©2018

OLIVER GRAHAM EVANS

ALL RIGHTS RESERVED

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

Approved:

Advisor
Dr. Kevin Kreider

Co-Advisor
Dr. Curtis Clemons

Faculty Reader
Dr. Gerald Young

Department Chair
Dr. Kevin Kreider

Accepted:

Dean of the College
Dr. John Green

Dean of the Graduate School
Dr. Chand Midha

Date

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. Future generations face major challenges regarding food, energy, and water security while addressing major issues associated with global climate change. 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, for increasing use of fertilizers that impair water quality, and for low return on energy investments for 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 biofuel production.

Nitrogen leakage into water bodies is a significant ecological problem, and is especially relevant near large conventional agriculture facilities due to run-off from nitrogen-based fertilizers, as well as near wastewater treatment plants. Waste water treatment plants (WWTPs) in particular are facing increasingly stringent regula-

tion of nutrients in their effluent discharges from the US Environmental Protection Agency (USEPA) and state regulatory agencies. Nutrient management at WWTPs requires significant infrastructure, operations, and maintenance investments for tertiary treatment processes. Many treatment works are constrained financially or by space limitations in their ability to expand their treatment works. As an alternative to conventional nutrient remediation techniques, the cultivation of the macroalgae *Saccharina latissima* (sugar kelp) within the nutrient plume of WWTP ocean outfalls has been proposed. The purpose of such an undertaking would be twofold: to prevent eutrophication of the surrounding ecosystem by sequestering nutrients, and to provide supplemental nutrients that benefit macroalgae cultivation.

Large scale macroalgae cultivation has long existed in Eastern Asia due to the popularity of seaweed in Asian cuisine, and low labor costs that facilitate its manual seeding and harvest. More recently, less labor-intense and more industrialized kelp aquaculture has been developing in Scandinavia and in the Northeastern United States and Canada. For example, the MACROSEA project is a four year international research collaboration led by SINTEF, an independent research organization in Norway, and 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.

Recently, a growth model[3] for S. latissima has been produced and integrated into the SINMOD[18] hydrodynamic and ecosystem model of SINTEF. 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. This thesis contributes 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. A finite difference solution to the radiative transfer equation is developed, followed by asymptotic approximations that prove to be sufficiently accurate for clear water, and significantly less computationally intensive. A detailed description of the numerical solution of this model is presented, accompanied by source code for a FORTRAN implementation of the solution. This model can be used independently, or in conjunction with a kelp growth model to determine the amount of light available for photosynthesis at a single time step.

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

Figure 1.1: *Saccharina latissima* being harvested

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 that 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]. The author takes into account respiration, frond decay, and sub-canopy light attenuation due to self-shading. Light attenuation is represented with a simple exponential model, and self-shading appears as an added term in the decay coefficient. The author does not consider radial or 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[4]. 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. In the same year, Nyman et. al. published a similar model alongside a Markov chain model, and compared the results with experimental data collected in New Zealand[11].

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[6, 19]. 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, of 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. The radiative transfer equation has been used primarily in stellar astrophysics; it's application to marine biology is fairly recent[9]. 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 other anomaly, and can therefore be safely ignored.

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, probabilistic model is developed to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. Chapter 3 begins with a survey of fundamental radiometric quantities and optical properties of matter. The spatial kelp distribution from Chapter 2 is used to determine optical properties of the combined water-kelp medium, and the radiative transfer equation, an integro-partial differential equation which describes the the light field as a function of position and angle, is discussed. 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 approx-

imation are described. Next, in Chapter 5, the availability of necessary parameters in the literature is discussed. For those which are not readily available, give rough estimates are given or describe experimental methods for their determination are described. Then, in Chapter 6, necessary grid resolution for adequate accuracy in the full finite difference solution is determined. The finite difference solution is compared to the asymptotic approximation for a few sets of optical properties. Further, we showcase the effect of varying a few key parameters on the light field predicted by the asymptotic approximation. Afterwards, the light model developed here is used in a numerical simulation of kelp growth, and the predicted light field and biomass production are compared to those predicted by a simpler 1D exponential decay light model. Finally, Chapter 7 concludes the thesis 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

The life of cultivated macroalgae generally begins in the laboratory, where microscopic kelp spores are inoculated onto a thread in a small laboratory pool. This thread is wrapped around a large rope, which is placed in the ocean and generally suspended by buoys in one of two configurations: horizontal or vertical. We consider only the case of a rigid vertical rope which does not sway in the current. The mature *Saccharina latissima* plant consists of a single frond (leaf), a stipe (stem) and a holdfast (root structure). For the sake of this model, only the kelp frond is considered, and its base is attached directly to the rope. The "gentle undulation approximation" is employed, whereby it is assumed that fronds are perfectly horizontal. While on at any point in time they may point up or down due to water

current and gravity, we consider the horizontal state to be an average configuration. This simplifaction allows for the three-dimensionally distributed population of kelp fronds to be considered a collection of independent populations in two-dimensional depth layers.

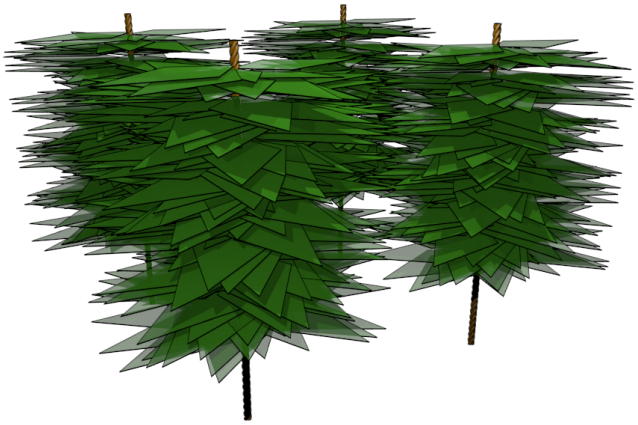

Figure 2.1: Rendering of four nearby vertical kelp ropes

2.2 Coordinate System

Consider the rectangular domain

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

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

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

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

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

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.

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

Then, calculating derivatives from (2.1) yields

$$\frac{\partial f}{\partial r} = \frac{\partial f}{\partial x}\sin\phi\cos\theta + \frac{\partial f}{\partial y}\sin\phi\sin\theta + \frac{\partial f}{\partial z}\cos\phi. \tag{2.2}$$

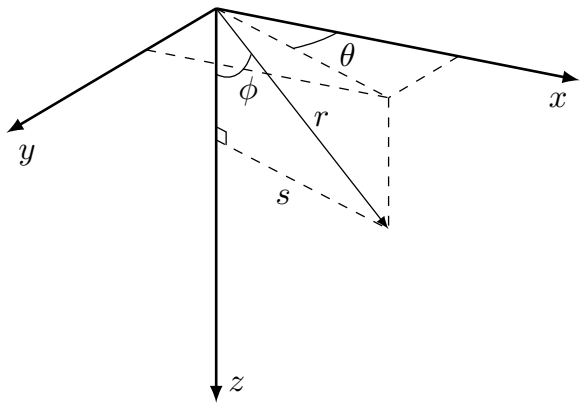

Figure 2.2: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ

2.3 Population Distributions

In order to construct a spatial distribution of kelp fronds, a simple kite-shaped geometry is introduced, and frond lengths and azimuthal orientations are assumed to be distributed predictably. It is assumed that fronds extend perfectly horizontally

2.3.1 Frond Shape

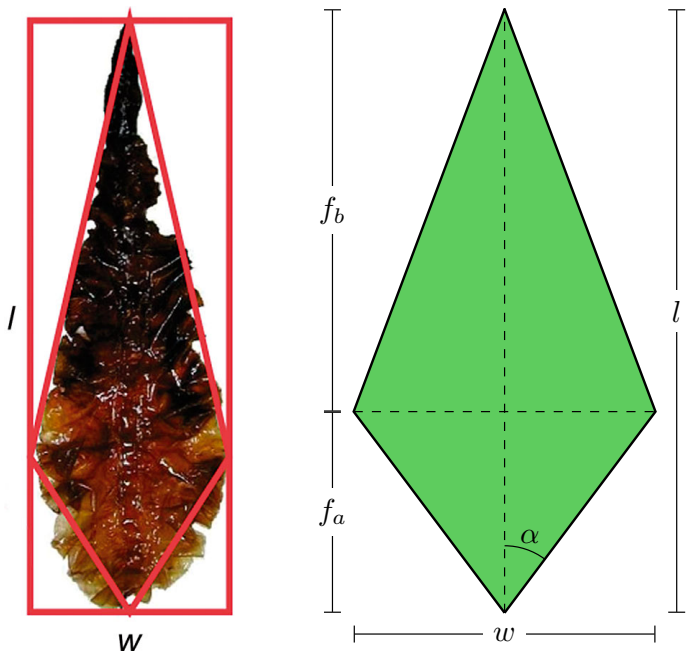

Figure 2.3: Simplified kite-shaped frond

The frond is assumed to be kite-shaped with length l from base to tip, and width w from left to right. In Figure 2.3, the base is shown at the bottom and the tip is shown at the top. The proximal length is the shortest distance from the base to the diagonal connecting the left and right corners, and is notated as f_a. Likewise, the distal length is the shortest distance from that diagonal to the tip, notated f_b.

We have

$$f_a + f_b = l$$

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

$$f_r = \frac{l}{w}$$
$$f_s = \frac{f_a}{f_b}$$

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.

$$w = \frac{l}{f_r}$$
$$f_a = \frac{l f_s}{1 + f_s}$$
$$f_b = \frac{l}{1 + f_s}$$

The angle α, half of the angle at the base corner, is also noteworthy. Using the above equations,

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

The area of the frond is given by

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

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

$$l = \sqrt{2Af_r}. \tag{2.3}$$

2.3.2 Length and Angle Distributions

Frond lengths are assumed to be normally distributed with mean μ_l and standard deviation σ_l. That is, the frond length distribution has the probability density function (PDF)

$$P_l(l) = \frac{1}{\sqrt{2\pi\sigma_l^2}} \exp\left(\frac{(l-\mu_l)^2}{2\sigma_l^2}\right).$$

It is further assumed that 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. 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, depending on the stiffness of the frond. Assuming a linear relationship between the current velocity and the steepness of the angular distribution, define the frond bending coefficient η, with units $\mathrm{s\,m^{-1}}$. Then, in use $\mu = \theta_w$ and $\kappa = \eta v_w$ as the von Mises distribution parameters. Note that θ_w and v_w vary over depth, while η is assumed constant for the population. Then, the PDF for the von Mises frond angle distribution is

$$P_{\theta_f}(\theta_f) = \frac{\exp\left(\eta v_w \cos(\theta_f - \theta_w)\right)}{2\pi I_0(\eta v_w)},$$

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, so

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

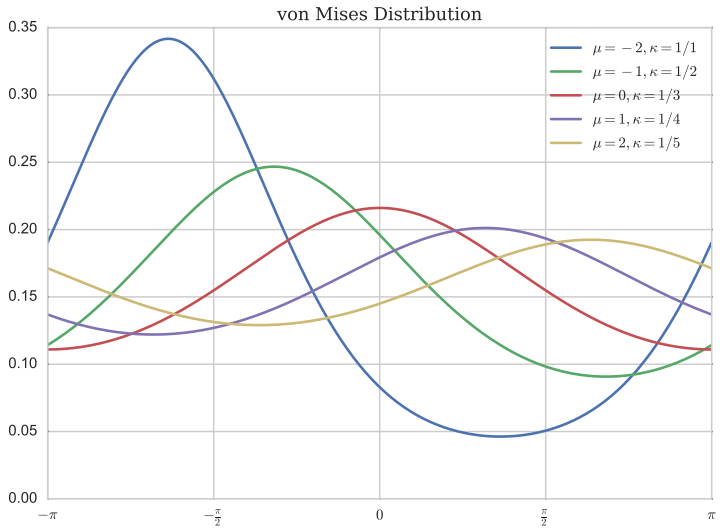

Figure 2.4: von Mises distribution for a variety of parameters

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.

Given independent events A and B,

$$P(A \cap B) = P(A)P(B)$$

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

$$P_{2D}(\theta_f, l) = P_{\theta_f}(\theta_f) \cdot P_l(l)$$

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, (2.3.3) would no longer hold, and it would be necessary to use the more general relation

$$P(A \cap B) = P(A)P(B|A) = P(B)P(B|A),$$

which is currently not taken into consideration in this model.

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 coordinates and then convert to

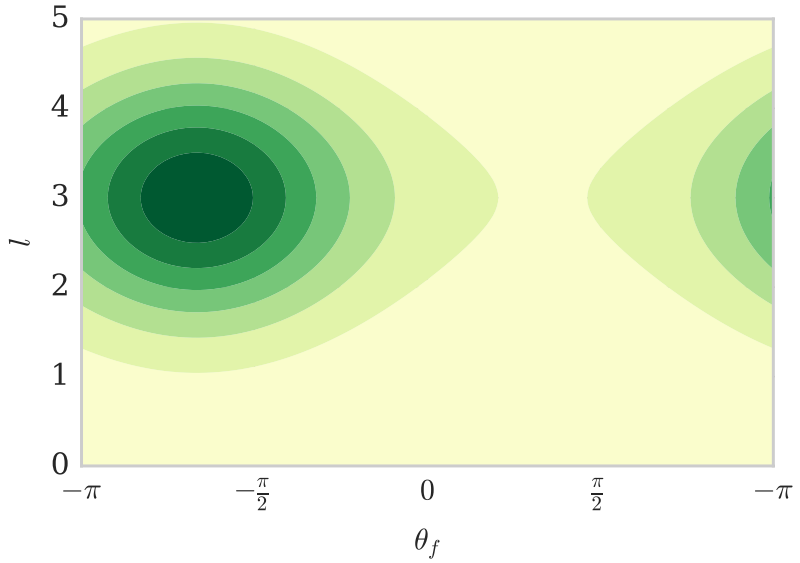

Figure 2.5: 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$

polar coordinates. Of primary interest are the edges connected to the frond tip. For convenience, 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 $(0, 0)$. Denote the Cartesian analogue of this coordinate system as (x', y') which is related to (θ', s) by

$$x' = s\cos\theta'$$

$$y' = s\sin\theta'$$

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

$$\theta' = \text{atan2}(y, x).$$

2.4.2 Functional Description of Frond Edge

With this coordinate system established, the outer two edges of the frond can be described in Cartesian coordinates as a piecewise linear function connecting the left

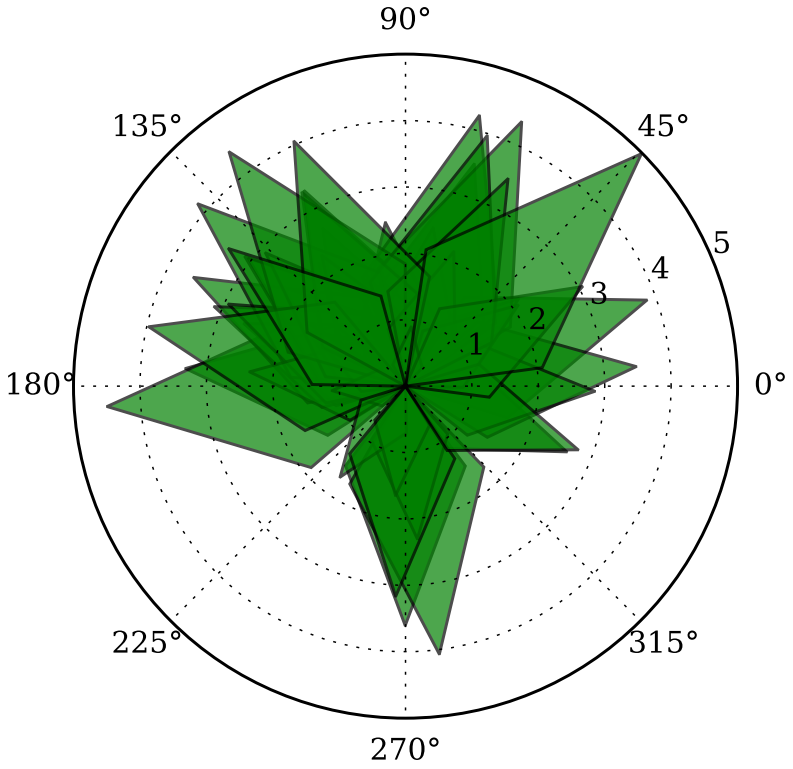

Figure 2.6: A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$.

corner: $(-w/2, f_a)$, the tip: $(0, l)$, and the right corner: $(w/2, f_a)$. This function has the form

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

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

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

where

$$S(\theta') = \text{sign}(\theta' - \pi/2).$$

Then, using the relationships in Section 2.3.1, the above equation can be rewritten in terms of the frond ratios f_s and f_r as

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

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

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

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

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

2.4.3 Conditions for Occupancy

We now formulate the conditions under which a kite shape frond occupies a point in the sense that the point lies within its interior. Combining these conditions with the size and orientation distributions from 2.3.2 allows a spatial distribution of the kelp fronds to be calculated.

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

$$|\theta_f - \theta| < \alpha, s < s_f(\theta).$$

Equivalently, the opposite perspective can be taken. Letting the point (θ, s) be fixed, a frond occupies the point if

$$\theta - \alpha < \theta_f < \theta + \alpha, \tag{2.4}$$

$$l > l_{min}(\theta, s), \tag{2.5}$$

where

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

Then, considering the point to be fixed, (2.4) and (2.5) 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.

2.4.4 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).

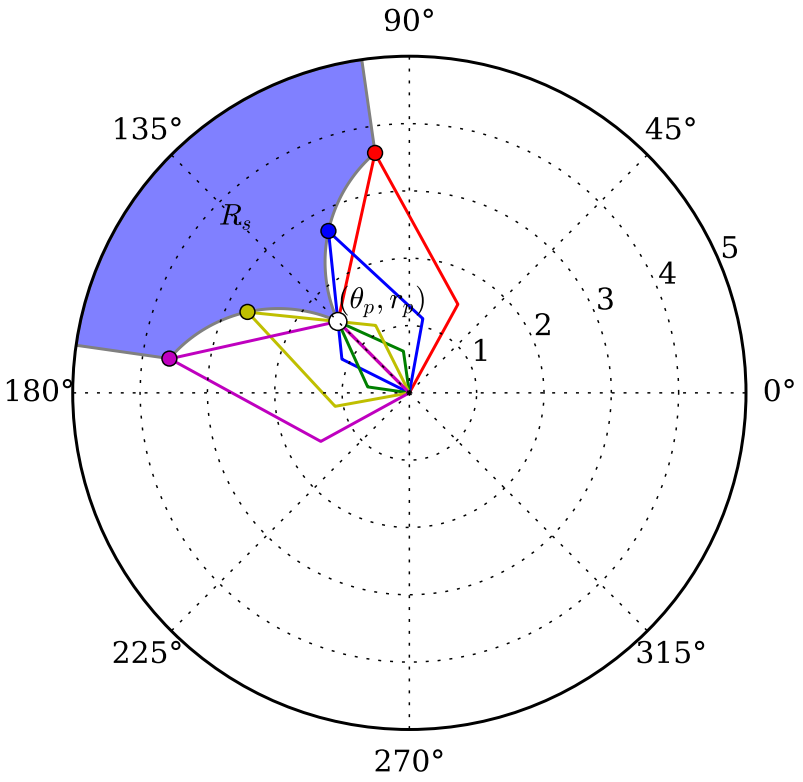

Figure 2.7: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$

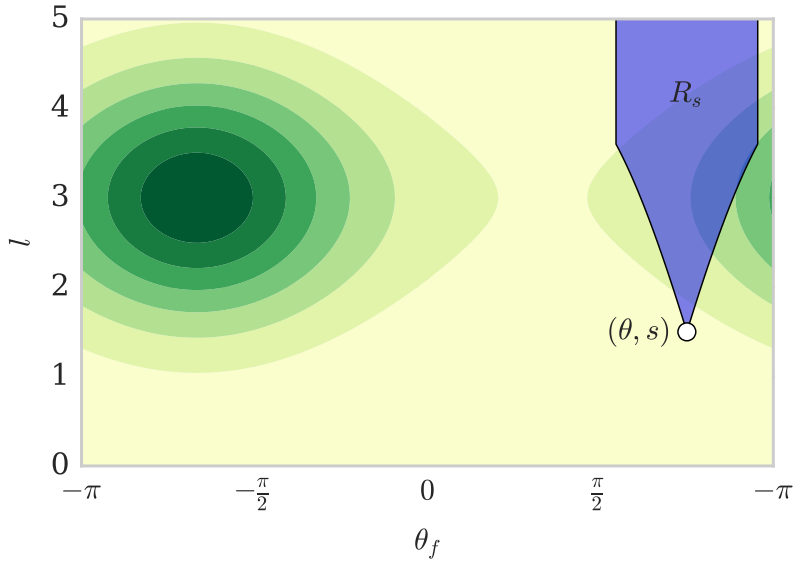

Figure 2.8: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the $\theta_f - l$ plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$

Integrating $P_{2D}(\theta_f, l)$ over $R_s(\theta, s)$ as depicted in Figure 2.8 yields the proportion of the population occupying the point (θ, s),

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

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. Assuming a uniform thickness t for all fronds, and a thickness dz of the depth layer, the proportion of the grid cell occupied by kelp is found to be

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

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

$$a(\boldsymbol{x}) = P_k(\boldsymbol{x})a_k + (1 - P_k(\boldsymbol{x}))a_w$$

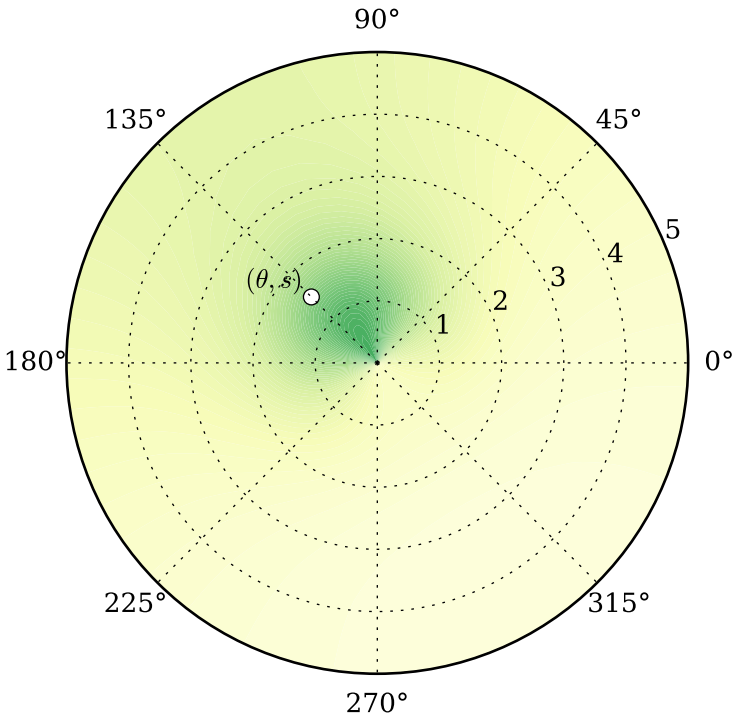

Figure 2.9: Colored plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$

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,

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

Once the radiance L is calculated everywhere, the irradiance is

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

Integrating $I(\boldsymbol{x})$, which has units W/m^2, over the surface of a frond, produces the power (with units W) transmitted to the frond. For details, see Section 3.1.2.

Irradiance is sometimes given in units of moles of photons (a mole of photons is also called an Einstein) per second, with the conversion [8] given by

$$1\,\mathrm{W/m^2} = 4.2\,\mu\mathrm{mol\,photons/s}. \tag{3.1}$$

3.1.2 Perceived Irradiance

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

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

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

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

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.

3.1.3 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 $a(\boldsymbol{x})$ (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 $\boldsymbol{\omega}$ and $\boldsymbol{\omega}'$, $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$ is the probability density of light scattering from $\boldsymbol{\omega}$ into $\boldsymbol{\omega}'$ (or vice-versa). Of course, since a single direction subtends no solid angle, the probability of scattering occurring exactly from $\boldsymbol{\omega}$ to $\boldsymbol{\omega}'$ is 0. Rather, we say that the probability of radiance being scattered from a direction ω into an element of solid angle Ω is $\int_\Omega \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')\, d\boldsymbol{\omega}'$.

The VSF is normalized such that

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

so that for any ω,

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

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

3.2 The Radiative Transfer Equation

We now present the Radiative Transfer Equation, whose solution is the radiance in the medium as a function of position and angle.

3.2.1 Ray Notation

Consider a fixed position $\boldsymbol{x}$ and direction $\boldsymbol{\omega}$ such that $\boldsymbol{\omega} \cdot \hat{z} \neq 0$. Let $\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s)$ denote the linear path containing $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$. Assume that the ray is not horizontal. Then, it originates either at the surface or bottom of the domain, with

initial z coordinate given by

$$z_0 = \begin{cases} 0, & \boldsymbol{\omega} \cdot \hat{z} < 0 \\ z_{\max}, & \boldsymbol{\omega} \cdot \hat{z} > 0. \end{cases}$$

Hence, the ray path is parameterized as

$$\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s) = \frac{1}{\tilde{s}}(s\boldsymbol{x} + (\tilde{s} - s)\boldsymbol{x_0}(\boldsymbol{x}, \boldsymbol{\omega})), \tag{3.2}$$

where

$$\boldsymbol{x_0}(\boldsymbol{x}, \boldsymbol{\omega}) = \boldsymbol{x} - \tilde{s}\boldsymbol{\omega}$$

is the origin of the ray, and

$$\tilde{s} = \frac{\boldsymbol{x} \cdot \hat{z} - z_0}{\boldsymbol{\omega} \cdot \hat{z}}$$

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

3.2.2 Colloquial Description

Denote the radiance at $\boldsymbol{x}$ in the direction $\boldsymbol{\omega}$ by $L(\boldsymbol{x}, \boldsymbol{\omega})$. As light travels along $\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s)$, 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

Combining these phenomena yields the Radiative Transfer Equation along $\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega})$,

$$\frac{dL}{ds}(\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}) = -(a(\boldsymbol{x}) + b)L(\boldsymbol{x}, \boldsymbol{\omega}) + b \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')L(\boldsymbol{x})\, d\boldsymbol{\omega}', \tag{3.3}$$

where $\int_{4\pi}$ denotes integration over the unit sphere. The derivative of L over the path can be rewritten as

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

which unveils the general form of the Radiative Transfer Equation,

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

or, equivalently,

$$\boldsymbol{\omega}\cdot\nabla L(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L(\boldsymbol{x},\boldsymbol{\omega}) = b\left(\int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L(\boldsymbol{x},\boldsymbol{\omega})\right). \quad (3.4)$$

3.2.4 Boundary Conditions

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

$$L\left((x_{\min},y,z),\boldsymbol{\omega}\right) = L\left((x_{\max},y,z),\boldsymbol{\omega}\right),$$

$$L\left((x,y_{\min},z),\boldsymbol{\omega}\right) = L\left((x,y_{\max},z),\boldsymbol{\omega}\right).$$

In the z direction, we specify a spatially uniform downwelling light just under the surface of the water by a function $f(\boldsymbol{\omega})$. Or if $z_{\min} > 0$, then the radiance at $z = z_{\min}$ 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, so

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

$$L(\boldsymbol{x_b},\boldsymbol{\omega}) = 0 \text{ if } \boldsymbol{\omega}\cdot\hat{z} < 0.$$

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

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

Substituting the above into (3.4) yields

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

Grouping like powers of b, we have the decoupled set of equations

$$\begin{aligned}
\boldsymbol{\omega}\cdot\nabla L_0(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_0(\boldsymbol{x}) &= 0, \qquad (3.5)\\
\boldsymbol{\omega}\cdot\nabla L_1(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_1(\boldsymbol{x}) &= \int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L_0(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L_0(\boldsymbol{x},\boldsymbol{\omega}),\\
\boldsymbol{\omega}\cdot\nabla L_2(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_2(\boldsymbol{x}) &= \int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L_1(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L_1(\boldsymbol{x},\boldsymbol{\omega}).\\
&\vdots
\end{aligned}$$

In general, for $n \geq 1$,

$$\boldsymbol{\omega} \cdot \nabla L_n(\boldsymbol{x}, \boldsymbol{\omega}) + a(\boldsymbol{x}) L_n(\boldsymbol{x}) = \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_{n-1}(\boldsymbol{x}, \boldsymbol{\omega}')\, d\boldsymbol{\omega}' - L_{n-1}(\boldsymbol{x}, \boldsymbol{\omega}). \quad (3.6)$$

For boundary conditions, let $\boldsymbol{x_s}$ be a point on the surface of the domain. Then,

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

which can be decomposed as

$$L_0(\boldsymbol{x}, \boldsymbol{\omega}) = \begin{cases} f(\omega), & \hat{z} \cdot \omega > 0, \\ 0, & \text{otherwise,} \end{cases} \quad (3.7)$$

$$L_1(\boldsymbol{x}, \boldsymbol{\omega}) = 0$$

$$L_2(\boldsymbol{x}, \boldsymbol{\omega}) = 0.$$

$$\vdots$$

In general, for $n \geq 1$,

$$L_n(\boldsymbol{x}, \boldsymbol{\omega}) = 0. \quad (3.8)$$

3.3.2 Analytical Solution

For all $\boldsymbol{x}, \boldsymbol{\omega}$, we consider the path $l(\boldsymbol{x}, \boldsymbol{\omega}, s)$ from (3.2). We extract the absorption coefficient along the path,

$$\tilde{a}(s) = a(\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}), s).$$

Then, the first equation from the asymptotic expansion, (3.5) and its associated boundary condition, (3.7), can be rewritten as

$$\begin{cases} 0 = \dfrac{du_0}{ds}(s) + \tilde{a}(s)u_0(s) \\ u_0(0) = f(\boldsymbol{\omega}), \end{cases}$$

which we can solve by multiplying by the appropriate integrating factor, as follows.

$$\begin{aligned} 0 &= \exp\left(\int_0^s \tilde{a}(s')\,ds'\right)\frac{du_0}{ds} + \exp\left(\int_0^s \tilde{a}(s')\,ds'\right)\tilde{a}(s)u_0(s) \\ &= \frac{d}{ds}\left[\exp\left(\int_0^s \tilde{a}(s')\,ds'\right)u_0(s)\right]. \end{aligned}$$

Then, integrating both sides yields

$$\begin{aligned} 0 &= \int_0^s \frac{d}{ds'}\left[\exp\left(\int_0^{s'} \tilde{a}(s'')\,ds''\right)u_0(s')\right]\,ds' \\ &= \exp\left(\int_0^s \tilde{a}(s')\,ds'\right)u_0(s) - f(\boldsymbol{\omega}). \end{aligned}$$

Hence,

$$u_0(s) = f(\omega)\exp\left(-\int_0^s \tilde{a}(s)\,ds\right). \tag{3.9}$$

Then, we convert back from path length s to the spatial coordinate $\boldsymbol{x}$ using

$$L_0(\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega},s),\boldsymbol{\omega}) = u_0(s).$$

The $n \geq 1$ equations have a nonzero right-hand side, which we call the effective source, $g_n(s)$. This can be similarly extracted along a ray path as

$$g_n(s) = \int_{4\pi} \beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega'})L_{n-1}(\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega'},s),\boldsymbol{\omega'})\,d\boldsymbol{\omega'} - L_{n-1}(\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega},s),\boldsymbol{\omega}).$$

Then, since g_n depends only on L_{n-1}, it is independent of u_n, which allows (3.6) and its boundary condition, (3.8), to be written as the first order, linear ordinary differential equation along the ray path,

$$\begin{cases} g_n(s) = \dfrac{du_n}{ds}(s) + \tilde{a}(s)u_n(s) \\ u_n(0) = 0 \end{cases}$$

As with the $n = 0$ equation, the solution is found by multiplying by the appropriate integrating factor.

$$\begin{aligned} \exp\left(\int_0^s \tilde{a}(s')\,ds'\right) g_n(s) &= \exp\left(\int_0^s \tilde{a}(s')\,ds'\right) \frac{du_n}{ds} + \exp\left(\int_0^s \tilde{a}(s')\,ds'\right) \tilde{a}(s)u_n(s) \\ &= \frac{d}{ds}\left[\exp\left(\int_0^s \tilde{a}(s')\,ds'\right) u_n(s)\right]. \end{aligned}$$

Integrating both sides yields

$$\begin{aligned} \int_0^s \exp\left(\int_0^{s'} \tilde{a}(s'')\,ds''\right) g_n(s')\,ds' &= \int_0^s \frac{d}{ds'}\left[\exp\left(\int_0^{s'} \tilde{a}(s'')\,ds''\right) u_n(s')\right] ds' \\ &= \exp\left(\int_0^s \tilde{a}(s')\,ds'\right) u_n(s). \end{aligned}$$

Hence,

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

which simplifies to

$$u_n(s) = \int_0^s g_n(s') \exp\left(-\int_{s''}^{s'} \tilde{a}(s'')\,ds''\right) ds'. \tag{3.10}$$

As before, the conversion back to spatial coordinates is

$$L_n(\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}) = u_n(s).$$

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

Rather than model each kelp frond, a subset of the population, called super-individuals, are modeled explicitly, and are considered to represent many identical individuals, as in [14]. 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 (2.3), 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 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. Other legitimate choices of grids exists; this one was chosen for its relative simplicity.

The domain of the radiative transfer equation is embedded in five dimensions: three spatial (x, y, and z) and two angular (azimuthal θ and polar ϕ). 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:

$$x \to i$$

$$y \to j$$

$$z \to k$$

$$\theta \to l$$

$$\phi \to m$$

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

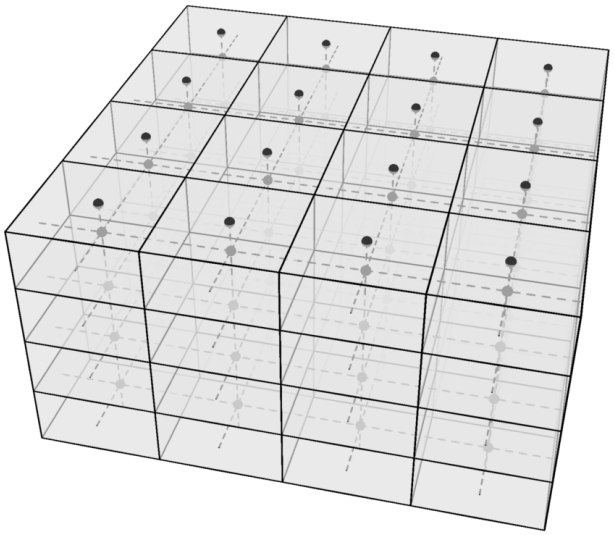

Figure 4.1: Spatial grid

Each spatial grid cell is the Cartesian product of x, y, and z intervals of width dx, dy, and dz respectively. The three-dimensional interval centered at (x_i, y_j, z_k) is denoted X_{ijk}, and has volume $|X_{ijk}| = dx\,dy\,dz$. Also, note that no grid center is located on the plane $z = 0$; the surface radiance boundary condition is treated separately.

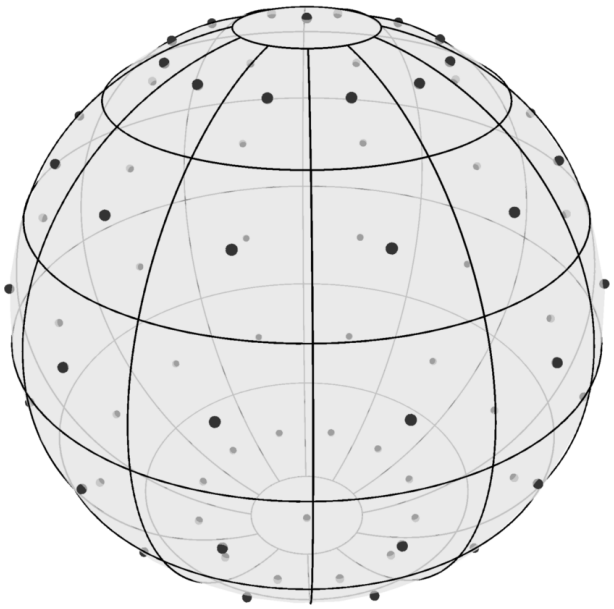

Figure 4.2: Angular grid at each point in space

As shown in Figure 4.2, $\phi = 0$ and $\phi = \pi$, called the north $(+z)$ and south $(-z)$ poles respectively, are treated separately from other angular grid cells. A generic interior angular grid cell centered at $\boldsymbol{\omega}_p$ is the Cartesian product of an azimuthal interval of width $d\theta$ and a polar interval of width $d\theta$. However, two pole cells are the Cartesian product of a polar interval of width $d\phi/2$ and the full azimuthal domain, $[0, 2\pi)$.

With this configuration, the total number of angles considered is $n_{\boldsymbol{\omega}} = n_\theta(n_\phi - 2) + 2$. Then, cells are indexed by $p = 1, \ldots, n_{\boldsymbol{\omega}}$ and are ordered such that $p = 1$ and $p = n_{\boldsymbol{\omega}}$ refer to the north and south poles respectively, $p \leq n_{\boldsymbol{\omega}}/2$ refers to the northern hemisphere, and $p > n_{\boldsymbol{\omega}}/2$ refers to the southern hemisphere. Further, the symbol Ω_p is used to refer to the two dimension angular interval centered at ω_p. The solid angle subtended by Ω_p is denoted $|\Omega_p|$. Refer to Appendix A for a more rigorous discussion of the discrete spatial-angular grid.

4.3 Quadrature Rules

Since it is assumed that all quantities are constant within a spatial-angular grid cell, the midpoint rule is employed for both spatial and angular integration. Presented here is a basic derivation of the formulas for integration in the spatial-angular grid. Further details are found in Appendix B.

4.3.1 Spatial Quadrature

Define the *spatial characteristic function* as

$$\mathcal{X}_{ijk}^{X}(\boldsymbol{x}) = \begin{cases} 1, & \boldsymbol{x} \in X_{ijk} \\ 0, & \text{otherwise.} \end{cases}$$

The double integral of a function $f(\boldsymbol{x})$ over a depth layer k is approximated as

$$\begin{aligned}\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} f(x,y,z_k)\,dy\,dx &\approx \int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} \mathcal{X}_{ijk}^{X}(x,y,z_k) f(x_i,y_j,z_k)\,dy\,dx \\ &= \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} f(x_i,y_j,z_k) \int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} \mathcal{X}_{ijk}^{X}(x,y,z_k)\,dy\,dx \\ &= \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} |X_{ijk}|\, f(x_i,y_j,z_k) \\ &= dx\,dy\,dx \sum_{i=1}^{n_x}\sum_{j=1}^{n_y} f(x_i,y_j,z_k).\end{aligned}$$

The path integral of $f(\boldsymbol{x})$ over a path $\boldsymbol{l}(s)$ from $s=0$ to $s=\tilde{s}$ is

$$\int_0^{\tilde{s}} f(\boldsymbol{l}(s))\,ds = \sum_{i=1}^{n_x}\sum_{j=1}^{n_y}\sum_{k=1}^{n_z} f(x_i,y_j,z_k)\,ds_{ijk},$$

where ds_{ijk} is the total path distance of $\boldsymbol{l}(s)$ through X_{ijk}. Full details of the path integral algorithm for the case of straight line paths are found in Appendix B.

4.3.2 Angular Quadrature

Define the *angular characteristic function* as

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

Then, the integral of a function $f(\boldsymbol{\omega})$ is approximated as

$$\begin{aligned}\int_{4\pi} f(\boldsymbol{\omega})\, d\boldsymbol{\omega} &\approx \int_{4\pi} \sum_{p=1}^{n_\omega} f(\boldsymbol{\omega}_p)\mathcal{X}_p^\Omega(\boldsymbol{\omega})\, d\boldsymbol{\omega} \\ &= \sum_{p=1}^{n_\omega} f(\boldsymbol{\omega}_p) \int_{4\pi} \mathcal{X}_p^\Omega(\boldsymbol{\omega})\, d\boldsymbol{\omega} \\ &= \sum_{p=1}^{n_\omega} f(\boldsymbol{\omega}_p) \int_{\Omega_p} d\boldsymbol{\omega} \\ &= \sum_{p=1}^{n_\omega} f(\boldsymbol{\omega}_p) \left|\Omega_p\right|.\end{aligned}$$

4.4 Numerical Asymptotics

Presented here are details of the evaluation of the asymptotic approximations (3.9) and (3.10) to the raditiave transfer equation (3.4).

4.4.1 Scattering Integral

Specifically, the amount of light scattered between angular grid cells is found by integrating β as follows. Consider two angular grid cells, Ω and Ω'. Since $\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')$ is the probability density of scattering between $\boldsymbol{\omega}$ and $\boldsymbol{\omega}'$, the average probability density of scattering from $\boldsymbol{\omega}\in\Omega$ to $\boldsymbol{\omega}'\in\Omega'$ (or vice versa) is

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

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

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

Hence, the average radiance scattered from $\Omega_{p'}$ into some $\boldsymbol{\omega} \in \Omega_p$ is $\beta_{pp'} \left|\Omega'\right| L_{ijkp'}$. Therefore, the radiance gain due to scattering into $\boldsymbol{\omega}_p$ from all other angles is

$$\int_{4\pi} \beta(\boldsymbol{\omega_p} \cdot \boldsymbol{\omega_{p'}}) L(\boldsymbol{x}, \boldsymbol{\omega'})\, d\boldsymbol{\omega} \approx \sum_{p=1}^{n_{\boldsymbol{\omega}}} \beta_{pp'} \left|\Omega'\right| L_{ijkp}. \tag{4.3}$$

4.4.2 Ray Integral

Given a position $\boldsymbol{x}$ and direction $\boldsymbol{\omega}$, a path through the discrete grid can be constructed using the ray tracing algorithm described in Appendix B. Let $\nu = 1, \ldots, N-1$ index the spatial grid cells traversed by the ray, and define the *path-length characteristic function*

$$\mathcal{X}_\nu^l(s) = \begin{cases} 1, & s_\nu \leq s < s_{\nu+1} \\ \\ 0, & \text{otherwise.} \end{cases}$$

Then, the piecewise constant representations of the path absorption coefficient $\tilde{a}(s)$ and the effective source $g_n(s)$ from Section 3.3.2 are

$$g_n(s) = \sum_{\nu=1}^{N-1} g_{n\nu} \mathcal{X}_\nu^l(s),$$
$$\tilde{a}(s) = \sum_{\nu=1}^{N-1} \tilde{a}_\nu \mathcal{X}_\nu^l(s).$$

As the ray traverses the spatial grids, it crosses $N-2$ spatial grid edges. Let the nondecreasing path lengths at which these crossings occur be denoted by $\{s_\nu\}_{\nu=1}^N$, with the convention $s_1 = 0$ and $s_N = \tilde{s}$. $\{s_\nu\}$ is not strictly increasing if the ray directly intersects a grid corner, which means that multiple edges are traversed at the same path length. Hence, for $\nu = 1, \ldots, N-1$, the path lengths through each grid cell are

$$ds_\nu = s_{\nu+1} - s_\nu.$$

Given s, the index next edge crossing occurs at

$$\hat{\nu}(s) = \min\left\{\nu \in \{1, \ldots, N\} : s_\nu > s\right\},$$

and the path length between s and the next edge crossing is

$$\tilde{d}(s) = s_{\hat{\nu}(s)} - s.$$

Then, evaluating (3.10) at $s = \tilde{s}$ is calculated as

$$\begin{aligned}
u_n(\tilde{s}) &= \int_0^{\tilde{s}} g_n(s') \exp\left(-\int_{s''}^{s'} \tilde{a}(s'')\, ds''\right) ds' \\
&= \int_0^{s_N} \sum_{\nu=1}^{N-1} g_{n\nu} \mathcal{X}_\nu^l(s') \exp\left(-\int_{s''}^{s'} \sum_{j=1}^{N-1} \tilde{a}_j \mathcal{X}_j^l(s'')\, ds''\right) ds' \\
&= \sum_{\nu=1}^{N-1} g_{n\nu} \int_0^{s_N} \mathcal{X}_\nu^l(s') \exp\left(-\sum_{j=1}^{N-1} \tilde{a}_j \int_{s''}^{s'} \mathcal{X}_j^l(s'')\, ds''\right) ds' \\
&= \sum_{\nu=1}^{N-1} g_{n\nu} \int_{s_\nu}^{s_{\nu+1}} \exp\left(-\tilde{a}_{\hat{\nu}(s')-1} \tilde{d}(s') - \sum_{j=\hat{\nu}(s')}^{N-1} \tilde{a}_j ds_j\right) ds' \\
&= \sum_{\nu=1}^{N-1} g_{n\nu} \int_{s_\nu}^{s_{\nu+1}} \exp\left(-\tilde{a}_\nu (s_{\nu+1} - s') - \sum_{j=\nu+1}^{N-1} \tilde{a}_j ds_j\right) ds'.
\end{aligned}$$

This integral is made straightforward by setting

$$b_\nu = -\tilde{a}_\nu s_{\nu+1} - \sum_{j=\nu+1}^{N-1} \tilde{a}_j ds_j,$$

which yields

$$\begin{aligned}
u_n(\tilde{s}) &= \sum_{\nu=1}^{N-1} g_{n\nu} \int_{s_\nu}^{s_{\nu+1}} \exp\left(\tilde{a}_\nu s' + b_\nu\right) ds' \\
&= \sum_{\nu=1}^{N-1} g_{n\nu} e^{b_\nu} \int_{s_\nu}^{s_{\nu+1}} \exp\left(\tilde{a}_\nu s'\right) ds'.
\end{aligned}$$

Define the intermediate variable

$$
\begin{aligned}
d_\nu &= \int_{s_\nu}^{s_{\nu+1}} \exp\left(\tilde{a}_\nu s'\right)\, ds' \\
&= \begin{cases} ds_\nu, & \tilde{a} = 0 \\ \left(\exp(\tilde{a}_\nu s_{\nu+1}) - \exp(\tilde{a}_\nu s_\nu)\right)/\tilde{a}_\nu, & \text{otherwise,} \end{cases}
\end{aligned}
$$

which permits the simple formula

$$
u_n(\tilde{s}) = \sum_{\nu=1}^{N-1} g_{n\nu} d_\nu e^{b_\nu}. \tag{4.4}
$$

4.5 Finite Difference

While the asymptotic solution is valid in case of low scattering, a more general solution is obtained via finite difference, whereby the derivatives and integrals in the integro-partial differential equation are discretized to differences and sums and evaluated at each grid cell in order to construct a linear system of equations whose solution approximates that of the analytical equation. The price of a general solution, however, is greatly increased computational cost, both in terms of memory and CPU usage. Not only is it necessary to store a potentially huge sparse matrix in memory, but its construction can take enormous amounts of time. Nonetheless, it is useful to have access to a general numerical solution, at least for verification of approximations if not for direct use in applications.

4.5.1 Discretization

For the spatial interior of the domain, we use the second 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

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

Truncating after the first few terms, we have

$$f'(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \mathcal{O}(\varepsilon^3) \tag{4.5}$$

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

$$f'(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \tag{4.6}$$

Rearranging (4.5) produces

$$f''(x)\varepsilon^2 = 2f(x+\varepsilon) - 2f(x) - 2f'(x)\varepsilon + \mathcal{O}(\varepsilon^3) \tag{4.7}$$

Combining (4.6) with (4.7) 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

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

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

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

4.5.2 Difference Equations

In general, we have

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

Then,

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

Interior:

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

Surface downwelling (BC):

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

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

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

Likewise for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

4.5.3 Structure of Linear System

For each spatial-angular grid cell, one of the above equations is applied. The equation applied at each grid cell involves adjacent radiance values due to the discretized derivatives. Thus, a coupled system of linear equations is produced, which can be written as a sparse matrix equation, $Ax = b$. In the coefficient matrix A, each row is asociated with the grid cell at which the discretized equation was evaluated. Each column is the coefficient of the radiance at a particular spatial-angular grid cell. In principle the order of the equations, i.e., the order of the rows and columns of the coefficient matrix, is not important so long as consistency is maintained with the solution vector and right-hand side. In general, some procedure is necessary for constructing an ordered list of the multidimensional grid cells. One option, employed here, is to use a block structure where dimensions are nested within one another. An ordering for the dimensions is chosen, from outermost to innermost. Adjacent rows and columns in the matrix are associated with adjacent grid cells in the innermost

dimension, adjacent blocks of the innermost dimension are adjacent in the second innermost dimension, etc.

In the numerical implementation of this model, we choose the order of dimensions to be $\boldsymbol{\omega}, z, y, x$, with $\boldsymbol{\omega}$ being the outermost and x being the innermost. Recall that θ and ϕ are already combined, both indexed by p, as discussed in Section 4.2 and Appendix A. This particular ordering is chosen for ease of programming in terms of deciding which of the equations from Section 4.5.2 to apply. Since the choice of equation does not depend on x or y, they are the outermost. Then, the surface and bottom z values have to be considered separately from the rest. And within the surface and bottom depth layers, there are further cases depepending on whether the light is upwelling or downwelling. Hence, the chosen ordering follows somewhat naturally from the boundary conditions.

Then, the discretized equation applied to $(x_i, y_j, z_k, \omega_p)$ is stored in row

$$r_{ijkp} = p + n_{\boldsymbol{\omega}}(k-1) + n_{\boldsymbol{\omega}} n_z (j-1) + n_{\boldsymbol{\omega}} n_z n_y (i-1).$$

Since the same ordering is used for rows and columns of the coefficient matrix A, L_{ijkp} is located at position r_{ijkp} of the solution vector x, and the right-hand side associated with that grid cell, if any, is also stored at position r_{ijkp} of the right-hand side vector b.

Also relevant is the total size of the system and of the sparse matrices necessary to store. The sizes of A, x, and b are the number of grid cells, which is just $n_x n_y n_z n_{\boldsymbol{\omega}}$. Most of these elements, though, are zero since derivatives only involve

adjacent spatial grid cells and the scattering integral only involves angles within a single spatial grid cell. Therefore, by saving only the locations and values of nonzero elements in the coefficient matrix, a considerable amount of storage space is saved. Table 4.1 shows a breakdown of the number of distinct radiance values involved in each application of the discretized equations from Section 4.5.2, as well as the number of times that each of the equations appears in the matrix.

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

Table 4.1: Breakdown of nonzero matrix elements by derivative case

By multiplying the first column of Table 4.1 by the second and summing

over the rows, the total number of nonzero matrix elements is calculated to be

$$\begin{aligned} N_A &= (n_{\boldsymbol{\omega}} + 6) \cdot n_x n_y (n_z - 2) n_{\boldsymbol{\omega}} \\ &\quad + (n_{\boldsymbol{\omega}} + 5) \cdot n_x n_y n_{\boldsymbol{\omega}} + (n_{\boldsymbol{\omega}} + 6) \cdot n_x n_y n_{\boldsymbol{\omega}} \\ &= n_x n_y n_z \left[(n_{\boldsymbol{\omega}} + 6)(n_z - 2 + 1) + n_{\boldsymbol{\omega}} + 5 \right] \\ &= n_x n_y n_z \left[(n_{\boldsymbol{\omega}} + 6)(n_z - 1) + n_{\boldsymbol{\omega}} + 5 \right] \\ &= n_x n_y n_z \left[n\omega n_z - n_{\boldsymbol{\omega}} + 6 n_z - 6 + n_{\boldsymbol{\omega}} + 5 \right] \\ &= n_x n_y n_z \left[n_z (n_{\boldsymbol{\omega}} + 6) - 1 \right] \end{aligned}$$

Also, note that b only has nonzero entries for the downwelling surface grid cells, of which there are $n_x n_y n_{\boldsymbol{\omega}}/2$.

4.5.4 Iterative Solution

For small systems of equations, direct methods such as Gaussian elimination, QR factorization, and singular value decomposition can be used. However, when the coefficient matrix becomes very large, the memory required to carry out a direct solution is prohibitively large, iterative solvers must be used. Many such solvers are available, including GMRES[13], LGMRES[2], IDR[16], and BI-CGSTAB[17]. In our case, GMRES is used.

CHAPTER V

PARAMETER VALUES

In this chapter, model parameters are discussed. In the case that this model is run in conjunction with a kelp growth model and ocean model, they will provide some necessary parameters. Other parameters not coming from the kelp or ocean model can be found in the literature, summarized in Table 5.1 and Table 5.2. Still, some parameters remain which are not well described in the literature.

5.1 Simulation Parameters

It is assumed that this model is run together with a kelp growth model such as described in [3], and an ocean model, as in [18]. Both models are assumed to use the same spatial grid, with n_z discrete depth layers of thickness dz_k for $k = 1, ldots, n_z$. It is assumed that the horizontal spacing for both models is quite large, and the light model therefore uses a much finer horizontal resolution, but retains the same vertical resolution as the encompassing calculations. The ocean model provides current speed and direction over depth, which is used in calculating the kelp distribution. The position of the sun and irradiance just below the surface of the water is also provided by the ocean model, which is used to generate the surface radiance boundary condition. The ocean model should also provide an absorption coefficient for each

depth layer, which may vary due to nutrient concentrations an biological specimen such as phytoplankton. The kelp model is expected to provide super-individual data describing the population in each depth layer. Then, (4.1) and (4.2) are used to calculate length and orientation distributions, as described in Section 4.1.

5.2 Parameters from Literature

Given here is a table of parameter values found in the literature which are used in Chapter 6 to test this light model. A few comments are in order. No values were available for the absorptance of *Saccharina latissima*, but a value for *Macrocystis pyrifera* was found. The surface irradiance from [3] was given in terms of photons per second, and was converted to $\mathrm{W\,m^{-2}}$ according to (3.1). No data in the literature for the frond thickness, so a best estimate is provided.

In [12], very detailed measurements of optical properties in various ocean waters are presented. A few of those measurements are reproduced here, using the same site names as in the original report. There are three categories of water provided: *AUTEC* is from Tounge of the Ocean, Bahama Islands, and represents very clear, pure water; `HAOCE` is from offshore southern California, and represents a more average coastal region, likely the most similar to water where kelp cultivation would occur; `NUC` data is from the San Diego Harbor, and represents very turbid water, likely more so than one would expect to find in a seaweed farm.

Table 5.1: Parameter values

Parameter Name	Symbol	Value(s)	Citation
Kelp absorptance	A_k	0.8	[5]
Water absorption coefficient	a_w	See Table 5.2	[12]
Scattering coefficient	b	See Table 5.2	[12]
Volume scattering function	β	tabulated	[12, 15],
Frond thickness	t	0.4 mm	estimated
Surface solar irradiance	I_0	$50\,\mathrm{W\,m^{-2}}$	[3]

Table 5.2: Field measurement data of optical properties in the ocean[12]. Site names from the source are used.

Site	$\boldsymbol{a}(\mathbf{m^{-1}})$	$\boldsymbol{b}(\mathbf{m^{-1}})$	$\boldsymbol{c}(\mathbf{m^{-1}})$	$\boldsymbol{a/c}$	$\boldsymbol{b/c}$
AUTEC 8	0.114	0.037	0.151	0.753	0.247
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 2240	0.125	1.205	1.33	0.094	0.906

5.3 Frond Bending Coefficient

The *frond bending coefficient*, η, describes the dependence of frond alignment on current speed. To the author's knowledge, no such parameter is available in the literature. However, similar measurements have been made in the MACROSEA project by Norvik et. al.[10] to describe the dependence of the elevation angle of the frond as a function of current speed. In that study, artificial seaweed was designed, suitable for use in fresh water laboratory flumes without fear of degradation. Using those synthetic kelp fronds, one could perform a simple experiment to determine the frond bending coefficient, sketched here.

Fix a taught vertical rope or rod in the center of a flume, and attach the fronds to it with a short string which acts as the stipe. To emulate the holdfast, the string should be tied tightly around the vertical rope or rod so as to prevent it from rotating at its attachment point, giving the frond a preferred orientation from which it has to bend. The preferred directions should be more or less evenly distributed. A camera should be mounted directly over the vertical rope, pointed straight down. If possible, a flourescent dye could be applied to the tip of the each frond to make their orientation more easily discernable in the recording. Turn on the flume to several current speeds, recording a video or many snapshots for each. If the fluorescent dye is applied, then a simple peak-finding image processing algorithm can be applied to locate the frond tips. By preprocessing the image to a gray scale such that the color of the dye has the highest intensity, the tip locations are located at local maxima.

Once the tip locations are determined, thee azimuthal orientations can be calculated relative to the vertical line. Data from all snapshots for the same current speed can be combined, and a von Mises distribution can be fitted to the combined data, noting the best fit values of μ and κ. Presumably, the best fit μ will be in the direction of current flow. After repeating the procedure for several current speeds, κ can be plotted as a function of current speed. Then, an optimal value for the frond bending coefficient η can be found by fitting $\kappa = \eta\mu$ to the data. It may, of course, turn out that this simple linear relationship does not hold, in which case a more appropriate description can be given.

CHAPTER VI

MODEL ANALYSIS

In this chapter, the numerical implementation of the model is probed. First, the finite difference solution at several vertical resolutions is compared to the exact solution in the case of a uniform medium with no scattering. Then, kelp is added to introduce inhomogeneity in the medium, and light scattering is enabled. The maximum feasible resolution for a finite difference solution is used as the "true" solution and compared to lower resolutions to judge their performance. Next, the low-scattering asymptotic approximation is compared to the finite difference solution for the four sets of optical properties given in Table 5.2. Finally, several model parameters are varied from a base case to determine how sensitive the model is to each of them.

6.1 Homogeneous medium

In this section, a homogeneous medium with $a_w = 0.5$ and $b = 0$ is used. In the case of no scattering, the zeroth order asymptotic approximation is in fact the true solution to the radiative transfer equation. Several vertical grid spacings are used for a finite difference solution, and resulting irradiance values are plotted at each depth layer. Absolute and relative differences from the exact solution are shown. Average

errors are then plotted as a function of grid resolution. For $n_z = 24$, an average relative error of about 5% is achieved.

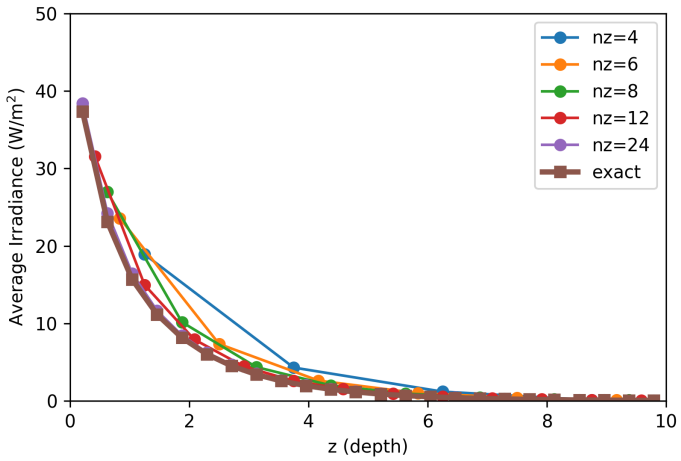

Figure 6.1: Exact v.s. finite difference irradiance, linear scale

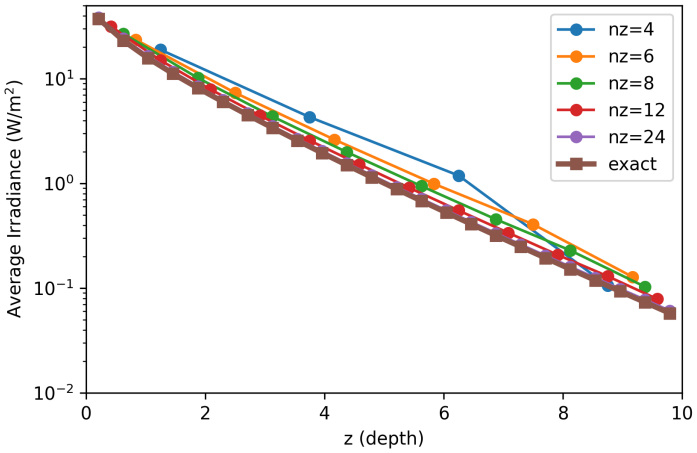

Figure 6.2: Exact v.s. finite difference irradiance, log scale

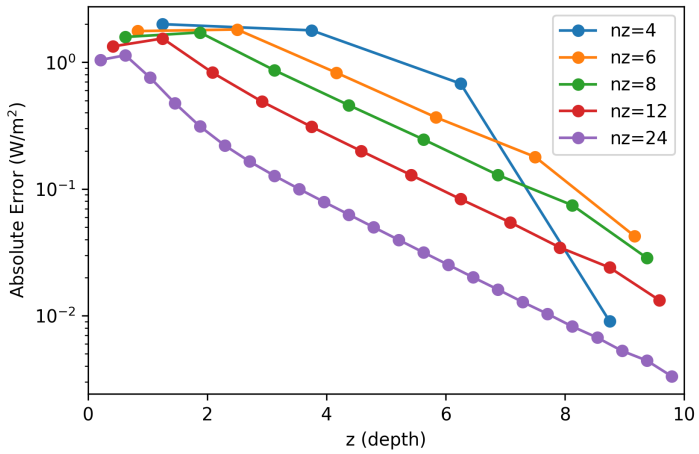

Figure 6.3: Exact v.s. finite difference irradiance, absolute error

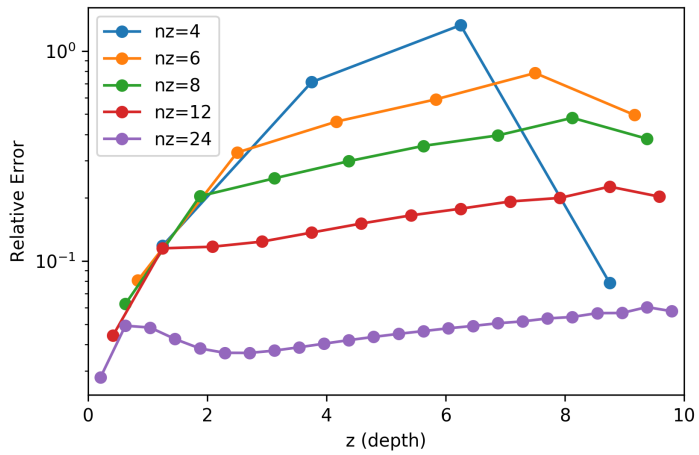

Figure 6.4: Exact v.s. finite difference irradiance, relative error

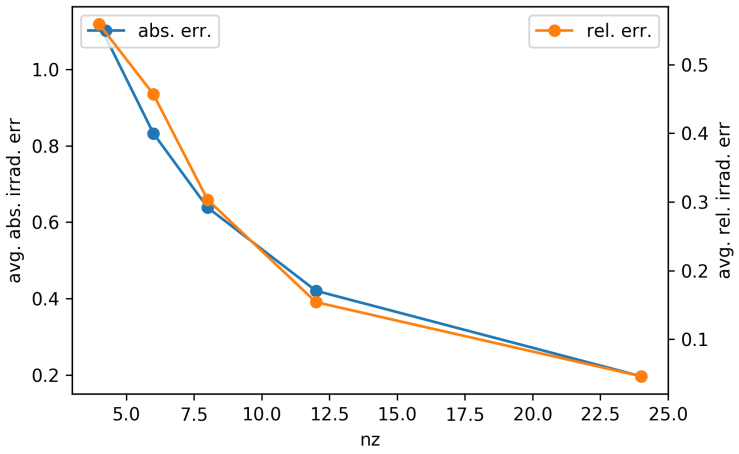

Figure 6.5: Exact v.s. finite difference irradiance, relative error v.s. grid resolution

6.2 Grid Study

A five dimensional (x, y, x, θ, ϕ) resolution space is nontrivial to characterize. For the sake of reducing dimensionality, we define generic spatial and angular resolutions n_s and n_a such that $n_s = n_x = n_y$ and $n_a = n_\theta = n_\phi$. Remaining is a three-dimensional resolution space, (n_s, n_z, n_a). Rather than perform calculations at every possible combination of resolutions in the space, we choose a maximum resolution of $20 \times 20 \times 20$. We hold two of the three resolutions at the maximum value while varying the third. For example, Figure 6.6 compares $4 \times 20 \times 20$, $6 \times 20 \times 20$, $8 \times 20 \times 20$, etc. The quantity that we compare is *perceived irradiance*, which is not the simple average irradiance in each depth layer. Rather, the average is weighted by the normalized

spatial kelp distribution to determine the average irradiance experienced by the kelp population. For more detail, see Section 3.1.2

Note the different natures of convergence in each dimension. In varying n_s, we see that the accuracy is very low for small n_s values. This is because in these cases, the horizontal grid cells are too large to capture any detail about the kelp fronds near the bottom where they are very small. The kelp is effectively not present in these layers, and therefore the perceived irradiance is zero. After increasing the resolution past this minimum threshold, however, little improvement results from increasing n_s further, as seen in Figure 6.6. On the other hand, Figure 6.7 shows that increasing the vertical resolution consistently improves the accuracy of the solution. Figure 6.8 shows that n_a is somewhere between the two, demonstrating clear improvement with increasing resolution, though the improvement is not uniform over depth. Figure 6.9 shows the trend of increasing accuracy with increasing resolution in each dimension.

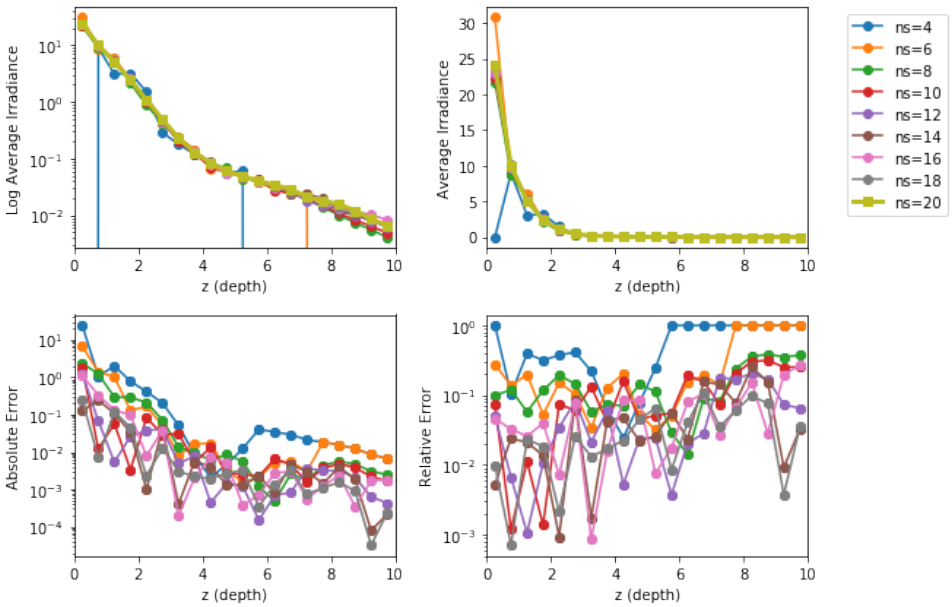

Figure 6.6: Grid study, n_s

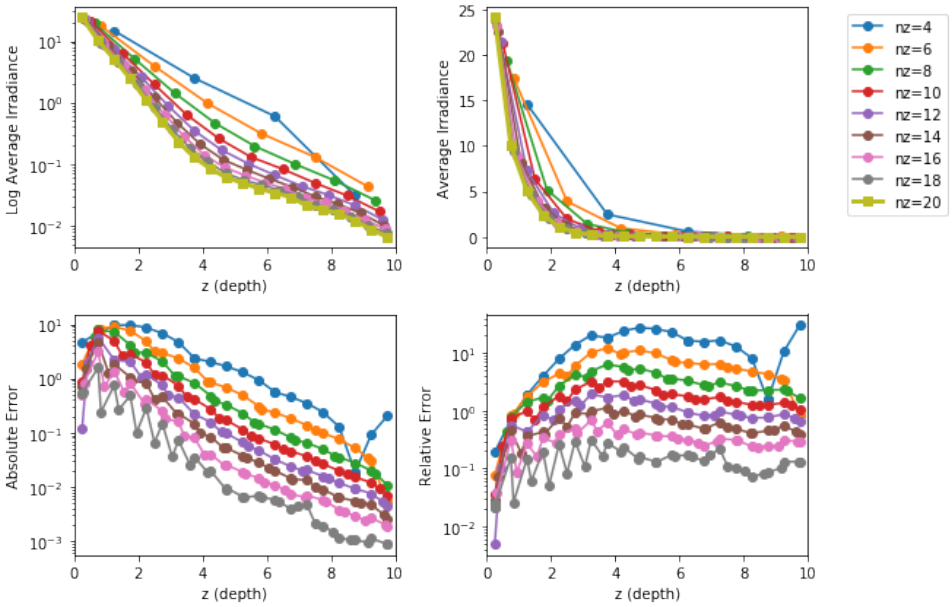

Figure 6.7: Grid study, n_z

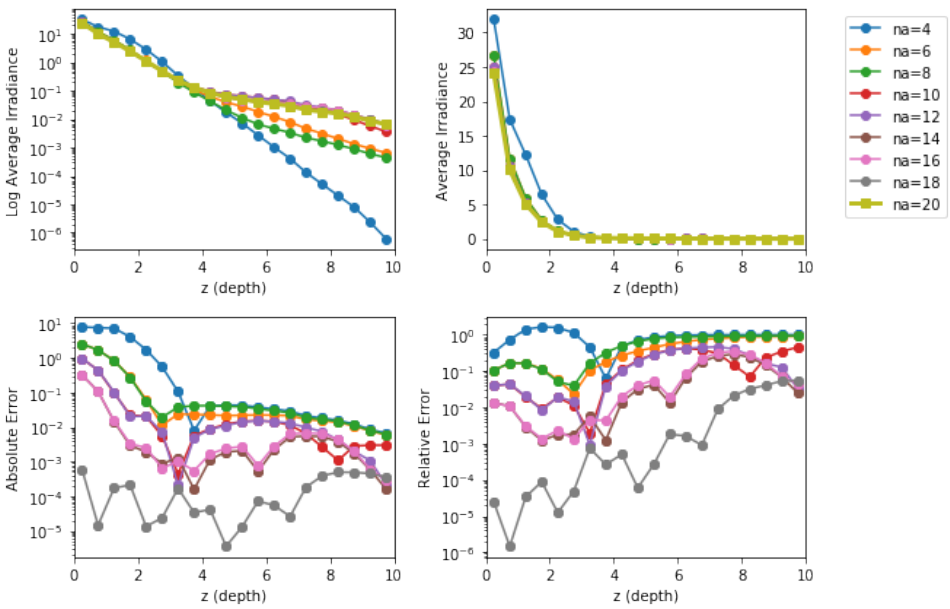

Figure 6.8: Grid study, n_a

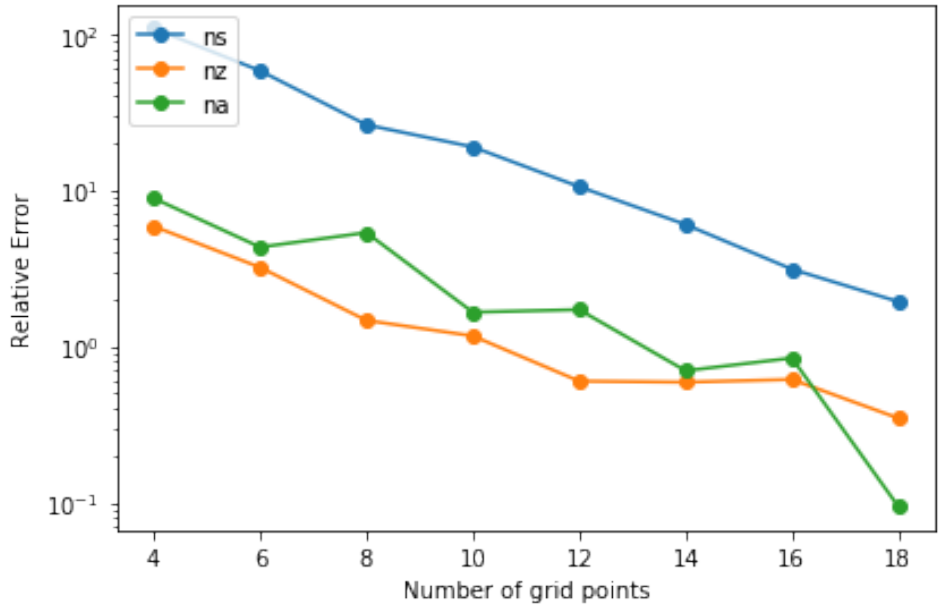

Figure 6.9: Grid study, summary

6.3 Asymptotic Convergence

In this section, the four water cases from Table 5.2 are considered. In each case, the full finite difference solution is calculated on an $18 \times 18 \times 18$ grid, and asymptotic approximations are given, varying the number of terms used in the asymptotic series. Perceived irradiances are shown, as well as errors from the finite difference solution.

In the first two cases, when the scattering coefficient is the same order or smaller as the absorption coefficient, the asymptotic approximation converges to the finite difference solution. However, in the very turbid water of the San Diego Harbor, the scattering coefficient is an order of magnitude higher than the absorption coefficient, causing the asymptotic solution to quickly diverge. In figure 6.16, average relative errors for the two converging cases are shown. In both cases, the accuracy improves with more scattering events until it plateaus. In the first case, 4 scattering events is sufficient, whereas in the second, the accuracy improves until 12 scattering events.

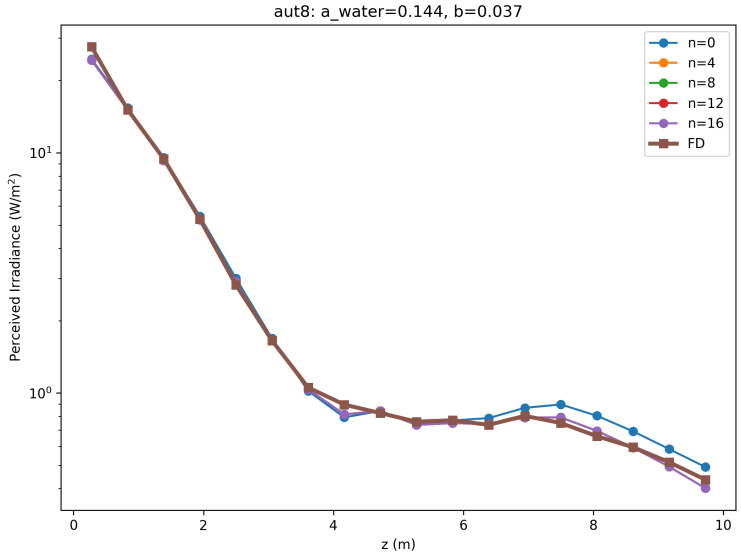

Figure 6.10: Successive asymptotic approximations, irradiance: `AUT8`

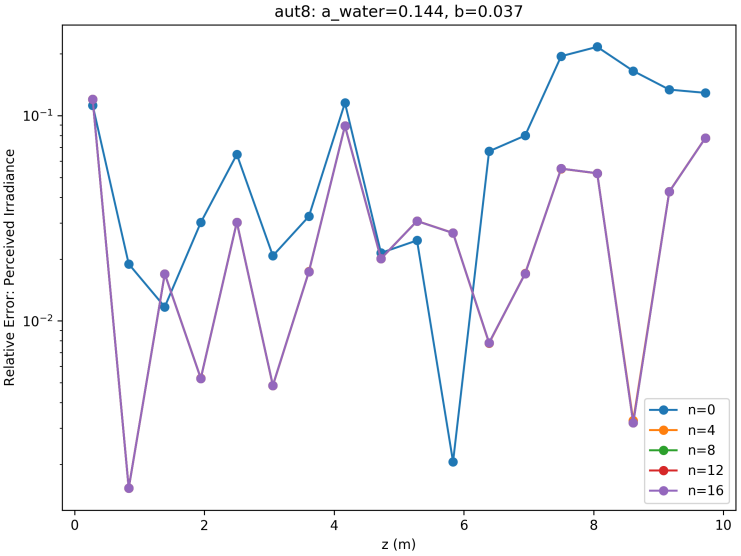

Figure 6.11: Successive asymptotic approximations, relative error: `AUT8`

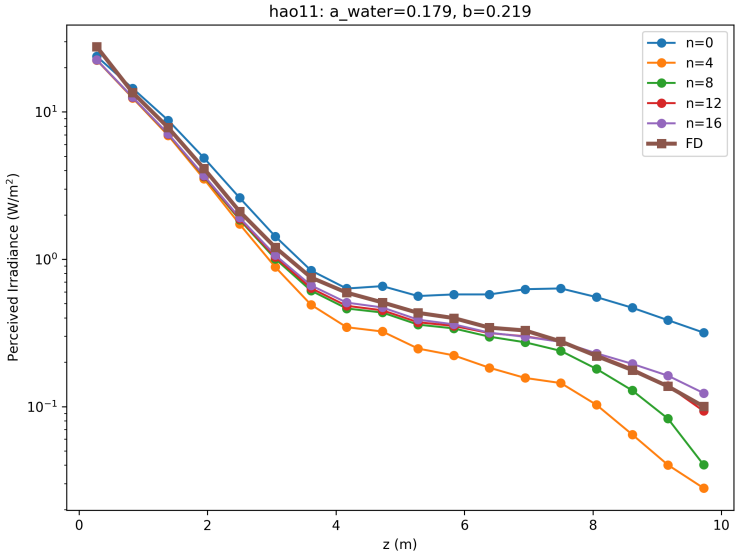

Figure 6.12: Successive asymptotic approximations, irradiance: `HAO11`

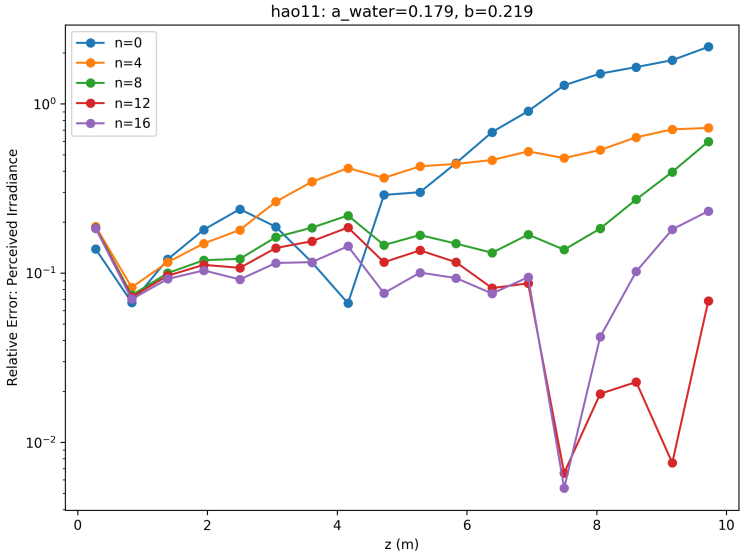

Figure 6.13: Successive asymptotic approximations, relative error: `HAO11`

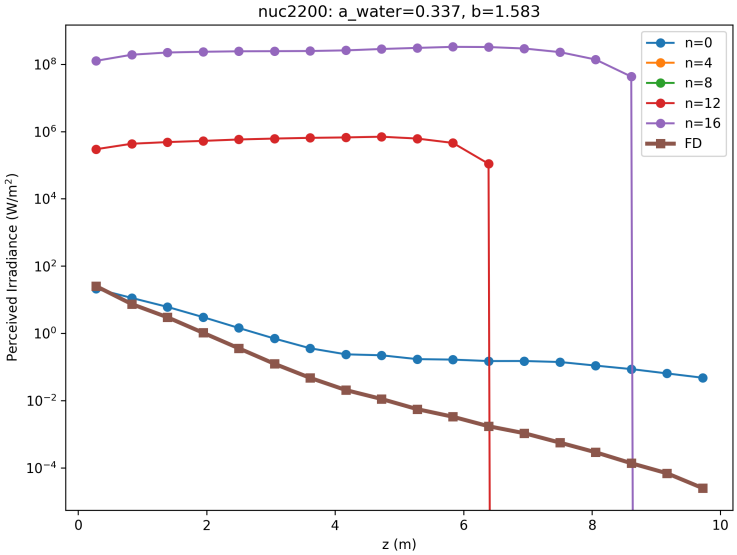

Figure 6.14: Successive asymptotic approximations, irradiance: `NUC2200`

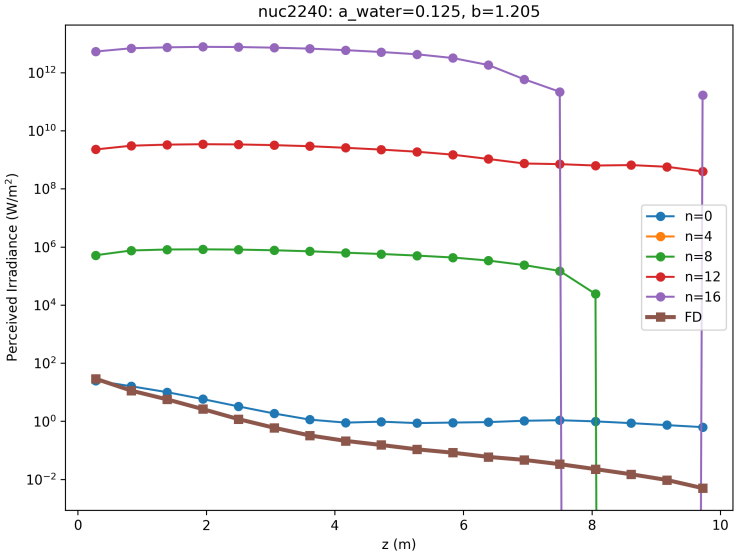

Figure 6.15: Successive asymptotic approximations, relative error: `NUC2240`

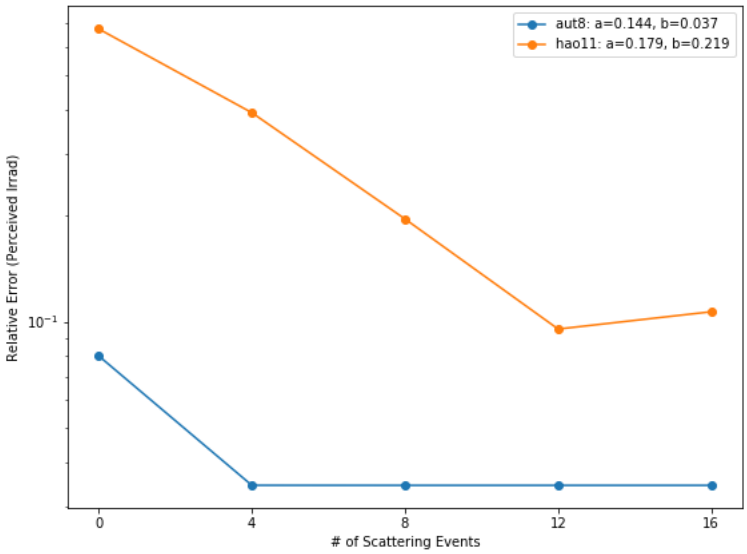

Figure 6.16: Comparison of asymptotic approximations for various waters.

6.4 Sensitivity Analysis

In this section, we demonstrate the effect of varying some of the parameters of the model. The 12-term asymptotic approximation is used. In Figure 6.22 and Figure 6.23, the solution is shown to diverge when the ratio b/a is too large, as in Section 6.3.

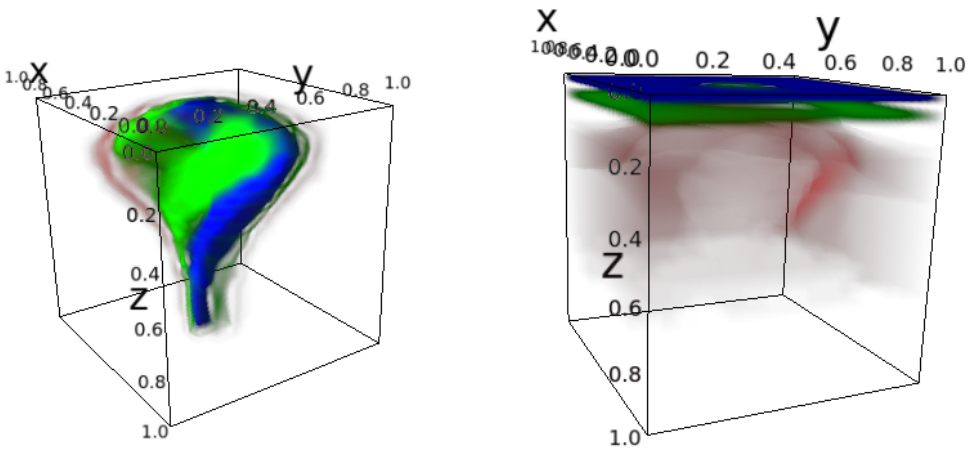

Figure 6.17: *top-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

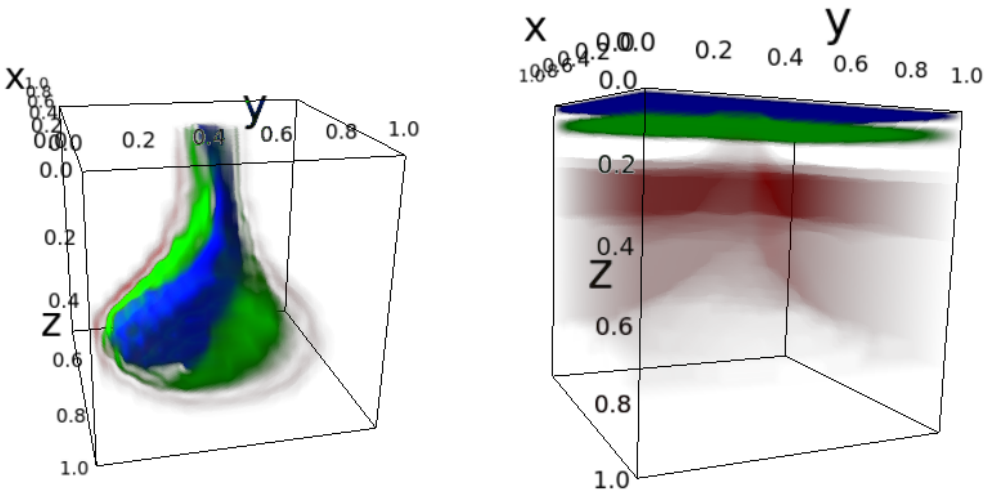

Figure 6.18: *bottom-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

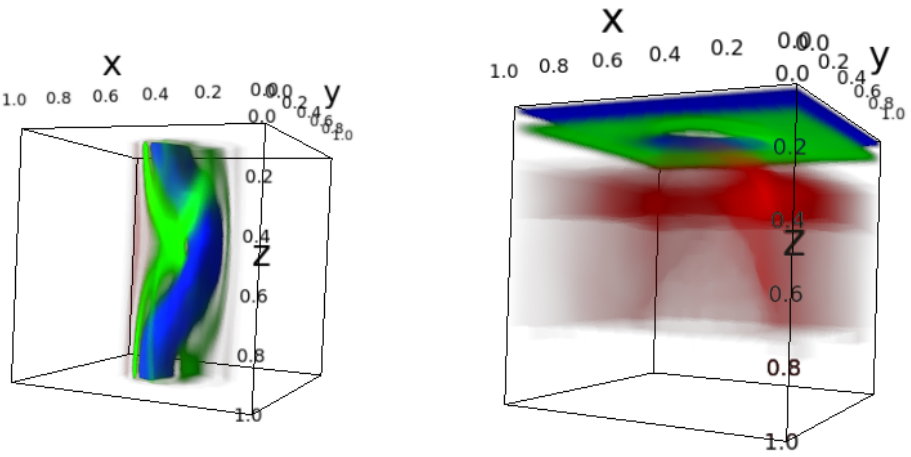

Figure 6.19: *uniform* kelp distribution (left) and no-scattering irradiance profile (right)

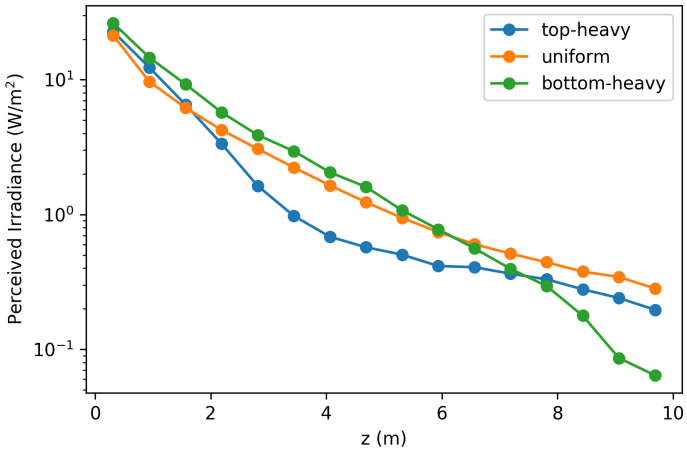

Figure 6.20: Several kelp profiles

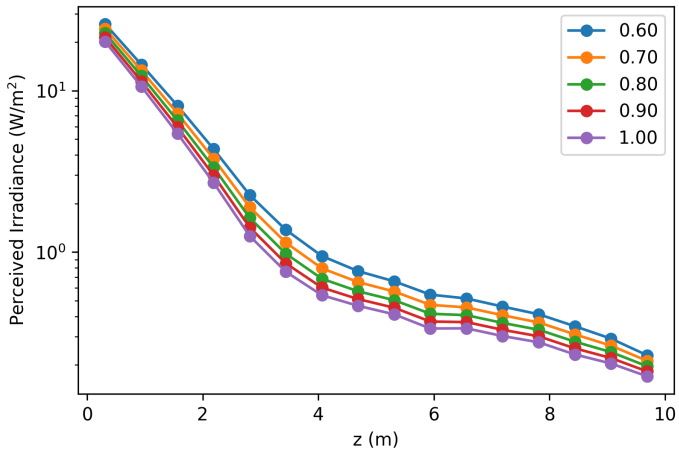

Figure 6.21: Several values of kelp absorptance

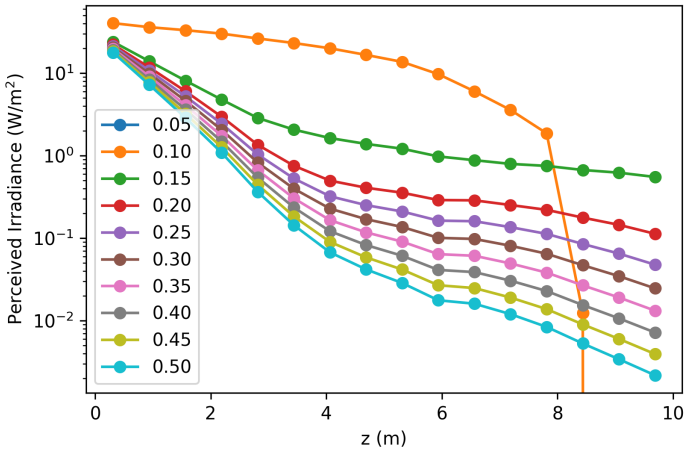

Figure 6.22: Several values of absorption coefficient of water

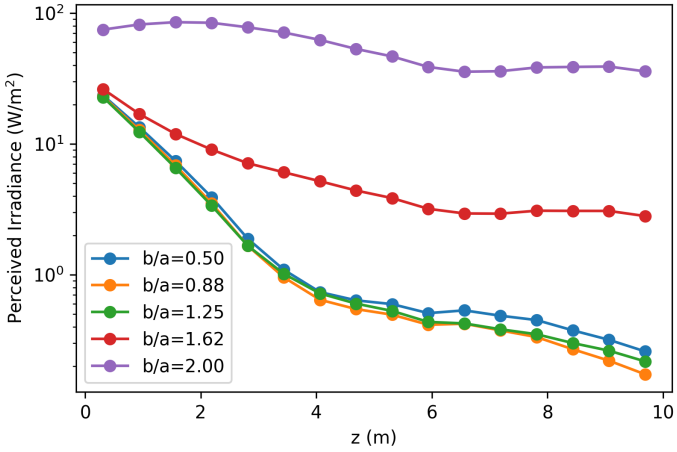

Figure 6.23: Several values of scattering coefficient

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. The asymptotic approximation is shown to converge to the finite difference solution in cases where the absorption coefficient is the same order of magnitude as the scattering coefficient or larger. Otherwise, the solution diverges.

Many aspects of the model have room for future improvement. The most pressing is probably the development of a model for long-lines, which is more popular in practice than the vertical lines studied here. Similar techniques can likely be applied, but the details will of course differ.

One major simplification in the calculation of the kelp model is the assumption that the fronds are perfectly horizontal. This could be improved in a straightforward way by including some probability distribution for the angular elevation as a function of current speed, similar to the study performed in [10]. The cost of implementing polar rotation is that depth layers are no longer isolated. Rather than integrating the two dimensional length-orientation distribution from Section 2.3.3 to

calculate the spatial kelp distribution, it would be necessary to perform a triple integral which includes the elevation distribution. Since frond elevation and azimuathal orientation are both related to current velocity, it would likely be impossible to ignore the remarks at the end of 2.3.3, and the assumption of independent distributions would have to be abandoned.

Of course, real fronds are not rotating planar kites, but have a very dynamic geometry. To consider out-of-plane frond bending would require a totally different approach. Whether or not any improved description of the seaweed would merit the substantial work is unclear.

BIBLIOGRAPHY

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

[2] A. H. Baker, E. R. Jessup, and T. Manteuffel. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, Jan. 2005.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[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. *Light and Water: Radiative Transfer in Natural Waters*. Academic Press, 1994.

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

[10] C. Norvik. Design of Artificial Seaweeds for Assessment of Hydrodynamic Properties of Seaweed Farms. 2017.

[11] 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.

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

[13] Y. Saad and M. H. Schults. GMRES: a Generalized Minimal Residual algorithm for solving nonsymmetric linear systems. Research Report YALEU/DCS/RR-254, Yale University, May 1985.

[14] 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.

[15] 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.

[16] P. Sonneveld and M. B. van Gijzen. IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. *SIAM Journal on Scientific Computing; Philadelphia*, 31(2):28, 2008.

[17] H. Van Der Vorst. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, Mar. 1992.

[18] 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.

[19] 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

GRID DETAILS

The width of the spatial grid cells in each dimension are

$$dx = \frac{x_{\max} - x_{\min}}{n_x},$$
$$dy = \frac{y_{\max} - y_{\min}}{n_y},$$
$$dz = \frac{z_{\max} - z_{\min}}{n_z}.$$

Denote the edges as

$$x_i^e = (i-1)dx \text{ for } i = 1, \ldots, n_x$$
$$y_j^e = (j-1)dy \text{ for } j = 1, \ldots, n_y$$
$$z_k^e = (k-1)dz \text{ for } k = 1, \ldots, n_z$$

and the cell centers as

$$x_i = (i-1/2)dx \text{ for } i = 1, \ldots, n_x$$
$$y_j = (j-1/2)dy \text{ for } j = 1, \ldots, n_y$$
$$z_k = (k-1/2)dz \text{ for } k = 1, \ldots, n_z$$

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

Now, we define the azimuthal angle such that

$$\theta_l = (l-1)d\theta.$$

For the sake of periodicity, we need

$$\theta_1 = 0,$$

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

which requires

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

For the polar angle, we similarly let

$$\phi_m = (m-1)d\phi$$

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

$$\phi_1 = 0,$$

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

This gives us

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

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

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

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

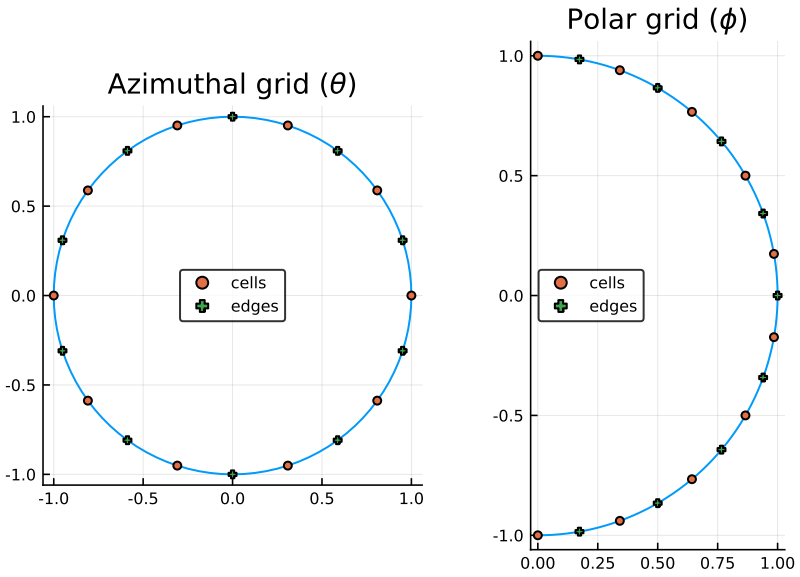

Figure A.1: Angular grid

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

The following notation is used.

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

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

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

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

Thus, it follows that

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

Accordingly, define

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

Further, we refer to the angular grid cell centered at $\boldsymbol{\omega}_p$ 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 infinitesimal differential and for finite grid spacing.

For the poles, we have

$$\begin{aligned}
|\Omega_1| = |\Omega_{n_{\boldsymbol{\omega}}}| &= \int_{\Omega_1} d\boldsymbol{\omega} \\
&= \int_0^{2\pi} \int_0^{d\phi/2} \sin\phi \, d\phi \, d\theta \\
&= 2\pi \cos\phi \Big|_{d\phi/2}^{0} \\
&= 2\pi(1 - \cos(d\phi/2))
\end{aligned}$$

And for all other angular grid cells,

$$\begin{aligned}
|\Omega_p| &= \int_{\Omega_p} d\boldsymbol{\omega} \\
&= \int_{\theta_l^e}^{\theta_{l+1}^e} \int_{\phi_m^e}^{\phi_{m+1}^e} \sin(\phi) \, d\phi \, d\theta \\
&= d\theta \int_{\phi_m^e}^{\phi_{m+1}^e} \sin(\phi) \, d\phi \\
&= d\theta \left(\cos(\phi_m^e) - \cos(\phi_{m+1}^e) \right).
\end{aligned}$$

APPENDIX B

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

Given

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

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

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

we have

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

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

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

$$\tag{B.7}$$

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

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

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

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

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

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

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

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

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 $x = x_{\min}$, but to $x = x_{\min} + dx$.

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

That is,

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

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

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

Then,

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

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

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

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

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

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

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

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).

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

* This doesn't quite make sense yet.

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

and

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

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 C

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

```
1  ! General utilities which might be useful in other settings
2  module utils
3  implicit none
4
5  ! Constants
6  double precision, parameter :: pi = 4.D0 * datan(1.D0)
7
8  contains
9
10 ! Determine base directory relative to current directory
11 ! by looking for Makefile, which is in the base dir
12 ! Assuming that this is executed from within the git repo.
13 function getbasedir()
14     implicit none
15
16     ! INPUTS:
17     ! Number of paths to check
18     integer, parameter :: numpaths = 3
19     ! Maximum length of path names
20     integer, parameter :: maxlength = numpaths * 2 - 1
21     ! Paths to check for Makefile
22     character(len=maxlength), parameter, dimension(numpaths) :: check_paths &
```

```
23              = (/ '.    ', '..   ', '../..' /)
24      ! Temporary path string
25      character(len=maxlength) tmp_path
26      ! Whether Makefile has been found yet
27      logical found
28      ! Path counter
29      integer ii
30      ! Lengths of paths
31      integer, dimension(numpaths) ::  pathlengths
32
33      ! OUTPUT:
34      ! getbasedir - relative path to base
          directory
35      ! Will either return '.', '..', or '../..'
36      character(len=maxlength) getbasedir
37
38
39      ! Determine length of each path
40      pathlengths(1) = 1
41      do ii = 2, numpaths
42          pathlengths(ii) = 2 + 3 * (ii - 2)
43      end do
44
45      ! Loop through paths
46      do ii = 1, numpaths
47          ! Determine this path
48          tmp_path = check_paths(ii)
49
50          ! Check whether Makefile is in this
              directory
51          !write(*,*) 'Checking "', tmp_path(1:
              pathlengths(ii)), '"'
52          inquire(file=tmp_path(1:pathlengths(ii))
                // '/Makefile', exist=found)
53          ! If so, stop. Otherwise, keep looking.
54          if(found) then
55              getbasedir = tmp_path(1:pathlengths(
                  ii))
56              exit
57          end if
58      end do
59
60      ! If it hasn't been found, then this script
          was probably called
61      ! from outside of the repository.
62      if(.not. found) then
63          write(*,*) 'BASE DIR NOT FOUND.'
64      end if
65
66  end function
67
68  ! Determine array size from min, max and step
```

```
69 ! If alignment is off, array will overstep the
       maximum
70 function bnd2max(xmin,xmax,dx)
71     implicit none
72
73     ! INPUTS:
74     ! xmin - minimum x value in array
75     ! xmax - maximum x value in array (inclusive
           )
76     ! dx - step size
77     double precision, intent(in) :: xmin, xmax,
           dx
78
79     ! OUTPUT:
80     ! step2max - maximum index of array
81     integer bnd2max
82
83     ! Calculate array size
84     bnd2max = int(ceiling((xmax-xmin)/dx))
85 end function
86
87 ! Create array from bounds and number of
       elements
88 ! xmax is not included in array
89 function bnd2arr(xmin,xmax,imax)
90     implicit none
91
92     ! INPUTS:
93     ! xmin - minimum x value in array
94     ! xmax - maximum x value in array (exclusive
           )
95     double precision, intent(in) :: xmin, xmax
96     ! imax - number of elements in array
97     integer imax
98
99     ! OUTPUT:
100    ! bnd2arr - array to generate
101    double precision, dimension(imax) :: bnd2arr
102
103    ! BODY:
104
105    ! Counter
106    integer ii
107    ! Step size
108    double precision dx
109
110    ! Calculate step size
111    dx = (xmax - xmin) / imax
112
113    ! Generate array
114    do ii = 1, imax
115        bnd2arr(ii) = xmin + (ii-1) * dx
```

```
116       end do
117
118   end function
119
120   function mod1(i, n)
121     implicit none
122     integer i, n, m
123     integer mod1
124
125     m = modulo(i, n)
126
127     if(m .eq. 0) then
128        mod1 = n
129     else
130        mod1 = m
131     end if
132
133   end function mod1
134
135   function sgn_int(x)
136     integer x, sgn_int
137     ! Standard signum function
138     sgn_int = sign(1,x)
139     if(x .eq. 0.) sgn_int = 0
140   end function sgn_int
141
142   function sgn(x)
143     double precision x, sgn
144     ! Standard signum function
145     sgn = sign(1.d0,x)
146     if(x .eq. 0.) sgn = 0
147   end function sgn
148
149   ! Interpolate single point from 1D data
150   function interp(x0,xx,yy,nn)
151       implicit none
152
153       ! INPUTS:
154       ! x0 - x value at which to interpolate
155       double precision, intent(in) :: x0
156       ! xx - ordered x values at which y data is
              sampled
157       ! yy - corresponding y values to interpolate
158       double precision, dimension (nn), intent(in)
              :: xx,yy
159       ! nn - length of data
160       integer, intent(in) :: nn
161
162       ! OUTPUT:
163       ! interp - interpolated y value
164       double precision interp
165
```

```
166     ! BODY:
167
168     ! Index of lower-adjacent data (xx(i) < x0 <
            xx(i+1))
169     integer ii
170     ! Slope of liine between (xx(ii),yy(ii)) and
            (xx(ii+1),yy(ii+1))
171     double precision mm
172
173     ! If out of bounds, then return endpoint
           value
174     if (x0 < xx(1)) then
175        interp = yy(1)
176     else if (x0 > xx(nn)) then
177        interp = yy(nn)
178     else
179
180       ! Determine ii
181       do ii = 1, nn
182           if (xx(ii) > x0) then
183               ! We've now gone one index too far
                      .
184               exit
185           end if
186       end do
187
188       ! Determine whether we're on the right
             endpoint
189       if(ii-1 < nn) then
190           ! If this is a legitimate
                 interpolation, then
191           ! subtract since we went one index too
                  far
192           ii = ii - 1
193
194           ! Calculate slope
195           mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
                  xx(ii))
196
197           ! Return interpolated value
198           interp = yy(ii) + mm * (x0 - xx(ii))
199       else
200           ! If we're actually interpolating the
                 right endpoint,
201           ! then just return it.
202           interp = yy(nn)
203       end if
204
205    end if
206
207 end function
208
```

```
209 ! Calculate unshifted position of periodic image
210 ! Assuming xmin, xmax are extreme attainable
      values of x
211 function shift_mod(x, xmin, xmax)
212   double precision x, xmin, xmax
213   double precision mod_part, shift_mod
214   mod_part = mod(x-xmin, xmax-xmin)
215   if(mod_part .ge. 0) then
216     ! In this case, mod_part is distance
        between image & lower bound
217     shift_mod = xmin + mod_part
218   else
219     ! In this case, mod_part is distance
        between image & upper bound
220     shift_mod = xmax + mod_part
221   endif
222 end function shift_mod
223
224 ! Bilinear interpolation on evenly spaced 2D
    grid
225 ! Assume upper endpoint is not included and is
    identical
226 ! to the lower endpoint, which is included.
227 function bilinear_array_periodic(x, y, nx, ny,
    x_vals, y_vals, fun_vals)
228   implicit none
229   double precision x, y
230   integer nx, ny
231   double precision, dimension(:) :: x_vals,
      y_vals
232   double precision, dimension(:,:) :: fun_vals
233
234   double precision dx, dy, xmin, ymin
235   integer i0, j0, i1, j1
236   double precision x0, x1, y0, y1
237   double precision z00, z10, z01, z11
238
239   double precision bilinear_array_periodic
240
241   xmin = x_vals(1)
242   ymin = y_vals(1)
243   dx = x_vals(2) - x_vals(1)
244   dy = y_vals(2) - y_vals(1)
245
246   ! Add 1 for one-indexing
247   i0 = int(floor((x-xmin)/dx))+1
248   j0 = int(floor((y-ymin)/dy))+1
249
250   x0 = x_vals(i0)
251   y0 = y_vals(j0)
252
253   ! Periodic wrap
```

```
254 |   if(i0 .lt. nx) then
255 |      i1 = i0 + 1
256 |      x1 = x_vals(i1)
257 |   else
258 |      i1 = 1
259 |      x1 = x_vals(nx) + dx
260 |   endif
261 |
262 |   if(j0 .lt. ny) then
263 |      j1 = j0 + 1
264 |      y1 = y_vals(j1)
265 |   else
266 |      j1 = 1
267 |      y1 = y_vals(ny) + dy
268 |   endif
269 |
270 |   z00 = fun_vals(i0,j0)
271 |   z10 = fun_vals(i1,j0)
272 |   z01 = fun_vals(i0,j1)
273 |   z11 = fun_vals(i1,j1)
274 |
275 |   bilinear_array_periodic = bilinear(x, y, x0,
    |      y0, x1, y1, z00, z01, z10, z11)
276 | end function bilinear_array_periodic
277 |
278 | ! Bilinear interpolation on evenly spaced 2D
    |   grid
279 | ! Assume upper and lower endpoints are included
280 | function bilinear_array(x, y, x_vals, y_vals,
    |   fun_vals)
281 |   implicit none
282 |   double precision x, y
283 |   double precision, dimension(:) :: x_vals,
    |      y_vals
284 |   double precision, dimension(:,:) :: fun_vals
285 |
286 |   double precision dx, dy, xmin, ymin
287 |   integer i0, j0, i1, j1
288 |   double precision x0, x1, y0, y1
289 |   double precision z00, z10, z01, z11
290 |
291 |   double precision bilinear_array
292 |
293 |   xmin = x_vals(1)
294 |   ymin = y_vals(1)
295 |   dx = x_vals(2) - x_vals(1)
296 |   dy = y_vals(2) - y_vals(1)
297 |
298 |   ! Add 1 for one-indexing
299 |   i0 = int(floor((x-xmin)/dx))+1
300 |   j0 = int(floor((y-ymin)/dy))+1
```

```
301   i1 = i0 + 1
302   j1 = j0 + 1
303
304   ! Bounds checking
305   ! if(i0 .lt. 1) then
306   !    i0 = 1
307   !    i1 = 1
308   ! else if(i1 .gt. nx) then
309   !    i0 = nx
310   !    i1 = nx
311   ! endif
312   ! if(j0 .lt. 1) then
313   !    j0 = 1
314   !    j1 = 1
315   ! else if(j1 .gt. ny) then
316   !    j0 = ny
317   !    j1 = ny
318   ! endif
319
320   x0 = x_vals(i0)
321   x1 = x_vals(i1)
322   y0 = y_vals(j0)
323   y1 = y_vals(j1)
324
325   z00 = fun_vals(i0,j0)
326   z10 = fun_vals(i1,j0)
327   z01 = fun_vals(i0,j1)
328   z11 = fun_vals(i1,j1)
329
330   bilinear_array = bilinear(x, y, x0, y0, x1, y1
        , z00, z01, z10, z11)
331 end function bilinear_array
332
333 ! ilinear interpolation of a function of two
      variables
334 ! over a rectangle of points.
335 ! Weight each point by the area of the sub-
      rectangle involving
336 ! the point (x,y) and the point diagonally
      across the rectangle
337 function bilinear(x, y, x0, y0, x1, y1, z00, z01
      , z10, z11)
338   implicit none
339   double precision x, y
340   double precision x0, y0, x1, y1, z00, z01, z10
        , z11
341   double precision a, b, c, d
342   double precision bilinear
343
344   a = (x-x0)*(y-y0)
345   b = (x1-x)*(y-y0)
```

```
346 |   c = (x-x0)*(y1-y)
347 |   d = (x1-x)*(y1-y)
348 |
349 |   bilinear = (a*z11 + b*z01 + c*z10 + d*z00) / (
    |     a + b + c + d)
350 | end function bilinear
351 |
352 | ! Integrate using left endpoint rule
353 | ! Assuming the right endpoint is not included in
    |   arr
354 | function lep_rule(arr,dx,nn)
355 |     implicit none
356 |
357 |     ! INPUTS:
358 |     ! arr - array to integrate
359 |     double precision, dimension(nn) :: arr
360 |     ! dx - array spacing (mesh size)
361 |     double precision dx
362 |     ! nn - length of arr
363 |     integer, intent(in) :: nn
364 |
365 |     ! OUTPUT:
366 |     ! lep_rule - integral w/ left endpoint rule
367 |     double precision lep_rule
368 |
369 |     ! BODY:
370 |
371 |     ! Counter
372 |     integer ii
373 |
374 |     ! Set output to zero
375 |     lep_rule = 0.0d0
376 |
377 |     ! Accumulate integral
378 |     do ii = 1, nn
379 |         lep_rule = lep_rule + arr(ii) * dx
380 |     end do
381 |
382 | end function
383 |
384 | ! Integrate using trapezoid rule
385 | ! Assuming both endpoints are included in arr
386 | function trap_rule_dx(arr, dx, nn)
387 |   implicit none
388 |   double precision, dimension(nn) :: arr
389 |   double precision dx
390 |   integer ii, nn
391 |   double precision trap_rule_dx
392 |
393 |   trap_rule_dx = 0.0d0
394 |
395 |   do ii=1, nn-1
```

```
396       trap_rule_dx = trap_rule_dx + 0.5d0 * dx *
              (arr(ii) + arr(ii+1))
397     end do
398
399   end function trap_rule_dx
400
401   ! Integrate using trapezoid rule
402   ! Assuming both endpoints are included in arr
403   function trap_rule_uneven(xx, yy, nn)
404     implicit none
405     double precision, dimension(nn) :: xx
406     double precision, dimension(nn) :: yy
407     integer ii, nn
408     double precision trap_rule_uneven
409
410     trap_rule_uneven = 0.0d0
411
412     do ii=1, nn-1
413        trap_rule_uneven = trap_rule_uneven + 0.5d0
               * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii
              +1))
414     end do
415   end function trap_rule_uneven
416
417   function trap_rule_dx_uneven(dx, yy, nn)
418     implicit none
419     double precision, dimension(nn-1) :: dx
420     double precision, dimension(nn) :: yy
421     integer ii, nn
422     double precision trap_rule_dx_uneven
423
424     trap_rule_dx_uneven = 0.0d0
425
426     do ii=1, nn-1
427        trap_rule_dx_uneven = trap_rule_dx_uneven +
               0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
428     end do
429   end function trap_rule_dx_uneven
430
431   ! Integrate using midpoint rule
432   ! First and last bins, only use inner half
433   function midpoint_rule_halfends(dx, yy, nn)
          result(integral)
434     implicit none
435     integer ii, nn
436     double precision, dimension(nn) :: dx, yy
437     double precision integral
438
439     if(nn > 1) then
440       integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(
              nn))
441
```

```
442 |     do ii=2, nn-1
443 |       integral = integral + dx(ii)*yy(ii)
444 |     end do
445 |   else
446 |     integral = 0.d0
447 |   end if
448 | end function midpoint_rule_halfends
449 |
450 | ! Normalize 1D array and return integral w/ left
    |     endpoint rule
451 | function normalize_dx(arr,dx,nn)
452 |     implicit none
453 |
454 |     ! INPUTS:
455 |     ! arr - array to normalize
456 |     double precision, dimension(nn) :: arr
457 |     ! dx - array spacing (mesh size)
458 |     double precision dx
459 |     ! nn - length of arr
460 |     integer, intent(in) :: nn
461 |
462 |     ! OUTPUT:
463 |     ! normalize - integral before normalization
    |        (left endpoint rule)
464 |     double precision normalize_dx
465 |
466 |     ! BODY:
467 |
468 |     ! Calculate integral
469 |     normalize_dx = lep_rule(arr,dx,nn)
470 |
471 |     ! Normalize array
472 |     arr = arr / normalize_dx
473 |
474 |   end function normalize_dx
475 |
476 | ! Normalize 1D unevenly-spaced array and
477 | ! return integral w/ trapezoid rule
478 | ! Will not be quite accurate if rightmost
    |    endpoint is not included
479 | ! (Very small for VSF, so not a big deal there)
480 | ! Modifies yy in place
481 | function normalize_uneven(xx, yy, nn) result(
    |    norm)
482 |   implicit none
483 |
484 |   ! INPUTS:
485 |   ! xx, yy - array values of data to normalize
486 |   double precision, dimension(nn) :: xx, yy
487 |   ! nn - length of arr
488 |   integer, intent(in) :: nn
489 |
```

```
490 |   ! OUTPUT:
491 |   ! normalize - integral before normalization (
    |      left endpoint rule)
492 |   double precision norm
493 |
494 |   ! BODY:
495 |
496 |   ! Calculate integral
497 |   ! PERHAPS WE SHOULD USE TRAPEZOID RULE
498 |   norm = trap_rule_uneven(xx, yy, nn)
499 |
500 |   ! Normalize array
501 |   yy(:) = yy(:) / norm
502 |
503 |end function normalize_uneven
504 |
505 |! Read 2D array from file
506 |function read_array(filename,fmtstr,nn,mm,
    |   skiplines_in)
507 |     implicit none
508 |
509 |     ! INPUTS:
510 |     ! filename - path to file to be read
511 |     ! fmtstr - input format (no parentheses, don
    |        't specify columns)
512 |     ! e.g. 'E10.2', not '(2E10.2)'
513 |     character(len=*), intent(in) :: filename,
    |        fmtstr
514 |     ! nn - Number of data rows in file
515 |     ! mm - number of data columns in file
516 |     integer, intent(in) :: nn, mm
517 |     ! skiplines - optional - number of lines to
    |        skip from header
518 |     integer, optional :: skiplines_in
519 |     integer skiplines
520 |
521 |     ! OUTPUT:
522 |     double precision, dimension(nn,mm) ::
    |        read_array
523 |
524 |     ! BODY:
525 |
526 |     ! Row counter
527 |     integer ii
528 |     ! File unit number
529 |     integer, parameter :: un = 10
530 |     ! Final format to use
531 |     character(len=256) finfmt
532 |
533 |     ! Generate final format string
534 |     write(finfmt,'(A,I1,A,A)') '(', mm, fmtstr,
    |        ')'
```

```
535 |
536 |     ! Print message
537 |     !write(*,*) 'Reading data from "', trim(
    |       filename), '"'
538 |     !write(*,*) 'using format "', trim(finfmt),
    |       '"'
539 |
540 |     ! Open file
541 |     open(unit=un, file=trim(filename), status='
    |       old', form='formatted')
542 |
543 |     ! Skip lines if desired
544 |     if(present(skiplines_in)) then
545 |         skiplines = skiplines_in
546 |         do ii = 1, skiplines
547 |             ! Read without variable ignores the
    |                line
548 |             read(un,*)
549 |         end do
550 |     else
551 |         skiplines = 0
552 |     end if
553 |
554 |     ! Loop through lines
555 |     do ii = 1, nn
556 |         ! Read one row at a time
557 |         read(unit=un, fmt=trim(finfmt))
    |           read_array(ii,:)
558 |     end do
559 |
560 |     ! Close file
561 |     close(unit=un)
562 |
563 | end function
564 |
565 | ! Print 2D array to stdout
566 | subroutine print_int_array(arr,nn,mm,fmtstr_in)
567 |   implicit none
568 |
569 |   ! INPUTS:
570 |   ! arr - array to print
571 |   integer, dimension (nn,mm), intent(in) :: arr
572 |   ! nn - number of data rows in file
573 |   ! nn - number of data columns in file
574 |   integer, intent(in) :: nn, mm
575 |   ! fmtstr - output format (no parentheses, don'
    |     t specify columns)
576 |   ! e.g. 'E10.2', not '(2E10.2)'
577 |   character(len=*), optional :: fmtstr_in
578 |   character(len=256) fmtstr
579 |
580 |   ! NO OUTPUTS
```

```
581 |
582 |    ! BODY
583 |
584 |    ! Row counter
585 |    integer ii
586 |    ! Final format to use
587 |    character(len=256) finfmt
588 |
589 |    ! Determine string format
590 |    if(present(fmtstr_in)) then
591 |       fmtstr = fmtstr_in
592 |    else
593 |       fmtstr = 'I10'
594 |    end if
595 |
596 |    ! Generate final format string
597 |    write(finfmt,'(A,I4,A,A)') '(', mm, trim(
    |       fmtstr), ')'
598 |
599 |    ! Loop through rows
600 |    do ii = 1, nn
601 |       ! Print one row at a time
602 |       write(*,finfmt) arr(ii,:)
603 |    end do
604 |
605 |    ! Print blank line after
606 |    write(*,*) ' '
607 |
608 | end subroutine print_int_array
609 |
610 | subroutine print_array(arr,nn,mm,fmtstr_in)
611 |     implicit none
612 |
613 |     ! INPUTS:
614 |     ! arr - array to print
615 |     double precision, dimension (nn,mm), intent(
    |        in) :: arr
616 |     ! nn - number of data rows in file
617 |     ! nn - number of data columns in file
618 |     integer, intent(in) :: nn, mm
619 |     ! fmtstr - output format (no parentheses,
    |        don't specify columns)
620 |     ! e.g. 'E10.2', not '(2E10.2)'
621 |     character(len=*), optional :: fmtstr_in
622 |     character(len=256) fmtstr
623 |
624 |     ! NO OUTPUTS
625 |
626 |     ! BODY
627 |
628 |     ! Row counter
629 |     integer ii
```

```
630 |     ! Final format to use
631 |     character(len=256) finfmt
632 |
633 |     ! Determine string format
634 |     if(present(fmtstr_in)) then
635 |         fmtstr = fmtstr_in
636 |     else
637 |         fmtstr = 'ES10.2'
638 |     end if
639 |
640 |     ! Generate final format string
641 |     write(finfmt,'(A,I4,A,A)') '(', mm, trim(
    |        fmtstr), ')'
642 |
643 |     ! Loop through rows
644 |     do ii = 1, nn
645 |         ! Include row number
646 |         !write(*,'(I10)', advance='no') ii
647 |         ! Print one row at a time
648 |         write(*,finfmt) arr(ii,:)
649 |     end do
650 |
651 |     ! Print blank line after
652 |     write(*,*) ' '
653 |
654 | end subroutine
655 |
656 | ! Write 2D array to file
657 | subroutine write_array(arr,nn,mm,filename,
    |    fmtstr_in)
658 |     implicit none
659 |
660 |     ! INPUTS:
661 |     ! arr - array to print
662 |     double precision, dimension (nn,mm), intent(
    |        in) :: arr
663 |     ! nn - number of data rows in file
664 |     ! nn - number of data columns in file
665 |     integer, intent(in) :: nn, mm
666 |     ! filename - file to write to
667 |     character(len=*) filename
668 |     ! fmtstr - output format (no parentheses,
    |        don't specify columns)
669 |     ! e.g. 'E10.2', not '(2E10.2)'
670 |     character(len=*), optional :: fmtstr_in
671 |     character(len=256) fmtstr
672 |
673 |     ! NO OUTPUTS
674 |
675 |     ! BODY
676 |
677 |     ! Row counter
```

```
678     integer ii
679     ! Final format to use
680     character(len=256) finfmt
681     ! Dummy file unit to use
682     integer, parameter :: un = 20
683 
684     ! Open file for writing
685     open(unit=un, file=trim(filename), status='
        replace', form='formatted')
686 
687     ! Determine string format
688     if(present(fmtstr_in)) then
689         fmtstr = fmtstr_in
690     else
691         fmtstr = 'E10.2'
692     end if
693 
694     ! Generate final format string
695     write(finfmt,'(A,I4,A,A)') '(', mm, trim(
        fmtstr), ')'
696 
697     ! Loop through rows
698     do ii = 1, nn
699         ! Print one row at a time
700         write(un,finfmt) arr(ii,:)
701     end do
702 
703     ! Close file
704     close(unit=un)
705 
706 end subroutine
707 
708 subroutine zeros(x, n)
709   implicit none
710   integer n, i
711   double precision, dimension(n) :: x
712 
713   do i=1, n
714      x(i) = 0
715   end do
716 end subroutine zeros
717 
718 end module
```

sag.f90

```
1 module sag
2 use utils
3 use fastgl
4 
5 implicit none
6 
```

```
7  ! Spatial grids do not include upper endpoints.
8  ! Angular grids do include upper endpoints.
9  ! Both include lower endpoints.
10
11 ! To use:
12 ! call grid%set_bounds(...)
13 ! call grid%set_num(...) (or set_uniform_spacing
   )
14 ! call grid%init()
15 ! ...
16 ! call grid%deinit()
17
18 !integer, parameter :: pi = 3.141592653589793D
   +00
19
20 type index_list
21    integer i, j, k, p
22  contains
23    procedure :: init => index_list_init
24    procedure :: print => index_list_print
25 end type index_list
26
27 type angle2d
28    integer ntheta, nphi, nomega
29    double precision dtheta, dphi
30    double precision, dimension(:), allocatable
         :: theta, phi, theta_edge, phi_edge
31    double precision, dimension(:), allocatable
         ::  theta_p, phi_p, theta_edge_p,
         phi_edge_p
32    double precision, dimension(:), allocatable
         :: cos_theta, sin_theta, cos_phi, sin_phi
33    double precision, dimension(:), allocatable
         :: cos_theta_edge, sin_theta_edge,
         cos_phi_edge, sin_phi_edge
34    double precision, dimension(:), allocatable
         :: cos_theta_p, sin_theta_p, cos_phi_p,
         sin_phi_p
35    double precision, dimension(:), allocatable
         :: cos_theta_edge_p, sin_theta_edge_p,
         cos_phi_edge_p, sin_phi_edge_p
36    double precision, dimension(:), allocatable
         :: area_p
37  contains
38    procedure :: set_num => angle_set_num
39    procedure :: phat, lhat, mhat
40    procedure :: init => angle_init ! Call after
         set_num
41    procedure :: integrate_points =>
         angle_integrate_points
42    procedure :: integrate_func =>
         angle_integrate_func
```

```
43    procedure :: deinit => angle_deinit
44  end type angle2d
45
46  type angle_dim
47    integer num
48    double precision minval, maxval, prefactor
49    double precision, dimension(:), allocatable
        :: vals, weights, sin, cos
50   contains
51    procedure :: set_bounds => angle_set_bounds
52    procedure :: set_num => angle1d_set_num
53    procedure :: deinit => angle1d_deinit
54    procedure :: integrate_points =>
        angle1d_integrate_points
55    procedure :: integrate_func =>
        angle1d_integrate_func
56    procedure :: assign_linspace =>
        angle1d_assign_linspace
57    procedure :: assign_legendre
58  end type angle_dim
59
60  type space_dim
61    integer num
62    double precision minval, maxval
63    double precision, dimension(:), allocatable
        :: vals, edges, spacing
64   contains
65    procedure :: integrate_points =>
        space_integrate_points
66    procedure :: trapezoid_rule
67    procedure :: set_bounds => space_set_bounds
68    procedure :: set_num => space_set_num
69    procedure :: set_uniform_spacing =>
        space_set_uniform_spacing
70    !procedure :: set_num_from_spacing
71    procedure :: set_uniform_spacing_from_num
72    procedure :: set_spacing_array =>
        space_set_spacing_array
73    procedure :: deinit => space_deinit
74    procedure :: assign_linspace
75  end type space_dim
76
77  type space_angle_grid !(sag)
78    type(space_dim) :: x, y, z
79    type(angle2d) :: angles
80    double precision, dimension(:), allocatable ::
        x_factor, y_factor
81  contains
82    procedure :: set_bounds => sag_set_bounds
83    procedure :: set_num => sag_set_num
84    procedure :: init => sag_init
85    procedure :: deinit => sag_deinit
```

```
86    !procedure :: set_num_from_spacing =>
        sag_set_num_from_spacing
87    procedure :: set_uniform_spacing_from_num =>
        sag_set_uniform_spacing_from_num
88    procedure :: calculate_factors =>
        sag_calculate_factors
89  end type space_angle_grid
90
91  contains
92
93    subroutine index_list_init(indices)
94      class(index_list) indices
95      indices%i = 1
96      indices%j = 1
97      indices%k = 1
98      indices%p = 1
99    end subroutine
100
101   subroutine index_list_print(indices)
102     class(index_list) indices
103
104     write(*,*) 'i, j, k, p =', indices%i,
          indices%j, indices%k, indices%p
105   end subroutine index_list_print
106
107   subroutine angle_set_num(angles, ntheta, nphi)
108     class(angle2d) :: angles
109     integer ntheta, nphi
110     angles%ntheta = ntheta
111     angles%nphi = nphi
112     angles%nomega = ntheta*(nphi-2) + 2
113   end subroutine angle_set_num
114
115   function lhat(angles, p) result(l)
116     class(angle2d) :: angles
117     integer l, p
118     if(p .eq. 1) then
119        l = 1
120     else if(p .eq. angles%nomega) then
121        l = 1
122     else
123        l = mod1(p-1, angles%ntheta)
124    end if
125   end function lhat
126
127   function mhat(angles, p) result(m)
128     class(angle2d) :: angles
129     integer m, p
130     if(p .eq. 1) then
131        m = 1
132     else if(p .eq. angles%nomega) then
```

```
133         m = angles%nphi
134       else
135         m = ceiling(dble(p-1)/dble(angles%ntheta)
              ) + 1
136       end if
137     end function mhat
138
139     function phat(angles, l, m) result(p)
140       class(angle2d) :: angles
141       integer l, m, p
142
143       if(m .eq. 1) then
144         p = 1
145       else if(m .eq. angles%nphi) then
146         p = angles%nomega
147       else
148         p = (m-2)*angles%ntheta + l + 1
149       end if
150     end function phat
151
152     subroutine angle_init(angles)
153       class(angle2d) :: angles
154       integer l, m, p
155       double precision area
156
157
158       ! TODO: CONSIDER REMOVING non-p
159       allocate(angles%theta(angles%ntheta))
160       allocate(angles%phi(angles%nphi))
161       allocate(angles%theta_edge(angles%ntheta))
162       allocate(angles%phi_edge(angles%nphi-1))
163       allocate(angles%theta_p(angles%nomega))
164       allocate(angles%phi_p(angles%nomega))
165       allocate(angles%theta_edge_p(angles%nomega))
166       allocate(angles%phi_edge_p(angles%nomega))
167       allocate(angles%cos_theta_p(angles%nomega))
168       allocate(angles%sin_theta_p(angles%nomega))
169       allocate(angles%cos_phi_p(angles%nomega))
170       allocate(angles%sin_phi_p(angles%nomega))
171       allocate(angles%cos_theta(angles%nomega))
172       allocate(angles%sin_theta(angles%nomega))
173       allocate(angles%cos_phi(angles%nomega))
174       allocate(angles%sin_phi(angles%nomega))
175       allocate(angles%cos_theta_edge(angles%ntheta
            ))
176       allocate(angles%sin_theta_edge(angles%ntheta
            ))
177       allocate(angles%cos_phi_edge(angles%nphi-1))
178       allocate(angles%sin_phi_edge(angles%nphi-1))
```

```
179 |     allocate(angles%cos_theta_edge_p(angles%
    |        nomega))
180 |     allocate(angles%sin_theta_edge_p(angles%
    |        nomega))
181 |     allocate(angles%cos_phi_edge_p(angles%nomega
    |        -1))
182 |     allocate(angles%sin_phi_edge_p(angles%nomega
    |        -1))
183 |     allocate(angles%area_p(angles%nomega))
184 |
185 |     ! Calculate spacing
186 |     angles%dtheta = 2.d0*pi/dble(angles%ntheta)
187 |     angles%dphi = pi/dble(angles%nphi-1)
188 |
189 |     ! Create grids
190 |     do l=1, angles%ntheta
191 |        angles%theta(l) = dble(l-1)*angles%dtheta
192 |        angles%cos_theta(l) = cos(angles%theta(l)
    |           )
193 |        angles%sin_theta(l) = sin(angles%theta(l)
    |           )
194 |        angles%theta_edge(l) = dble(l-0.5d0)*
    |           angles%dtheta
195 |        angles%cos_theta_edge(l) = cos(angles%
    |           theta_edge(l))
196 |        angles%sin_theta_edge(l) = sin(angles%
    |           theta_edge(l))
197 |     end do
198 |
199 |     do m=1, angles%nphi
200 |        angles%phi(m) = dble(m-1.d0)*angles%dphi
201 |        angles%cos_phi(m) = cos(angles%phi(m))
202 |        angles%sin_phi(m) = sin(angles%phi(m))
203 |        if(m<angles%nphi) then
204 |           angles%phi_edge(m) = dble(m-0.5d0)*
    |              angles%dphi
205 |           angles%cos_phi_edge(m) = cos(angles%
    |              phi_edge(m))
206 |           angles%sin_phi_edge(m) = sin(angles%
    |              phi_edge(m))
207 |        end if
208 |     end do
209 |
210 |     ! Create p arrays
211 |     do m=2, angles%nphi-1
212 |        area = angles%dtheta &
213 |             * (angles%cos_phi_edge(m-1) - angles
    |                %cos_phi_edge(m))
214 |        do l=1, angles%ntheta
```

```
215 |           p = angles%phat(l, m)
216 |
217 |           angles%theta_p(p) = angles%theta(l)
218 |           angles%phi_p(p) = angles%phi(m)
219 |           angles%theta_edge_p(p) = angles%
    |              theta_edge(l)
220 |           angles%phi_edge_p(p) = angles%phi_edge
    |              (m)
221 |
222 |           angles%cos_theta_p(p) = cos(angles%
    |              theta_p(p))
223 |           angles%sin_theta_p(p) = sin(angles%
    |              theta_p(p))
224 |           angles%cos_phi_p(p) = cos(angles%phi_p
    |              (p))
225 |           angles%sin_phi_p(p) = sin(angles%phi_p
    |              (p))
226 |
227 |           angles%cos_theta_edge_p(p) = cos(
    |              angles%theta_edge_p(p))
228 |           angles%sin_theta_edge_p(p) = sin(
    |              angles%theta_edge_p(p))
229 |           angles%cos_phi_edge_p(p) = cos(angles%
    |              phi_edge_p(p))
230 |           angles%sin_phi_edge_p(p) = sin(angles%
    |              phi_edge_p(p))
231 |
232 |           angles%area_p(p) = area
233 |        end do
234 |     end do
235 |
236 |     ! Poles
237 |     l=1
238 |     area = 2.d0*pi*(1.d0-cos(angles%dphi/2.d0))
239 |
240 |     ! North Pole
241 |     p = 1
242 |     m=1
243 |     angles%theta_p(p) = angles%theta(l)
244 |     angles%theta_edge_p(p) = angles%theta_edge(l
    |        )
245 |     angles%phi_p(p) = angles%phi(m)
246 |     ! phi_edge_p only defined up to nphi-1.
247 |     angles%phi_edge_p(p) = angles%phi_edge(m)
248 |     angles%cos_theta_p(p) = cos(angles%theta_p(p
    |        ))
249 |     angles%sin_theta_p(p) = sin(angles%theta_p(p
    |        ))
250 |     angles%cos_phi_p(p) = cos(angles%phi_p(p))
251 |     angles%sin_phi_p(p) = sin(angles%phi_p(p))
```

```
252 |     angles%cos_theta_edge_p(p) = cos(angles%
    |       theta_edge_p(p))
253 |     angles%sin_theta_edge_p(p) = sin(angles%
    |       theta_edge_p(p))
254 |     angles%cos_phi_edge_p(p) = cos(angles%
    |       phi_edge_p(p))
255 |     angles%sin_phi_edge_p(p) = sin(angles%
    |       phi_edge_p(p))
256 |     angles%area_p(p) = area
257 |
258 |     ! South Pole
259 |     p = angles%nomega
260 |     m = angles%nphi
261 |     angles%theta_p(p) = angles%theta(1)
262 |     angles%theta_edge_p(p) = angles%theta_edge(1
    |       )
263 |     angles%phi_p(p) = angles%phi(m)
264 |     angles%cos_theta_p(p) = cos(angles%theta_p(p
    |       ))
265 |     angles%sin_theta_p(p) = sin(angles%theta_p(p
    |       ))
266 |     angles%cos_phi_p(p) = cos(angles%phi_p(p))
267 |     angles%sin_phi_p(p) = sin(angles%phi_p(p))
268 |     angles%area_p(p) = area
269 |   end subroutine angle_init
270 |
271 |   ! Integrate function given function values at
    |      grid cells
272 |   function angle_integrate_points(angles,
    |      func_vals) result(integral)
273 |     class(angle2d) :: angles
274 |     double precision, dimension(angles%nomega)
    |       :: func_vals
275 |     double precision integral
276 |     integer p
277 |
278 |     integral = 0.d0
279 |
280 |     do p=1, angles%nomega
281 |        integral = integral + angles%area_p(p) *
    |          func_vals(p)
282 |     end do
283 |
284 |   end function angle_integrate_points
285 |
286 |   function angle_integrate_func(angles,
    |      func_callable) result(integral)
287 |     class(angle2d) :: angles
288 |     double precision, external :: func_callable
```

```
289     double precision, dimension(:), allocatable
          :: func_vals
290     double precision integral
291     integer p
292     double precision theta, phi
293
294     allocate(func_vals(angles%nomega))
295
296     do p=1, angles%nomega
297        theta = angles%theta_p(p)
298        phi = angles%phi_p(p)
299        func_vals(p) = func_callable(theta, phi)
300     end do
301
302     integral = angles%integrate_points(func_vals
          )
303
304     deallocate(func_vals)
305   end function angle_integrate_func
306
307   subroutine angle_deinit(angles)
308     class(angle2d) :: angles
309     deallocate(angles%theta)
310     deallocate(angles%phi)
311     deallocate(angles%theta_edge)
312     deallocate(angles%phi_edge)
313     deallocate(angles%theta_p)
314     deallocate(angles%phi_p)
315     deallocate(angles%theta_edge_p)
316     deallocate(angles%phi_edge_p)
317     deallocate(angles%cos_theta)
318     deallocate(angles%sin_theta)
319     deallocate(angles%cos_phi)
320     deallocate(angles%sin_phi)
321     deallocate(angles%cos_theta_p)
322     deallocate(angles%sin_theta_p)
323     deallocate(angles%cos_phi_p)
324     deallocate(angles%sin_phi_p)
325     deallocate(angles%cos_theta_edge)
326     deallocate(angles%sin_theta_edge)
327     deallocate(angles%cos_phi_edge)
328     deallocate(angles%sin_phi_edge)
329     deallocate(angles%cos_theta_edge_p)
330     deallocate(angles%sin_theta_edge_p)
331     deallocate(angles%cos_phi_edge_p)
332     deallocate(angles%sin_phi_edge_p)
333     deallocate(angles%area_p)
334   end subroutine angle_deinit
335
336
```

```
337   !!! ANGLE 1D !!!
338
339   subroutine angle_set_bounds(angle, minval,
          maxval)
340     class(angle_dim) :: angle
341     double precision minval, maxval
342     angle%minval = minval
343     angle%maxval = maxval
344   end subroutine angle_set_bounds
345
346   subroutine angle1d_set_num(angle, num)
347     class(angle_dim) :: angle
348     integer num
349     angle%num = num
350   end subroutine angle1d_set_num
351
352   subroutine angle1d_assign_linspace(angle)
353     class(angle_dim) :: angle
354     double precision spacing
355     integer i
356
357     spacing = (angle%maxval - angle%minval) /
            dble(angle%num)
358     do i=1, angle%num
359        angle%vals(i) = (i-1) * spacing
360     end do
361   end subroutine angle1d_assign_linspace
362
363   ! To calculate \int_{xmin}^{xmax} f(x) dx :
364   ! int = prefactor * sum(weights * f(roots))
365   subroutine assign_legendre(angle)
366     class(angle_dim) :: angle
367     double precision root, weight, theta
368     integer i
369     ! glpair produces both x and theta, where x=
          cos(theta). We'll throw out theta.
370
371     allocate(angle%vals(angle%num))
372     allocate(angle%weights(angle%num))
373     allocate(angle%sin(angle%num))
374     allocate(angle%cos(angle%num))
375
376     ! Prefactor for integration
377     ! From change of variables
378     angle%prefactor = (angle%maxval - angle%
          minval) / 2.d0
379
380     do i = 1, angle%num
381        call glpair(angle%num, i, theta, weight,
             root)
```

```
382         call affine_transform(root, -1.d0, 1.d0,
                angle%minval, angle%maxval)
383         angle%vals(i) = root
384         angle%weights(i) = weight
385         angle%sin(i) = sin(root)
386         angle%cos(i) = cos(root)
387       end do
388
389     end subroutine assign_legendre
390
391     ! Integrate callable function over angle via
          Gauss-Legendre quadrature
392
393     function angle1d_integrate_func(angle,
          func_callable) result(integral)
394       class(angle_dim) :: angle
395       double precision, external :: func_callable
396       double precision, dimension(:), allocatable
            :: func_vals
397       double precision integral
398       integer i
399
400       allocate(func_vals(angle%num))
401
402       do i=1, angle%num
403         func_vals(i) = func_callable(angle%vals(i
              ))
404       end do
405
406       integral = angle%integrate_points(func_vals)
407
408       deallocate(func_vals)
409     end function angle1d_integrate_func
410
411     ! Integrate function given function values
          sampled at legendre theta values
412     function angle1d_integrate_points(angle,
          func_vals) result(integral)
413       class(angle_dim) :: angle
414       double precision, dimension(angle%num) ::
            func_vals
415       double precision integral
416
417       integral = angle%prefactor * sum(angle%
            weights * func_vals)
418     end function angle1d_integrate_points
419
420     subroutine angle1d_deinit(angle)
421       class(angle_dim) :: angle
422       deallocate(angle%vals)
423       deallocate(angle%weights)
```

```
424 |     deallocate(angle%sin)
425 |     deallocate(angle%cos)
426 |   end subroutine angle1d_deinit
427 |
428 |
429 |   !! SPACE !!
430 |
431 |   ! Integrate function given function values
    |      sampled at even grid points
432 |   function space_integrate_points(space,
    |      func_vals) result(integral)
433 |     class(space_dim) :: space
434 |     double precision, dimension(space%num) ::
    |        func_vals
435 |     double precision integral
436 |
437 |     ! Encapsulate actual method for easy
    |        switching
438 |     integral = space%trapezoid_rule(func_vals)
439 |
440 |   end function space_integrate_points
441 |
442 |   function trapezoid_rule(space, func_vals)
    |      result(integral)
443 |     class(space_dim) :: space
444 |     double precision, dimension(space%num) ::
    |        func_vals
445 |     double precision integral
446 |
447 |     integral = 0.5d0 * sum(func_vals * space%
    |        spacing)
448 |   end function
449 |
450 |   subroutine space_set_bounds(space, minval,
    |      maxval)
451 |     class(space_dim) :: space
452 |     double precision minval, maxval
453 |     space%minval = minval
454 |     space%maxval = maxval
455 |   end subroutine space_set_bounds
456 |
457 |   subroutine space_set_num(space, num)
458 |     class(space_dim) :: space
459 |     integer num
460 |     space%num = num
461 |   end subroutine space_set_num
462 |
463 |   subroutine space_set_uniform_spacing(space,
    |      spacing)
464 |     class(space_dim) :: space
465 |     double precision spacing
```

```
466 |     integer k
467 |     do k=1, space%num
468 |       space%spacing(k) = spacing
469 |    end do
470 |   end subroutine space_set_uniform_spacing
471 |
472 |   subroutine space_set_spacing_array(space,
    |      spacing)
473 |     class(space_dim) :: space
474 |     double precision, dimension(space%num) ::
    |        spacing
475 |     space%spacing = spacing
476 |   end subroutine space_set_spacing_array
477 |
478 |   subroutine assign_linspace(space)
479 |     class(space_dim) :: space
480 |     double precision spacing
481 |     integer i
482 |
483 |     allocate(space%vals(space%num))
484 |     allocate(space%edges(space%num))
485 |     allocate(space%spacing(space%num))
486 |
487 |     spacing = spacing_from_num(space%minval,
    |        space%maxval, space%num)
488 |     call space%set_uniform_spacing(spacing)
489 |
490 |     do i=1, space%num
491 |        space%edges(i) = space%minval + dble(i-1)
    |            * space%spacing(i)
492 |        space%vals(i) = space%minval + dble(i-0.5
    |           d0) * space%spacing(i)
493 |     end do
494 |
495 |   end subroutine assign_linspace
496 |
497 |   subroutine set_uniform_spacing_from_num(space)
498 |     ! Create evenly spaced grid (linspace)
499 |     class(space_dim) :: space
500 |     double precision spacing
501 |
502 |     spacing = spacing_from_num(space%minval,
    |        space%maxval, space%num)
503 |     call space%set_uniform_spacing(spacing)
504 |
505 |   end subroutine set_uniform_spacing_from_num
506 |
507 | !   subroutine set_num_from_spacing(space)
508 | !     class(space_dim) :: space
```

```
509 !     !space%num = num_from_spacing(space%minval
    |    , space%maxval, space%spacing)
510 |
511 !   end subroutine set_num_from_spacing
512 |
513   subroutine space_deinit(space)
514     class(space_dim) :: space
515     deallocate(space%vals)
516     deallocate(space%edges)
517     deallocate(space%spacing)
518   end subroutine space_deinit
519 |
520   !! SAG !!
521 |
522   subroutine sag_set_bounds(grid, xmin, xmax,
    |     ymin, ymax, zmin, zmax)
523     class(space_angle_grid) :: grid
524     double precision xmin, xmax, ymin, ymax,
    |       zmin, zmax
525 |
526     call grid%x%set_bounds(xmin, xmax)
527     call grid%y%set_bounds(ymin, ymax)
528     call grid%z%set_bounds(zmin, zmax)
529   end subroutine sag_set_bounds
530 |
531   subroutine sag_set_uniform_spacing(grid, dx,
    |     dy, dz)
532     class(space_angle_grid) :: grid
533     double precision dx, dy, dz
534     call grid%x%set_uniform_spacing(dx)
535     call grid%y%set_uniform_spacing(dy)
536     call grid%z%set_uniform_spacing(dz)
537   end subroutine sag_set_uniform_spacing
538 |
539   subroutine sag_set_num(grid, nx, ny, nz,
    |     ntheta, nphi)
540     class(space_angle_grid) :: grid
541     integer nx, ny, nz, ntheta, nphi
542     call grid%x%set_num(nx)
543     call grid%y%set_num(ny)
544     call grid%z%set_num(nz)
545     call grid%angles%set_num(ntheta, nphi)
546   end subroutine sag_set_num
547 |
548   subroutine sag_init(grid)
549     class(space_angle_grid) :: grid
550 |
551     call grid%x%assign_linspace()
552     call grid%y%assign_linspace()
553     call grid%z%assign_linspace()
```

```
554 |
555 |     call grid%angles%init()
556 |     call grid%calculate_factors()
557 |
558 |   end subroutine sag_init
559 |
560 |   subroutine sag_calculate_factors(grid)
561 |     ! Factors by which depth difference is
          multiplied
562 |     ! in order to calculate distance traveled in
          the
563 |     ! (x, y) direction along a ray in the (theta
          , phi)
564 |     ! direction
565 |     class(space_angle_grid) :: grid
566 |     integer p, nomega
567 |     double precision theta, phi
568 |
569 |     nomega = grid%angles%nomega
570 |
571 |     allocate(grid%x_factor(nomega))
572 |     allocate(grid%y_factor(nomega))
573 |
574 |     do p=1, nomega
575 |        theta = grid%angles%theta_p(p)
576 |        phi = grid%angles%phi_p(p)
577 |        grid%x_factor(p) = tan(phi) * cos(theta)
578 |        grid%y_factor(p) = tan(phi) * sin(theta)
579 |     end do
580 |
581 |   end subroutine sag_calculate_factors
582 |
583 |   subroutine sag_set_uniform_spacing_from_num(
          grid)
584 |     class(space_angle_grid) :: grid
585 |     call grid%x%set_uniform_spacing_from_num()
586 |     call grid%y%set_uniform_spacing_from_num()
587 |     call grid%z%set_uniform_spacing_from_num()
588 |   end subroutine
          sag_set_uniform_spacing_from_num
589 |
590 |   ! subroutine sag_set_num_from_spacing(grid)
591 |   !   class(space_angle_grid) :: grid
592 |   !   call grid%x%set_num_from_spacing()
593 |   !   call grid%y%set_num_from_spacing()
594 |   !   call grid%z%set_num_from_spacing()
595 |
596 |   ! end subroutine sag_set_num_from_spacing
597 |
598 |   subroutine sag_deinit(grid)
599 |     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

```

```
15 subroutine generate_grid(xmin, xmax, nx, ymin,
       ymax, ny, zmin, zmax, nz, ntheta, nphi, grid,
       p_kelp)
16   double precision xmin, xmax, ymin, ymax, zmin,
         zmax
17   integer nx, ny, nz, ntheta, nphi
18   type(space_angle_grid) grid
19   double precision, dimension(:,:,:),
       allocatable :: p_kelp
20
21   call grid%set_bounds(xmin, xmax, ymin, ymax,
       zmin, zmax)
22   call grid%set_num(nx, ny, nz, ntheta, nphi)
23
24   allocate(p_kelp(nx,ny,nz))
25
26 end subroutine generate_grid
27
28 subroutine kelp3d_deinit(grid, rope, p_kelp)
29   type(space_angle_grid) grid
30   type(rope_state) rope
31   double precision, dimension(:,:,:),
       allocatable :: p_kelp
32   call rope%deinit()
33   call grid%deinit()
34   deallocate(p_kelp)
35 end subroutine kelp3d_deinit
36
37 subroutine calculate_kelp_on_grid(grid, p_kelp,
       frond, rope, quadrature_degree)
38   type(space_angle_grid), intent(in) :: grid
39   type(frond_shape), intent(in) :: frond
40   type(rope_state), intent(in) :: rope
41   type(point3d) point
42   integer, intent(in) :: quadrature_degree
43   double precision, dimension(grid%x%num, grid%y
       %num, grid%z%num) :: p_kelp
44   type(depth_state) depth
45
46   integer i, j, k, nx, ny, nz
47   double precision x, y, z
48
49   nx = grid%x%num
50   ny = grid%y%num
51   nz = grid%z%num
52
53   do k=1, nz
54     z = grid%z%vals(k)
55     call depth%set_depth(rope, grid, k)
56     do i=1, nx
```

```
57        x = grid%x%vals(i)
58        do j=1, ny
59          y = grid%y%vals(j)
60          call point%set_cart(x, y, z)
61          p_kelp(i, j, k) = kelp_proportion(point,
                frond, grid, depth,
              quadrature_degree)
62          !p_kelp(i, j, k) = prob_kelp(point,
              frond, depth, quadrature_degree)
63        end do
64      end do
65    end do
66  end subroutine calculate_kelp_on_grid
67
68  subroutine shading_region_limits(theta_low_lim,
      theta_high_lim, point, frond)
69    type(point3d), intent(in) :: point
70    type(frond_shape), intent(in) :: frond
71    double precision, intent(out) :: theta_low_lim
        , theta_high_lim
72
73    theta_low_lim = point%theta - frond%alpha
74    theta_high_lim = point%theta + frond%alpha
75  end subroutine shading_region_limits
76
77  function prob_kelp(point, frond, depth,
      quadrature_degree)
78  ! 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.
79    type(point3d), intent(in) :: point
80    type(frond_shape), intent(in) :: frond
81    type(depth_state), intent(in) :: depth
82    integer, intent(in) :: quadrature_degree
83    double precision prob_kelp
84    double precision theta_low_lim, theta_high_lim
85
86    call shading_region_limits(theta_low_lim,
        theta_high_lim, point, frond)
87    prob_kelp = integrate_ps(theta_low_lim,
        theta_high_lim, quadrature_degree, point,
        frond, depth)
88  end function prob_kelp
89
90  function kelp_proportion(point, frond, grid,
      depth, quadrature_degree)
91    ! This is the proportion of the volume of the
        Cartesian grid cell occupied by kelp
92    type(point3d), intent(in) :: point
93    type(frond_shape), intent(in) :: frond
```

```
94  |   type(depth_state), intent(in) :: depth
95  |   type(space_angle_grid), intent(in) :: grid
96  |   integer, intent(in) :: quadrature_degree
97  |   double precision p_k, n, t, dz
98  |   double precision kelp_proportion
99  |
100 |   n = depth%num_fronds
101 |   dz = grid%z%spacing(depth%depth_layer)
102 |   t = frond%ft
103 |   !write(*,*) 'KELP PROPORTION'
104 |   !write(*,*) 'n=', n
105 |   !write(*,*) 'dz=', dz
106 |   !write(*,*) 't=', t
107 |   !write(*,*) 'coef=', n*t/dz
108 |   p_k = prob_kelp(point, frond, depth,
    |      quadrature_degree)
109 |   kelp_proportion = n*t/dz * p_k
110 | end function kelp_proportion
111 |
112 | function integrate_ps(theta_low_lim,
    |    theta_high_lim, quadrature_degree, point,
    |    frond, depth) result(integral)
113 |   type(point3d), intent(in) :: point
114 |   type(frond_shape), intent(in) :: frond
115 |   double precision, intent(in) :: theta_low_lim,
    |       theta_high_lim
116 |   integer, intent(in) :: quadrature_degree
117 |   type(depth_state), intent(in) :: depth
118 |   double precision integral
119 |   double precision, dimension(:), allocatable ::
    |       integrand_vals
120 |   integer i
121 |
122 |   type(angle_dim) :: theta_f
123 |   call theta_f%set_bounds(theta_low_lim,
    |      theta_high_lim)
124 |   call theta_f%set_num(quadrature_degree)
125 |   call theta_f%assign_legendre()
126 |
127 |   allocate(integrand_vals(theta_f%num))
128 |
129 |   do i=1, theta_f%num
130 |     integrand_vals(i) = ps_integrand(theta_f%
    |        vals(i), point, frond, depth)
131 |   end do
132 |
133 |   integral = theta_f%integrate_points(
    |      integrand_vals)
134 |
135 |   deallocate(integrand_vals)
```

```
136 |   call theta_f%deinit()
137 |
138 | end function integrate_ps
139 |
140 | function ps_integrand(theta_f, point, frond,
    |    depth)
141 |   type(point3d), intent(in) :: point
142 |   type(frond_shape), intent(in) :: frond
143 |   type(depth_state), intent(in) :: depth
144 |   double precision theta_f, l_min
145 |   double precision angular_part, length_part
146 |   double precision ps_integrand
147 |
148 |   l_min = min_shading_length(theta_f, point,
    |      frond)
149 |
150 |   angular_part = depth%angle_distribution_pdf(
    |      theta_f)
151 |   length_part =  1 - depth%
    |      length_distribution_cdf(l_min)
152 |
153 |   ps_integrand = angular_part * length_part
154 | end function ps_integrand
155 |
156 |
157 | function min_shading_length(theta_f, point,
    |    frond) result(l_min)
158 | ! L_min(\theta)
159 |   type(point3d), intent(in) :: point
160 |   type(frond_shape), intent(in) :: frond
161 |   double precision, intent(in) :: theta_f
162 |   double precision l_min
163 |   double precision tpp
164 |   double precision frond_frac
165 |
166 |   ! tpp === theta_p_prime
167 |   tpp = point%theta - theta_f + pi / 2.d0
168 |   frond_frac = 2.d0 * frond%fr / (1.d0 + frond%
    |      fs)
169 |   l_min = point%r * (sin(tpp) + angular_sign(tpp
    |      ) * frond_frac * cos(tpp))
170 | end function min_shading_length
171 |
172 | ! function frond_edge(theta, theta_f, L, fs, fr)
173 | ! ! r_f(\theta)
174 | !   double precision, intent(in) :: theta,
    |    theta_f, L, fs, fr
175 | !   double precision, intent(out) :: frond_edge
176 | !
177 | !   frond_edge = relative_frond_edge(theta -
    |    theta_f + pi/2.d0)
```

```
178 !
179 ! end function frond_edge
180 !
181 ! function relative_frond_edge(theta_prime, L,
    fs, fr)
182 ! ! r_f'(\theta')
183 !   double precision, intent(in) :: theta_prime,
    L, fs, fr
184 !   double precision, intent(out) ::
    relative_frond_edge
185 !
186 !   relative_frond_edge = L / (sin(theta_prime)
    + angular_sign(theta_prime * alpha(fs, fr) *
    cos(theta_prime)))
187 ! end function relative_frond_edge
188
189 function angular_sign(theta_prime)
190 ! S(\theta')
191   double precision, intent(in) :: theta_prime
192   double precision angular_sign
193
194   ! This seems to be incorrect in summary.pdf as
        of 9/9/18
195   ! In the report, it's written as sgn(
      theta_print - pi/2.d0)
196   ! This results in L_min < 0 - not good!
197   angular_sign = sgn(pi/2.d0 - theta_prime)
198 end function angular_sign
199
200 end module kelp3d
```

rte_sparse_matrices.f90

```
1  module rte_sparse_matrices
2  use sag
3  use kelp_context
4  use mgmres
5  !use hdf5_utils
6  implicit none
7
8  type solver_params
9     integer maxiter_inner, maxiter_outer
10    double precision tol_abs, tol_rel
11 end type solver_params
12
13 type rte_mat
14    type(space_angle_grid) grid
15    type(optical_properties) iops
16    type(solver_params) params
17    integer nx, ny, nz, nomega
18    integer i, j, k, p
```

```
19    integer nonzero, n_total
20    integer x_block_size, y_block_size,
         z_block_size, omega_block_size
21
22    double precision, dimension(:), allocatable
         :: surface_vals
23
24    ! A stored in coordinate form in row, col,
         data
25    integer, dimension(:), allocatable :: row,
         col
26    double precision, dimension(:), allocatable
         :: data
27    ! b and x stored in rhs in full form
28    double precision, dimension(:), allocatable
         :: rhs, sol
29
30    ! Pointer to solver subroutine
31    ! Set to mgmres by default
32    procedure(solver_interface), pointer, nopass
         :: solver => mgmres_st
33
34  contains
35    procedure :: init => mat_init
36    procedure :: deinit => mat_deinit
37    procedure :: calculate_size
38    procedure :: set_solver_params =>
         mat_set_solver_params
39    procedure :: assign => mat_assign
40    procedure :: add => mat_add
41    procedure :: assign_rhs => mat_assign_rhs
42    !procedure :: store_index => mat_store_index
43    !procedure :: find_index => mat_find_index
44    procedure :: set_bc => mat_set_bc
45    procedure :: solve => mat_solve
46    procedure :: ind => mat_ind
47    !procedure :: to_hdf => mat_to_hdf
48    procedure attenuate
49    procedure angular_integral
50
51    ! Derivative subroutines
52    procedure x_cd2
53    procedure x_cd2_first
54    procedure x_cd2_last
55    procedure y_cd2
56    procedure y_cd2_first
57    procedure y_cd2_last
58    procedure z_cd2
59    procedure z_fd2
60    procedure z_bd2
61    procedure z_surface_bc
62    procedure z_bottom_bc
63
```

```
64  end type rte_mat
65
66  interface
67     ! Define interface for external procedure
68     ! https://stackoverflow.com/questions
          /8549415/how-to-declare-the-interface-
          section-for-a-procedure-argument-which-in-
          turn-ref
69     subroutine solver_interface(n_total, nonzero,
           row, col, data, &
70           sol, rhs, maxiter_outer, maxiter_inner,
              &
71           tol_abs, tol_rel)
72       integer ::  n_total, nonzero
73       integer, dimension(nonzero) :: row, col
74       double precision, dimension(nonzero) ::
           data
75       double precision, dimension(nonzero) :: sol
76       double precision, dimension(n_total) :: rhs
77       integer :: maxiter_outer, maxiter_inner
78       double precision :: tol_abs, tol_rel
79     end subroutine solver_interface
80  end interface
81
82  contains
83
84    subroutine mat_init(mat, grid, iops)
85      class(rte_mat) mat
86      type(space_angle_grid) grid
87      type(optical_properties) iops
88      integer nnz, n_total
89
90      mat%grid = grid
91      mat%iops = iops
92
93      call mat%calculate_size()
94
95      n_total = mat%n_total
96      nnz = mat%nonzero
97      allocate(mat%surface_vals(grid%angles%nomega
          ))
98      allocate(mat%row(nnz))
99      allocate(mat%col(nnz))
100     allocate(mat%data(nnz))
101     allocate(mat%rhs(n_total))
102     allocate(mat%sol(n_total))
103
104     call zeros(mat%rhs, n_total)
105     call zeros(mat%sol, n_total)
106
107   end subroutine mat_init
```

```
108 |
109 |   subroutine mat_deinit(mat)
110 |     class(rte_mat) mat
111 |     deallocate(mat%row)
112 |     deallocate(mat%col)
113 |     deallocate(mat%data)
114 |     deallocate(mat%rhs)
115 |     deallocate(mat%sol)
116 |     deallocate(mat%surface_vals)
117 |   end subroutine mat_deinit
118 |
119 |   subroutine calculate_size(mat)
120 |     class(rte_mat) mat
121 |     integer nx, ny, nz, nomega
122 |
123 |     nx = mat%grid%x%num
124 |     ny = mat%grid%y%num
125 |     nz = mat%grid%z%num
126 |     nomega = mat%grid%angles%nomega
127 |
128 |     !mat%nonzero = nx * ny * ntheta * nphi * ( (
    |       nz-1) * (6 + ntheta * nphi) + 1)
129 |     mat%nonzero = nx * ny * nomega * (nz * (
    |       nomega + 6) - 1)
130 |     mat%n_total = nx * ny * nz * nomega
131 |
132 |     !mat%theta_block_size = 1
133 |     !mat%phi_block_size = mat%theta_block_size *
    |        ntheta
134 |     mat%omega_block_size = 1
135 |     mat%y_block_size = mat%omega_block_size *
    |       nomega
136 |     mat%x_block_size = mat%y_block_size * ny
137 |     mat%z_block_size = mat%x_block_size * nx
138 |
139 |   end subroutine calculate_size
140 |
141 | !  subroutine mat_to_hdf(mat,filename)
142 | !    class(rte_mat) mat
143 | !    character(len=*) filename
144 | !    call write_coo(filename, mat%row, mat%col,
    |   mat%data, mat%nonzero)
145 | !  end subroutine mat_to_hdf
146 |
147 |   subroutine mat_set_bc(mat, bc)
148 |     class(rte_mat) mat
149 |     class(boundary_condition) bc
150 |     integer p
151 |
152 |     do p=1, mat%grid%angles%nomega/2
```

```
153         mat%surface_vals(p) = bc%bc_grid(p)
154       end do
155     end subroutine mat_set_bc
156
157     subroutine mat_solve(mat)
158       class(rte_mat) mat
159       type(solver_params) params
160
161       params = mat%params
162
163       write(*,*) 'mat%n_total =', mat%n_total
164       write(*,*) 'mat%nonzero =', mat%nonzero
165       write(*,*) 'size(mat%row) =', size(mat%row)
166       write(*,*) 'size(mat%col) =', size(mat%col)
167       write(*,*) 'size(mat%data) =', size(mat%data
            )
168       write(*,*) 'size(mat%sol) =', size(mat%sol)
169       write(*,*) 'size(mat%rhs) =', size(mat%rhs)
170       write(*,*) 'params%maxiter_outer =', params%
            maxiter_outer
171       write(*,*) 'params%maxiter_inner =', params%
            maxiter_inner
172       write(*,*) 'params%tol_rel =', params%
            tol_rel
173       write(*,*) 'params%tol_abs =', params%
            tol_abs
174       ! open(unit=1, file='row.txt')
175       ! open(unit=2, file='col.txt')
176       ! open(unit=3, file='data.txt')
177       ! open(unit=4, file='rhs.txt')
178       ! open(unit=5, file='sol.txt')
179       ! write(1,*) mat%row
180       ! write(2,*) mat%col
181       ! write(3,*) mat%data
182       ! write(4,*) mat%rhs
183
184       ! close(1)
185       ! close(2)
186       ! close(3)
187       ! close(4)
188
189       call mat%solver(mat%n_total, mat%nonzero, &
190            mat%row, mat%col, mat%data, mat%sol,
                 mat%rhs, &
191            params%maxiter_outer, params%
                 maxiter_inner, &
192            params%tol_abs, params%tol_rel)
193
194       ! write(5,*) mat%sol
195       ! close(5)
```

```
196 |
197 |   end subroutine mat_solve
198 |
199 |   subroutine mat_set_solver_params(mat,
    |      maxiter_outer, &
200 |         maxiter_inner, tol_abs, tol_rel)
201 |     class(rte_mat) mat
202 |     integer maxiter_outer, maxiter_inner
203 |     double precision tol_abs, tol_rel
204 |
205 |     mat%params%maxiter_outer = maxiter_outer
206 |     mat%params%maxiter_inner = maxiter_inner
207 |     mat%params%tol_abs = tol_abs
208 |     mat%params%tol_rel = tol_rel
209 |   end subroutine mat_set_solver_params
210 |
211 |   function mat_ind(mat, i, j, k, p) result(ind)
212 |     ! Assuming var ordering: z, x, y, omega
213 |     class(rte_mat) mat
214 |     integer i, j, k, p
215 |     integer ind
216 |
217 |     ind = (i-1) * mat%x_block_size + (j-1) * mat
    |        %y_block_size + &
218 |           (k-1) * mat%z_block_size + p * mat%
    |              omega_block_size
219 |   end function mat_ind
220 |
221 |   subroutine mat_assign(mat, row_num, ent, val,
    |      i, j, k, p)
222 |     ! It's assumed that this is the only time
    |        this entry is defined
223 |     class(rte_mat) mat
224 |     double precision val
225 |     integer i, j, k, p
226 |     integer row_num, ent
227 |
228 |     mat%row(ent) = row_num
229 |     mat%col(ent) = mat%ind(i, j, k, p)
230 |     mat%data(ent) = val
231 |
232 |     ent = ent + 1
233 |   end subroutine mat_assign
234 |
235 |   subroutine mat_add(mat, repeat_ent, val)
236 |     ! Use this when you know that this entry has
    |        already been assigned
237 |     ! and you'd like to add this value to the
    |        existing value.
238 |
239 |     class(rte_mat) mat
```

```
240     double precision val
241     integer repeat_ent
242
243     ! Entry number where value is already stored
244     mat%data(repeat_ent) = mat%data(repeat_ent)
           + val
245   end subroutine mat_add
246
247   subroutine mat_assign_rhs(mat, row_num, data)
248     class(rte_mat) mat
249     double precision data
250     integer row_num
251
252     mat%rhs(row_num) = data
253   end subroutine mat_assign_rhs
254
255   ! subroutine mat_store_index(mat, row_num,
          col_num)
256   !   ! Remember where we stored information for
          this matrix element
257   !   class(rte_mat) mat
258   !   integer row_num, col_num
259   !   !mat%index_map(row_num, col_num) = mat%ent
260   ! end subroutine
261
262   ! function mat_find_index(mat, row_num,
          col_num) result(index)
263   !   ! Find the position in row, col, data
          where this entry
264   !   ! is defined.
265   !   class(rte_mat) mat
266   !   integer row_num, col_num, index
267
268   !   index = mat%index_map(row_num, col_num)
269
270   !   ! This took up 95% of execution time.
271   !   ! Only search up to most recently assigned
          index
272   !   ! do index=1, mat%ent-1
273   !   !    if( (mat%row(index) .eq. row_num) .
          and. (mat%col(index) .eq. col_num)) then
274   !   !       exit
275   !   !    end if
276   !   ! end do
277   ! end function mat_find_index
278
279   subroutine attenuate(mat, indices, repeat_ent)
280     ! Has to be called after angular_integral
281     ! Because they both write to the same matrix
          entry
282     ! And adding here is more efficient than a
          conditional
```

```
283 |     ! in the angular loop.
284 |     class(rte_mat) mat
285 |     double precision attenuation
286 |     type(index_list) indices
287 |     double precision aa, bb
288 |     integer repeat_ent
289 |
290 |     aa = mat%iops%abs_grid(indices%i, indices%j,
    |         indices%k)
291 |     bb = mat%iops%scat
292 |     attenuation = aa + bb
293 |
294 |     call mat%add(repeat_ent, attenuation)
295 |   end subroutine attenuate
296 |
297 |   subroutine x_cd2(mat, indices, row_num, ent)
298 |     class(rte_mat) mat
299 |     double precision val, dx
300 |     type(index_list) indices
301 |     integer i, j, k, p
302 |     integer row_num, ent
303 |
304 |     i = indices%i
305 |     j = indices%j
306 |     k = indices%k
307 |     p = indices%p
308 |
309 |     dx = mat%grid%x%spacing(1)
310 |
311 |     val = mat%grid%angles%sin_phi_p(p) &
312 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
313 |
314 |     call mat%assign(row_num,ent,-val,i-1,j,k,p)
315 |     call mat%assign(row_num,ent,val,i+1,j,k,p)
316 |   end subroutine x_cd2
317 |
318 |   subroutine x_cd2_first(mat, indices, row_num,
    |    ent)
319 |     class(rte_mat) mat
320 |     double precision val, dx
321 |     integer nx
322 |     type(index_list) indices
323 |     integer i, j, k, p
324 |     integer row_num, ent
325 |
326 |     i = indices%i
327 |     j = indices%j
328 |     k = indices%k
329 |     p = indices%p
330 |
```

```
331 |     dx = mat%grid%x%spacing(1)
332 |     nx = mat%grid%x%num
333 |
334 |     val = mat%grid%angles%sin_phi_p(p) &
335 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
336 |
337 |     call mat%assign(row_num,ent,-val,nx,j,k,p)
338 |     call mat%assign(row_num,ent,val,i+1,j,k,p)
339 |   end subroutine x_cd2_first
340 |
341 |   subroutine x_cd2_last(mat, indices, row_num,
    |      ent)
342 |     class(rte_mat) mat
343 |     double precision val, dx
344 |     type(index_list) indices
345 |     integer i, j, k, p
346 |     integer row_num, ent
347 |
348 |     i = indices%i
349 |     j = indices%j
350 |     k = indices%k
351 |     p = indices%p
352 |
353 |     dx = mat%grid%x%spacing(1)
354 |
355 |     val = mat%grid%angles%sin_phi_p(p) &
356 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
357 |
358 |     call mat%assign(row_num,ent,-val,i-1,j,k,p)
359 |     call mat%assign(row_num,ent,val,1,j,k,p)
360 |   end subroutine x_cd2_last
361 |
362 |   subroutine y_cd2(mat, indices, row_num, ent)
363 |     class(rte_mat) mat
364 |     double precision val, dy
365 |     type(index_list) indices
366 |     integer i, j, k, p
367 |     integer row_num, ent
368 |
369 |     i = indices%i
370 |     j = indices%j
371 |     k = indices%k
372 |     p = indices%p
373 |
374 |     dy = mat%grid%y%spacing(1)
375 |
376 |     val = mat%grid%angles%sin_phi_p(p) &
```

```
377 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
378 |
379 |     call mat%assign(row_num,ent,-val,i,j-1,k,p)
380 |     call mat%assign(row_num,ent,val,i,j+1,k,p)
381 |   end subroutine y_cd2
382 |
383 |   subroutine y_cd2_first(mat, indices, row_num,
    |      ent)
384 |     class(rte_mat) mat
385 |     double precision val, dy
386 |     integer ny
387 |     type(index_list) indices
388 |     integer i, j, k, p
389 |     integer row_num, ent
390 |
391 |     i = indices%i
392 |     j = indices%j
393 |     k = indices%k
394 |     p = indices%p
395 |
396 |     dy = mat%grid%y%spacing(1)
397 |     ny = mat%grid%y%num
398 |
399 |     val = mat%grid%angles%sin_phi_p(p) &
400 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
401 |
402 |     call mat%assign(row_num,ent,-val,i,ny,k,p)
403 |     call mat%assign(row_num,ent,val,i,j+1,k,p)
404 |   end subroutine y_cd2_first
405 |
406 |   subroutine y_cd2_last(mat, indices, row_num,
    |      ent)
407 |     class(rte_mat) mat
408 |     double precision val, dy
409 |     type(index_list) indices
410 |     integer i, j, k, p
411 |     integer row_num, ent
412 |
413 |     i = indices%i
414 |     j = indices%j
415 |     k = indices%k
416 |     p = indices%p
417 |
418 |     dy = mat%grid%y%spacing(1)
419 |
420 |     val = mat%grid%angles%sin_phi_p(p) &
421 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
```

```
422 |
423 |     call mat%assign(row_num,ent,-val,i,j-1,k,p)
424 |     call mat%assign(row_num,ent,val,i,1,k,p)
425 |   end subroutine y_cd2_last
426 |
427 |   subroutine z_cd2(mat, indices, row_num, ent)
428 |     class(rte_mat) mat
429 |     double precision val, dz
430 |     type(index_list) indices
431 |     integer i, j, k, p
432 |     integer row_num, ent
433 |
434 |     i = indices%i
435 |     j = indices%j
436 |     k = indices%k
437 |     p = indices%p
438 |
439 |     dz = mat%grid%z%spacing(indices%k)
440 |
441 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
          dz)
442 |
443 |     call mat%assign(row_num,ent,-val,i,j,k-1,p)
444 |     call mat%assign(row_num,ent,val,i,j,k+1,p)
445 |   end subroutine z_cd2
446 |
447 |   subroutine z_fd2(mat, indices, row_num, ent,
        repeat_ent)
448 |     ! Has to be called after angular_integral
449 |     ! Because they both write to the same matrix
          entry
450 |     ! And adding here is more efficient than a
          conditional
451 |     ! in the angular loop.
452 |     class(rte_mat) mat
453 |     double precision val, val1, val2, val3, dz
454 |     type(index_list) indices
455 |     integer i, j, k, p
456 |     integer row_num, ent, repeat_ent
457 |
458 |     i = indices%i
459 |     j = indices%j
460 |     k = indices%k
461 |     p = indices%p
462 |
463 |     dz = mat%grid%z%spacing(indices%k)
464 |
465 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
          dz)
466 |
467 |     val1 = -3.d0 * val
```

```
468       val2 = 4.d0 * val
469       val3 = -val
470
471       call mat%add(repeat_ent, val1)
472       call mat%assign(row_num,ent,val2,i,j,k+1,p)
473       call mat%assign(row_num,ent,val3,i,j,k+2,p)
474     end subroutine z_fd2
475
476     subroutine z_bd2(mat, indices, row_num, ent,
          repeat_ent)
477       ! Has to be called after angular_integral
478       ! Because they both write to the same matrix
            entry
479       ! And adding here is more efficient than a
           conditional
480       ! in the angular loop.
481       class(rte_mat) mat
482       double precision val, val1, val2, val3, dz
483       type(index_list) indices
484       integer i, j, k, p
485       integer row_num, ent, repeat_ent
486
487       i = indices%i
488       j = indices%j
489       k = indices%k
490       p = indices%p
491
492       dz = mat%grid%z%spacing(indices%k)
493
494       val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
              dz)
495
496       val1 = 3.d0 * val
497       val2 = -4.d0 * val
498       val3 = val
499
500       call mat%add(repeat_ent, val1)
501       call mat%assign(row_num,ent,val2,i,j,k-1,p)
502       call mat%assign(row_num,ent,val3,i,j,k-2,p)
503     end subroutine z_bd2
504
505     subroutine angular_integral(mat, indices,
          row_num, ent)
506       class(rte_mat) mat
507       ! Primed angular integration variables
508       integer pp
509       double precision val
510       type(index_list) indices
511       integer row_num, ent
512
513       ! Interior
```

```
514     do pp=1, mat%grid%angles%nomega
515        ! TODO: Make sure I don't have p and pp
              backwards
516        val = -mat%iops%scat * mat%iops%
              vsf_integral(indices%p, pp)
517        call mat%assign(row_num, ent, val,
              indices%i, indices%j, indices%k, pp)
518     end do
519   end subroutine angular_integral
520
521   subroutine z_surface_bc(mat, indices, row_num,
        ent, repeat_ent)
522     class(rte_mat) mat
523     double precision bc_val
524     type(index_list) indices
525     double precision val1, val2, dz
526     integer row_num, ent, repeat_ent
527
528     dz = mat%grid%z%spacing(1)
529
530     val1 = mat%grid%angles%cos_phi_p(indices%p)
          / (5.d0 * dz)
531     val2 = 7.d0 * val1
532     bc_val = 8.d0 * val1 * mat%surface_vals(
          indices%p)
533
534     call mat%assign(row_num,ent,val1,indices%i,
          indices%j,2,indices%p)
535     call mat%add(repeat_ent, val2)
536     call mat%assign_rhs(row_num, bc_val)
537
538   end subroutine z_surface_bc
539
540     subroutine z_bottom_bc(mat, indices, row_num
          , ent, repeat_ent)
541     class(rte_mat) mat
542     type(index_list) indices
543     double precision val1, val2, dz
544     integer nz
545     integer row_num, ent, repeat_ent
546
547     dz = mat%grid%z%spacing(1)
548     nz = mat%grid%z%num
549
550     val1 = -mat%grid%angles%cos_phi_p(indices%p)
          / (5.d0 * dz)
551     val2 = 7.d0 * val1
552
553     call mat%assign(row_num,ent,val1,indices%i,
          indices%j,nz-1,indices%p)
554     call mat%add(repeat_ent, val2)
```

```
555 |
556 |   end subroutine z_bottom_bc
557 |
558 |   ! Finite difference wrappers
559 |
560 |   ! subroutine wrap_x_cd2(mat, indices)
561 |   !   type(rte_mat) mat
562 |   !   type(index_list) indices
563 |   !   call mat%x_cd2(indices)
564 |   ! end subroutine wrap_x_cd2
565 |
566 |   ! subroutine wrap_x_cd2_last(mat, indices)
567 |   !   type(rte_mat) mat
568 |   !   type(index_list) indices
569 |   !   call mat%x_cd2_last(indices)
570 |   ! end subroutine wrap_x_cd2_last
571 |
572 |   ! subroutine wrap_x_cd2_first(mat, indices)
573 |   !   type(rte_mat) mat
574 |   !   type(index_list) indices
575 |   !   call mat%x_cd2_first(indices)
576 |   ! end subroutine wrap_x_cd2_first
577 |
578 |   ! subroutine wrap_y_cd2(mat, indices)
579 |   !   type(rte_mat) mat
580 |   !   type(index_list) indices
581 |   !   call mat%y_cd2(indices)
582 |   ! end subroutine wrap_y_cd2
583 |
584 |   ! subroutine wrap_y_cd2_last(mat, indices)
585 |   !   type(rte_mat) mat
586 |   !   type(index_list) indices
587 |   !   call mat%y_cd2_last(indices)
588 |   ! end subroutine wrap_y_cd2_last
589 |
590 |   ! subroutine wrap_y_cd2_first(mat, indices)
591 |   !   type(rte_mat) mat
592 |   !   type(index_list) indices
593 |   !   call mat%y_cd2_first(indices)
594 |   ! end subroutine wrap_y_cd2_first
595 |
596 |   ! subroutine wrap_z_cd2(mat, indices)
597 |   !   type(rte_mat) mat
598 |   !   type(index_list) indices
599 |   !   call mat%z_cd2(indices)
600 |   ! end subroutine wrap_z_cd2
601 |
602 | end module rte_sparse_matrices
```

```
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( &
```

```
46 |   !$    omp_get_num_procs()/num_threads_z, &
47 |   !$    mat%grid%x%num)
48 |   !$ num_threads_y = min( &
49 |   !$    omp_get_num_procs()/(num_threads_z*
   |      num_threads_x), &
50 |   !$    mat%grid%y%num)
51 |
52 |   !$ write(*,*) 'num_procs =', omp_get_num_procs
   |      ()
53 |   !$ write(*,*) 'ntz =', num_threads_z
54 |   !$ write(*,*) 'ntx =', num_threads_x
55 |   !$ write(*,*) 'nty =', num_threads_y
56 |
57 |   !$omp parallel do default(none) shared(mat) &
58 |   !$omp private(ddx,ddy,angle_loop, k, i, j)
   |      private(indices) &
59 |   !$omp shared(num_threads_x,num_threads_y,
   |      num_threads_z) &
60 |   !$omp num_threads(num_threads_z) if(
   |      num_threads_z .gt. 1)
61 |   do k=1, mat%grid%z%num
62 |      write(*,*) 'k =', k
63 |      indices%k = k
64 |      if(k .eq. 1) then
65 |         angle_loop => surface_angle_loop
66 |      else if(k .eq. mat%grid%z%num) then
67 |         angle_loop => bottom_angle_loop
68 |      else
69 |         angle_loop => interior_angle_loop
70 |      end if
71 |
72 |      !$omp parallel do default(none) shared(mat)
   |          private(i,j) &
73 |      !$omp firstprivate(indices,angle_loop, k)
   |         private(ddx,ddy) &
74 |      !$omp shared(num_threads_x,num_threads_y,
   |         num_threads_z) &
75 |      !$omp num_threads(num_threads_x) if(
   |         num_threads_x .gt. 1)
76 |      do i=1, mat%grid%x%num
77 |         indices%i = i
78 |         if(indices%i .eq. 1) then
79 |            ddx => x_cd2_first
80 |         else if(indices%i .eq. mat%grid%x%num)
   |            then
81 |            ddx => x_cd2_last
82 |         else
83 |            ddx => x_cd2
84 |         end if
```

```
85 |          !$omp parallel do default(none) shared(
   |             mat) private(j) &
86 |          !$omp firstprivate(indices,ddx,ddy,
   |             angle_loop, i, k) &
87 |          !$omp shared(num_threads_x,num_threads_y
   |             ,num_threads_z) &
88 |          !$omp num_threads(num_threads_y) if(
   |             num_threads_y .gt. 1)
89 |          do j=1, mat%grid%y%num
90 |             indices%j = j
91 |             if(indices%j .eq. 1) then
92 |                ddy => y_cd2_first
93 |             else if(indices%j .eq. mat%grid%y%num
   |                ) then
94 |                ddy => y_cd2_last
95 |             else
96 |                ddy => y_cd2
97 |             end if
98 |
99 |             call angle_loop(mat, indices, ddx,
   |                ddy)
100|          end do
101|          !$omp end parallel do
102|       end do
103|       !$omp end parallel do
104|    end do
105|    !$omp end parallel do
106|end subroutine whole_space_loop
107|
108|function calculate_start_ent(grid, indices)
   |    result(ent)
109|    type(space_angle_grid) grid
110|    type(index_list) indices
111|    integer ent
112|    integer boundary_nnz, interior_nnz
113|    integer num_boundary, num_interior
114|    integer num_this_x, num_this_z
115|
116|    ! Nonzero matrix entries for an surface or
   |       bottom spatial grid cell
117|    ! Definitely an integer since nomega is even
118|    boundary_nnz = grid%angles%nomega * (2 * grid%
   |       angles%nomega + 11) / 2
119|    ! Nonzero matrix entries for an interior
   |       spatial grid cell
120|    interior_nnz = grid%angles%nomega * (grid%
   |       angles%nomega + 6)
121|
122|    ! Order: z, x, y, omega
123|    ! Total number traversed so far in each
   |       spatial category
```

```
124 |   ! row
125 |   num_this_x = indices%j - 1
126 |   ! depth layer
127 |   num_this_z = (indices%i - 1) * grid%y%num +
    |      num_this_x
128 |
129 |   ! Calculate number of spatial grid cells of
    |      each type which have
130 |   ! already been traversed up to this point
131 |   if(indices%k .eq. 1) then
132 |      num_boundary = num_this_z
133 |      num_interior = 0
134 |   else if(indices%k .eq. grid%z%num) then
135 |      num_boundary = (grid%x%num * grid%y%num) +
    |         num_this_z
136 |      num_interior = (grid%z%num-2) * grid%x%num
    |         * grid%y%num
137 |   else
138 |      num_boundary = grid%x%num * grid%y%num
139 |      num_interior = num_this_z + (indices%k-2) *
    |          grid%x%num * grid%y%num
140 |   end if
141 |
142 |   ent = num_boundary * boundary_nnz +
    |      num_interior * interior_nnz + 1
143 | end function calculate_start_ent
144 |
145 | function calculate_repeat_ent(ent, p) result(
    |    repeat_ent)
146 |   integer ent, p, repeat_ent
147 |   ! Entry number for row=mat%ind(i,j,k,p), col=
    |      mat%ind(i,j,k,p),
148 |   ! which will be modified multiple times in
    |      this matrix row
149 |   repeat_ent = ent + p - 1
150 | end function calculate_repeat_ent
151 |
152 | subroutine interior_angle_loop(mat, indices, ddx
    |    , ddy)
153 |   type(rte_mat) mat
154 |   type(index_list) indices
155 |   procedure(deriv_interface) :: ddx, ddy
156 |   integer p
157 |   integer ent, repeat_ent
158 |   integer row_num
159 |
160 |   ! Determine which matrix row to start at
161 |   ent = calculate_start_ent(mat%grid, indices)
162 |   indices%p = 1
163 |   row_num = mat%ind(indices%i, indices%j,
    |      indices%k, indices%p)
```

```
164 |
165 |   do p=1, mat%grid%angles%nomega
166 |      indices%p = p
167 |      repeat_ent = calculate_repeat_ent(ent, p)
168 |      call mat%angular_integral(indices, row_num,
    |          ent)
169 |      call ddx(mat, indices, row_num, ent)
170 |      call ddy(mat, indices, row_num, ent)
171 |      call mat%z_cd2(indices, row_num, ent)
172 |      call mat%attenuate(indices, repeat_ent)
173 |      row_num = row_num + 1
174 |   end do
175 | end subroutine
176 |
177 | subroutine surface_angle_loop(mat, indices, ddx,
    |     ddy)
178 |   type(rte_mat) mat
179 |   type(index_list) indices
180 |   integer p
181 |   procedure(deriv_interface) :: ddx, ddy
182 |   integer ent, repeat_ent
183 |   integer row_num
184 |
185 |   ! Determine which matrix row to start at
186 |   ent = calculate_start_ent(mat%grid, indices)
187 |   indices%p = 1
188 |   row_num = mat%ind(indices%i, indices%j,
    |      indices%k, indices%p)
189 |
190 |   ! Downwelling
191 |   do p=1, mat%grid%angles%nomega / 2
192 |      indices%p = p
193 |      repeat_ent = calculate_repeat_ent(ent, p)
194 |      call mat%angular_integral(indices, row_num,
    |          ent)
195 |      call ddx(mat, indices, row_num, ent)
196 |      call ddy(mat, indices, row_num, ent)
197 |      call mat%z_surface_bc(indices, row_num, ent
    |         , repeat_ent)
198 |      call mat%attenuate(indices, repeat_ent)
199 |      row_num = row_num + 1
200 |   end do
201 |   ! Upwelling
202 |   do p=mat%grid%angles%nomega/2+1, mat%grid%
    |      angles%nomega
203 |      indices%p = p
204 |      repeat_ent = calculate_repeat_ent(ent, p)
205 |      call mat%angular_integral(indices, row_num,
    |          ent)
206 |      call ddx(mat, indices, row_num, ent)
```

```
207 |       call ddy(mat, indices, row_num, ent)
208 |       call mat%z_fd2(indices, row_num, ent,
    |          repeat_ent)
209 |       call mat%attenuate(indices, repeat_ent)
210 |       row_num = row_num + 1
211 |   end do
212 | end subroutine surface_angle_loop
213 |
214 | subroutine bottom_angle_loop(mat, indices, ddx,
    |    ddy)
215 |   type(rte_mat) mat
216 |   type(index_list) indices
217 |   integer p
218 |   integer row_num, ent, repeat_ent
219 |   procedure(deriv_interface) :: ddx, ddy
220 |
221 |   ! Determine which matrix row to start at
222 |   ent = calculate_start_ent(mat%grid, indices)
223 |   indices%p = 1
224 |   row_num = mat%ind(indices%i, indices%j,
    |      indices%k, indices%p)
225 |
226 |   ! Downwelling
227 |   do p=1, mat%grid%angles%nomega/2
228 |       indices%p = p
229 |       repeat_ent = calculate_repeat_ent(ent, p)
230 |       call mat%angular_integral(indices, row_num,
    |           ent)
231 |       call ddx(mat, indices, row_num, ent)
232 |       call ddy(mat, indices, row_num, ent)
233 |       call mat%z_bd2(indices, row_num, ent,
    |          repeat_ent)
234 |       call mat%attenuate(indices, repeat_ent)
235 |       row_num = row_num + 1
236 |   end do
237 |   ! Upwelling
238 |   do p=mat%grid%angles%nomega/2+1, mat%grid%
    |      angles%nomega
239 |       indices%p = p
240 |       repeat_ent = calculate_repeat_ent(ent, p)
241 |       call mat%angular_integral(indices, row_num,
    |           ent)
242 |       call ddx(mat, indices, row_num, ent)
243 |       call ddy(mat, indices, row_num, ent)
244 |       call mat%z_bottom_bc(indices, row_num, ent,
    |           repeat_ent)
245 |       call mat%attenuate(indices, repeat_ent)
246 |       row_num = row_num + 1
247 |   end do
248 | end subroutine bottom_angle_loop
```

```
249 |
250 |subroutine gen_matrix(mat)
251 |  type(rte_mat) mat
252 |  type(index_list) indices
253 |
254 |  call indices%init()
255 |
256 |  call whole_space_loop(mat, indices)
257 |  ! call surface_space_loop(mat, indices)
258 |  ! call interior_space_loop(mat, indices)
259 |  ! call bottom_space_loop(mat, indices)
260 |end subroutine gen_matrix
261 |
262 |subroutine rte3d_deinit(mat, iops, light)
263 |  type(rte_mat) mat
264 |  type(optical_properties) iops
265 |  type(light_state) light
266 |
267 |  call mat%deinit()
268 |  call iops%deinit()
269 |  call light%deinit()
270 |end subroutine
271 |
272 |end module rte3d
```

kelp_context.f90

```
 1 |module kelp_context
 2 |use sag
 3 |use prob
 4 |implicit none
 5 |
 6 |! Point in cylindrical coordinates
 7 |type point3d
 8 |   double precision x, y, z, theta, r
 9 | contains
10 |   procedure :: set_cart => point_set_cart
11 |   procedure :: set_cyl => point_set_cyl
12 |   procedure :: cartesian_to_polar
13 |   procedure :: polar_to_cartesian
14 |end type point3d
15 |
16 |type frond_shape
17 |  double precision fs, fr, tan_alpha, alpha, ft
18 |contains
19 |  procedure :: set_shape => frond_set_shape
20 |  procedure :: calculate_angles =>
   |     frond_calculate_angles
21 |end type frond_shape
22 |
23 |type rope_state
24 |   integer nz
```

```
25    double precision, dimension(:), allocatable
        :: frond_lengths, frond_stds, num_fronds,
        water_speeds, water_angles
26  contains
27      procedure :: init => rope_init
28      procedure :: deinit => rope_deinit
29  end type rope_state
30
31  type depth_state
32    double precision frond_length, frond_std,
        num_fronds, water_speeds, water_angles,
        depth
33    integer depth_layer
34  contains
35    procedure :: set_depth
36    procedure :: length_distribution_cdf
37    procedure :: angle_distribution_pdf
38  end type depth_state
39
40  type optical_properties
41    integer num_vsf
42    type(space_angle_grid) grid
43    double precision, dimension(:), allocatable
        :: vsf_angles, vsf_vals, vsf_cos
44    double precision, dimension(:), allocatable
        :: abs_water
45    double precision abs_kelp, vsf_scat_coef,
        scat
46    ! On x, y, z grid - including water & kelp.
47    double precision, dimension(:,:,:),
        allocatable :: abs_grid
48    ! On theta, phi, theta_prime, phi_prime grid
49    double precision, dimension(:,:), allocatable
         :: vsf, vsf_integral
50   contains
51    procedure :: init => iop_init
52    procedure :: calculate_coef_grids
53    procedure :: load_vsf
54    procedure :: eval_vsf
55    procedure :: calc_vsf_on_grid
56    procedure :: deinit => iop_deinit
57    procedure :: vsf_from_function
58  end type optical_properties
59
60  type boundary_condition
61    double precision I0, decay, theta_s, phi_s
62    type(space_angle_grid) grid
63    double precision, dimension(:), allocatable
        :: bc_grid
64   contains
65    procedure :: bc_gaussian
66    procedure :: init => bc_init
67    procedure :: deinit => bc_deinit
```

```
68 end type boundary_condition
69
70 contains
71
72   function bc_gaussian(bc, theta, phi)
73     class(boundary_condition) bc
74     double precision theta, phi, diff
75     double precision bc_gaussian
76     diff = angle_diff_3d(theta, phi, bc%theta_s,
       bc%phi_s)
77     bc_gaussian = exp(-bc%decay * diff)
78   end function bc_gaussian
79
80   subroutine bc_init(bc, grid, theta_s, phi_s,
       decay, I0)
81     class(boundary_condition) bc
82     type(space_angle_grid) grid
83     double precision theta_s, phi_s, decay, I0
84     integer p
85     double precision theta, phi
86
87     allocate(bc%bc_grid(grid%angles%nomega))
88
89     bc%theta_s = theta_s
90     bc%phi_s = phi_s
91     bc%decay = decay
92     bc%I0 = I0
93
94     ! Only set BC for downwelling light
95     do p=1, grid%angles%nomega/2
96        theta = grid%angles%theta_p(p)
97        phi = grid%angles%phi_p(p)
98        bc%bc_grid(p) = bc%bc_gaussian(theta, phi
            )
99     end do
100    ! Zero upwelling light specified at surface
101    do p=grid%angles%nomega/2+1, grid%angles%
       nomega
102       bc%bc_grid(p) = 0.d0
103    end do
104
105    ! Normalize
106    bc%bc_grid = bc%I0 * bc%bc_grid &
107         / grid%angles%integrate_points(bc%
              bc_grid)
108
109  end subroutine bc_init
110
111  subroutine bc_deinit(bc)
112    class(boundary_condition) bc
113    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
```

```
163     type(space_angle_grid) grid
164
165     iops%grid = grid
166
167     ! Assume that these are preallocated and
            passed to function
168     ! Nevermind, don't assume this.
169     allocate(iops%abs_water(grid%z%num))
170
171     ! Assume that these must be allocated here
172     allocate(iops%vsf_angles(iops%num_vsf))
173     allocate(iops%vsf_vals(iops%num_vsf))
174     allocate(iops%vsf_cos(iops%num_vsf))
175     allocate(iops%vsf(grid%angles%nomega,grid%
            angles%nomega))
176     allocate(iops%vsf_integral(grid%angles%
            nomega,grid%angles%nomega))
177     allocate(iops%abs_grid(grid%x%num, grid%y%
            num, grid%z%num))
178   end subroutine iop_init
179
180   subroutine calculate_coef_grids(iops, p_kelp)
181     class(optical_properties) iops
182     double precision, dimension(:,:,:) :: p_kelp
183
184     integer k
185
186     ! Allow water IOPs to vary over depth
187     do k=1, iops%grid%z%num
188       iops%abs_grid(:,:,k) = (iops%abs_kelp -
              iops%abs_water(k)) * p_kelp(:,:,k) +
              iops%abs_water(k)
189    end do
190
191   end subroutine calculate_coef_grids
192
193
194   subroutine load_vsf(iops, filename, fmtstr)
195     class(optical_properties) :: iops
196     character(len=*) :: filename, fmtstr
197     double precision, dimension(:,:),
            allocatable :: tmp_2d_arr
198     integer num_rows, num_cols, skiplines_in
199
200     ! First column is the angle at which the
            measurement is taken
201     ! Second column is the value of the VSF at
            that angle
202     num_rows = iops%num_vsf
203     num_cols = 2
```

```
204     skiplines_in = 1 ! Ignore comment on first
          line
205
206     allocate(tmp_2d_arr(num_rows, num_cols))
207
208     tmp_2d_arr = read_array(filename, fmtstr,
          num_rows, num_cols, skiplines_in)
209     iops%vsf_angles = tmp_2d_arr(:,1)
210     iops%vsf_vals = tmp_2d_arr(:,2)
211
212     ! write(*,*) 'vsf_angles = ', iops%
          vsf_angles
213     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
214
215     ! Pre-evaluate for all pair of angles
216     call iops%calc_vsf_on_grid()
217   end subroutine load_vsf
218
219   function eval_vsf(iops, theta)
220     class(optical_properties) iops
221     double precision theta
222     double precision eval_vsf
223     ! No need to set vsf(0) = 0.
224     ! It's the area under the curve that matters
          , not the value.
225     eval_vsf = interp(theta, iops%vsf_angles,
          iops%vsf_vals, iops%num_vsf)
226
227   end function eval_vsf
228
229   subroutine rope_init(rope, grid)
230     class(rope_state) :: rope
231     type(space_angle_grid) :: grid
232
233     rope%nz = grid%z%num
234     allocate(rope%frond_lengths(rope%nz))
235     allocate(rope%frond_stds(rope%nz))
236     allocate(rope%water_speeds(rope%nz))
237     allocate(rope%water_angles(rope%nz))
238     allocate(rope%num_fronds(rope%nz))
239   end subroutine rope_init
240
241   subroutine rope_deinit(rope)
242     class(rope_state) rope
243     deallocate(rope%frond_lengths)
244     deallocate(rope%frond_stds)
245     deallocate(rope%water_speeds)
246     deallocate(rope%water_angles)
247     deallocate(rope%num_fronds)
248   end subroutine rope_deinit
```

```
249 |
250 |   subroutine set_depth(depth, rope, grid,
    |      depth_layer)
251 |     class(depth_state) depth
252 |     type(rope_state) rope
253 |     type(space_angle_grid) grid
254 |     integer depth_layer
255 |
256 |     depth%frond_length = rope%frond_lengths(
    |        depth_layer)
257 |     depth%frond_std = rope%frond_stds(
    |        depth_layer)
258 |     depth%water_speeds = rope%water_speeds(
    |        depth_layer)
259 |     depth%water_angles = rope%water_angles(
    |        depth_layer)
260 |     depth%num_fronds = rope%num_fronds(
    |        depth_layer)
261 |     depth%depth_layer = depth_layer
262 |     depth%depth = grid%z%vals(depth_layer)
263 |   end subroutine set_depth
264 |
265 |   function length_distribution_cdf(depth, L)
    |      result(output)
266 |     ! C_L(L)
267 |     class(depth_state) depth
268 |     double precision L, L_mean, L_std
269 |     double precision output
270 |
271 |     L_mean = depth%frond_length
272 |     L_std = depth%frond_std
273 |
274 |     call normal_cdf(L, L_mean, L_std, output)
275 |   end function length_distribution_cdf
276 |
277 |   function angle_distribution_pdf(depth, theta_f
    |      ) result(output)
278 |     ! P_{\theta_f}(\theta_f)
279 |     class(depth_state) depth
280 |     double precision theta_f, v_w, theta_w
281 |     double precision output
282 |     double precision diff
283 |
284 |     v_w = depth%water_speeds
285 |     theta_w = depth%water_angles
286 |
287 |     ! von_mises_pdf is only defined on [-pi, pi]
288 |     ! So take difference of angles and input
    |        into
289 |     ! von_mises dist. centered & x=0.
```

```
290 |
291 |     diff = angle_diff_2d(theta_f, theta_w)
292 |
293 |     call von_mises_pdf(diff, 0.d0, v_w, output)
294 |   end function angle_distribution_pdf
295 |
296 |   function angle_mod(theta) result(mod_theta)
297 |     ! Shift theta to the interval [-pi, pi]
298 |     ! which is where von_mises_pdf is defined.
299 |
300 |     double precision theta, mod_theta
301 |
302 |     mod_theta = mod(theta + pi, 2.d0*pi) - pi
303 |   end function angle_mod
304 |
305 |   function angle_diff_2d(theta1, theta2) result(
    |      diff)
306 |     ! Shortest difference between two angles
    |        which may be
307 |     ! in different periods.
308 |     double precision theta1, theta2, diff
309 |     double precision modt1, modt2
310 |
311 |     ! Shift to [0, 2*pi]
312 |     modt1 = mod(theta1, 2*pi)
313 |     modt2 = mod(theta2, 2*pi)
314 |
315 |     ! https://gamedev.stackexchange.com/
    |        questions/4467/comparing-angles-and-
    |        working-out-the-difference
316 |
317 |     diff = pi - abs(abs(modt1-modt2) - pi)
318 |   end function angle_diff_2d
319 |
320 |   function angle_diff_3d(theta, phi, theta_prime
    |      , phi_prime) result(diff)
321 |     ! Angle between two vectors in spherical
    |        coordinates
322 |     double precision theta, phi, theta_prime,
    |        phi_prime
323 |     double precision alpha, diff
324 |
325 |     ! Faster, but produces lots of NaNs (at
    |        least in Python)
326 |     !alpha = sin(theta)*sin(theta_prime)*cos(
    |        theta-theta_prime) + cos(phi)*cos(
    |        phi_prime)
327 |
328 |
329 |     ! Slower, but more accurate
330 |     alpha = (sin(phi)*sin(phi_prime) &
```

```
331 |       * (cos(theta)*cos(theta_prime) + sin(theta
    |          )*sin(theta_prime)) &
332 |       + cos(phi)*cos(phi_prime))
333 |
334 |     ! Avoid out-of-bounds errors due to rounding
335 |     alpha = min(1.d0, alpha)
336 |     alpha = max(-1.d0, alpha)
337 |
338 |     diff = acos(alpha)
339 |   end function angle_diff_3d
340 |
341 |   subroutine vsf_from_function(iops, func)
342 |     class(optical_properties) iops
343 |     double precision, external :: func
344 |     integer i
345 |     type(angle_dim) :: angle1d
346 |
347 |     call angle1d%set_bounds(-1.d0, 1.d0)
348 |     call angle1d%set_num(iops%num_vsf)
349 |     call angle1d%assign_legendre()
350 |
351 |     iops%vsf_angles(:) = acos(angle1d%vals(:))
352 |     do i=1, iops%num_vsf
353 |        iops%vsf_vals(i) = func(iops%vsf_angles(i
    |          ))
354 |     end do
355 |
356 |     call iops%calc_vsf_on_grid()
357 |
358 |     call angle1d%deinit()
359 |   end subroutine vsf_from_function
360 |
361 |   subroutine calc_vsf_on_grid(iops)
362 |     class(optical_properties) iops
363 |     double precision th, ph, thp, php
364 |     integer p, pp
365 |     integer nomega
366 |     double precision norm
367 |
368 |     nomega = iops%grid%angles%nomega
369 |
370 |     ! Calculate cos VSF
371 |     iops%vsf_cos = cos(iops%vsf_angles)
372 |
373 |     ! Normalize cos VSF to 1/(2pi) on [-1, 1]
374 |     iops%vsf_scat_coef = abs(trap_rule_uneven(
    |        iops%vsf_cos, iops%vsf_vals, iops%num_vsf
    |        ))
375 |     iops%vsf_vals(:) = iops%vsf_vals(:) / (2*pi
    |        * iops%vsf_scat_coef)
376 |
```

```
377 |     ! write(*,*) 'norm = ', iops%vsf_scat_coef
378 |     ! write(*,*) 'now: ', trap_rule_uneven(iops%
    |        vsf_cos, iops%vsf_vals, iops%num_vsf)
379 |     ! write(*,*) 'cos: ', iops%vsf_cos
380 |     ! write(*,*) 'vals: ', iops%vsf_vals
381 |
382 |     do p=1, nomega
383 |        th = iops%grid%angles%theta_p(p)
384 |        ph = iops%grid%angles%phi_p(p)
385 |        do  pp=1, nomega
386 |           thp = iops%grid%angles%theta_p(pp)
387 |           php = iops%grid%angles%phi_p(pp)
388 |           ! TODO: Might be better to calculate
    |              average scattering
389 |           ! from angular cell rather than only
    |              using center
390 |           iops%vsf(p, pp) = iops%eval_vsf(
    |              angle_diff_3d(th,ph,thp,php))
391 |        end do
392 |
393 |        ! Normalize each row of VSF (midpoint
    |           rule)
394 |        norm = sum(iops%vsf(p,:) * iops%grid%
    |           angles%area_p(:))
395 |        iops%vsf(p,:) = iops%vsf(p,:) / norm
396 |
397 |        ! % / meter light scattered from cell pp
    |           into direction p.
398 |        ! TODO: Could integrate VSF instead of
    |           just using value at center
399 |        iops%vsf_integral(p, :) = iops%vsf(p, :)
    |           &
400 |              * iops%grid%angles%area_p(:)
401 |        !write(*,*) 'vsf_integral (beta_pp)', p,
    |           ' = ', iops%vsf_integral(p, :)
402 |     end do
403 |
404 |     ! Normalize VSF on unit sphere w.r.t. north
    |        pole
405 |     !iops%vsf_scat_coef = sum(iops%vsf(1,:) *
    |        iops%grid%angles%area_p)
406 |     !iops%vsf = iops%vsf / iops%vsf_scat_coef
407 |     !iops%vsf_integral = iops%vsf_integral /
    |        iops%vsf_scat_coef
408 |   end subroutine calc_vsf_on_grid
409 |
410 |   subroutine iop_deinit(iops)
411 |     class(optical_properties) iops
412 |     deallocate(iops%vsf_angles)
413 |     deallocate(iops%vsf_vals)
```

```
414 |     deallocate(iops%vsf_cos)
415 |     deallocate(iops%vsf)
416 |     deallocate(iops%vsf_integral)
417 |     deallocate(iops%abs_water)
418 |     deallocate(iops%abs_grid)
419 |
420 |   end subroutine iop_deinit
421 |
422 | end module kelp_context
```

light_context.f90

```
  1 | module light_context
  2 |   use sag
  3 |   use rte_sparse_matrices
  4 |   !use hdf5
  5 |   implicit none
  6 |
  7 |   type light_state
  8 |      double precision, dimension(:,:,:),
           allocatable :: irradiance
  9 |      double precision, dimension(:,:,:,:),
           allocatable :: radiance
 10 |      type(space_angle_grid) :: grid
 11 |      type(rte_mat) :: mat
 12 |    contains
 13 |      procedure :: init => light_init
 14 |      procedure :: init_grid => light_init_grid
 15 |      procedure :: calculate_radiance
 16 |      procedure :: calculate_irradiance
 17 |      procedure :: deinit => light_deinit
 18 |      !procedure :: to_hdf => light_to_hdf
 19 |   end type light_state
 20 |
 21 | contains
 22 |
 23 |   ! Init for use with mat
 24 |   subroutine light_init(light, mat)
 25 |     class(light_state) light
 26 |     type(rte_mat) mat
 27 |     integer nx, ny, nz, nomega
 28 |
 29 |     light%mat = mat
 30 |     light%grid = mat%grid
 31 |
 32 |     nx = light%grid%x%num
 33 |     ny = light%grid%y%num
 34 |     nz = light%grid%z%num
 35 |     nomega = light%grid%angles%nomega
 36 |
 37 |     allocate(light%irradiance(nx, ny, nz))
```

```
38      allocate(light%radiance(nx, ny, nz, nomega))
39    end subroutine light_init
40
41    ! Init for use without mat
42    subroutine light_init_grid(light, grid)
43      class(light_state) light
44      type(space_angle_grid) grid
45      integer nx, ny, nz, nomega
46
47      light%grid = grid
48
49      nx = light%grid%x%num
50      ny = light%grid%y%num
51      nz = light%grid%z%num
52      nomega = light%grid%angles%nomega
53
54      allocate(light%irradiance(nx, ny, nz))
55      allocate(light%radiance(nx, ny, nz, nomega))
56    end subroutine light_init_grid
57
58    subroutine calculate_radiance(light)
59      class(light_state) light
60      integer i, j, k, p
61      integer nx, ny, nz, nomega
62      integer index
63
64      nx = light%grid%x%num
65      ny = light%grid%y%num
66      nz = light%grid%z%num
67      nomega = light%grid%angles%nomega
68
69      index = 1
70
71      ! Set initial guess from provided radiance
72      ! Traverse solution vector in order
73      ! so as to avoid calculating index
74      do k=1, nz
75         do i=1, nx
76             do j=1, ny
77               do p=1, nomega
78                  light%mat%sol(index) = light%
                       radiance(i,j,k,p)
79                  index = index + 1
80               end do
81            end do
82         end do
83      end do
84
85      !call light%mat%initial_guess()
86
87      ! Solve (MGMRES)
```

```
88      call light%mat%solve()
89
90      index = 1
91
92      ! Extract solution
93      do k=1, nz
94         do i=1, nx
95            do j=1, ny
96               do p=1, nomega
97                  light%radiance(i,j,k,p) = light%
                       mat%sol(index)
98                  index = index + 1
99               end do
100           end do
101        end do
102     end do
103   end subroutine calculate_radiance
104
105   subroutine calculate_irradiance(light)
106     class(light_state) light
107     integer i, j, k
108     integer nx, ny, nz
109     double precision, dimension(light%grid%
          angles%nomega) :: tmp_rad
110
111     nx = light%grid%x%num
112     ny = light%grid%y%num
113     nz = light%grid%z%num
114
115     do i=1, nx
116        do j=1, ny
117           do k=1, nz
118              ! Use temporary array to avoid
                    creating one
119              ! implicitly at every spatial grid
                    point
120              tmp_rad = light%radiance(i,j,k,:)
121              light%irradiance(i,j,k) = &
122                   light%grid%angles%
                        integrate_points(tmp_rad)
123           end do
124        end do
125     end do
126
127   end subroutine calculate_irradiance
128
129 !  subroutine light_to_hdf(light, radfile,
      irradfile)
130 !    class(light_state) light
131 !    character(len=*) radfile
132 !    character(len=*) irradfile
133 !
```

```
!    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

```
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))
```

```
32 |
33 |     call calculate_light_before_scattering(grid,
   |         bc, iops, source, radiance, path_length,
   |         path_spacing, a_tilde, gn)
34 |
35 |     if (num_scatters .gt. 0) then
36 |        call calculate_light_after_scattering(&
37 |             grid, iops, source, radiance, &
38 |             num_scatters, path_length,
   |                path_spacing, &
39 |             a_tilde, gn)
40 |     end if
41 |
42 |     deallocate(path_length)
43 |     deallocate(path_spacing)
44 |     deallocate(a_tilde)
45 |     deallocate(gn)
46 |     deallocate(source)
47 |   end subroutine calculate_light_with_scattering
48 |
49 |   subroutine calculate_light_before_scattering(
   |      grid, bc, iops, source, radiance,
   |      path_length, path_spacing, a_tilde, gn)
50 |     type(space_angle_grid) grid
51 |     type(boundary_condition) bc
52 |     type(optical_properties) iops
53 |     double precision, dimension(:,:,:,:) ::
   |        radiance, source
54 |     double precision, dimension(:) ::
   |        path_length, path_spacing, a_tilde, gn
55 |     integer i, j, k, p
56 |
57 |     ! !$ integer omp_get_num_procs
58 |     ! !$ integer num_threads_z, num_threads_x
59 |
60 |     ! ! Enable nested parallelism
61 |     ! !$ call omp_set_nested(.true.)
62 |
63 |     ! ! Use nz procs for outer loop,
64 |     ! ! or num_procs if num_procs < nz
65 |     ! ! Divide the rest of the tasks as
   |        appropriate
66 |
67 |     ! !$ num_threads_z = min(omp_get_num_procs()
   |        , grid%z%num)
68 |     ! !$ num_threads_x = min( &
69 |     ! !$     omp_get_num_procs()/num_threads_z, &
70 |     ! !$     grid%x%num)
71 |
72 |     ! !$omp parallel do default(none) private(i,
   |        j,k,p) &
```

```
73 |     ! !$omp shared(grid,iops,radiance,bc,
   |        num_threads_x) &
74 |     ! !$omp private(source,path_length,
   |        path_spacing,a_tilde,gn) &
75 |     ! !$omp num_threads(num_threads_z) if(
   |        num_threads_z .gt. 1)
76 |     do k=1, grid%z%num
77 |        ! !$omp parallel do default(none) private
   |           (i,j,p) &
78 |        ! !$omp firstprivate(k) shared(grid,iops,
   |           radiance,bc) &
79 |        ! !$omp private(source,path_length,
   |           path_spacing,a_tilde,gn) &
80 |        ! !$omp num_threads(num_threads_x) if(
   |           num_threads_x .gt. 1)
81 |        do i=1, grid%x%num
82 |           do j=1, grid%y%num
83 |              do p=1, grid%angles%nomega/2
84 |                 ! Downwelling light
85 |                 call
   |                    attenuate_light_from_surface
   |                    (&
86 |                     grid, iops, source, i, j, k,
   |                         p,&
87 |                     radiance, path_length,
   |                        path_spacing,&
88 |                     a_tilde, gn, bc)
89 |
90 |                 ! No upwelling light before
   |                    scattering
91 |                 radiance(i,j,k,p+grid%angles%
   |                    nomega/2) = 0.d0
92 |              end do
93 |           end do
94 |        end do
95 |        ! !$omp end parallel do
96 |     end do
97 |     ! !$omp end parallel do
98 |   end subroutine
   |      calculate_light_before_scattering
99 |
100|   subroutine attenuate_light_from_surface(&
101|        grid, iops, source, i, j, k, p, radiance,
   |            &
102|        path_length, path_spacing, a_tilde, gn,
   |           bc)
103|     type(space_angle_grid) grid
104|     type(boundary_condition) bc
105|     type(optical_properties) iops
```

```
106     double precision, dimension(:,:,:,:) ::
          radiance, source
107     double precision, dimension(:) ::
          path_length, path_spacing, a_tilde, gn
108     integer i, j, k, p
109     integer num_cells
110     double precision atten
111
112     ! Don't need gn here, so just ignore it
113     call traverse_ray(grid, iops, source, i, j,
          k, p, path_length, path_spacing, a_tilde,
           gn, num_cells)
114
115     ! Start with surface bc and attenuate along
          path
116     atten = sum(path_spacing(1:num_cells) *
          a_tilde(1:num_cells))
117     ! Avoid underflow
118     if(atten .lt. 100.d0) then
119        radiance(i,j,k,p) = bc%bc_grid(p) * exp(-
             atten)
120     else
121        radiance(i,j,k,p) = 0.d0
122     end if
123
124   end subroutine attenuate_light_from_surface
125
126   subroutine calculate_light_after_scattering(
        grid, iops, source, radiance,&
127        num_scatters, path_length, path_spacing,
             a_tilde, gn)
128     type(space_angle_grid) grid
129     type(optical_properties) iops
130     double precision, dimension(:,:,:,:) ::
          radiance, source
131     integer num_scatters
132     double precision, dimension(:) ::
          path_length, path_spacing, a_tilde, gn
133     double precision, dimension(:,:,:,:),
          allocatable :: rad_scatter
134     integer n
135     double precision bb
136
137     allocate(rad_scatter(grid%x%num, grid%y%num,
           grid%z%num, grid%angles%nomega))
138     rad_scatter = radiance
139     bb = iops%scat
140
141     do n=1, num_scatters
142        write(*,*) 'scatter #', n
143        call scatter(grid, iops, source,
```

```
            rad_scatter, path_length, path_spacing
            , a_tilde, gn)
144         radiance = radiance + bb**n * rad_scatter
145       end do
146
147       deallocate(rad_scatter)
148     end subroutine
          calculate_light_after_scattering
149
150     ! Perform one scattering event
151     subroutine scatter(grid, iops, source,
          rad_scatter, path_length, path_spacing,
          a_tilde, gn)
152       type(space_angle_grid) grid
153       type(optical_properties) iops
154       double precision, dimension(:,:,:,:) ::
            rad_scatter, source
155       double precision, dimension(:,:,:,:),
            allocatable :: scatter_integral
156       double precision, dimension(:) ::
            path_length, path_spacing, a_tilde, gn
157       integer nx, ny, nz, nomega
158
159       nx = grid%x%num
160       ny = grid%y%num
161       nz = grid%z%num
162       nomega = grid%angles%nomega
163
164       allocate(scatter_integral(nx, ny, nz, nomega
            ))
165
166       call calculate_source(grid, iops,
            rad_scatter, source, scatter_integral)
167       call advect_light(grid, iops, source,
            rad_scatter, path_length, path_spacing,
            a_tilde, gn)
168
169       deallocate(scatter_integral)
170     end subroutine scatter
171
172     ! Calculate source from no-scatter or previous
          scattering layer
173     subroutine calculate_source(grid, iops,
          rad_scatter, source, scatter_integral)
174       type(space_angle_grid) grid
175       type(optical_properties) iops
176       double precision, dimension(:,:,:,:) ::
            rad_scatter
177       double precision, dimension(:,:,:,:) ::
            source
```

```
178 |     double precision, dimension(:,:,:,:) ::
    |        scatter_integral
179 |     type(index_list) indices
180 |     integer nx, ny, nz, nomega
181 |     integer i, j, k, p
182 |
183 |     !$ integer omp_get_num_procs
184 |     !$ integer num_threads_z, num_threads_x
185 |
186 |     nx = grid%x%num
187 |     ny = grid%y%num
188 |     nz = grid%z%num
189 |     nomega = grid%angles%nomega
190 |
191 |     ! Enable nested parallelism
192 |     !$ call omp_set_nested(.true.)
193 |
194 |     ! Use nz procs for outer loop,
195 |     ! or num_procs if num_procs < nz
196 |     ! Divide the rest of the tasks as
    |        appropriate
197 |
198 |     !$ num_threads_z = min(omp_get_num_procs(),
    |        grid%z%num)
199 |     !$ num_threads_x = min( &
200 |     !$    omp_get_num_procs()/num_threads_z, &
201 |     !$    grid%x%num)
202 |
203 |     !$omp parallel do default(none) private(
    |        indices) &
204 |     !$omp private(i,j,k,p) shared(nx,ny,nz,
    |        nomega) &
205 |     !$omp shared(iops, rad_scatter,
    |        scatter_integral) &
206 |     !$omp shared(num_threads_x) &
207 |     !$omp num_threads(num_threads_z) if(
    |        num_threads_z .gt. 1)
208 |     do k=1, nz
209 |        indices%k = k
210 |        !$omp parallel do default(none)
    |           firstprivate(indices,k) &
211 |        !$omp private(i,j,p) shared(nx,ny,nz,
    |           nomega) &
212 |        !$omp shared(iops, rad_scatter,
    |           scatter_integral) &
213 |        !$omp num_threads(num_threads_x) if(
    |           num_threads_x .gt. 1)
214 |        do i=1, nx
215 |          indices%i = i
216 |          do j=1, ny
```

```
217 |              indices%j = j
218 |              do p=1, nomega
219 |                 indices%p = p
220 |                 call calculate_scatter_integral
    |                    (&
221 |                     iops, rad_scatter,&
222 |                     scatter_integral,&
223 |                     indices)
224 |              end do
225 |           end do
226 |        end do
227 |        !$omp end parallel do
228 |     end do
229 |     !$omp end parallel do
230 |
231 |     source(:,:,:,:) = -rad_scatter(:,:,:,:) +
    |        scatter_integral(:,:,:,:)
232 |
233 |     write(*,*) 'source: ', sum(source)/size(
    |        source), minval(source), maxval(source)
234 |
235 |   end subroutine calculate_source
236 |
237 |   subroutine calculate_scatter_integral(iops,
    |      rad_scatter, scatter_integral, indices)
238 |     type(optical_properties) iops
239 |     double precision, dimension(:,:,:,:) ::
    |        rad_scatter, scatter_integral
240 |     type(index_list) indices
241 |
242 |     scatter_integral(indices%i,indices%j,indices
    |        %k,indices%p) &
243 |           = sum(iops%vsf_integral(indices%p, :) &
244 |           * rad_scatter(&
245 |             indices%i,&
246 |             indices%j,&
247 |             indices%k,:))
248 |
249 |   end subroutine calculate_scatter_integral
250 |
251 |   subroutine advect_light(grid, iops, source,
    |      rad_scatter, path_length, path_spacing,
    |      a_tilde, gn)
252 |     type(space_angle_grid) grid
253 |     type(optical_properties) iops
254 |     double precision, dimension(:,:,:,:) ::
    |        rad_scatter, source
255 |     double precision, dimension(:) ::
    |        path_length, path_spacing, a_tilde, gn
256 |     integer i, j, k, p
257 |
```

```
258 |     !$ integer omp_get_num_procs
259 |     !$ integer num_threads_z, num_threads_x
260 |
261 |     ! Enable nested parallelism
262 |     !$ call omp_set_nested(.true.)
263 |
264 |     ! Use nz procs for outer loop,
265 |     ! or num_procs if num_procs < nz
266 |     ! Divide the rest of the tasks as
          |    appropriate
267 |
268 |     !$ num_threads_z = min(omp_get_num_procs(),
          |    grid%z%num)
269 |     !$ num_threads_x = min( &
270 |     !$     omp_get_num_procs()/num_threads_z, &
271 |     !$     grid%x%num)
272 |
273 |     !$omp parallel do default(none) &
274 |     !$omp private(i,j,k,p) &
275 |     !$omp shared(rad_scatter,source,grid,iops,
          |    num_threads_x) &
276 |     !$omp private(path_length,path_spacing,
          |    a_tilde,gn) &
277 |     !$omp num_threads(num_threads_z) if(
          |    num_threads_z .gt. 1)
278 |     do k=1, grid%z%num
279 |        !$omp parallel do default(none) &
280 |        !$omp firstprivate(k) private(i,j,p) &
281 |        !$omp shared(rad_scatter,source,grid,iops
          |       ) &
282 |        !$omp private(path_length,path_spacing,
          |       a_tilde,gn) &
283 |        !$omp num_threads(num_threads_x) if(
          |       num_threads_x .gt. 1)
284 |        do i=1, grid%x%num
285 |          do j=1, grid%y%num
286 |              do p=1, grid%angles%nomega
287 |                 call integrate_ray(grid, iops,
                           source,&
288 |                      rad_scatter, path_length,
                             path_spacing,&
289 |                      a_tilde, gn, i, j, k, p)
290 |            end do
291 |          end do
292 |        end do
293 |        !$omp end parallel do
294 |     end do
295 |     !$omp end parallel do
296 |   end subroutine advect_light
297 |
```

```
298   ! New algorithm, double integral over
          piecewise-constant 1d funcs
299   subroutine integrate_ray(grid, iops, source,
          rad_scatter, path_length, path_spacing,
          a_tilde, gn, i, j, k, p)
300     type(space_angle_grid) :: grid
301     type(optical_properties) :: iops
302     double precision, dimension(:,:,:,:) ::
            source
303     double precision, dimension(:,:,:,:) ::
            rad_scatter
304     integer :: i, j, k, p
305     ! The following are only passed to avoid
            unnecessary allocation
306     double precision, dimension(:) ::
            path_length, path_spacing, a_tilde, gn
307     integer num_cells
308
309     call traverse_ray(grid, iops, source, i, j,
            k, p, path_length, path_spacing, a_tilde,
            gn, num_cells)
310     rad_scatter(i,j,k,p) =
            calculate_ray_integral(num_cells,
            path_length, path_spacing, a_tilde, gn)
311   end subroutine integrate_ray
312
313   function calculate_ray_integral(num_cells, s,
          ds, a_tilde, gn) result(integral)
314     ! Double integral which accumulates all
            scattered light along the path
315     ! via an angular integral and attenuates it
            by integrating along the path
316     integer :: num_cells
317     double precision, dimension(num_cells) :: ds
            , a_tilde, gn
318     double precision, dimension(num_cells+1) ::
            s
319     double precision :: integral
320     double precision bi, di
321     integer i, j
322
323     integral = 0
324     do i=1, num_cells
325        bi = -a_tilde(i)*s(i+1)
326        do j=i+1, num_cells
327           bi = bi - a_tilde(j)*ds(j)
328        end do
329
330        ! WARNING: This will overflow if a_tilde
              is too large.
331        if(a_tilde(i) .eq. 0) then
```

```
332           di = ds(i)
333         else
334           di = (exp(a_tilde(i)*s(i+1))-exp(
                  a_tilde(i)*s(i)))/a_tilde(i)
335         end if
336
337         integral = integral + gn(i)*di * exp(bi)
338       end do
339
340     end function calculate_ray_integral
341
342     ! Calculate maximum number of cells a path
           through the grid could take
343     ! This is a loose upper bound
344     function calculate_max_cells(grid) result(
          max_cells)
345       type(space_angle_grid) :: grid
346       integer :: max_cells
347       double precision dx, dy, zrange, phi_middle
348
349       ! Angle that will have the longest ray
350       phi_middle = grid%angles%phi(grid%angles%
            nphi/2)
351       dx = grid%x%spacing(1)
352       dy = grid%y%spacing(1)
353       zrange = grid%z%maxval - grid%z%minval
354
355       max_cells = grid%z%num + ceiling((1/dx+1/dy)
            *zrange*tan(phi_middle))
356     end function calculate_max_cells
357
358     ! Traverse from surface or bottom to point (xi
           , yj, zk)
359     ! in the direction omega_p, extracting path
           lengths (ds) and
360     ! function values (f) along the way,
361     ! as well as number of cells traversed (n).
362     subroutine traverse_ray(grid, iops, source, i,
            j, k, p, s_array, ds, a_tilde, gn,
          num_cells)
363       type(space_angle_grid) :: grid
364       type(optical_properties) :: iops
365       double precision, dimension(:,:,:,:) ::
            source
366       integer :: i, j, k, p
367       double precision, dimension(:) :: s_array,
            ds, a_tilde, gn
368       integer :: num_cells
369
370       integer t
371       double precision p0x, p0y, p0z
```

```
372     double precision p1x, p1y, p1z
373     double precision z0
374     double precision s_tilde, s
375     integer dir_x, dir_y, dir_z
376     integer shift_x, shift_y, shift_z
377     integer cell_x, cell_y, cell_z
378     integer edge_x, edge_y, edge_z
379     integer first_x, last_x, first_y, last_y,
           last_z
380     double precision s_next_x, s_next_y,
           s_next_z, s_next
381     double precision x_factor, y_factor,
           z_factor
382     double precision ds_x, ds_y
383     double precision, dimension(grid%z%num) ::
           ds_z
384     double precision smx, smy
385
386     ! Divide by these numbers to get path
           separation
387     ! from separation in individual dimensions
388     x_factor = grid%angles%sin_phi_p(p) * grid%
           angles%cos_theta_p(p)
389     y_factor = grid%angles%sin_phi_p(p) * grid%
           angles%sin_theta_p(p)
390     z_factor = grid%angles%cos_phi_p(p)
391
392     ! Destination point
393     p1x = grid%x%vals(i)
394     p1y = grid%y%vals(j)
395     p1z = grid%z%vals(k)
396
397     !write(*,*) 'START PATH.'
398     !write(*,*) 'ijk = ', i, j, k
399
400     ! Direction
401     if(p .le. grid%angles%nomega/2) then
402        ! Downwelling light originates from
              surface
403        z0 = grid%z%minval
404        dir_z = 1
405     else
406        ! Upwelling light originates from bottom
407        z0 = grid%z%maxval
408        dir_z = -1
409     end if
410
411     ! Total path length from origin to
           destination
412     ! (sign is correct for upwelling and
           downwelling)
```

```
413     s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p
            )
414
415     ! Path spacings between edge intersections
            in each dimension
416     ! Set to 2*s_tilde if infinite in this
            dimension so that it's unreachable
417     ! Assume x & y spacings are uniform,
418     ! so it's okay to just use the first value.
419     if(x_factor .eq. 0) then
420        ds_x = 2*s_tilde
421     else
422        ds_x = abs(grid%x%spacing(1)/x_factor)
423     end if
424     if(y_factor .eq. 0) then
425        ds_y = 2*s_tilde
426     else
427        ds_y = abs(grid%y%spacing(1)/y_factor)
428     end if
429
430     ! This one is an array because z spacing can
            vary
431     ! z_factor should never be 0, because the
            ray will never
432     ! reach the surface or bottom.
433     ds_z(1:grid%z%num) = dir_z * grid%z%spacing
            (1:grid%z%num)/z_factor
434
435     ! Origin point
436     p0x = p1x - s_tilde * x_factor
437     p0y = p1y - s_tilde * y_factor
438     p0z = p1z - s_tilde * z_factor
439
440     ! Direction of ray in each dimension. 1 =>
            increasing. -1 => decreasing.
441     dir_x = int(sgn(p1x-p0x))
442     dir_y = int(sgn(p1y-p0y))
443
444     ! Shifts
445     ! Conversion from cell_inds to edge_inds
446     ! merge is fortran's ternary operator
447     shift_x = merge(1,0,dir_x>0)
448     shift_y = merge(1,0,dir_y>0)
449     shift_z = merge(1,0,dir_z>0)
450
451     ! Indices for cell containing origin point
452     cell_x = floor((p0x-grid%x%minval)/grid%x%
            spacing(1)) + 1
453     cell_y = floor((p0y-grid%y%minval)/grid%y%
            spacing(1)) + 1
454     ! x and y may be in periodic image, so shift
            back.
```

```
455 |     cell_x = mod1(cell_x, grid%x%num)
456 |     cell_y = mod1(cell_y, grid%y%num)
457 |
458 |     ! z starts at top or bottom depending on
    |        direction.
459 |     if(dir_z > 0) then
460 |        cell_z = 1
461 |     else
462 |        cell_z = grid%z%num
463 |     end if
464 |
465 |     ! Edge indices preceeding starting cells
466 |     edge_x = mod1(cell_x + shift_x, grid%x%num)
467 |     edge_y = mod1(cell_y + shift_y, grid%y%num)
468 |     edge_z = mod1(cell_z + shift_z, grid%z%num)
469 |
470 |     ! First and last cells given direction
471 |     if(dir_x .gt. 0) then
472 |        first_x = 1
473 |        last_x = grid%x%num
474 |     else
475 |        first_x = grid%x%num
476 |        last_x = 1
477 |     end if
478 |     if(dir_y .gt. 0) then
479 |        first_y = 1
480 |        last_y = grid%y%num
481 |     else
482 |        first_y = grid%y%num
483 |        last_y = 1
484 |     end if
485 |     if(dir_z .gt. 0) then
486 |        last_z = grid%z%num
487 |     else
488 |        last_z = 1
489 |     end if
490 |
491 |     ! Calculate periodic images
492 |     smx = shift_mod(p0x, grid%x%minval, grid%x%
    |        maxval)
493 |     smy = shift_mod(p0y, grid%y%minval, grid%y%
    |        maxval)
494 |
495 |     ! Path length to next edge plane in each
    |        dimension
496 |     if(abs(x_factor) .lt. 1.d-10) then
497 |        ! Will never cross, so set above total
    |           path length
498 |        s_next_x = 2*s_tilde
499 |     else if(cell_x .eq. last_x) then
```

```
500         ! If starts out at last cell, then
               compare to periodic image
501         s_next_x = (grid%x%edges(first_x) + dir_x
               * (grid%x%maxval - grid%x%minval)&
502               - smx) / x_factor
503      else
504         ! Otherwise, just compare to next cell
505         s_next_x = (grid%x%edges(edge_x) - smx) /
               x_factor
506      end if
507
508      ! Path length to next edge plane in each
            dimension
509      if(abs(y_factor) .lt. 1.d-10) then
510         ! Will never cross, so set above total
               path length
511         s_next_y = 2*s_tilde
512      else if(cell_y .eq. last_y) then
513         ! If starts out at last cell, then
               compare to periodic image
514         s_next_y = (grid%y%edges(first_y) + dir_y
               * (grid%y%maxval - grid%y%minval)&
515               - smy) / y_factor
516      else
517         ! Otherwise, just compare to next cell
518         s_next_y = (grid%y%edges(edge_y) - smy) /
               y_factor
519      end if
520
521      s_next_z = ds_z(cell_z)
522
523      ! Initialize path
524      s = 0.d0
525      s_array(1) = 0.d0
526
527      ! Start with t=0 so that we can increment
            before storing,
528      ! so that t will be the number of grid cells
            at the end of the loop.
529      t=0
530
531      ! s is the beginning of the current cell,
532      ! s_next is the end of the current cell.
533      do while (s .lt. s_tilde)
534         ! Move cell counter
535         t = t + 1
536
537         ! Extract function values
538         a_tilde(t) = iops%abs_grid(cell_x, cell_y
               , cell_z)
539         gn(t) = source(cell_x, cell_y, cell_z, p)
540
```

```
541 |         !write(*,*) ''
542 |         !write(*,*) 's_next_x = ', s_next_x
543 |         !write(*,*) 's_next_y = ', s_next_y
544 |         !write(*,*) 's_next_z = ', s_next_z
545 |         !write(*,*) 'theta, phi =', grid%angles%
    |            theta_p(p)*180.d0/pi, grid%angles%
    |            phi_p(p)*180.d0/pi
546 |         !write(*,*) 's = ', s, '/', s_tilde
547 |         !write(*,*) 'cell_z =', cell_z, '/', grid
    |            %z%num
548 |         !write(*,*) 's_next_z =', s_next_z
549 |         !write(*,*) 'last_z =', last_z
550 |         !write(*,*) 'new'
551 |
552 |         ! Move to next cell in path
553 |         if(s_next_x .le. min(s_next_y, s_next_z))
    |             then
554 |            ! x edge is closest
555 |            s_next = s_next_x
556 |
557 |            ! Increment indices (periodic)
558 |            cell_x = mod1(cell_x + dir_x, grid%x%
    |               num)
559 |            edge_x = mod1(edge_x + dir_x, grid%x%
    |               num)
560 |
561 |            ! x intersection after the one at s=
    |               s_next
562 |            s_next_x = s_next + ds_x
563 |
564 |         else if (s_next_y .le. min(s_next_x,
    |            s_next_z)) then
565 |            ! y edge is closest
566 |            s_next = s_next_y
567 |
568 |            ! Increment indices (periodic)
569 |            cell_y = mod1(cell_y + dir_y, grid%y%
    |               num)
570 |            edge_y = mod1(edge_y + dir_y, grid%y%
    |               num)
571 |
572 |            ! y intersection after the one at s=
    |               s_next
573 |            s_next_y = s_next + ds_y
574 |
575 |         else if(s_next_z .le. min(s_next_x,
    |            s_next_y)) then
576 |            ! z edge is closest
577 |            s_next = s_next_z
578 |
579 |            ! Increment indices
```

```
580             cell_z = cell_z + dir_z
581             edge_z = edge_z + dir_z
582
583             !write(*,*) 'z edge, s_next =', s_next
584
585             ! z intersection after the one at s=
                    s_next
586             if(cell_z .lt. last_z) then
587                ! Only look ahead if we aren't at
                      the end
588                s_next_z = s_next + ds_z(cell_z)
589             else
590                ! Otherwise, no need to continue.
591                ! this is our final destination.
592             !   exit
593                s_next_z = 2*s_tilde
594                !write(*,*) 'end. s_next_z =',
                      s_next_z
595             end if
596
597          end if
598
599          ! Cut off early if this is the end
600          ! This will be the last cell traversed if
                s_next >= s_tilde
601          s_next = min(s_tilde, s_next)
602
603          ! Store path length
604          s_array(t+1) = s_next
605          ! Extract path length from same cell as
                function vals
606          ds(t) = s_next - s
607
608          ! Update path length
609          s = s_next
610       end do
611
612       ! Return number of cells traversed
613       num_cells = t
614
615     end subroutine traverse_ray
616 end module asymptotics
```

light_interface.f90

```
1  module light_interface_module
2    use rte3d
3    use kelp3d
4    use asymptotics
5    implicit none
6
7  contains
8    subroutine full_light_calculations( &
```

```
9       ! OPTICAL PROPERTIES
10      absorptance_kelp, & ! NOT THE SAME AS
            ABSORPTION COEFFICIENT
11      abs_water, &
12      scat, &
13      num_vsf, &
14      vsf_file, &
15      ! SUNLIGHT
16      solar_zenith, &
17      solar_azimuthal, &
18      surface_irrad, &
19      ! KELP &
20      num_si, &
21      si_area, &
22      si_ind, &
23      frond_thickness, &
24      frond_aspect_ratio, &
25      frond_shape_ratio, &
26      ! WATER CURRENT
27      current_speeds, &
28      current_angles, &
29      ! SPACING
30      rope_spacing, &
31      depth_spacing, &
32      ! SOLVER PARAMETERS
33      nx, &
34      ny, &
35      nz, &
36      ntheta, &
37      nphi, &
38      num_scatters, &
39      ! FINAL RESULTS
40      perceived_irrad, &
41      avg_irrad)
42
43      implicit none
44
45      ! OPTICAL PROPERTIES
46      integer, intent(in) :: nx, ny, nz, ntheta,
            nphi
47      ! Absorption and scattering coefficients
48      double precision, intent(in) ::
            absorptance_kelp, scat
49      double precision, dimension(nz), intent(in)
            :: abs_water
50      ! Volume scattering function
51      integer, intent(in) :: num_vsf
52      character(len=*) :: vsf_file
53      !double precision, dimension(num_vsf),
            intent(int) :: vsf_angles
54      !double precision, dimension(num_vsf),
            intent(int) :: vsf_vals
```

```
55
56      ! SUNLIGHT
57      double precision, intent(in) :: solar_zenith
58      double precision, intent(in) ::
           solar_azimuthal
59      double precision, intent(in) ::
           surface_irrad
60
61      ! KELP
62      ! Number of Superindividuals in each depth
           level
63      integer, intent(in) :: num_si
64      ! si_area(i,j) = area of superindividual j
           at depth i
65      double precision, dimension(nz, num_si),
           intent(in) :: si_area
66      ! si_ind(i,j) = number of inidividuals
           represented by superindividual j at depth
            i
67      double precision, dimension(nz, num_si),
           intent(in) :: si_ind
68      ! Thickness of each frond
69      double precision, intent(in) ::
           frond_thickness
70      ! Ratio of length to width (0,infty)
71      double precision, intent(in) ::
           frond_aspect_ratio
72      ! Rescaled position of greatest width (0=
           base, 1=tip)
73      double precision, intent(in) ::
           frond_shape_ratio
74
75      ! WATER CURRENT
76      double precision, dimension(nz), intent(in)
           :: current_speeds
77      double precision, dimension(nz), intent(in)
           :: current_angles
78
79      ! SPACING
80      double precision, intent(in) :: rope_spacing
81      double precision, dimension(nz), intent(in)
           :: depth_spacing
82      ! SOLVER PARAMETERS
83      integer, intent(in) :: num_scatters
84
85      ! FINAL RESULT
86      real, dimension(nz), intent(out) ::
           avg_irrad, perceived_irrad
87
88      !-------------!
89
```

```
90      double precision xmin, xmax, ymin, ymax,
           zmin, zmax
91      character(len=5), parameter :: fmtstr = 'E13
           .4'
92      !double precision, dimension(num_vsf) ::
           vsf_angles, vsf_vals
93      double precision max_rad, decay
94      integer quadrature_degree
95
96      type(space_angle_grid) grid
97      type(optical_properties) iops
98      type(light_state) light
99      type(rope_state) rope
100     type(frond_shape) frond
101     type(boundary_condition) bc
102
103     double precision, dimension(:), allocatable
           :: pop_length_means, pop_length_stds
104     ! Number of fronds in each depth layer
105     double precision, dimension(:), allocatable
           :: num_fronds
106     double precision, dimension(:,:,:),
           allocatable :: p_kelp
107
108     write(*,*) 'Light calculation'
109
110     allocate(pop_length_means(nz))
111     allocate(pop_length_stds(nz))
112     allocate(num_fronds(nz))
113     allocate(p_kelp(nx, ny, nz))
114
115     xmin = -rope_spacing/2
116     xmax = rope_spacing/2
117
118     ymin = -rope_spacing/2
119     ymax = rope_spacing/2
120
121     zmin = 0.d0
122     zmax = sum(depth_spacing)
123
124     write(*,*) 'Grid'
125     call grid%set_bounds(xmin, xmax, ymin, ymax,
            zmin, zmax)
126     call grid%set_num(nx, ny, nz, ntheta, nphi)
127     call grid%init()
128     !call grid%set_uniform_spacing_from_num()
129     call grid%z%set_spacing_array(depth_spacing)
130
131     call rope%init(grid)
132
```

```
133     write(*,*) 'Rope'
134     ! Calculate kelp distribution
135     call calculate_length_dist_from_superinds( &
136     nz, &
137     num_si, &
138     si_area, &
139     si_ind, &
140     frond_aspect_ratio, &
141     num_fronds, &
142     pop_length_means, &
143     pop_length_stds)
144
145     rope%frond_lengths = pop_length_means
146     rope%frond_stds = pop_length_stds
147     rope%num_fronds = num_fronds
148     rope%water_speeds = current_speeds
149     rope%water_angles = current_angles
150
151     write(*,*) 'frond_lengths  =', rope%
            frond_lengths
152     write(*,*) 'frond_stds  =', rope%frond_stds
153     write(*,*) 'num_fronds  =', rope%num_fronds
154     write(*,*) 'water_speeds  =', rope%
            water_speeds
155     write(*,*) 'water_angles  =', rope%
            water_angles
156
157     write(*,*) 'Frond'
158     ! INIT FROND
159     call frond%set_shape(frond_shape_ratio,
            frond_aspect_ratio, frond_thickness)
160     ! CALCULATE KELP
161     quadrature_degree = 5
162     call calculate_kelp_on_grid(grid, p_kelp,
            frond, rope, quadrature_degree)
163     ! INIT IOPS
164     iops%num_vsf = num_vsf
165     call iops%init(grid)
166     write(*,*) 'IOPs'
167     iops%abs_kelp = absorptance_kelp /
            frond_thickness
168     iops%abs_water = abs_water
169     iops%scat = scat
170
171     !write(*,*) 'iop init'
172     !iops%vsf_angles = vsf_angles
173     !iops%vsf_vals = vsf_vals
174     call iops%load_vsf(vsf_file, fmtstr)
175
176     ! load_vsf already calls calc_vsf_on_grid
177     !call iops%calc_vsf_on_grid()
```

```
178       call iops%calculate_coef_grids(p_kelp)
179
180       !write(*,*) 'BC'
181       max_rad = 1.d0 ! Doesn't matter because we'
             ll rescale
182       decay = 1.d0 ! Does matter, but maybe not
             much. Determines drop-off from angle
183       call bc%init(grid, solar_zenith,
             solar_azimuthal, decay, max_rad)
184       ! Rescale surface radiance to match surface
             irradiance
185       bc%bc_grid = bc%bc_grid * surface_irrad /
             grid%angles%integrate_points(bc%bc_grid)
186
187       write(*,*) 'bc'
188       write(*,*) bc%bc_grid
189
190       ! write(*,*) 'bc'
191       ! do i=1, grid%y%num
192       !     write(*,'(10F15.3)') bc%bc_grid(i,:)
193       ! end do
194
195       call light%init_grid(grid)
196
197       write(*,*) 'Scatter'
198       call calculate_light_with_scattering(grid,
             bc, iops, light%radiance, num_scatters)
199
200       write(*,*) 'Irrad'
201       call light%calculate_irradiance()
202
203       ! Calculate output variables
204       call calculate_average_irradiance(grid,
             light, avg_irrad)
205       call calculate_perceived_irrad(grid, p_kelp,
              &
206            perceived_irrad, light%irradiance)
207
208       !write(*,*) 'vsf_angles = ', iops%vsf_angles
209       !write(*,*) 'vsf_vals = ', iops%vsf_vals
210       !write(*,*) 'vsf norm  = ', grid%
             integrate_angle_2d(iops%vsf(1,1,:,:))
211
212       ! write(*,*) 'abs_water = ', abs_water
213       ! write(*,*) 'scat_water = ', scat_water
214       write(*,*) 'kelp '
215       write(*,*) p_kelp(:,:,:)
216       write(*,*) 'ft =', frond%ft
217
218       write(*,*) 'irrad'
```

```
219     write(*,*) light%irradiance
220
221     write(*,*) 'avg_irrad = ', avg_irrad
222     write(*,*) 'perceived_irrad = ',
           perceived_irrad
223
224     write(*,*) 'deinit'
225     call bc%deinit()
226     !write(*,*) 'a'
227     call iops%deinit()
228     !write(*,*) 'b'
229     call light%deinit()
230     !write(*,*) 'c'
231     call rope%deinit()
232     !write(*,*) 'd'
233     call grid%deinit()
234     !write(*,*) 'e'
235
236     deallocate(pop_length_means)
237     deallocate(pop_length_stds)
238     deallocate(num_fronds)
239     deallocate(p_kelp)
240
241     !write(*,*) 'done'
242   end subroutine full_light_calculations
243
244   subroutine
          calculate_length_dist_from_superinds( &
245     nz, &
246     num_si, &
247     si_area, &
248     si_ind, &
249     frond_aspect_ratio, &
250     num_fronds, &
251     pop_length_means, &
252     pop_length_stds)
253
254     implicit none
255
256     ! Number of depth levels
257     integer, intent(in) :: nz
258     ! Number of Superindividuals in each depth
           level
259     integer, intent(in) :: num_si
260     ! si_area(i,j) = area of superindividual j
           at depth i
261     double precision, dimension(nz, num_si),
           intent(in) :: si_area
262     ! si_area(i,j) = number of inidividuals
           represented by superindividual j at depth
            i
```

```
263     double precision, dimension(nz, num_si),
           intent(in) :: si_ind
264     double precision, intent(in) ::
           frond_aspect_ratio
265
266     double precision, dimension(nz), intent(out)
           :: num_fronds
267     ! Population mean area at each depth level
268     double precision, dimension(nz), intent(out)
           :: pop_length_means
269     ! Population area standard deviation at each
           depth level
270     double precision, dimension(nz), intent(out)
           :: pop_length_stds
271
272     !---------------!
273
274     integer i, k
275     ! Numerators for mean and std
276     double precision mean_num, std_num
277     ! Convert area to length
278     double precision, dimension(num_si) ::
           si_length
279
280     do k=1, nz
281        mean_num = 0.d0
282        std_num = 0.d0
283        num_fronds(k) = 0
284
285        do i=1, num_si
286           si_length(i) = sqrt(2.d0*
                 frond_aspect_ratio*si_area(k,i))
287           mean_num = mean_num + si_length(i)
288           num_fronds(k) = num_fronds(k) + si_ind
                 (k,i)
289        end do
290
291        pop_length_means(k) = mean_num /
              num_fronds(k)
292
293        do i=1, num_si
294           std_num = std_num + (si_length(i) -
                 pop_length_means(k))**2
295        end do
296
297        pop_length_stds(k) = std_num / (
              num_fronds(k) - 1)
298
299     end do
300
301  end subroutine
        calculate_length_dist_from_superinds
```

```
302 |
303 |   subroutine calculate_average_irradiance(grid,
    |      light, avg_irrad)
304 |     type(space_angle_grid) grid
305 |     type(light_state) light
306 |     real, dimension(:) :: avg_irrad
307 |     integer k, nx, ny, nz
308 |
309 |     nx = grid%x%num
310 |     ny = grid%y%num
311 |     nz = grid%z%num
312 |
313 |     do k=1, nz
314 |        avg_irrad(k) = real(sum(light%irradiance
    |           (:,:,k)) / nx / ny)
315 |     end do
316 |   end subroutine calculate_average_irradiance
317 |
318 |   subroutine calculate_perceived_irrad(grid,
    |      p_kelp, &
319 |        perceived_irrad, irradiance)
320 |     type(space_angle_grid) grid
321 |     double precision, dimension(:,:,:) :: p_kelp
322 |     real, dimension(:) :: perceived_irrad
323 |     double precision, dimension(:,:,:) ::
    |        irradiance
324 |
325 |     integer k
326 |
327 |     ! Calculate the average irradiance
    |        experienced over the frond.
328 |     ! Has same units as irradiance.
329 |     do k=1, grid%z%num
330 |        perceived_irrad(k) = real( &
331 |             sum(p_kelp(:,:,k)*irradiance(:,:,k))
    |                 &
332 |             / sum(p_kelp(:,:,k)))
333 |     end do
334 |
335 |   end subroutine calculate_perceived_irrad
336 |
337 | end module light_interface_module
```