ignore comment on first line

    allocate(tmp_2d_arr(num_rows, num_cols))

    tmp_2d_arr = read_array(filename, fmtstr, num_rows, num_cols, skiplines_in)
    iops%vsf_angles = tmp_2d_arr(:,1)
    iops%vsf_vals = tmp_2d_arr(:,2)

    ! write(*,*) 'vsf_angles = ', iops%vsf_angles
    ! write(*,*) 'vsf_vals = ', iops%vsf_vals

    ! Pre-evaluate for all pair of angles
    call iops%calc_vsf_on_grid()
  end subroutine load_vsf

  function eval_vsf(iops, theta)
    class(optical_properties) iops
    double precision theta
    double precision eval_vsf
    ! No need to set vsf(0) = 0.
    ! It's the area under the curve that matters, not the value.
    eval_vsf = interp(theta, iops%vsf_angles, iops%vsf_vals, iops%num_vsf)

  end function eval_vsf

  subroutine rope_init(rope, grid)
    class(rope_state) :: rope
    type(space_angle_grid) :: grid

    rope%nz = grid%z%num
    allocate(rope%frond_lengths(rope%nz))
    allocate(rope%frond_stds(rope%nz))
    allocate(rope%water_speeds(rope%nz))
    allocate(rope%water_angles(rope%nz))
    allocate(rope%num_fronds(rope%nz))
  end subroutine rope_init

  subroutine rope_deinit(rope)
    class(rope_state) rope
    deallocate(rope%frond_lengths)
    deallocate(rope%frond_stds)
    deallocate(rope%water_speeds)
    deallocate(rope%water_angles)
    deallocate(rope%num_fronds)
  end subroutine rope_deinit

  subroutine set_depth(depth, rope, grid,
     depth_layer)
    class(depth_state) depth
    type(rope_state) rope
    type(space_angle_grid) grid
    integer depth_layer

    depth%frond_length = rope%frond_lengths(
       depth_layer)
    depth%frond_std = rope%frond_stds(
       depth_layer)
    depth%water_speeds = rope%water_speeds(
       depth_layer)
    depth%water_angles = rope%water_angles(
       depth_layer)
    depth%num_fronds = rope%num_fronds(
       depth_layer)
    depth%depth_layer = depth_layer
    depth%depth = grid%z%vals(depth_layer)
  end subroutine set_depth

  function length_distribution_cdf(depth, L)
     result(output)
    ! C_L(L)
    class(depth_state) depth
    double precision L, L_mean, L_std
    double precision output

    L_mean = depth%frond_length
    L_std = depth%frond_std

    call normal_cdf(L, L_mean, L_std, output)
  end function length_distribution_cdf

  function angle_distribution_pdf(depth, theta_f
     ) result(output)
    ! P_{\theta_f}(\theta_f)
    class(depth_state) depth
    double precision theta_f, v_w, theta_w
    double precision output
    double precision diff

    v_w = depth%water_speeds
    theta_w = depth%water_angles

    ! von_mises_pdf is only defined on [-pi, pi]
    ! So take difference of angles and input
       into
    ! von_mises dist. centered & x=0.
```

```fortran

    diff = angle_diff_2d(theta_f, theta_w)

    call von_mises_pdf(diff, 0.d0, v_w, output)
  end function angle_distribution_pdf

  function angle_mod(theta) result(mod_theta)
    ! Shift theta to the interval [-pi, pi]
    ! which is where von_mises_pdf is defined.

    double precision theta, mod_theta

    mod_theta = mod(theta + pi, 2.d0*pi) - pi
  end function angle_mod

  function angle_diff_2d(theta1, theta2) result(diff)
    ! Shortest difference between two angles which may be
    ! in different periods.
    double precision theta1, theta2, diff
    double precision modt1, modt2

    ! Shift to [0, 2*pi]
    modt1 = mod(theta1, 2*pi)
    modt2 = mod(theta2, 2*pi)

    ! https://gamedev.stackexchange.com/questions/4467/comparing-angles-and-working-out-the-difference

    diff = pi - abs(abs(modt1-modt2) - pi)
  end function angle_diff_2d

  function angle_diff_3d(theta, phi, theta_prime, phi_prime) result(diff)
    ! Angle between two vectors in spherical coordinates
    double precision theta, phi, theta_prime, phi_prime
    double precision alpha, diff

    ! Faster, but produces lots of NaNs (at least in Python)
    !alpha = sin(theta)*sin(theta_prime)*cos(theta-theta_prime) + cos(phi)*cos(phi_prime)


    ! Slower, but more accurate
    alpha = (sin(phi)*sin(phi_prime) &
      * (cos(theta)*cos(theta_prime) + sin(theta
         )*sin(theta_prime)) &
      + cos(phi)*cos(phi_prime))

    ! Avoid out-of-bounds errors due to rounding
    alpha = min(1.d0, alpha)
    alpha = max(-1.d0, alpha)

    diff = acos(alpha)
  end function angle_diff_3d

  subroutine vsf_from_function(iops, func)
    class(optical_properties) iops
    double precision, external :: func
    integer i
    type(angle_dim) :: angle1d

    call angle1d%set_bounds(-1.d0, 1.d0)
    call angle1d%set_num(iops%num_vsf)
    call angle1d%assign_legendre()

    iops%vsf_angles(:) = acos(angle1d%vals(:))
    do i=1, iops%num_vsf
       iops%vsf_vals(i) = func(iops%vsf_angles(i
          ))
    end do

    call iops%calc_vsf_on_grid()

    call angle1d%deinit()
  end subroutine vsf_from_function

  subroutine calc_vsf_on_grid(iops)
    class(optical_properties) iops
    double precision th, ph, thp, php
    integer p, pp
    integer nomega
    double precision norm

    nomega = iops%grid%angles%nomega

    ! Calculate cos VSF
    iops%vsf_cos = cos(iops%vsf_angles)

    ! Normalize cos VSF to 1/(2pi) on [-1, 1]
    iops%vsf_scat_coef = abs(trap_rule_uneven(
       iops%vsf_cos, iops%vsf_vals, iops%num_vsf
       ))
    iops%vsf_vals(:) = iops%vsf_vals(:) / (2*pi
       * iops%vsf_scat_coef)

    ! write(*,*) 'norm = ', iops%vsf_scat_coef
    ! write(*,*) 'now: ', trap_rule_uneven(iops%vsf_cos, iops%vsf_vals, iops%num_vsf)
    ! write(*,*) 'cos: ', iops%vsf_cos
    ! write(*,*) 'vals: ', iops%vsf_vals

    do p=1, nomega
       th = iops%grid%angles%theta_p(p)
       ph = iops%grid%angles%phi_p(p)
       do  pp=1, nomega
          thp = iops%grid%angles%theta_p(pp)
          php = iops%grid%angles%phi_p(pp)
          ! TODO: Might be better to calculate average scattering
          ! from angular cell rather than only using center
          iops%vsf(p, pp) = iops%eval_vsf(angle_diff_3d(th,ph,thp,php))
       end do

       ! Normalize each row of VSF (midpoint rule)
       norm = sum(iops%vsf(p,:) * iops%grid%angles%area_p(:))
       iops%vsf(p,:) = iops%vsf(p,:) / norm

       ! % / meter light scattered from cell pp into direction p.
       ! TODO: Could integrate VSF instead of just using value at center
       iops%vsf_integral(p, :) = iops%vsf(p, :) &
            * iops%grid%angles%area_p(:)
       !write(*,*) 'vsf_integral (beta_pp)', p, ' = ', iops%vsf_integral(p, :)
    end do

    ! Normalize VSF on unit sphere w.r.t. north pole
    !iops%vsf_scat_coef = sum(iops%vsf(1,:) * iops%grid%angles%area_p)
    !iops%vsf = iops%vsf / iops%vsf_scat_coef
    !iops%vsf_integral = iops%vsf_integral / iops%vsf_scat_coef
  end subroutine calc_vsf_on_grid

  subroutine iop_deinit(iops)
    class(optical_properties) iops
    deallocate(iops%vsf_angles)
    deallocate(iops%vsf_vals)
    deallocate(iops%vsf_cos)
    deallocate(iops%vsf)
    deallocate(iops%vsf_integral)
    deallocate(iops%abs_water)
    deallocate(iops%abs_grid)

  end subroutine iop_deinit

end module kelp_context
```

light_context.f90

```
module light_context
  use sag
  use rte_sparse_matrices
  !use hdf5
  implicit none

  type light_state
     double precision, dimension(:,:,:),
        allocatable :: irradiance
     double precision, dimension(:,:,:,:),
        allocatable :: radiance
     type(space_angle_grid) :: grid
     type(rte_mat) :: mat
   contains
     procedure :: init => light_init
     procedure :: init_grid => light_init_grid
     procedure :: calculate_radiance
     procedure :: calculate_irradiance
     procedure :: deinit => light_deinit
     !procedure :: to_hdf => light_to_hdf
  end type light_state

contains

  ! Init for use with mat
  subroutine light_init(light, mat)
    class(light_state) light
    type(rte_mat) mat
    integer nx, ny, nz, nomega

    light%mat = mat
    light%grid = mat%grid

    nx = light%grid%x%num
    ny = light%grid%y%num
    nz = light%grid%z%num
    nomega = light%grid%angles%nomega

    allocate(light%irradiance(nx, ny, nz))
    allocate(light%radiance(nx, ny, nz, nomega))
  end subroutine light_init

  ! Init for use without mat
  subroutine light_init_grid(light, grid)
    class(light_state) light
    type(space_angle_grid) grid
    integer nx, ny, nz, nomega

    light%grid = grid

    nx = light%grid%x%num
    ny = light%grid%y%num
    nz = light%grid%z%num
    nomega = light%grid%angles%nomega

    allocate(light%irradiance(nx, ny, nz))
    allocate(light%radiance(nx, ny, nz, nomega))
  end subroutine light_init_grid

  subroutine calculate_radiance(light)
    class(light_state) light
    integer i, j, k, p
    integer nx, ny, nz, nomega
    integer index

    nx = light%grid%x%num
    ny = light%grid%y%num
    nz = light%grid%z%num
    nomega = light%grid%angles%nomega

    index = 1

    ! Set initial guess from provided radiance
    ! Traverse solution vector in order
    ! so as to avoid calculating index
    do k=1, nz
       do i=1, nx
           do j=1, ny
             do p=1, nomega
                light%mat%sol(index) = light%
                   radiance(i,j,k,p)
                index = index + 1
             end do
          end do
       end do
    end do

    !call light%mat%initial_guess()

    ! Solve (MGMRES)
```

```fortran
    call light%mat%solve()

    index = 1

    ! Extract solution
    do k=1, nz
       do i=1, nx
          do j=1, ny
             do p=1, nomega
                light%radiance(i,j,k,p) = light%mat%sol(index)
                index = index + 1
             end do
          end do
       end do
    end do
  end subroutine calculate_radiance

  subroutine calculate_irradiance(light)
    class(light_state) light
    integer i, j, k
    integer nx, ny, nz
    double precision, dimension(light%grid%angles%nomega) :: tmp_rad

    nx = light%grid%x%num
    ny = light%grid%y%num
    nz = light%grid%z%num

    do i=1, nx
       do j=1, ny
          do k=1, nz
             ! Use temporary array to avoid creating one
             ! implicitly at every spatial grid point
             tmp_rad = light%radiance(i,j,k,:)
             light%irradiance(i,j,k) = &
                  light%grid%angles%integrate_points(tmp_rad)
          end do
       end do
    end do

  end subroutine calculate_irradiance

!  subroutine light_to_hdf(light, radfile, irradfile)
!    class(light_state) light
!    character(len=*) radfile
!    character(len=*) irradfile
!
```

```fortran
!     call hdf_write_radiance(radfile, light%radiance, light%grid)
!     call hdf_write_irradiance(irradfile, light%irradiance, light%grid)
!   end subroutine light_to_hdf

  subroutine light_deinit(light)
    class(light_state) light

    deallocate(light%irradiance)
    deallocate(light%radiance)
  end subroutine light_deinit
end module
```

asymptotics.f90

```fortran
module asymptotics
  use kelp_context
  !use rte_sparse_matrices
  !use light_context
  implicit none
  contains

  subroutine calculate_light_with_scattering(grid, bc, iops, radiance, num_scatters)
    type(space_angle_grid) grid
    type(boundary_condition) bc
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) :: radiance
    double precision, dimension(:,:,:,:), allocatable :: source
    integer num_scatters
    integer nx, ny, nz, nomega
    integer max_cells

    double precision, dimension(:), allocatable :: path_length, path_spacing, a_tilde, gn

    nx = grid%x%num
    ny = grid%y%num
    nz = grid%z%num
    nomega = grid%angles%nomega

    max_cells = calculate_max_cells(grid)

    allocate(path_length(max_cells+1))
    allocate(path_spacing(max_cells))
    allocate(a_tilde(max_cells))
    allocate(gn(max_cells))
    allocate(source(nx, ny, nz, nomega))

    call calculate_light_before_scattering(grid,
        bc, iops, source, radiance, path_length,
        path_spacing, a_tilde, gn)

    if (num_scatters .gt. 0) then
       call calculate_light_after_scattering(&
            grid, iops, source, radiance, &
            num_scatters, path_length,
               path_spacing, &
            a_tilde, gn)
    end if

    deallocate(path_length)
    deallocate(path_spacing)
    deallocate(a_tilde)
    deallocate(gn)
    deallocate(source)
  end subroutine calculate_light_with_scattering

  subroutine calculate_light_before_scattering(
     grid, bc, iops, source, radiance,
     path_length, path_spacing, a_tilde, gn)
    type(space_angle_grid) grid
    type(boundary_condition) bc
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) ::
       radiance, source
    double precision, dimension(:) ::
       path_length, path_spacing, a_tilde, gn
    integer i, j, k, p

    ! !$ integer omp_get_num_procs
    ! !$ integer num_threads_z, num_threads_x

    ! ! Enable nested parallelism
    ! !$ call omp_set_nested(.true.)

    ! ! Use nz procs for outer loop,
    ! ! or num_procs if num_procs < nz
    ! ! Divide the rest of the tasks as
       appropriate

    ! !$ num_threads_z = min(omp_get_num_procs()
       , grid%z%num)
    ! !$ num_threads_x = min( &
    ! !$    omp_get_num_procs()/num_threads_z, &
    ! !$    grid%x%num)

    ! !$omp parallel do default(none) private(i,
       j,k,p) &
    ! !$omp shared(grid,iops,radiance,bc, num_threads_x) &
    ! !$omp private(source,path_length, path_spacing,a_tilde,gn) &
    ! !$omp num_threads(num_threads_z) if( num_threads_z .gt. 1)
    do k=1, grid%z%num
       ! !$omp parallel do default(none) private(i,j,p) &
       ! !$omp firstprivate(k) shared(grid,iops, radiance,bc) &
       ! !$omp private(source,path_length, path_spacing,a_tilde,gn) &
       ! !$omp num_threads(num_threads_x) if( num_threads_x .gt. 1)
       do i=1, grid%x%num
          do j=1, grid%y%num
             do p=1, grid%angles%nomega/2
                ! Downwelling light
                call attenuate_light_from_surface(&
                 grid, iops, source, i, j, k, p,&
                 radiance, path_length, path_spacing,&
                 a_tilde, gn, bc)

                ! No upwelling light before scattering
                radiance(i,j,k,p+grid%angles%nomega/2) = 0.d0
             end do
          end do
       end do
       ! !$omp end parallel do
     end do
     ! !$omp end parallel do
  end subroutine calculate_light_before_scattering

  subroutine attenuate_light_from_surface(&
       grid, iops, source, i, j, k, p, radiance, &
       path_length, path_spacing, a_tilde, gn, bc)
    type(space_angle_grid) grid
    type(boundary_condition) bc
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) :: radiance, source
    double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn
    integer i, j, k, p
    integer num_cells
    double precision atten

    ! Don't need gn here, so just ignore it
    call traverse_ray(grid, iops, source, i, j, k, p, path_length, path_spacing, a_tilde, gn, num_cells)

    ! Start with surface bc and attenuate along path
    atten = sum(path_spacing(1:num_cells) * a_tilde(1:num_cells))
    ! Avoid underflow
    if(atten .lt. 100.d0) then
      radiance(i,j,k,p) = bc%bc_grid(p) * exp(-atten)
    else
      radiance(i,j,k,p) = 0.d0
    end if

  end subroutine attenuate_light_from_surface

  subroutine calculate_light_after_scattering(grid, iops, source, radiance,&
      num_scatters, path_length, path_spacing, a_tilde, gn)
    type(space_angle_grid) grid
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) :: radiance, source
    integer num_scatters
    double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn
    double precision, dimension(:,:,:,:), allocatable :: rad_scatter
    integer n
    double precision bb

    allocate(rad_scatter(grid%x%num, grid%y%num, grid%z%num, grid%angles%nomega))
    rad_scatter = radiance
    bb = iops%scat

    do n=1, num_scatters
      write(*,*) 'scatter #', n
      call scatter(grid, iops, source,
          rad_scatter, path_length, path_spacing
          , a_tilde, gn)
        radiance = radiance + bb**n * rad_scatter
    end do

    deallocate(rad_scatter)
  end subroutine
     calculate_light_after_scattering

  ! Perform one scattering event
  subroutine scatter(grid, iops, source,
     rad_scatter, path_length, path_spacing,
     a_tilde, gn)
    type(space_angle_grid) grid
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) ::
       rad_scatter, source
    double precision, dimension(:,:,:,:),
       allocatable :: scatter_integral
    double precision, dimension(:) ::
       path_length, path_spacing, a_tilde, gn
    integer nx, ny, nz, nomega

    nx = grid%x%num
    ny = grid%y%num
    nz = grid%z%num
    nomega = grid%angles%nomega

    allocate(scatter_integral(nx, ny, nz, nomega
       ))

    call calculate_source(grid, iops,
       rad_scatter, source, scatter_integral)
    call advect_light(grid, iops, source,
       rad_scatter, path_length, path_spacing,
       a_tilde, gn)

    deallocate(scatter_integral)
  end subroutine scatter

  ! Calculate source from no-scatter or previous
      scattering layer
  subroutine calculate_source(grid, iops,
     rad_scatter, source, scatter_integral)
    type(space_angle_grid) grid
    type(optical_properties) iops
    double precision, dimension(:,:,:,:) ::
       rad_scatter
    double precision, dimension(:,:,:,:) ::
       source
    double precision, dimension(:,:,:,:) ::
       scatter_integral
    type(index_list) indices
    integer nx, ny, nz, nomega
    integer i, j, k, p

    !$ integer omp_get_num_procs
    !$ integer num_threads_z, num_threads_x

    nx = grid%x%num
    ny = grid%y%num
    nz = grid%z%num
    nomega = grid%angles%nomega

    ! Enable nested parallelism
    !$ call omp_set_nested(.true.)

    ! Use nz procs for outer loop,
    ! or num_procs if num_procs < nz
    ! Divide the rest of the tasks as
       appropriate

    !$ num_threads_z = min(omp_get_num_procs(),
       grid%z%num)
    !$ num_threads_x = min( &
    !$    omp_get_num_procs()/num_threads_z, &
    !$    grid%x%num)

    !$omp parallel do default(none) private(
       indices) &
    !$omp private(i,j,k,p) shared(nx,ny,nz,
       nomega) &
    !$omp shared(iops, rad_scatter,
       scatter_integral) &
    !$omp shared(num_threads_x) &
    !$omp num_threads(num_threads_z) if(
       num_threads_z .gt. 1)
    do k=1, nz
       indices%k = k
       !$omp parallel do default(none)
          firstprivate(indices,k) &
       !$omp private(i,j,p) shared(nx,ny,nz,
          nomega) &
       !$omp shared(iops, rad_scatter,
          scatter_integral) &
       !$omp num_threads(num_threads_x) if(
          num_threads_x .gt. 1)
       do i=1, nx
         indices%i = i
         do j=1, ny
```

```
           indices%j = j
           do p=1, nomega
              indices%p = p
              call calculate_scatter_integral
                 (&
                  iops, rad_scatter,&
                  scatter_integral,&
                  indices)
           end do
        end do
     end do
     !$omp end parallel do
  end do
  !$omp end parallel do

  source(:,:,:,:) = -rad_scatter(:,:,:,:) +
     scatter_integral(:,:,:,:)

  write(*,*) 'source: ', sum(source)/size(
     source), minval(source), maxval(source)

end subroutine calculate_source

subroutine calculate_scatter_integral(iops,
   rad_scatter, scatter_integral, indices)
  type(optical_properties) iops
  double precision, dimension(:,:,:,:) ::
     rad_scatter, scatter_integral
  type(index_list) indices

  scatter_integral(indices%i,indices%j,indices
     %k,indices%p) &
       = sum(iops%vsf_integral(indices%p, :) &
       * rad_scatter(&
         indices%i,&
         indices%j,&
         indices%k,:))

end subroutine calculate_scatter_integral

subroutine advect_light(grid, iops, source,
   rad_scatter, path_length, path_spacing,
   a_tilde, gn)
  type(space_angle_grid) grid
  type(optical_properties) iops
  double precision, dimension(:,:,:,:) ::
     rad_scatter, source
  double precision, dimension(:) ::
     path_length, path_spacing, a_tilde, gn
  integer i, j, k, p

```

```
    !$ integer omp_get_num_procs
    !$ integer num_threads_z, num_threads_x

    ! Enable nested parallelism
    !$ call omp_set_nested(.true.)

    ! Use nz procs for outer loop,
    ! or num_procs if num_procs < nz
    ! Divide the rest of the tasks as
      appropriate

    !$ num_threads_z = min(omp_get_num_procs(),
      grid%z%num)
    !$ num_threads_x = min( &
    !$    omp_get_num_procs()/num_threads_z, &
    !$    grid%x%num)

    !$omp parallel do default(none) &
    !$omp private(i,j,k,p) &
    !$omp shared(rad_scatter,source,grid,iops,
      num_threads_x) &
    !$omp private(path_length,path_spacing,
      a_tilde,gn) &
    !$omp num_threads(num_threads_z) if(
      num_threads_z .gt. 1)
    do k=1, grid%z%num
       !$omp parallel do default(none) &
       !$omp firstprivate(k) private(i,j,p) &
       !$omp shared(rad_scatter,source,grid,iops
         ) &
       !$omp private(path_length,path_spacing,
         a_tilde,gn) &
       !$omp num_threads(num_threads_x) if(
         num_threads_x .gt. 1)
       do i=1, grid%x%num
         do j=1, grid%y%num
             do p=1, grid%angles%nomega
                call integrate_ray(grid, iops,
                  source,&
                     rad_scatter, path_length,
                       path_spacing,&
                     a_tilde, gn, i, j, k, p)
            end do
          end do
       end do
       !$omp end parallel do
    end do
    !$omp end parallel do
  end subroutine advect_light

  ! New algorithm, double integral over
     piecewise-constant 1d funcs
  subroutine integrate_ray(grid, iops, source,
     rad_scatter, path_length, path_spacing,
     a_tilde, gn, i, j, k, p)
    type(space_angle_grid) :: grid
    type(optical_properties) :: iops
    double precision, dimension(:,:,:,:) ::
       source
    double precision, dimension(:,:,:,:) ::
       rad_scatter
    integer :: i, j, k, p
    ! The following are only passed to avoid
       unnecessary allocation
    double precision, dimension(:) ::
       path_length, path_spacing, a_tilde, gn
    integer num_cells

    call traverse_ray(grid, iops, source, i, j,
       k, p, path_length, path_spacing, a_tilde,
        gn, num_cells)
    rad_scatter(i,j,k,p) =
       calculate_ray_integral(num_cells,
       path_length, path_spacing, a_tilde, gn)
  end subroutine integrate_ray

  function calculate_ray_integral(num_cells, s,
     ds, a_tilde, gn) result(integral)
    ! Double integral which accumulates all
       scattered light along the path
    ! via an angular integral and attenuates it
       by integrating along the path
    integer :: num_cells
    double precision, dimension(num_cells) :: ds
       , a_tilde, gn
    double precision, dimension(num_cells+1) ::
       s
    double precision :: integral
    double precision bi, di
    integer i, j

    integral = 0
    do i=1, num_cells
       bi = -a_tilde(i)*s(i+1)
       do j=i+1, num_cells
          bi = bi - a_tilde(j)*ds(j)
       end do

       ! WARNING: This will overflow if a_tilde
          is too large.
       if(a_tilde(i) .eq. 0) then
```

```
          di = ds(i)
       else
          di = (exp(a_tilde(i)*s(i+1))-exp(a_tilde(i)*s(i)))/a_tilde(i)
       end if

       integral = integral + gn(i)*di * exp(bi)
    end do

  end function calculate_ray_integral

  ! Calculate maximum number of cells a path through the grid could take
  ! This is a loose upper bound
  function calculate_max_cells(grid) result(max_cells)
    type(space_angle_grid) :: grid
    integer :: max_cells
    double precision dx, dy, zrange, phi_middle

    ! Angle that will have the longest ray
    phi_middle = grid%angles%phi(grid%angles%nphi/2)
    dx = grid%x%spacing(1)
    dy = grid%y%spacing(1)
    zrange = grid%z%maxval - grid%z%minval

    max_cells = grid%z%num + ceiling((1/dx+1/dy)*zrange*tan(phi_middle))
  end function calculate_max_cells

  ! Traverse from surface or bottom to point (xi, yj, zk)
  ! in the direction omega_p, extracting path lengths (ds) and
  ! function values (f) along the way,
  ! as well as number of cells traversed (n).
  subroutine traverse_ray(grid, iops, source, i, j, k, p, s_array, ds, a_tilde, gn, num_cells)
    type(space_angle_grid) :: grid
    type(optical_properties) :: iops
    double precision, dimension(:,:,:,:) :: source
    integer :: i, j, k, p
    double precision, dimension(:) :: s_array, ds, a_tilde, gn
    integer :: num_cells

    integer t
    double precision p0x, p0y, p0z
```

```fortran
    double precision p1x, p1y, p1z
    double precision z0
    double precision s_tilde, s
    integer dir_x, dir_y, dir_z
    integer shift_x, shift_y, shift_z
    integer cell_x, cell_y, cell_z
    integer edge_x, edge_y, edge_z
    integer first_x, last_x, first_y, last_y,
       last_z
    double precision s_next_x, s_next_y,
       s_next_z, s_next
    double precision x_factor, y_factor,
       z_factor
    double precision ds_x, ds_y
    double precision, dimension(grid%z%num) ::
       ds_z
    double precision smx, smy

    ! Divide by these numbers to get path
       separation
    ! from separation in individual dimensions
    x_factor = grid%angles%sin_phi_p(p) * grid%
       angles%cos_theta_p(p)
    y_factor = grid%angles%sin_phi_p(p) * grid%
       angles%sin_theta_p(p)
    z_factor = grid%angles%cos_phi_p(p)

    ! Destination point
    p1x = grid%x%vals(i)
    p1y = grid%y%vals(j)
    p1z = grid%z%vals(k)

    !write(*,*) 'START PATH.'
    !write(*,*) 'ijk = ', i, j, k

    ! Direction
    if(p .le. grid%angles%nomega/2) then
       ! Downwelling light originates from
          surface
       z0 = grid%z%minval
       dir_z = 1
    else
       ! Upwelling light originates from bottom
       z0 = grid%z%maxval
       dir_z = -1
    end if

    ! Total path length from origin to
       destination
    ! (sign is correct for upwelling and
       downwelling)
    s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p)

    ! Path spacings between edge intersections in each dimension
    ! Set to 2*s_tilde if infinite in this dimension so that it's unreachable
    ! Assume x & y spacings are uniform,
    ! so it's okay to just use the first value.
    if(x_factor .eq. 0) then
       ds_x = 2*s_tilde
    else
       ds_x = abs(grid%x%spacing(1)/x_factor)
    end if
    if(y_factor .eq. 0) then
       ds_y = 2*s_tilde
    else
       ds_y = abs(grid%y%spacing(1)/y_factor)
    end if

    ! This one is an array because z spacing can vary
    ! z_factor should never be 0, because the ray will never
    ! reach the surface or bottom.
    ds_z(1:grid%z%num) = dir_z * grid%z%spacing(1:grid%z%num)/z_factor

    ! Origin point
    p0x = p1x - s_tilde * x_factor
    p0y = p1y - s_tilde * y_factor
    p0z = p1z - s_tilde * z_factor

    ! Direction of ray in each dimension. 1 => increasing. -1 => decreasing.
    dir_x = int(sgn(p1x-p0x))
    dir_y = int(sgn(p1y-p0y))

    ! Shifts
    ! Conversion from cell_inds to edge_inds
    ! merge is fortran's ternary operator
    shift_x = merge(1,0,dir_x>0)
    shift_y = merge(1,0,dir_y>0)
    shift_z = merge(1,0,dir_z>0)

    ! Indices for cell containing origin point
    cell_x = floor((p0x-grid%x%minval)/grid%x%spacing(1)) + 1
    cell_y = floor((p0y-grid%y%minval)/grid%y%spacing(1)) + 1
    ! x and y may be in periodic image, so shift back.
    cell_x = mod1(cell_x, grid%x%num)
    cell_y = mod1(cell_y, grid%y%num)

    ! z starts at top or bottom depending on
       direction.
    if(dir_z > 0) then
       cell_z = 1
    else
       cell_z = grid%z%num
    end if

    ! Edge indices preceeding starting cells
    edge_x = mod1(cell_x + shift_x, grid%x%num)
    edge_y = mod1(cell_y + shift_y, grid%y%num)
    edge_z = mod1(cell_z + shift_z, grid%z%num)

    ! First and last cells given direction
    if(dir_x .gt. 0) then
       first_x = 1
       last_x = grid%x%num
    else
       first_x = grid%x%num
       last_x = 1
    end if
    if(dir_y .gt. 0) then
       first_y = 1
       last_y = grid%y%num
    else
       first_y = grid%y%num
       last_y = 1
    end if
    if(dir_z .gt. 0) then
       last_z = grid%z%num
    else
       last_z = 1
    end if

    ! Calculate periodic images
    smx = shift_mod(p0x, grid%x%minval, grid%x%
       maxval)
    smy = shift_mod(p0y, grid%y%minval, grid%y%
       maxval)

    ! Path length to next edge plane in each
       dimension
    if(abs(x_factor) .lt. 1.d-10) then
       ! Will never cross, so set above total
          path length
       s_next_x = 2*s_tilde
    else if(cell_x .eq. last_x) then
```

```fortran
       ! If starts out at last cell, then compare to periodic image
       s_next_x = (grid%x%edges(first_x) + dir_x * (grid%x%maxval - grid%x%minval)&
           - smx) / x_factor
    else
       ! Otherwise, just compare to next cell
       s_next_x = (grid%x%edges(edge_x) - smx) / x_factor
    end if

    ! Path length to next edge plane in each dimension
    if(abs(y_factor) .lt. 1.d-10) then
       ! Will never cross, so set above total path length
       s_next_y = 2*s_tilde
    else if(cell_y .eq. last_y) then
       ! If starts out at last cell, then compare to periodic image
       s_next_y = (grid%y%edges(first_y) + dir_y * (grid%y%maxval - grid%y%minval)&
           - smy) / y_factor
    else
       ! Otherwise, just compare to next cell
       s_next_y = (grid%y%edges(edge_y) - smy) / y_factor
    end if

    s_next_z = ds_z(cell_z)

    ! Initialize path
    s = 0.d0
    s_array(1) = 0.d0

    ! Start with t=0 so that we can increment before storing,
    ! so that t will be the number of grid cells at the end of the loop.
    t=0

    ! s is the beginning of the current cell,
    ! s_next is the end of the current cell.
    do while (s .lt. s_tilde)
       ! Move cell counter
       t = t + 1

       ! Extract function values
       a_tilde(t) = iops%abs_grid(cell_x, cell_y, cell_z)
       gn(t) = source(cell_x, cell_y, cell_z, p)

```

```fortran
        !write(*,*) ''
        !write(*,*) 's_next_x = ', s_next_x
        !write(*,*) 's_next_y = ', s_next_y
        !write(*,*) 's_next_z = ', s_next_z
        !write(*,*) 'theta, phi =', grid%angles%theta_p(p)*180.d0/pi, grid%angles%phi_p(p)*180.d0/pi
        !write(*,*) 's = ', s, '/', s_tilde
        !write(*,*) 'cell_z =', cell_z, '/', grid%z%num
        !write(*,*) 's_next_z =', s_next_z
        !write(*,*) 'last_z =', last_z
        !write(*,*) 'new'

        ! Move to next cell in path
        if(s_next_x .le. min(s_next_y, s_next_z)) then
           ! x edge is closest
           s_next = s_next_x

           ! Increment indices (periodic)
           cell_x = mod1(cell_x + dir_x, grid%x%num)
           edge_x = mod1(edge_x + dir_x, grid%x%num)

           ! x intersection after the one at s=s_next
           s_next_x = s_next + ds_x

        else if (s_next_y .le. min(s_next_x, s_next_z)) then
           ! y edge is closest
           s_next = s_next_y

           ! Increment indices (periodic)
           cell_y = mod1(cell_y + dir_y, grid%y%num)
           edge_y = mod1(edge_y + dir_y, grid%y%num)

           ! y intersection after the one at s=s_next
           s_next_y = s_next + ds_y

        else if(s_next_z .le. min(s_next_x, s_next_y)) then
           ! z edge is closest
           s_next = s_next_z

           ! Increment indices
```

```
            cell_z = cell_z + dir_z
            edge_z = edge_z + dir_z

            !write(*,*) 'z edge, s_next =', s_next

            ! z intersection after the one at s=
               s_next
            if(cell_z .lt. last_z) then
               ! Only look ahead if we aren't at
                  the end
               s_next_z = s_next + ds_z(cell_z)
            else
               ! Otherwise, no need to continue.
               ! this is our final destination.
            !   exit
               s_next_z = 2*s_tilde
               !write(*,*) 'end. s_next_z =',
                  s_next_z
            end if

         end if

         ! Cut off early if this is the end
         ! This will be the last cell traversed if
            s_next >= s_tilde
         s_next = min(s_tilde, s_next)

         ! Store path length
         s_array(t+1) = s_next
         ! Extract path length from same cell as
           function vals
         ds(t) = s_next - s

         ! Update path length
         s = s_next
      end do

      ! Return number of cells traversed
      num_cells = t

   end subroutine traverse_ray
end module asymptotics
```

light_interface.f90

```
module light_interface_module
  use rte3d
  use kelp3d
  use asymptotics
  implicit none

contains
  subroutine full_light_calculations( &
    ! OPTICAL PROPERTIES
    absorptance_kelp, & ! NOT THE SAME AS
       ABSORPTION COEFFICIENT
    abs_water, &
    scat, &
    num_vsf, &
    vsf_file, &
    ! SUNLIGHT
    solar_zenith, &
    solar_azimuthal, &
    surface_irrad, &
    ! KELP &
    num_si, &
    si_area, &
    si_ind, &
    frond_thickness, &
    frond_aspect_ratio, &
    frond_shape_ratio, &
    ! WATER CURRENT
    current_speeds, &
    current_angles, &
    ! SPACING
    rope_spacing, &
    depth_spacing, &
    ! SOLVER PARAMETERS
    nx, &
    ny, &
    nz, &
    ntheta, &
    nphi, &
    num_scatters, &
    ! FINAL RESULTS
    perceived_irrad, &
    avg_irrad)

    implicit none

    ! OPTICAL PROPERTIES
    integer, intent(in) :: nx, ny, nz, ntheta,
       nphi
    ! Absorption and scattering coefficients
    double precision, intent(in) ::
       absorptance_kelp, scat
    double precision, dimension(nz), intent(in)
       :: abs_water
    ! Volume scattering function
    integer, intent(in) :: num_vsf
    character(len=*) :: vsf_file
    !double precision, dimension(num_vsf),
       intent(int) :: vsf_angles
    !double precision, dimension(num_vsf),
       intent(int) :: vsf_vals

    ! SUNLIGHT
    double precision, intent(in) :: solar_zenith
    double precision, intent(in) :: solar_azimuthal
    double precision, intent(in) :: surface_irrad

    ! KELP
    ! Number of Superindividuals in each depth level
    integer, intent(in) :: num_si
    ! si_area(i,j) = area of superindividual j at depth i
    double precision, dimension(nz, num_si), intent(in) :: si_area
    ! si_ind(i,j) = number of inidividuals represented by superindividual j at depth i
    double precision, dimension(nz, num_si), intent(in) :: si_ind
    ! Thickness of each frond
    double precision, intent(in) :: frond_thickness
    ! Ratio of length to width (0,infty)
    double precision, intent(in) :: frond_aspect_ratio
    ! Rescaled position of greatest width (0=base, 1=tip)
    double precision, intent(in) :: frond_shape_ratio

    ! WATER CURRENT
    double precision, dimension(nz), intent(in) :: current_speeds
    double precision, dimension(nz), intent(in) :: current_angles

    ! SPACING
    double precision, intent(in) :: rope_spacing
    double precision, dimension(nz), intent(in) :: depth_spacing
    ! SOLVER PARAMETERS
    integer, intent(in) :: num_scatters

    ! FINAL RESULT
    real, dimension(nz), intent(out) :: avg_irrad, perceived_irrad

    !-------------!

```

```fortran
    double precision xmin, xmax, ymin, ymax, zmin, zmax
    character(len=5), parameter :: fmtstr = 'E13.4'
    !double precision, dimension(num_vsf) :: vsf_angles, vsf_vals
    double precision max_rad, decay
    integer quadrature_degree

    type(space_angle_grid) grid
    type(optical_properties) iops
    type(light_state) light
    type(rope_state) rope
    type(frond_shape) frond
    type(boundary_condition) bc

    double precision, dimension(:), allocatable :: pop_length_means, pop_length_stds
    ! Number of fronds in each depth layer
    double precision, dimension(:), allocatable :: num_fronds
    double precision, dimension(:,:,:), allocatable :: p_kelp

    write(*,*) 'Light calculation'

    allocate(pop_length_means(nz))
    allocate(pop_length_stds(nz))
    allocate(num_fronds(nz))
    allocate(p_kelp(nx, ny, nz))

    xmin = -rope_spacing/2
    xmax = rope_spacing/2

    ymin = -rope_spacing/2
    ymax = rope_spacing/2

    zmin = 0.d0
    zmax = sum(depth_spacing)

    write(*,*) 'Grid'
    call grid%set_bounds(xmin, xmax, ymin, ymax, zmin, zmax)
    call grid%set_num(nx, ny, nz, ntheta, nphi)
    call grid%init()
    !call grid%set_uniform_spacing_from_num()
    call grid%z%set_spacing_array(depth_spacing)

    call rope%init(grid)

    write(*,*) 'Rope'
    ! Calculate kelp distribution
    call calculate_length_dist_from_superinds( &
    nz, &
    num_si, &
    si_area, &
    si_ind, &
    frond_aspect_ratio, &
    num_fronds, &
    pop_length_means, &
    pop_length_stds)

    rope%frond_lengths = pop_length_means
    rope%frond_stds = pop_length_stds
    rope%num_fronds = num_fronds
    rope%water_speeds = current_speeds
    rope%water_angles = current_angles

    write(*,*) 'frond_lengths  =', rope%
       frond_lengths
    write(*,*) 'frond_stds  =', rope%frond_stds
    write(*,*) 'num_fronds  =', rope%num_fronds
    write(*,*) 'water_speeds  =', rope%
       water_speeds
    write(*,*) 'water_angles  =', rope%
       water_angles

    write(*,*) 'Frond'
    ! INIT FROND
    call frond%set_shape(frond_shape_ratio,
       frond_aspect_ratio, frond_thickness)
    ! CALCULATE KELP
    quadrature_degree = 5
    call calculate_kelp_on_grid(grid, p_kelp,
       frond, rope, quadrature_degree)
    ! INIT IOPS
    iops%num_vsf = num_vsf
    call iops%init(grid)
    write(*,*) 'IOPs'
    iops%abs_kelp = absorptance_kelp /
       frond_thickness
    iops%abs_water = abs_water
    iops%scat = scat

    !write(*,*) 'iop init'
    !iops%vsf_angles = vsf_angles
    !iops%vsf_vals = vsf_vals
    call iops%load_vsf(vsf_file, fmtstr)

    ! load_vsf already calls calc_vsf_on_grid
    !call iops%calc_vsf_on_grid()
    call iops%calculate_coef_grids(p_kelp)

    !write(*,*) 'BC'
    max_rad = 1.d0 ! Doesn't matter because we'll rescale
    decay = 1.d0 ! Does matter, but maybe not much. Determines drop-off from angle
    call bc%init(grid, solar_zenith, solar_azimuthal, decay, max_rad)
    ! Rescale surface radiance to match surface irradiance
    bc%bc_grid = bc%bc_grid * surface_irrad / grid%angles%integrate_points(bc%bc_grid)

    write(*,*) 'bc'
    write(*,*) bc%bc_grid

    ! write(*,*) 'bc'
    ! do i=1, grid%y%num
    !     write(*,'(10F15.3)') bc%bc_grid(i,:)
    ! end do

    call light%init_grid(grid)

    write(*,*) 'Scatter'
    call calculate_light_with_scattering(grid, bc, iops, light%radiance, num_scatters)

    write(*,*) 'Irrad'
    call light%calculate_irradiance()

    ! Calculate output variables
    call calculate_average_irradiance(grid, light, avg_irrad)
    call calculate_perceived_irrad(grid, p_kelp, &
        perceived_irrad, light%irradiance)

    !write(*,*) 'vsf_angles = ', iops%vsf_angles
    !write(*,*) 'vsf_vals = ', iops%vsf_vals
    !write(*,*) 'vsf norm  = ', grid%integrate_angle_2d(iops%vsf(1,1,:,:))

    ! write(*,*) 'abs_water = ', abs_water
    ! write(*,*) 'scat_water = ', scat_water
    write(*,*) 'kelp '
    write(*,*) p_kelp(:,:,:)
    write(*,*) 'ft =', frond%ft

    write(*,*) 'irrad'
```

```fortran
    write(*,*) light%irradiance

    write(*,*) 'avg_irrad = ', avg_irrad
    write(*,*) 'perceived_irrad = ', perceived_irrad

    write(*,*) 'deinit'
    call bc%deinit()
    !write(*,*) 'a'
    call iops%deinit()
    !write(*,*) 'b'
    call light%deinit()
    !write(*,*) 'c'
    call rope%deinit()
    !write(*,*) 'd'
    call grid%deinit()
    !write(*,*) 'e'

    deallocate(pop_length_means)
    deallocate(pop_length_stds)
    deallocate(num_fronds)
    deallocate(p_kelp)

    !write(*,*) 'done'
  end subroutine full_light_calculations

  subroutine calculate_length_dist_from_superinds( &
    nz, &
    num_si, &
    si_area, &
    si_ind, &
    frond_aspect_ratio, &
    num_fronds, &
    pop_length_means, &
    pop_length_stds)

    implicit none

    ! Number of depth levels
    integer, intent(in) :: nz
    ! Number of Superindividuals in each depth level
    integer, intent(in) :: num_si
    ! si_area(i,j) = area of superindividual j at depth i
    double precision, dimension(nz, num_si), intent(in) :: si_area
    ! si_area(i,j) = number of inidividuals represented by superindividual j at depth i
    double precision, dimension(nz, num_si),
       intent(in) :: si_ind
    double precision, intent(in) ::
       frond_aspect_ratio

    double precision, dimension(nz), intent(out)
        :: num_fronds
    ! Population mean area at each depth level
    double precision, dimension(nz), intent(out)
        :: pop_length_means
    ! Population area standard deviation at each
        depth level
    double precision, dimension(nz), intent(out)
        :: pop_length_stds

    !---------------!

    integer i, k
    ! Numerators for mean and std
    double precision mean_num, std_num
    ! Convert area to length
    double precision, dimension(num_si) ::
       si_length

    do k=1, nz
       mean_num = 0.d0
       std_num = 0.d0
       num_fronds(k) = 0

       do i=1, num_si
          si_length(i) = sqrt(2.d0*
             frond_aspect_ratio*si_area(k,i))
          mean_num = mean_num + si_length(i)
          num_fronds(k) = num_fronds(k) + si_ind
             (k,i)
       end do

       pop_length_means(k) = mean_num /
          num_fronds(k)

       do i=1, num_si
          std_num = std_num + (si_length(i) -
             pop_length_means(k))**2
       end do

       pop_length_stds(k) = std_num / (
          num_fronds(k) - 1)

    end do

  end subroutine
     calculate_length_dist_from_superinds
```

```fortran

  subroutine calculate_average_irradiance(grid,
     light, avg_irrad)
    type(space_angle_grid) grid
    type(light_state) light
    real, dimension(:) :: avg_irrad
    integer k, nx, ny, nz

    nx = grid%x%num
    ny = grid%y%num
    nz = grid%z%num

    do k=1, nz
       avg_irrad(k) = real(sum(light%irradiance
          (:,:,k)) / nx / ny)
    end do
  end subroutine calculate_average_irradiance

  subroutine calculate_perceived_irrad(grid,
     p_kelp, &
       perceived_irrad, irradiance)
    type(space_angle_grid) grid
    double precision, dimension(:,:,:) :: p_kelp
    real, dimension(:) :: perceived_irrad
    double precision, dimension(:,:,:) ::
       irradiance

    integer k

    ! Calculate the average irradiance
       experienced over the frond.
    ! Has same units as irradiance.
    do k=1, grid%z%num
       perceived_irrad(k) = real( &
            sum(p_kelp(:,:,k)*irradiance(:,:,k))
                &
            / sum(p_kelp(:,:,k)))
    end do

  end subroutine calculate_perceived_irrad

end module light_interface_module
```