

©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:

Accepted:

Advisor
Dr. Kevin Kreider

Dean of the College
Dr. John Green

Co-Advisor
Dr. Curtis Clemons

Dean of the Graduate School
Dr. Chand Midha

Faculty Reader
Dr. Gerald Young

Date

Department Chair
Dr. Kevin Kreider

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

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Background on Kelp Models	3
1.3 Background on Radiative Transfer	6
1.4 Overview of Thesis	7
II. KELP MODEL	9
2.1 Physical Setup	9
2.2 Coordinate System	10
2.3 Population Distributions	11
2.4 Spatial Distribution	16
III. LIGHT MODEL	23
3.1 Optical Definitions	23
3.2 The Radiative Transfer Equation	25
3.3 Low-Scattering Approximation	27

IV. NUMERICAL SOLUTION	32
4.1 Super-Individuals	32
4.2 Discrete Grid	33
4.3 Finite Difference	41
4.4 Numerical Asymptotics	46
V. PARAMETER VALUES	49
5.1 Parameters from Literature	49
5.2 Frond Distribution Parameters	49
VI. MODEL ANALYSIS	52
6.1 Grid Study	52
6.2 Asymptotic Convergence	52
6.3 Sensitivity Analysis	52
6.4 Kelp Cultivation Simulation	53
VII. CONCLUSION	54
APPENDICES	58
APPENDIX A. RAY TRACING ALGORITHM	59
APPENDIX B. FORTRAN CODE	63

LIST OF TABLES

Table		Page
4.1	Breakdown of nonzero matrix elements by derivative case	45
5.1	Parameter values	50
5.2	Petzold IOP summary [11]. I'll pull a few cases from here and point out when the asymptotic approximation will work.	51

LIST OF FIGURES

Figure	Page
1.1 <i>Saccharina latissima</i> being harvested	4
2.1 Rendering of four nearby vertical kelp ropes	10
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	11
2.3 Simplified kite-shaped frond	12
2.4 von Mises distribution for a variety of parameters	15
2.5 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$	16
2.6 A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$	17
2.7 Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$	20
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 $(\theta, s) = (3\pi/4, 3/2)$	21
2.9 Contour plot of the probability of occupying sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$	22
4.1 Spatial grid	34
4.2 Angular grid at each point in space	36
4.3 Angular grid	38

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-intensive 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 [2] for *S. latissima* has been produced and integrated into the SINMOD [14] 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 numerical finite difference solution to the Radiative Transfer Equation, followed by asymptotic approximations that prove to be sufficiently accurate and less computationally intensive. A detailed description of the numerical solution of this model, accompanied by source code for a FORTRAN implementation of the solution. This model can be used independently, or in conjunction with a 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 for the Kelp Habitat Improvement Project at the California Institute of Technology



Figure 1.1: *Saccharina latissima* being harvested

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 [3]. This approach is quite different, and functions by dividing kelp plants into groups based on size and age and generating random numbers to determine how the population distribution over these groups changes over time based on measured rates of growth, death, decay, light availability, etc. 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 [10].

In 1996 and 1998 respectively, P. Duarte and J.G. Ferreira used the size-class approach to create a more general model of macroalgae growth, and Yoshimori et. al. created a differential equation model of *Laminaria religiosa* with specific

emphasis on temperature dependence of growth rate [5, 15]. 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, we develop a probabilistic model to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. We begin Chapter 3 with a survey of fundamental radiometric quantities and optical properties of matter. We use the spatial kelp distribution from Chapter 2 to determine optical properties of the combined water-kelp medium. We then present the Radiative Transfer Equation, an integro-partial differential equation which describes the the light field as a function of position and angle. An asymptotic expansion is explored for cases where absorption dominates scattering in the medium, such as in very clear water or high kelp density. In Chapter 4, details are given for the numerical solution of the equations from Chapters 2 and 3. Both the full finite difference solution and the asymptotic approx-

imation are described. Next, in Chapter 5, we discuss the availability of necessary parameters in the literature. For those which are not readily available, we give rough estimates and briefly describe experimental methods for their determination. Then, in Chapter 6, we investigate the necessary grid resolution for adequate accuracy in the full finite difference solution and compare to the asymptotic approximation for a few parameter sets. Further, we determine the effect of varying a few key parameters on the light field predicted by the asymptotic approximation. Afterwards, we use the light model developed here in a full lifecycle simulation of kelp growth and compare the light field and biomass production to those predicted by a simpler 1D exponential decay light model. Finally, we conclude with Chapter 7 by giving a brief summary of the model, discuss 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 vertical rope case, in which the kelp plants extend radially outward from the rope in all directions. The mature *Saccharina latissima* plant consists of a single frond (leaf), a stipe (stem) and a holdfast (root structure). Plants extending from each rope will shade both themselves and their neighbors to varying degrees based on the depth of the kelp, the rope spacing, the angle of incident light on the surface of the water, and the optical properties of the medium.

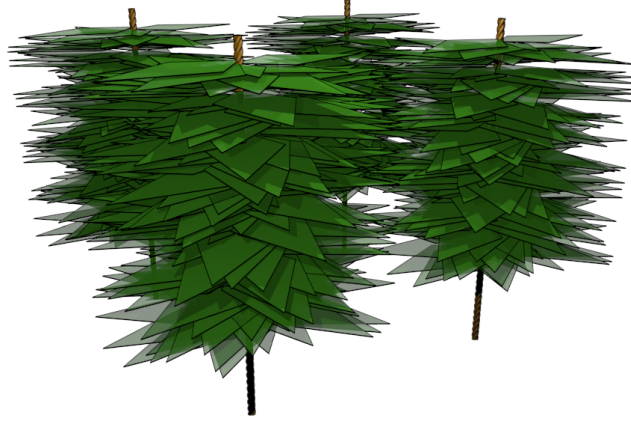


Figure 2.1: Rendering of four nearby vertical kelp ropes

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

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

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

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

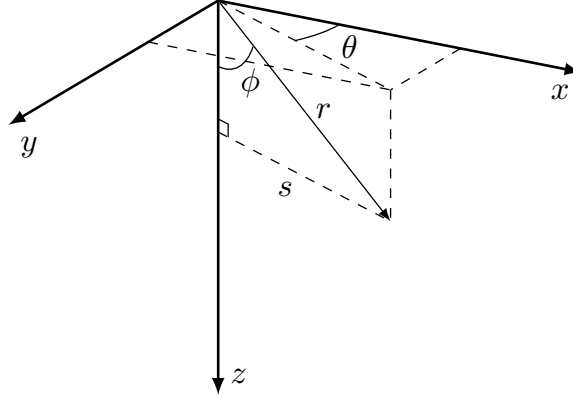


Figure 2.2: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ

2.3 Population Distributions

2.3.1 Frond Shape

We assume the frond is a kite with length l from base to tip, and width w from left to right. 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

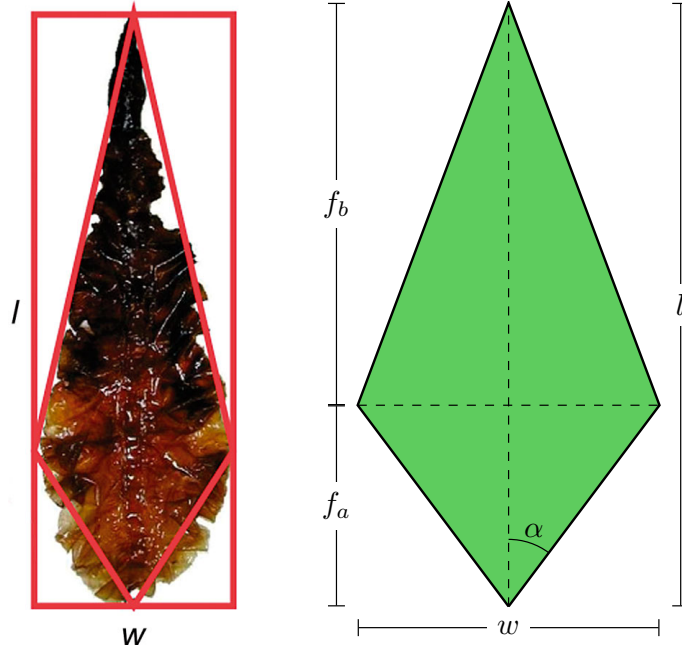


Figure 2.3: Simplified kite-shaped frond

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 important in our analysis. Using the above equations,

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

The area of the frond is given by

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

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

$$l = \sqrt{2A f_r} \tag{2.3}$$

2.3.2 Length and Angle Distributions

We assume that frond lengths are normally distributed with mean μ_l and standard deviation σ_l . That is, the frond length distribution has the probability density function (PDF)

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

We assume the frond angle varies according to the von Mises distribution, which is the periodic analogue of the normal distribution, defined on $[-\pi, \pi]$ rather

than $(-\infty, \infty)$. The von Mises distribution has two parameters, μ and κ , which shift and sharpen its peak respectively, as shown in Figure 2.4. κ can be considered analogous to $1/\sigma$ in the normal distribution. Here, we use $\mu = \theta_w$ and $\kappa = v_w$. That is, in the case of zero current velocity, the frond angles are distributed uniformly, while as current velocity increases, they become increasingly likely to be pointing in the direction of the current. Note that θ_w and v_w vary over depth.

The PDF for the von Mises distribution is

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

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.

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

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 ,

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

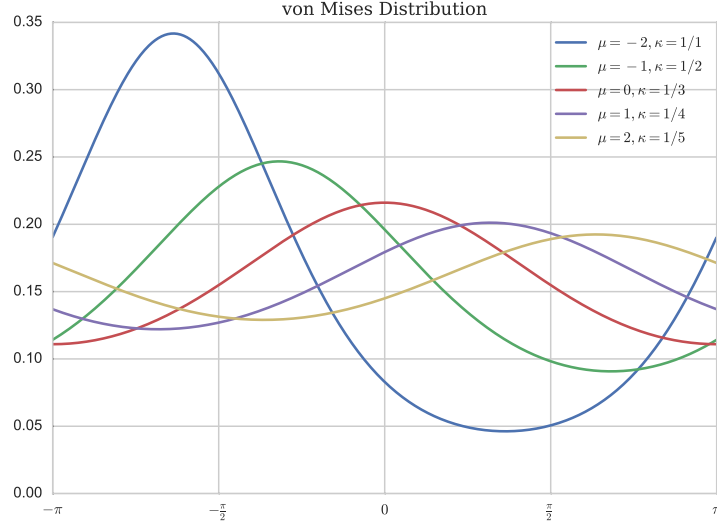


Figure 2.4: von Mises distribution for a variety of parameters

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 following more

general relation.

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

This is currently not taken into consideration in this model.

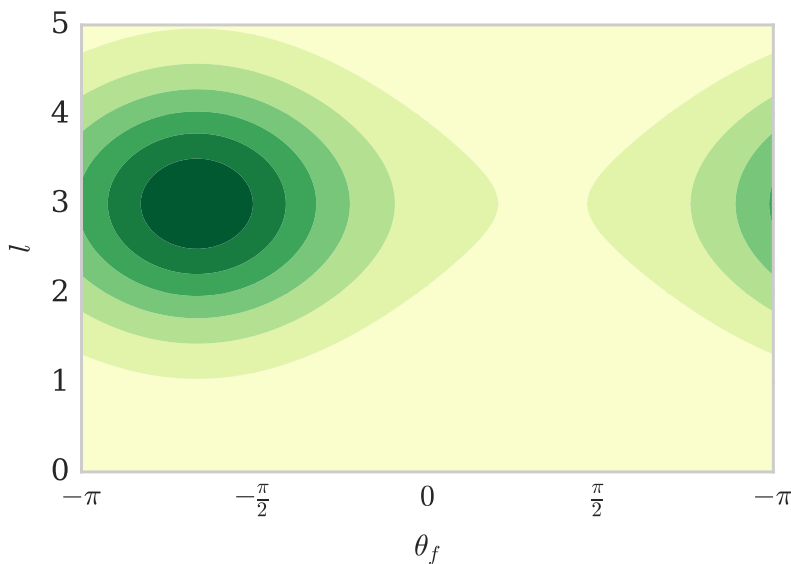


Figure 2.5: 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$

2.4 Spatial Distribution

2.4.1 Rotated Coordinate System

To determine under what conditions a frond will occupy a given point, we begin by describing the shape of the frond in Cartesian and then converting to polar coordinates. Of primary interest are the edges connected to the frond tip. For convenience, we will use a rotated coordinate system (θ', s) such that the line connecting the base to the tip is vertical, with the base at $(0, 0)$. The Cartesian analogue of this coordi-

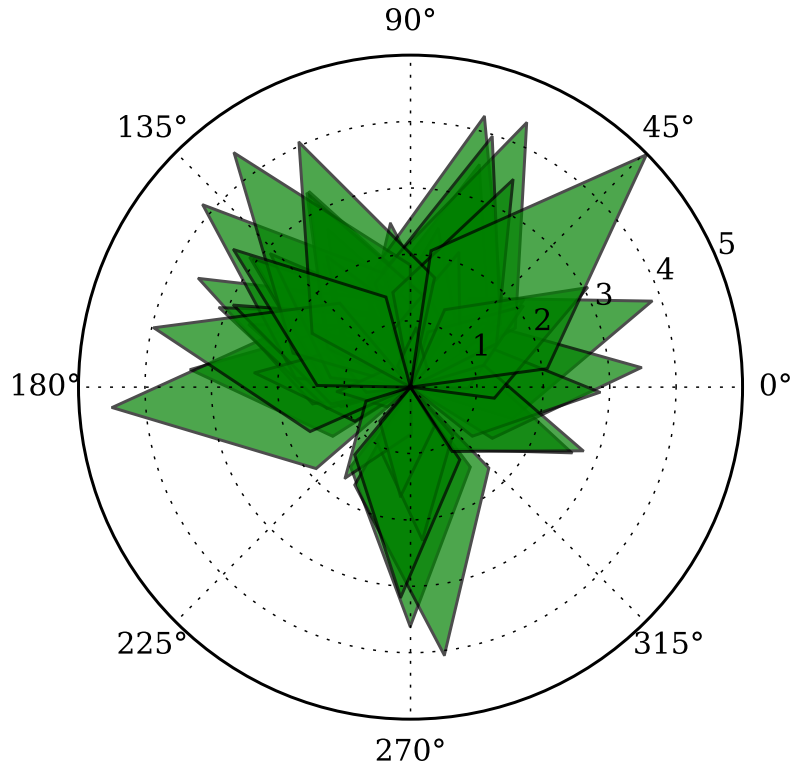


Figure 2.6: A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$.

nate system, (x', y') , has the following properties.

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

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

and

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

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

2.4.2 Functional Description of Frond Edge

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

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

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, we can rewrite the above equation in terms of our frond ratios f_s and f_r .

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

2.4.3 Absolute Coordinates

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

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

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

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

2.4.4 Conditions for Occupancy

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

$$|\theta_f - \theta| < \alpha$$

and

$$s < s_f(\theta)$$

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

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

and

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

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.

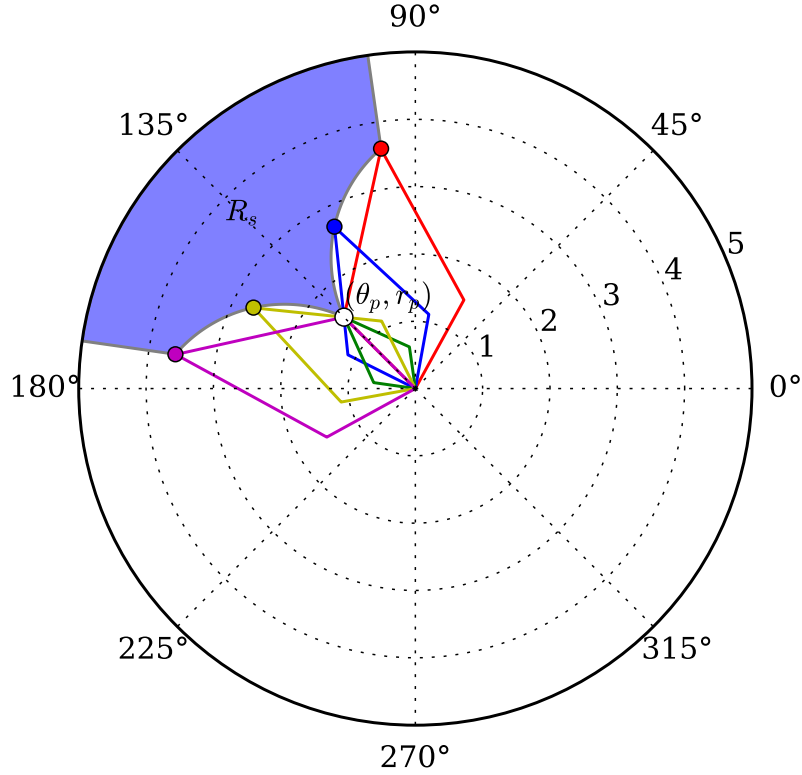


Figure 2.7: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$

2.4.5 Probability of Occupancy

We are interested in the probability that, given a fixed point (θ, s) , values of l and θ_f chosen from the distributions described in Section 2.3.2 will fall in the occupancy

region. This is found by integrating P_{2D} over the occupancy region for (θ, s) , as depicted in figure 2.8.

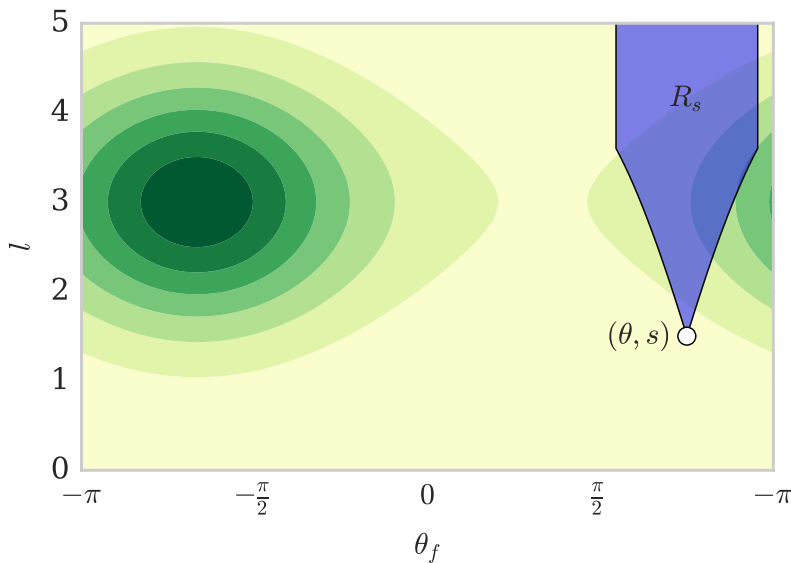


Figure 2.8: Contour plot of $P_{2D}(\theta_f, l)$ overlaid with the region in the $\theta_f - l$ plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$

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

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

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

the proportion of the grid cell occupied by kelp to be

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

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

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

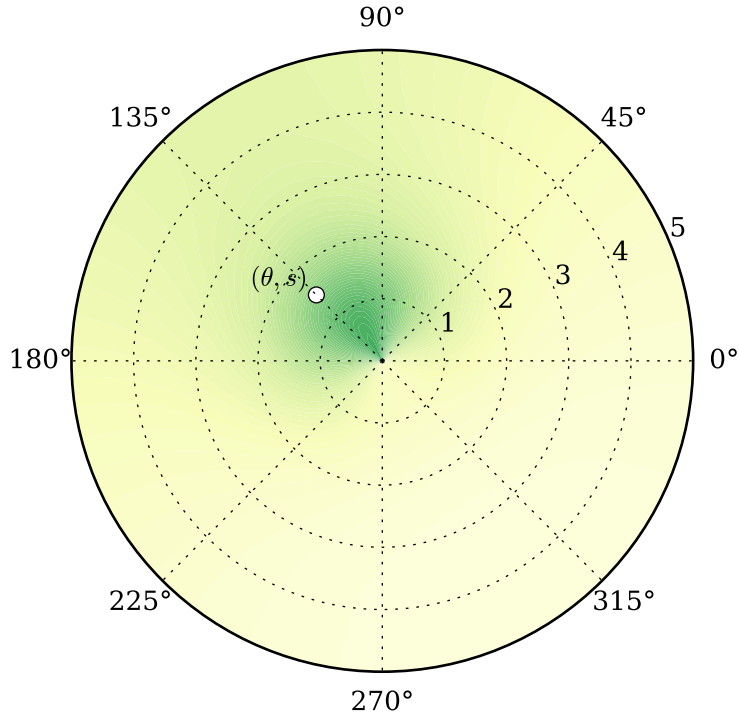


Figure 2.9: Contour plot of the probability of occupying sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$

CHAPTER III

LIGHT MODEL

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

3.1 Optical Definitions

3.1.1 Radiometric Quantities

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

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

Once the radiance L is calculated everywhere, the irradiance is

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

Integrating $I(\mathbf{x})$, 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 4.4.1.

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

$$1 \text{ W/m}^2 = 4.2 \text{ } \mu\text{mol photons/s}.$$

3.1.2 Inherent Optical Properties

We must now define a few inherent optical properties (IOPs) which depend only on the medium of propagation. These phenomena are governed by three inherent optical properties (IOPs) of the medium. The absorption coefficient $a(\mathbf{x})$ (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) $\beta(\Delta) : [-1, 1] \rightarrow \mathbb{R}^+$ (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 \mathbf{x} and direction $\boldsymbol{\omega}$ such that $\boldsymbol{\omega} \cdot \hat{\mathbf{z}} \neq 0$.

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

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

Then,

$$\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s) = \frac{1}{\tilde{s}}(s\mathbf{x} + (\tilde{s} - s)\mathbf{x}_0(\mathbf{x}, \boldsymbol{\omega})) \quad (3.1)$$

where

$$\mathbf{x}_0(\mathbf{x}, \boldsymbol{\omega}) = \mathbf{x} - \tilde{s}\boldsymbol{\omega}$$

is the origin of the ray, and

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

is the path length from $\mathbf{x}_0(\mathbf{x}, \boldsymbol{\omega})$ to \mathbf{x} .

3.2.2 Colloquial Description

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

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

$$\frac{dL}{ds}(\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}) = -(a(\mathbf{x}) + b)L(\mathbf{x}, \boldsymbol{\omega}) + b \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L(\mathbf{x}, \boldsymbol{\omega}') d\omega', \quad (3.2)$$

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

Now, we have

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

Then, the general form of the Radiative Transfer Equation is

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

or, equivalently,

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

3.2.4 Boundary Conditions

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

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

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

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.

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

$$L(\mathbf{x}_b, \boldsymbol{\omega}) = 0 \text{ if } \boldsymbol{\omega} \cdot \hat{\mathbf{z}} < 0$$

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

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

Then, substituting the above into the RTE,

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

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

$$\begin{aligned}
\boldsymbol{\omega} \cdot \nabla L_0(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_0(\mathbf{x}) &= 0 \\
\boldsymbol{\omega} \cdot \nabla L_1(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_1(\mathbf{x}) &= \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_0(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_0(\mathbf{x}, \boldsymbol{\omega}) \\
\boldsymbol{\omega} \cdot \nabla L_2(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_2(\mathbf{x}) &= \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_1(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_1(\mathbf{x}, \boldsymbol{\omega}) \\
&\vdots
\end{aligned} \tag{3.3}$$

In general, for $n \geq 1$, we have

$$\boldsymbol{\omega} \cdot \nabla L_n(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_n(\mathbf{x}) = \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_{n-1}(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_{n-1}(\mathbf{x}, \boldsymbol{\omega}) \tag{3.4}$$

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

Then,

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

which becomes

$$L_0(\mathbf{x}, \boldsymbol{\omega}) = \begin{cases} f(\boldsymbol{\omega}), & \hat{\mathbf{z}} \cdot \boldsymbol{\omega} > 0, \\ 0, & \text{otherwise,} \end{cases} \quad (3.5)$$

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

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

$$\vdots$$

In general, for $n \geq 1$,

$$L_n(\mathbf{x}, \boldsymbol{\omega}) = 0. \quad (3.6)$$

3.3.2 Analytical Solution

For all $\mathbf{x}, \boldsymbol{\omega}$, we consider the path $l(\mathbf{x}, \boldsymbol{\omega}, s)$ from (3.1). We extract the absorption coefficient along the path,

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

Then, the first equation from the asymptotic expansion, (3.3) and its associated boundary condition, (3.5), can be rewritten as

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

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,

$$u_0(s) = f(\boldsymbol{\omega}) \exp \left(- \int_0^s \tilde{a}(s) ds \right). \quad (3.7)$$

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

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

Now, 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 (3.4) and its boundary condition, (3.6), to be written as the first order, linear ordinary differential equation along the ray path,

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

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

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

Then, 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

$$u_n(s) = \int_0^s g_n(s') \exp \left(- \int_{s'}^s \tilde{a}(s'') ds'' \right) ds'. \quad (3.8)$$

As before, the conversion back to spatial coordinates is

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

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

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

4.1.1 Frond Length Distribution

Rather than model each kelp frond, a subset of the population, called super-individuals, are modelled explicitly, and are considered to represent many identical individuals, as in [12]. 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:

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

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

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

4.2 Discrete Grid

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

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

The following indices are assigned to each dimension:

$$x \rightarrow i$$

$$y \rightarrow j$$

$$z \rightarrow k$$

$$\theta \rightarrow l$$

$$\phi \rightarrow m$$

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

4.2.1 Spatial Grid

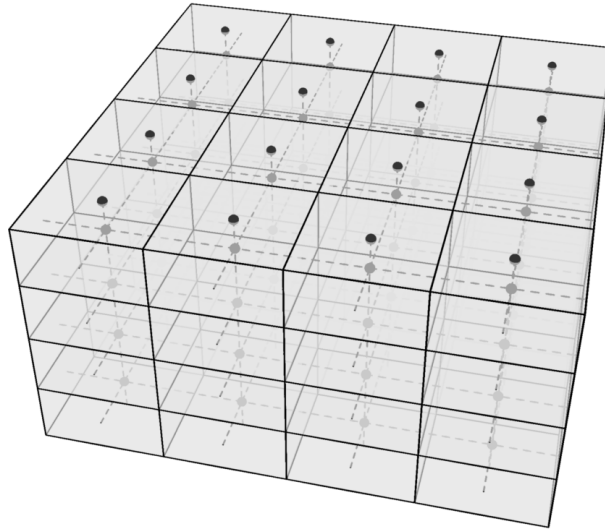


Figure 4.1: Spatial grid

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

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

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

Denote the edges as

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

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

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

and the cell centers as

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

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

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

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

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

4.2.2 Angular Grid

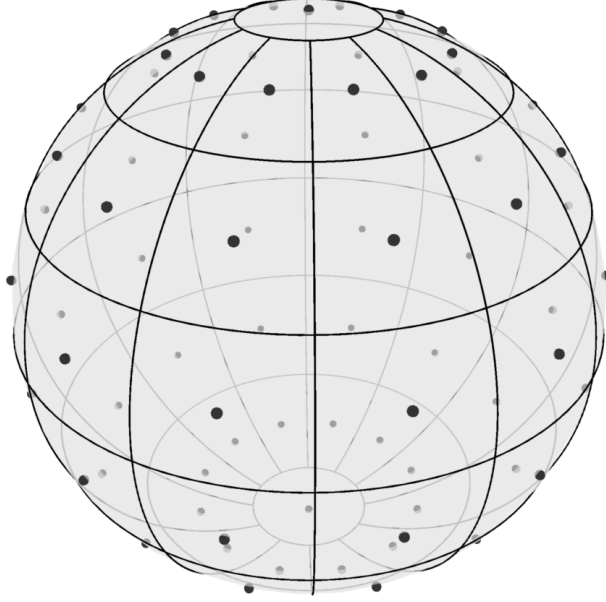


Figure 4.2: Angular grid at each point in space

Now, we define the azimuthal angle such that

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

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

$$\theta_l^e = (l - 1/2)d\theta, \quad l = 1, \dots, n_\theta \quad (4.1)$$

$$\phi_m^e = (m - 1/2)d\phi, \quad m = 1, \dots, n_\phi - 1. \quad (4.2)$$

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

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

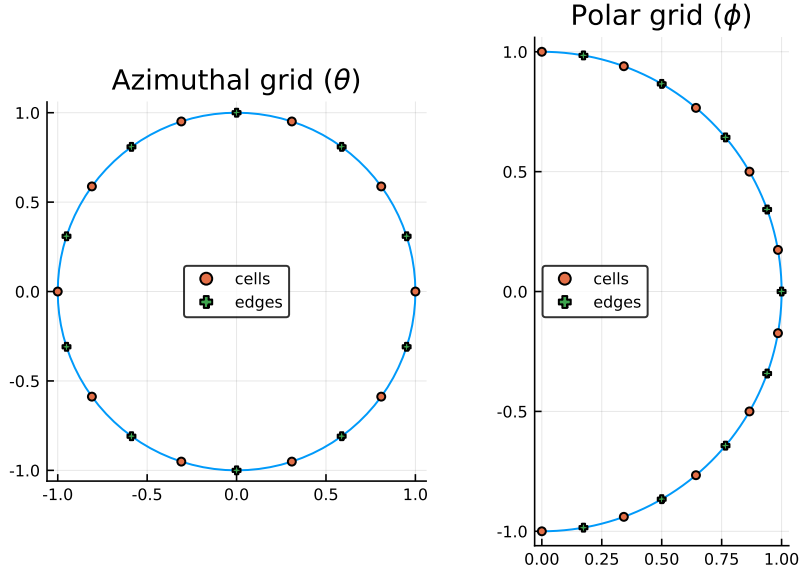


Figure 4.3: Angular grid

south poles respectively. 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 ω_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

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

And for all other angular grid cells,

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

4.2.3 Angular Quadrature

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

Define the *angular characteristic function*

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

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

4.2.4 Scattering Integral

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

Consider two angular grid cells, Ω and Ω' . The average probability density of scattering from $\boldsymbol{\omega} \in \Omega$ 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 is $\beta_{pp'} |\Omega'| L_{ijkp'}$.

4.3 Finite Difference

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

4.3.1 Discretization

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

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

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

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

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

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

$$f(x+\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} \varepsilon^n$$

Truncating after the first few terms, we have

$$f(x+\varepsilon) = f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \mathcal{O}(\varepsilon^3) \quad (4.3)$$

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

$$f(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \quad (4.4)$$

Rearranging (4.3) produces

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

Combining (4.4) with (4.5) 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 $\varepsilon \rightarrow -\varepsilon$, we have

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

4.3.2 Difference Equation

In general, we have

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

Then,

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

Interior:

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

Surface downwelling (BC):

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

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

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

Likewise for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

4.3.3 Structure of Linear System

Describe layout of matrix.

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

Table 4.1: Breakdown of nonzero matrix elements by derivative case

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

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

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

4.3.4 GMRES

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

4.4 Numerical Asymptotics

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

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

Let

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

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

and

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

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

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

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

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

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

Let

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

Then,

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

Let

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

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

Then,

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

4.4.1 Perceived Irradiance

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

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

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

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

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

CHAPTER V

PARAMETER VALUES

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

5.1 Parameters from Literature

* More to come

5.2 Frond Distribution Parameters

5.2.1 Rotation

5.2.2 Lift

Parameter Name		Symbol	Value(s)	Citation	Notes
Kelp	Absorp- tance	A_k	0.8	[4]	Actually for <i>Macrocystis</i> <i>Pyrifera</i>
Water	absorp- tion coefficient	a_w	?	?	?
Scattering	coeffi- cient	b	0.366	[13]	Table 2, $b_{\lambda 0}$, mean
VSF		β	tabulated	[11, 13],	Currently using Petzold
Fron	thickness	t	0.4 mm	Ole Jacob	Carina? ***
Water	absorp- tion coefficient	a_w	0.03 1 1/m	[6]	Fig. 6, dense cluster. Sam- nanger Fjord, Western Norway.
Water	scattering coefficient	a_w	0.5 1 1/m	[6]	Fig. 7, dense cluster. Sam- nanger Fjord, Western Norway.
Surface	solar ir- radiance	I_0	50 W m ⁻²	[2]	Irradiance for maximal pho- tosynthesis, converted from photons

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

Table 5.2: Petzold IOP summary [11]. I'll pull a few cases from here and point out when the asymptotic approximation will work.

CHAPTER VI

MODEL ANALYSIS

6.1 Grid Study

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

6.2 Asymptotic Convergence

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

6.3 Sensitivity Analysis

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

- absorption coefficient
- scattering coefficient

- VSF
- frond bending coefficient

6.4 Kelp Cultivation Simulation

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

- irradiance over time for several depths
- computation time
- harvestable biomass

CHAPTER VII

CONCLUSION

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

Future work:

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

BIBLIOGRAPHY

- [1] N. Anderson. A mathematical model for the growth of giant kelp. *Simulation*, 22(4):97–105, 1974.
- [2] O. J. Broch and D. Slagstad. Modelling seasonal growth and composition of the kelp *Saccharina latissima*. *Journal of Applied Phycology*, 24(4):759–776, Aug. 2012.
- [3] M. A. Burgman and V. A. Gerard. A stage-structured, stochastic population model for the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 105(1):15–23, 1990.
- [4] M. F. Colombo-Pallotta, E. Garca-Mendoza, and L. B. Ladah. Photosynthetic Performance, Light Absorption, and Pigment Composition of *Macrocystis Pyrifera* (laminariales, Phaeophyceae) Blades from Different Depths1. *Journal of Phycology*, 42(6):1225–1234, Dec. 2006.
- [5] P. Duarte and J. G. Ferreira. A model for the simulation of macroalgal population dynamics and productivity. *Ecological modelling*, 98(2-3):199–214, 1997.
- [6] B. Hamre, . Frette, S. R. Erga, J. J. Stamnes, and K. Stamnes. Parameterization and analysis of the optical absorption and scattering coefficients in a western

- Norwegian fjord: a case II water study. *Applied Optics*, 42(6):883, Feb. 2003.
- [7] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 95(4):611–624, 1987.
 - [8] C. Mobley. *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] 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.
 - [11] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.
 - [12] 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.
 - [13] 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.

- [14] 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.
- [15] 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):136–146, 1998.

APPENDICES

APPENDIX A

RAY TRACING ALGORITHM

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

Consider a grid center $\mathbf{p}_1 = (p_{1x}, p_{1y}, p_{1z})$ and a corresponding path $\mathbf{l}(\mathbf{x}_1, \boldsymbol{\omega}, s)$.

To find the location of discontinuities in the integrand, we first calculate the distance from its origin, $\mathbf{p}_0 = \mathbf{x}_0(\mathbf{p}_1, \boldsymbol{\omega}) = (p_{0x}, p_{0y}, p_{0z})$ to grid edges in each dimension separately.

Given

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

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

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

we have

$$s_i^x = \tilde{s} \frac{x_i - p_{0x}}{p_{1x} - p_{0x}} \quad (\text{A.4})$$

$$s_i^y = \tilde{s} \frac{y_i - p_{0y}}{p_{1y} - p_{0y}} \quad (\text{A.5})$$

$$s_i^z = \tilde{s} \frac{z_i - p_{0z}}{p_{1z} - p_{0z}} \quad (\text{A.6})$$

$$(\text{A.7})$$

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

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

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

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

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

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

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

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

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

Let (i_0, j_0, k_0) be the indices of the grid cell containing \mathbf{p}_0 .

That is,

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

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

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

Then,

$$i_0^e = i_0 + \sigma_x \quad (\text{A.17})$$

$$j_0^e = j_0 + \sigma_y \quad (\text{A.18})$$

$$k_0^e = k_0 + \sigma_z \quad (\text{A.19})$$

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

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

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

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

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

* We also track s , the path length.

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

$$d = \operatorname{argmin}_{x,y,z} \{s_i^x - s, s_j^y - s, s_k^z\} \quad (\text{A.23})$$

* This doesn't quite make sense yet.

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

and

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

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

APPENDIX B

FORTRAN CODE

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

<https://github.com/OliverEvans96/kelp>

<https://gitlab.com/OliverEvans96/kelp>

```
utils.f90
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
35     ! directory
36     ! Will either return '.', '..', or '..../..'
37     character(len=maxlength) getbasedir
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
51         ! directory
52         !write(*,*) 'Checking "', tmp_path(1:
53         ! pathlengths(ii)), '"'
54         inquire(file=tmp_path(1:pathlengths(ii))
55         // '/Makefile', exist=found)
56         ! If so, stop. Otherwise, keep looking.
57         if(found) then
58             getbasedir = tmp_path(1:pathlengths(
59             ii))
60             exit
61         end if
62     end do
63
64     ! If it hasn't been found, then this script
65     ! was probably called
66     ! from outside of the repository.
67     if(.not. found) then
68         write(*,*) 'BASE DIR NOT FOUND.'
69     end if
70
71 end function
72
73 ! 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
157             sampled
158         ! yy - corresponding y values to interpolate
159         double precision, dimension (nn), intent(in)
160             :: xx,yy
161         ! nn - length of data
162         integer, intent(in) :: nn
163
164         ! OUTPUT:
165         ! interp - interpolated y value
166         double precision interp

```

```

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      end if
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
331 | , z00, z01, z10, z11)
332 | end function bilinear_array
333 |
334 | ! ilinear interpolation of a function of two
335 | ! variables
336 | ! over a rectangle of points.
337 | ! Weight each point by the area of the sub-
338 | ! rectangle involving
339 | ! the point (x,y) and the point diagonally
340 | ! across the rectangle
341 | function bilinear(x, y, x0, y0, x1, y1, z00, z01
342 | , z10, z11)
343 | implicit none
344 | double precision x, y
345 | double precision x0, y0, x1, y1, z00, z01, z10
346 | , z11
347 | double precision a, b, c, d
348 | double precision bilinear
349 |
350 | a = (x-x0)*(y-y0)
351 | 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
451 ! endpoint rule
452 function normalize_dx(arr,dx,nn)
453     implicit none
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
464     ! (left endpoint rule)
465     double precision normalize_dx
466
467     ! BODY:
468
469     ! Calculate integral
470     normalize_dx = lep_rule(arr,dx,nn)
471
472     ! Normalize array
473     arr = arr / normalize_dx
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
479 ! endpoint is not included
480 ! (Very small for VSF, so not a big deal there)
481 ! Modifies yy in place
482 function normalize_uneven(xx, yy, nn) result(
483     norm)
484     implicit none
485     ! INPUTS:
486     ! xx, yy - array values of data to normalize
487     double precision, dimension(nn) :: xx, yy
488     ! nn - length of arr
489     integer, intent(in) :: nn

```

```

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      ! mm - 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
19   +00
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
31     :: theta, phi, theta_edge, phi_edge
32   double precision, dimension(:), allocatable
33     :: theta_p, phi_p, theta_edge_p,
34       phi_edge_p
35   double precision, dimension(:), allocatable
36     :: cos_theta, sin_theta, cos_phi, sin_phi
37   double precision, dimension(:), allocatable
38     :: cos_theta_edge, sin_theta_edge,
39       cos_phi_edge, sin_phi_edge
40   double precision, dimension(:), allocatable
41     :: cos_theta_p, sin_theta_p, cos_phi_p,
42       sin_phi_p
43   double precision, dimension(:), allocatable
44     :: cos_theta_edge_p, sin_theta_edge_p,
45       cos_phi_edge_p, sin_phi_edge_p
46   double precision, dimension(:), allocatable
47     :: area_p
48   contains
49     procedure :: set_num => angle_set_num
50     procedure :: phat, lhat, mhat
51     procedure :: init => angle_init ! Call after
52       set_num
53     procedure :: integrate_points =>
54       angle_integrate_points
55     procedure :: integrate_func =>
56       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 => angleld_set_num
53     procedure :: deinit => angleld_deinit
54     procedure :: integrate_points =>
        angleld_integrate_points
55     procedure :: integrate_func =>
        angleld_integrate_func
56     procedure :: assign_linspace =>
        angleld_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)
136             ) + 1
137     end if
138 end function mhat
139
140 function phat(angles, l, m) result(p)
141     class(angle2d) :: angles
142     integer l, m, p
143
144     if(m .eq. 1) then
145         p = 1
146     else if(m .eq. angles%nphi) then
147         p = angles%nomega
148     else
149         p = (m-2)*angles%ntheta + 1 + 1
150     end if
151 end function phat
152
153 subroutine angle_init(angles)
154     class(angle2d) :: angles
155     integer l, m, p
156     double precision area
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         ))
177     allocate(angles%sin_theta_edge(angles%ntheta
178         ))
179     allocate(angles%cos_phi_edge(angles%nphi-1))
180     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/dbble(angles%ntheta)
187 | angles%dphi = pi/dbble(angles%nphi-1)
188 |
189 | ! Create grids
190 | do l=1, angles%ntheta
191 |     angles%theta(l) = dbble(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) = dbble(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) = dbble(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) = dbble(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(1, m)
216
217         angles%theta_p(p) = angles%theta(1)
218         angles%phi_p(p) = angles%phi(m)
219         angles%theta_edge_p(p) = angles%
220             theta_edge(1)
221         angles%phi_edge_p(p) = angles%phi_edge
222             (m)
223
224         angles%cos_theta_p(p) = cos(angles%
225             theta_p(p))
226         angles%sin_theta_p(p) = sin(angles%
227             theta_p(p))
228         angles%cos_phi_p(p) = cos(angles%phi_p
229             (p))
230         angles%sin_phi_p(p) = sin(angles%phi_p
231             (p))
232
233         angles%cos_theta_edge_p(p) = cos(
234             angles%theta_edge_p(p))
235         angles%sin_theta_edge_p(p) = sin(
236             angles%theta_edge_p(p))
237         angles%cos_phi_edge_p(p) = cos(angles%
238             phi_edge_p(p))
239         angles%sin_phi_edge_p(p) = sin(angles%
240             phi_edge_p(p))
241
242         angles%area_p(p) = area
243     end do
244 end do
245
246 ! Poles
247 l=1
248 area = 2.d0*pi*(1.d0-cos(angles%dphi/2.d0))
249
250 ! North Pole
251 p = 1
252 m=1
253 angles%theta_p(p) = angles%theta(1)
254 angles%theta_edge_p(p) = angles%theta_edge(1
255 )
256 angles%phi_p(p) = angles%phi(m)
257 ! phi_edge_p only defined up to nphi-1.
258 angles%phi_edge_p(p) = angles%phi_edge(m)
259 angles%cos_theta_p(p) = cos(angles%theta_p(p
260 ))
261 angles%sin_theta_p(p) = sin(angles%theta_p(p
262 ))
263 angles%cos_phi_p(p) = cos(angles%phi_p(p))
264 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,
340                                maxval)
341      class(angle_dim) :: angle
342      double precision minval, maxval
343      angle%minval = minval
344      angle%maxval = maxval
345      end subroutine angle_set_bounds
346
347      subroutine angle1d_set_num(angle, num)
348      class(angle_dim) :: angle
349      integer num
350      angle%num = num
351      end subroutine angle1d_set_num
352
353      subroutine angle1d_assign_linspace(angle)
354      class(angle_dim) :: angle
355      double precision spacing
356      integer i
357
358      spacing = (angle%maxval - angle%minval) /
359               dble(angle%num)
360      do i=1, angle%num
361         angle%vals(i) = (i-1) * spacing
362      end do
363      end subroutine angle1d_assign_linspace
364
365      ! To calculate \int_{xmin}^{xmax} f(x) dx :
366      ! int = prefactor * sum(weights * f(roots))
367      subroutine assign_legendre(angle)
368      class(angle_dim) :: angle
369      double precision root, weight, theta
370      integer i
371      ! glpair produces both x and theta, where x=
372      !   cos(theta). We'll throw out theta.
373
374      allocate(angle%vals(angle%num))
375      allocate(angle%weights(angle%num))
376      allocate(angle%sin(angle%num))
377      allocate(angle%cos(angle%num))
378
379      ! Prefactor for integration
380      ! From change of variables
381      angle%prefactor = (angle%maxval - angle%
382                        minval) / 2.d0
383
384      do i = 1, angle%num
385         call glpair(angle%num, i, theta, weight,
386                    root)

```



```

382         call affine_transform(root, -1.d0, 1.d0,
383             angle%minval, angle%maxval)
384         angle%vals(i) = root
385         angle%weights(i) = weight
386         angle%sin(i) = sin(root)
387         angle%cos(i) = cos(root)
388     end do
389 end subroutine assign_legendre
390
391 ! Integrate callable function over angle via
392   Gauss-Legendre quadrature
393
394 function angle1d_integrate_func(angle,
395     func_callable) result(integral)
396     class(angle_dim) :: angle
397     double precision, external :: func_callable
398     double precision, dimension(:), allocatable
399         :: func_vals
400     double precision integral
401     integer i
402
403     allocate(func_vals(angle%num))
404
405     do i=1, angle%num
406         func_vals(i) = func_callable(angle%vals(i)
407             )
408     end do
409
410     integral = angle%integrate_points(func_vals)
411
412     deallocate(func_vals)
413 end function angle1d_integrate_func
414
415 ! Integrate function given function values
416   sampled at legendre theta values
417
418 function angle1d_integrate_points(angle,
419     func_vals) result(integral)
420     class(angle_dim) :: angle
421     double precision, dimension(angle%num) ::
422         func_vals
423     double precision integral
424
425     integral = angle%prefactor * sum(angle%
426         weights * func_vals)
427 end function angle1d_integrate_points
428
429 subroutine angle1d_deinit(angle)
430     class(angle_dim) :: angle
431     deallocate(angle%vals)
432     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,
473     spacing)
474     class(space_dim) :: space
475     double precision, dimension(space%num) ::
476         spacing
477     space%spacing = spacing
478 end subroutine space_set_spacing_array
479
480 subroutine assign_linspace(space)
481     class(space_dim) :: space
482     double precision spacing
483     integer i
484
485     allocate(space%vals(space%num))
486     allocate(space%edges(space%num))
487     allocate(space%spacing(space%num))
488
489     spacing = spacing_from_num(space%minval,
490         space%maxval, space%num)
491     call space%set_uniform_spacing(spacing)
492
493     do i=1, space%num
494         space%edges(i) = space%minval + dble(i-1)
495             * space%spacing(i)
496         space%vals(i) = space%minval + dble(i-0.5
497             d0) * space%spacing(i)
498     end do
499 end subroutine assign_linspace
500
501 subroutine set_uniform_spacing_from_num(space)
502     ! Create evenly spaced grid (linspace)
503     class(space_dim) :: space
504     double precision spacing
505
506     spacing = spacing_from_num(space%minval,
507         space%maxval, space%num)
508     call space%set_uniform_spacing(spacing)
509 end subroutine set_uniform_spacing_from_num
510
511 ! subroutine set_num_from_spacing(space)
512 !     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
562     ! multiplied
563     ! in order to calculate distance traveled in
564     ! the
565     ! (x, y) direction along a ray in the (theta
566     ! , phi)
567     ! direction
568     class(space_angle_grid) :: grid
569     integer p, nomega
570     double precision theta, phi
571
572     nomega = grid%angles%nomega
573
574     allocate(grid%x_factor(nomega))
575     allocate(grid%y_factor(nomega))
576
577     do p=1, nomega
578         theta = grid%angles%theta_p(p)
579         phi = grid%angles%phi_p(p)
580         grid%x_factor(p) = tan(phi) * cos(theta)
581         grid%y_factor(p) = tan(phi) * sin(theta)
582     end do
583
584 end subroutine sag_calculate_factors
585
586 subroutine sag_set_uniform_spacing_from_num(
587     grid)
588     class(space_angle_grid) :: grid
589     call grid%x%set_uniform_spacing_from_num()
590     call grid%y%set_uniform_spacing_from_num()
591     call grid%z%set_uniform_spacing_from_num()
592 end subroutine
593     sag_set_uniform_spacing_from_num
594
595 ! subroutine sag_set_num_from_spacing(grid)
596 !     class(space_angle_grid) :: grid
597 !     call grid%x%set_num_from_spacing()
598 !     call grid%y%set_num_from_spacing()
599 !     call grid%z%set_num_from_spacing()
600
601 ! end subroutine sag_set_num_from_spacing
602
603 subroutine sag_deinit(grid)
604     class(space_angle_grid) :: grid

```

```

600 |     call grid%x%deinit()
601 |     call grid%y%deinit()
602 |     call grid%z%deinit()
603 |     call grid%angles%deinit()
604 |
605 |     deallocate(grid%x_factor)
606 |     deallocate(grid%y_factor)
607 | end subroutine sag_deinit
608 |
609 | ! Affine shift on x from [xmin, xmax] to [ymin
610 |   , ymax]
611 | subroutine affine_transform(x, xmin, xmax,
612 |   ymin, ymax)
613 |   double precision x, xmin, xmax, ymin, ymax
614 |   x = ymin + (ymax-ymin)/(xmax-xmin) * (x-xmin
615 |     )
616 | end subroutine affine_transform
617 |
618 | function num_from_spacing(xmin, xmax, dx)
619 |   result(n)
620 |   double precision xmin, xmax, dx
621 |   integer n
622 |   n = floor( (xmax - xmin) / dx )
623 | end function num_from_spacing
624 |
625 | function spacing_from_num(xmin, xmax, nx)
626 |   result(dx)
627 |   double precision xmin, xmax, dx
628 |   integer nx
629 |   dx = (xmax - xmin) / dble(nx)
630 | end function spacing_from_num
631 | end module sag

```

kelp3d.f90

```

1 | ! Kelp 3D
2 | ! Oliver Evans
3 | ! 8/31/2017
4 |
5 | ! Given superindividual/water current data at
6 |   each depth, generate kelp distribution at
7 |   each point in 3D space
8 |
9 | module kelp3d
10 |
11 | use kelp_context
12 |
13 | implicit none
14 | 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,
62 |                 frond, grid, depth,
63 |                 quadrature_degree)
64 |             !p_kelp(i, j, k) = prob_kelp(point,
65 |                 frond, depth, quadrature_degree)
66 |         end do
67 |     end do
68 | end subroutine calculate_kelp_on_grid
69 |
70 | subroutine shading_region_limits(theta_low_lim,
71 |     theta_high_lim, point, frond)
72 |     type(point3d), intent(in) :: point
73 |     type(frond_shape), intent(in) :: frond
74 |     double precision, intent(out) :: theta_low_lim
75 |     , theta_high_lim
76 |
77 |     theta_low_lim = point%theta - frond%alpha
78 |     theta_high_lim = point%theta + frond%alpha
79 | end subroutine shading_region_limits
80 |
81 | function prob_kelp(point, frond, depth,
82 |     quadrature_degree)
83 | ! P_s(theta_p, r_p) - This is the proportion of
84 |   the population of this depth layer which can
85 |   be found in this Cartesian grid cell.
86 |     type(point3d), intent(in) :: point
87 |     type(frond_shape), intent(in) :: frond
88 |     type(depth_state), intent(in) :: depth
89 |     integer, intent(in) :: quadrature_degree
90 |     double precision prob_kelp
91 |     double precision theta_low_lim, theta_high_lim
92 |
93 |     call shading_region_limits(theta_low_lim,
94 |         theta_high_lim, point, frond)
95 |     prob_kelp = integrate_ps(theta_low_lim,
96 |         theta_high_lim, quadrature_degree, point,
97 |         frond, depth)
98 | end function prob_kelp
99 |
100 | function kelp_proportion(point, frond, grid,
101 |     depth, quadrature_degree)
102 | ! This is the proportion of the volume of the
103 |   Cartesian grid cell occupied by kelp
104 |     type(point3d), intent(in) :: point
105 |     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_fronnd_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_fronnd_edge
185 | !
186 | !     relative_fronnd_edge = L / (sin(theta_prime)
    | !     + angular_sign(theta_prime * alpha(fs, fr) *
    | !     cos(theta_prime)))
187 | ! end function relative_fronnd_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_prime - 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 ent, i, j, k, p

```

```

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

```

```

63     procedure z_bd2
64     procedure z_surface_bc
65     procedure z_bottom_bc
66
67 end type rte_mat
68
69 interface
70     ! Define interface for external procedure
71     ! https://stackoverflow.com/questions/8549415/how-to-declare-the-interface-section-for-a-procedure-argument-which-in-turn-ref
72     subroutine solver_interface(n_total, nonzero,
73         row, col, data, &
74         sol, rhs, maxiter_outer, maxiter_inner,
75         &
76         tol_abs, tol_rel)
77         integer :: n_total, nonzero
78         integer, dimension(nonzero) :: row, col
79         double precision, dimension(nonzero) ::
80             data
81         double precision, dimension(nonzero) :: sol
82         double precision, dimension(n_total) :: rhs
83         integer :: maxiter_outer, maxiter_inner
84         double precision :: tol_abs, tol_rel
85     end subroutine solver_interface
86 end interface
87
88 contains
89
90 subroutine mat_init(mat, grid, iops)
91     class(rte_mat) mat
92     type(space_angle_grid) grid
93     type(optical_properties) iops
94     integer nnz, n_total
95
96     mat%grid = grid
97     mat%iops = iops
98
99     call mat%calculate_size()
100
101     n_total = mat%n_total
102     nnz = mat%nonzero
103     allocate(mat%surface_vals(grid%angles%nomega
104         ))
105     allocate(mat%row(nnz))
106     allocate(mat%col(nnz))
107     allocate(mat%data(nnz))
108     allocate(mat%rhs(n_total))
109     allocate(mat%sol(n_total))

```

```

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

```

```

151 | ! end subroutine mat_to_hdf
152 |
153 | subroutine mat_set_bc(mat, bc)
154 |     class(rte_mat) mat
155 |     class(boundary_condition) bc
156 |     integer p
157 |
158 |     do p=1, mat%grid%angles%nomega/2
159 |         mat%surface_vals(p) = bc%bc_grid(p)
160 |     end do
161 | end subroutine mat_set_bc
162 |
163 | subroutine mat_solve(mat)
164 |     class(rte_mat) mat
165 |     type(solver_params) params
166 |
167 |     params = mat%params
168 |
169 |     write(*,*) 'mat%n_total =', mat%n_total
170 |     write(*,*) 'mat%nonzero =', mat%nonzero
171 |     write(*,*) 'size(mat%row) =', size(mat%row)
172 |     write(*,*) 'size(mat%col) =', size(mat%col)
173 |     write(*,*) 'size(mat%data) =', size(mat%data
174 |         )
175 |     write(*,*) 'size(mat%sol) =', size(mat%sol)
176 |     write(*,*) 'size(mat%rhs) =', size(mat%rhs)
177 |     write(*,*) 'params%maxiter_outer =', params%
178 |         maxiter_outer
179 |     write(*,*) 'params%maxiter_inner =', params%
180 |         maxiter_inner
181 |     write(*,*) 'params%tol_rel =', params%
182 |         tol_rel
183 |     write(*,*) 'params%tol_abs =', params%
184 |         tol_abs
185 |     open(unit=1, file='row.txt')
186 |     open(unit=2, file='col.txt')
187 |     open(unit=3, file='data.txt')
188 |     open(unit=4, file='rhs.txt')
189 |     open(unit=5, file='sol.txt')
190 |     write(1,*) mat%row
191 |     write(2,*) mat%col
192 |     write(3,*) mat%data
193 |     write(4,*) mat%rhs
194 |
195 |     close(1)
196 |     close(2)
197 |     close(3)
198 |     close(4)
199 |
200 |     call mat%solver(mat%n_total, mat%nonzero, &

```

```

196         mat%row, mat%col, mat%data, mat%sol,
197         mat%rhs, &
198         params%maxiter_outer, params%
199         maxiter_inner, &
200         params%tol_abs, params%tol_rel)
201     write(5,*) mat%sol
202     close(5)
203 end subroutine mat_solve
204
205 subroutine mat_set_solver_params(mat,
206     maxiter_outer, &
207     maxiter_inner, tol_abs, tol_rel)
208     class(rte_mat) mat
209     integer maxiter_outer, maxiter_inner
210     double precision tol_abs, tol_rel
211
212     mat%params%maxiter_outer = maxiter_outer
213     mat%params%maxiter_inner = maxiter_inner
214     mat%params%tol_abs = tol_abs
215     mat%params%tol_rel = tol_rel
216 end subroutine mat_set_solver_params
217
218 subroutine mat_calculate_repeat_ent(mat)
219     ! Should be called when incrementing row
220     class(rte_mat) mat
221
222     ! Index of L_{ijklp}
223     ! whose coefficient will be modified
224     ! several times per row
225     mat%repeat_ent = mat%ent + mat%p - 1
226
227 end subroutine mat_calculate_repeat_ent
228
229 function mat_ind(mat, i, j, k, p) result(ind)
230     ! Assuming var ordering: z, x, y, omega
231     class(rte_mat) mat
232     type(space_angle_grid) grid
233     integer i, j, k, p
234     integer ind
235     grid = mat%grid
236
237     ind = (i-1) * mat%x_block_size + (j-1) * mat
238         %y_block_size + &
239         (k-1) * mat%z_block_size + p * mat%
240         omega_block_size
241 end function mat_ind
242
243 subroutine mat_set_row(mat, indices)
244     ! These indices act as a row counter

```



```

242      ! Row should always be incremented in
243      ! angular_integral, which should be called
244      ! before derivatives, bcs, and attenuation
245      class(rte_mat) mat
246      type(index_list) indices
247
248      mat%i = indices%i
249      mat%j = indices%j
250      mat%k = indices%k
251      mat%p = indices%p
252
253      call mat%calculate_repeat_ent()
254
255  end subroutine mat_set_row
256
257  subroutine mat_assign(mat, val, i, j, k, p)
258      ! It's assumed that this is the only time
259      ! this entry is defined
260      class(rte_mat) mat
261      double precision val
262      integer i, j, k, p
263      integer row_num, col_num
264
265      row_num = mat%ind(mat%i, mat%j, mat%k, mat%
266      p)
267      col_num = mat%ind(i, j, k, p)
268
269      mat%row(mat%ent) = row_num
270      mat%col(mat%ent) = col_num
271      if(isnan(val)) then
272          write(*,*) 'ISNAN'
273          write(*,*) 'row = ', row_num
274          write(*,*) 'col = ', col_num
275          write(*,*) 'mat_index =', mat%i, mat%j,
276          mat%k, mat%p
277          write(*,*) 'index =', i, j, k, p
278          write(*,*) 'entry =', mat%ent
279      endif
280
281      ! if(i.eq.mat%i .and. j.eq.mat%j .and. k.eq.
282      ! mat%j .and. l.eq.mat%l .and. p.eq.mat%p)
283      ! then
284      !     write(*,*) 'diag: ', val
285      ! endif
286
287      mat%data(mat%ent) = val
288
289      ! Remember where we stored information for
290      ! this matrix element
291      ! call mat%store_index(row_num, col_num)

```

```

287     mat%ent = mat%ent + 1
288 end subroutine mat_assign
289
290 subroutine mat_add(mat, val)
291     ! Use this when you know that this entry has
292     ! already been assigned
293     ! and you'd like to add this value to the
294     ! existing value.
295
296     class(rte_mat) mat
297     double precision val
298     integer index
299
300     ! Entry number where value is already stored
301     index = mat%repeat_ent
302
303     mat%data(index) = mat%data(index) + val
304 end subroutine mat_add
305
306 subroutine mat_assign_rhs(mat, data)
307     class(rte_mat) mat
308     double precision data
309     integer row_num
310
311     row_num = mat%ind(mat%i, mat%j, mat%k, mat%p
312     )
313     mat%rhs(row_num) = data
314 end subroutine mat_assign_rhs
315
316 ! subroutine mat_store_index(mat, row_num,
317 !     col_num)
318 !     ! Remember where we stored information for
319 !     ! this matrix element
320 !     class(rte_mat) mat
321 !     integer row_num, col_num
322 !     !mat%index_map(row_num, col_num) = mat%ent
323 ! end subroutine
324
325 ! function mat_find_index(mat, row_num,
326 !     col_num) result(index)
327 !     ! Find the position in row, col, data
328 !     ! where this entry
329 !     ! is defined.
330 !     class(rte_mat) mat
331 !     integer row_num, col_num, index
332
333     ! index = mat%index_map(row_num, col_num)
334
335     ! ! This took up 95% of execution time.
336     ! ! Only search up to most recently assigned
337     ! ! index
338     ! do index=1, mat%ent-1

```

```

331 | !      !      if( (mat%row(index) .eq. row_num) .
      |         and. (mat%col(index) .eq. col_num)) then
332 | !      !      exit
333 | !      !      end if
334 | !      !      end do
335 | ! end function mat_find_index
336 |
337 | subroutine attenuate(mat, indices)
338 |   ! Has to be called after angular_integral
339 |   ! Because they both write to the same matrix
      |   entry
340 |   ! And adding here is more efficient than a
      |   conditional
341 |   ! in the angular loop.
342 |   class(rte_mat) mat
343 |   type(optical_properties) iops
344 |   double precision attenuation
345 |   type(index_list) indices
346 |   double precision aa, bb
347 |   iops = mat%iops
348 |
349 |   aa = iops%abs_grid(indices%i, indices%j,
      |         indices%k)
350 |   bb = iops%scat
351 |
352 |   attenuation = aa + bb*(1-iops%vsf_integral(
      |         indices%p, indices%p))
353 |   call mat%add(attenuation)
354 | end subroutine attenuate
355 |
356 | subroutine x_cd2(mat, indices)
357 |   class(rte_mat) mat
358 |   type(space_angle_grid) grid
359 |   double precision val, dx
360 |   type(index_list) indices
361 |   integer i, j, k, p
362 |   i = indices%i
363 |   j = indices%j
364 |   k = indices%k
365 |   p = indices%p
366 |   grid = mat%grid
367 |
368 |   dx = grid%x%spacing(1)
369 |
370 |   val = grid%angles%sin_phi_p(p) * grid%angles
      |         %cos_theta_p(p) / (2.d0 * dx)
371 |
372 |   call mat%assign(-val,i-1,j,k,p)
373 |   call mat%assign(val,i+1,j,k,p)
374 | end subroutine x_cd2
375 |

```

```

376 | subroutine x_cd2_first(mat, indices)
377 |   class(rte_mat) mat
378 |   type(space_angle_grid) grid
379 |   double precision val, dx
380 |   integer nx
381 |   type(index_list) indices
382 |   integer i, j, k, p
383 |
384 |   i = indices%i
385 |   j = indices%j
386 |   k = indices%k
387 |   p = indices%p
388 |   grid = mat%grid
389 |
390 |   dx = grid%x%spacing(1)
391 |   nx = grid%x%num
392 |
393 |   val = grid%angles%sin_phi_p(p) * grid%angles
394 |         %cos_theta_p(p) / (2.d0 * dx)
395 |
396 |   call mat%assign(-val,nx,j,k,p)
397 |   call mat%assign(val,i+1,j,k,p)
398 | end subroutine x_cd2_first
399 |
400 | subroutine x_cd2_last(mat, indices)
401 |   class(rte_mat) mat
402 |   type(space_angle_grid) grid
403 |   double precision val, dx
404 |   type(index_list) indices
405 |   integer i, j, k, p
406 |   i = indices%i
407 |   j = indices%j
408 |   k = indices%k
409 |   p = indices%p
410 |   grid = mat%grid
411 |
412 |   dx = grid%x%spacing(1)
413 |
414 |   val = grid%angles%sin_phi_p(p) * grid%angles
415 |         %cos_theta_p(p) / (2.d0 * dx)
416 |
417 |   call mat%assign(-val,i-1,j,k,p)
418 |   call mat%assign(val,1,j,k,p)
419 | end subroutine x_cd2_last
420 |
421 | subroutine y_cd2(mat, indices)
422 |   class(rte_mat) mat
423 |   type(space_angle_grid) grid
424 |   double precision val, dy
425 |   type(index_list) indices

```

```

424 | integer i, j, k, p
425 | i = indices%i
426 | j = indices%j
427 | k = indices%k
428 | p = indices%p
429 | grid = mat%grid
430 |
431 | dy = grid%y%spacing(1)
432 |
433 | val = grid%angles%sin_phi_p(p) * grid%angles
      | %sin_theta_p(p) / (2.d0 * dy)
434 |
435 | call mat%assign(-val,i,j-1,k,p)
436 | call mat%assign(val,i,j+1,k,p)
437 | end subroutine y_cd2
438 |
439 | subroutine y_cd2_first(mat, indices)
440 |   class(rte_mat) mat
441 |   type(space_angle_grid) grid
442 |   double precision val, dy
443 |   integer ny
444 |   type(index_list) indices
445 |   integer i, j, k, p
446 |   i = indices%i
447 |   j = indices%j
448 |   k = indices%k
449 |   p = indices%p
450 |   grid = mat%grid
451 |
452 |   dy = grid%y%spacing(1)
453 |   ny = grid%y%num
454 |
455 |   val = grid%angles%sin_phi_p(p) * grid%angles
      | %sin_theta_p(p) / (2.d0 * dy)
456 |
457 |   call mat%assign(-val,i,ny,k,p)
458 |   call mat%assign(val,i,j+1,k,p)
459 | end subroutine y_cd2_first
460 |
461 | subroutine y_cd2_last(mat, indices)
462 |   class(rte_mat) mat
463 |   type(space_angle_grid) grid
464 |   double precision val, dy
465 |   type(index_list) indices
466 |   integer i, j, k, p
467 |   i = indices%i
468 |   j = indices%j
469 |   k = indices%k
470 |   p = indices%p
471 |   grid = mat%grid

```

```

472
473     dy = grid%y%spacing(1)
474
475     val = grid%angles%sin_phi_p(p) * grid%angles
         %sin_theta_p(p) / (2.d0 * dy)
476
477     call mat%assign(-val,i,j-1,k,p)
478     call mat%assign(val,i,1,k,p)
479 end subroutine y_cd2_last
480
481 subroutine z_cd2(mat, indices)
482     class(rte_mat) mat
483     type(space_angle_grid) grid
484     double precision val, dz
485     type(index_list) indices
486     integer i, j, k, p
487     i = indices%i
488     j = indices%j
489     k = indices%k
490     p = indices%p
491     grid = mat%grid
492
493     dz = grid%z%spacing(indices%k)
494
495     val = grid%angles%cos_phi_p(p) / (2.d0 * dz)
496
497     call mat%assign(-val,i,j,k-1,p)
498     call mat%assign(val,i,j,k+1,p)
499 end subroutine z_cd2
500
501 subroutine z_fd2(mat, indices)
502     ! Has to be called after angular_integral
503     ! Because they both write to the same matrix
         entry
504     ! And adding here is more efficient than a
         conditional
505     ! in the angular loop.
506     class(rte_mat) mat
507     type(space_angle_grid) grid
508     double precision val, val1, val2, val3, dz
509     type(index_list) indices
510     integer i, j, k, p
511     i = indices%i
512     j = indices%j
513     k = indices%k
514     p = indices%p
515     grid = mat%grid
516
517     dz = grid%z%spacing(indices%k)
518

```

```

519     val = grid%angles%cos_phi_p(p) / (2.d0 * dz)
520
521     val1 = -3.d0 * val
522     val2 = 4.d0 * val
523     val3 = -val
524
525     call mat%add(val1)
526     call mat%assign(val2,i,j,k+1,p)
527     call mat%assign(val3,i,j,k+2,p)
528 end subroutine z_fd2
529
530 subroutine z_bd2(mat, indices)
531     ! Has to be called after angular_integral
532     ! Because they both write to the same matrix
533     ! And adding here is more efficient than a
534     ! conditional
535     ! in the angular loop.
536     class(rte_mat) mat
537     type(space_angle_grid) grid
538     double precision val, val1, val2, val3, dz
539     type(index_list) indices
540     integer i, j, k, p
541     i = indices%i
542     j = indices%j
543     k = indices%k
544     p = indices%p
545     grid = mat%grid
546
547     dz = grid%z%spacing(indices%k)
548
549     val = grid%angles%cos_phi_p(p) / (2.d0 * dz)
550
551     val1 = 3.d0 * val
552     val2 = -4.d0 * val
553     val3 = val
554
555     call mat%add(val1)
556     call mat%assign(val2,i,j,k-1,p)
557     call mat%assign(val3,i,j,k-2,p)
558 end subroutine z_bd2
559
560 subroutine angular_integral(mat, indices)
561     class(rte_mat) mat
562     ! Primed angular integration variables
563     integer pp
564     double precision val
565     type(index_list) indices
566
567     call mat%set_row(indices)

```

```

568
569     ! Interior
570     do pp=1, mat%grid%angles%nomega
571         ! TODO: Make sure I don't have p and pp
            backwards
572         val = mat%iops%scat * mat%iops%
            vsf_integral(indices%p, pp)
573         call mat%assign(val, indices%i, indices%j
            , indices%k, pp)
574     end do
575
576 end subroutine angular_integral
577
578 subroutine z_surface_bc(mat, indices)
579     class(rte_mat) mat
580     type(space_angle_grid) grid
581     double precision bc_val
582     type(index_list) indices
583     double precision val1, val2
584
585     grid = mat%grid
586
587     val1 = grid%angles%cos_phi_p(indices%p) /
            (5.d0 * grid%z%spacing(1))
588     val2 = 7.d0 * val1
589
590     call mat%assign(val1, indices%i, indices%j, 2,
            indices%p)
591     call mat%add(val2)
592
593     bc_val = 8.d0 * mat%surface_vals(indices%p)
            / (5.d0 * grid%z%spacing(1))
594     call mat%assign_rhs(bc_val)
595
596 end subroutine z_surface_bc
597
598 subroutine z_bottom_bc(mat, indices)
599     class(rte_mat) mat
600     type(space_angle_grid) grid
601     type(index_list) indices
602     double precision val1, val2
603
604     grid = mat%grid
605
606     val1 = -grid%angles%cos_phi_p(indices%p) /
            (5.d0 * grid%z%spacing(1))
607     val2 = 7.d0 * val1
608
609     call mat%assign(val1, indices%i, indices%j,
            grid%z%num-1, indices%p)
610     call mat%add(val2)

```



```

611
612     end subroutine z_bottom_bc
613
614     ! Finite difference wrappers
615
616     subroutine wrap_x_cd2(mat, indices)
617         type(rte_mat) mat
618         type(index_list) indices
619         call mat%x_cd2(indices)
620     end subroutine wrap_x_cd2
621
622     subroutine wrap_x_cd2_last(mat, indices)
623         type(rte_mat) mat
624         type(index_list) indices
625         call mat%x_cd2_last(indices)
626     end subroutine wrap_x_cd2_last
627
628     subroutine wrap_x_cd2_first(mat, indices)
629         type(rte_mat) mat
630         type(index_list) indices
631         call mat%x_cd2_first(indices)
632     end subroutine wrap_x_cd2_first
633
634     subroutine wrap_y_cd2(mat, indices)
635         type(rte_mat) mat
636         type(index_list) indices
637         call mat%y_cd2(indices)
638     end subroutine wrap_y_cd2
639
640     subroutine wrap_y_cd2_last(mat, indices)
641         type(rte_mat) mat
642         type(index_list) indices
643         call mat%y_cd2_last(indices)
644     end subroutine wrap_y_cd2_last
645
646     subroutine wrap_y_cd2_first(mat, indices)
647         type(rte_mat) mat
648         type(index_list) indices
649         call mat%y_cd2_first(indices)
650     end subroutine wrap_y_cd2_first
651
652     subroutine wrap_z_cd2(mat, indices)
653         type(rte_mat) mat
654         type(index_list) indices
655         call mat%z_cd2(indices)
656     end subroutine wrap_z_cd2
657
658 end module rte_sparse_matrices

```

rte3d.f90

```

1 module rte3d
2 use kelp_context
3 use rte_sparse_matrices
4 use light_context
5 implicit none
6
7 contains
8
9 subroutine interior_space_loop(mat, indices)
10   type(rte_mat) mat
11   type(space_angle_grid) grid
12   type(index_list) indices
13   integer i, j, k
14
15   grid = mat%grid
16
17   ! z interior
18   !$OMP PARALLEL DO FIRSTPRIVATE(indices)
19   do k=2, grid%z%num - 1
20     indices%k = k
21     write(*,*) 'k =', indices%k, '/', grid%z%
22       num
23     ! x first
24     indices%i=1
25     ! y first
26     indices%j=1
27     call interior_angle_loop(mat, indices,
28       wrap_x_cd2_first, wrap_y_cd2_first)
29
30     ! y interior
31     do j=2, grid%y%num - 1
32       indices%j = j
33       call interior_angle_loop(mat, indices,
34         wrap_x_cd2_first, wrap_y_cd2)
35     end do
36
37     ! y last
38     indices%j=grid%y%num
39     call interior_angle_loop(mat, indices,
40       wrap_x_cd2_first, wrap_y_cd2_last)
41
42     ! x interior
43     do i=2, grid%x%num - 1
44       indices%i = i
45       ! y first
46       indices%j=1
47       call interior_angle_loop(mat, indices,
48         wrap_x_cd2, wrap_y_cd2_first)
49
50       ! y interior
51       do j=2, grid%y%num - 1
52         indices%j = j

```

```

46         call interior_angle_loop(mat, indices
47             , wrap_x_cd2, wrap_y_cd2)
48     end do
49     ! y last
49     indices%j=grid%y%num
50     call interior_angle_loop(mat, indices
51         , wrap_x_cd2, wrap_y_cd2_last)
51 end do
52
53 ! x last
54 indices%i=grid%x%num
55 ! y first
56 indices%j=1
57 call interior_angle_loop(mat, indices,
58     wrap_x_cd2_last, wrap_y_cd2_first)
58 ! y interior
59 do j=2, grid%y%num - 1
60     indices%j = j
61     call interior_angle_loop(mat, indices,
62         wrap_x_cd2_last, wrap_y_cd2)
62 end do
63 ! y last
64 indices%j=grid%y%num
65 call interior_angle_loop(mat, indices,
66     wrap_x_cd2_last, wrap_y_cd2_last)
66
67 end do
68 !$OMP END PARALLEL DO
69
70 end subroutine
71
72
73 subroutine surface_space_loop(mat, indices)
74     type(rte_mat) mat
75     type(space_angle_grid) grid
76     type(index_list) indices
77     integer i, j
78
79     grid = mat%grid
80
81     ! z surface
82     indices%k=1
83     write(*,*) 'k =', indices%k, '/', grid%z%
84         num
85     ! x first
86     indices%i=1
87     ! y first
88     indices%j=1
89     call surface_angle_loop(mat, indices,
90         wrap_x_cd2_first,
91         wrap_y_cd2_first)

```

```

89         ! y interior
90         do j=2, grid%y%num - 1
91             indices%j = j
92             call surface_angle_loop(mat, indices,
                                     wrap_x_cd2_first, wrap_y_cd2)
93         end do
94         ! y last
95         indices%j=grid%y%num
96         call surface_angle_loop(mat, indices,
                                     wrap_x_cd2_first, wrap_y_cd2_last
                                     )
97
98     ! x interior
99     !$OMP PARALLEL DO FIRSTPRIVATE(indices)
100    do i=2, grid%x%num - 1
101        indices%i = i
102        ! y first
103        indices%j=1
104        call surface_angle_loop(mat, indices,
                                wrap_x_cd2, wrap_y_cd2_first)
105        ! y interior
106        do j=2, grid%y%num - 1
107            indices%j = j
108            call surface_angle_loop(mat, indices,
                                    wrap_x_cd2, wrap_y_cd2)
109        end do
110        ! y last
111        indices%j=grid%y%num
112        call surface_angle_loop(mat, indices,
                                wrap_x_cd2, wrap_y_cd2_last)
113    end do
114    !$OMP END PARALLEL DO
115
116    ! x last
117    indices%i=grid%x%num
118    ! y first
119    indices%j=1
120    call surface_angle_loop(mat, indices,
                            wrap_x_cd2_last, wrap_y_cd2_first)
121    ! y surface
122    do j=2, grid%y%num - 1
123        indices%j = j
124        call surface_angle_loop(mat, indices,
                                wrap_x_cd2_last, wrap_y_cd2)
125    end do
126    ! y last
127    indices%j=grid%y%num
128    call surface_angle_loop(mat, indices,
                            wrap_x_cd2_last, wrap_y_cd2_last)
129

```

```

130 | end subroutine surface_space_loop
131 |
132 | subroutine bottom_space_loop(mat, indices)
133 |     type(rte_mat) mat
134 |     type(space_angle_grid) grid
135 |     type(index_list) indices
136 |     integer i, j
137 |
138 |     grid = mat%grid
139 |
140 |     ! z bottom
141 |     indices%k=grid%z%num
142 |     write(*,*) 'k =', indices%k, '/', grid%z%
143 |         num
144 |     ! x first
145 |     indices%i=1
146 |     ! y first
147 |     indices%j=1
148 |     call bottom_angle_loop(mat, indices,
149 |         wrap_x_cd2_first, wrap_y_cd2_first)
150 |     ! y interior
151 |     do j=2, grid%y%num - 1
152 |         indices%j = j
153 |         call bottom_angle_loop(mat, indices,
154 |             wrap_x_cd2_first, wrap_y_cd2)
155 |     end do
156 |     ! y last
157 |     indices%j=grid%y%num
158 |     call bottom_angle_loop(mat, indices,
159 |         wrap_x_cd2_first, wrap_y_cd2_last)
160 |
161 |     ! x interior
162 |     !$OMP PARALLEL DO FIRSTPRIVATE(indices)
163 |     do i=2, grid%x%num - 1
164 |         indices%i = i
165 |         ! y first
166 |         indices%j=1
167 |         call bottom_angle_loop(mat, indices,
168 |             wrap_x_cd2, wrap_y_cd2_first)
169 |         ! y bottom
170 |         do j=2, grid%y%num - 1
171 |             indices%j = j
172 |             call bottom_angle_loop(mat, indices,
173 |                 wrap_x_cd2, wrap_y_cd2)
174 |         end do
175 |         ! y last
176 |         indices%j=grid%y%num
177 |         call bottom_angle_loop(mat, indices,
178 |             wrap_x_cd2, wrap_y_cd2_last)
179 |     end do

```

```

173         !$OMP END PARALLEL DO
174
175         ! x last
176         indices%i=grid%x%num
177         ! y first
178         indices%j=1
179         call bottom_angle_loop(mat, indices,
180                                wrap_x_cd2_last, wrap_y_cd2_first)
181         ! y interior
182         do j=2, grid%y%num - 1
183             indices%j = j
184             call bottom_angle_loop(mat, indices,
185                                    wrap_x_cd2_last, wrap_y_cd2)
186         end do
187         ! y last
188         indices%j=grid%y%num
189         call bottom_angle_loop(mat, indices,
190                                wrap_x_cd2_last, wrap_y_cd2_last)
191     end subroutine
192
193     subroutine interior_angle_loop(mat, indices, ddx
194                                     , ddy)
195         type(space_angle_grid) grid
196         type(rte_mat) mat
197         type(index_list) indices
198         integer p
199
200         ! Allow derivative subroutines to be passed as
201         arguments
202         interface
203             subroutine ddx(mat, indices)
204                 use sag
205                 use rte_sparse_matrices
206                 type(rte_mat) mat
207                 type(index_list) indices
208             end subroutine ddx
209             subroutine ddy(mat, indices)
210                 use sag
211                 use rte_sparse_matrices
212                 type(rte_mat) mat
213                 type(index_list) indices
214             end subroutine ddy
215         end interface
216
217         grid = mat%grid
218
219         do p=1, grid%angles%nomega
220             indices%p = p
221             call mat%angular_integral(indices)
222             call ddx(mat, indices)

```

```

219 |         call ddy(mat, indices)
220 |         call mat%z_cd2(indices)
221 |         call mat%attenuate(indices)
222 |     end do
223 | end subroutine
224 |
225 |
226 | subroutine surface_angle_loop(mat, indices, ddx,
    |         ddy)
227 |     type(space_angle_grid) grid
228 |     type(rte_mat) mat
229 |     type(index_list) indices
230 |     integer p
231 |
232 |     ! Allow derivative subroutines to be passed as
    |         arguments
233 |     interface
234 |         subroutine ddx(mat, indices)
235 |             use sag
236 |             use rte_sparse_matrices
237 |             type(rte_mat) mat
238 |             type(index_list) indices
239 |         end subroutine ddx
240 |         subroutine ddy(mat, indices)
241 |             use sag
242 |             use rte_sparse_matrices
243 |             type(rte_mat) mat
244 |             type(index_list) indices
245 |         end subroutine ddy
246 |     end interface
247 |
248 |     grid = mat%grid
249 |
250 |     ! Downwelling
251 |     do p=1, grid%angles%nomega / 2
252 |         indices%p = p
253 |         call mat%angular_integral(indices)
254 |         call ddx(mat, indices)
255 |         call ddy(mat, indices)
256 |         call mat%z_surface_bc(indices)
257 |         call mat%attenuate(indices)
258 |     end do
259 |
260 |     ! Upwelling
261 |     do p=grid%angles%nomega/2+1, grid%angles%
    |         nomega
262 |         indices%p = p
263 |         call mat%angular_integral(indices)
264 |         call ddx(mat, indices)
265 |         call ddy(mat, indices)
266 |         call mat%z_fd2(indices)

```

```

267     call mat%attenuate(indices)
268 end do
269
270 end subroutine surface_angle_loop
271
272 subroutine bottom_angle_loop(mat, indices, ddx,
    ddy)
273     type(space_angle_grid) grid
274     type(rte_mat) mat
275     type(index_list) indices
276     integer p
277
278     ! Allow derivative subroutines to be passed as
        arguments
279     interface
280         subroutine ddx(mat, indices)
281             use sag
282             use rte_sparse_matrices
283             type(rte_mat) mat
284             type(index_list) indices
285         end subroutine ddx
286         subroutine ddy(mat, indices)
287             use sag
288             use rte_sparse_matrices
289             type(rte_mat) mat
290             type(index_list) indices
291         end subroutine ddy
292     end interface
293
294     grid = mat%grid
295
296     ! Downwelling
297     do p=1, grid%angles%nomega/2
298         indices%p = p
299         call mat%angular_integral(indices)
300         call ddx(mat, indices)
301         call ddy(mat, indices)
302         call mat%z_bd2(indices)
303         call mat%attenuate(indices)
304     end do
305
306     ! Upwelling
307     do p=grid%angles%nomega/2+1, grid%angles%
        nomega
308         indices%p = p
309         call mat%angular_integral(indices)
310         call ddx(mat, indices)
311         call ddy(mat, indices)
312         call mat%z_bottom_bc(indices)
313         call mat%attenuate(indices)
314     end do

```



```

315
316 end subroutine bottom_angle_loop
317
318 subroutine gen_matrix(mat)
319     type(rte_mat) mat
320     type(index_list) indices
321
322     call indices%init()
323
324     call surface_space_loop(mat, indices)
325     call interior_space_loop(mat, indices)
326     call bottom_space_loop(mat, indices)
327 end subroutine gen_matrix
328
329 subroutine rte3d_deinit(mat, iops, light)
330     type(rte_mat) mat
331     type(optical_properties) iops
332     type(light_state) light
333
334     call mat%deinit()
335     call iops%deinit()
336     call light%deinit()
337 end subroutine
338
339 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 =>
21         frond_calculate_angles
22 end type frond_shape
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 max_rad, 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,
77 |             bc%phi_s)
78 |         bc_gaussian = bc%max_rad * exp(-bc%decay *
79 |             diff)
80 |     end function bc_gaussian
81 |
82 |     subroutine bc_init(bc, grid, theta_s, phi_s,
83 |         decay, max_rad)
84 |         class(boundary_condition) bc
85 |         type(space_angle_grid) grid
86 |         double precision theta_s, phi_s, decay,
87 |             max_rad
88 |         integer p
89 |         double precision theta, phi
90 |
91 |         allocate(bc%bc_grid(grid%angles%nomega))
92 |
93 |         bc%theta_s = theta_s
94 |         bc%phi_s = phi_s
95 |         bc%decay = decay
96 |         bc%max_rad = max_rad
97 |
98 |         ! Only set BC for downwelling light
99 |         do p=1, grid%angles%nomega/2
100 |             theta = grid%angles%theta_p(p)
101 |             phi = grid%angles%phi_p(p)
102 |             bc%bc_grid(p) = bc%bc_gaussian(theta, phi
103 |                 )
104 |         end do
105 |         ! Zero upwelling light specified at surface
106 |         do p=grid%angles%nomega/2+1, grid%angles%
107 |             nomega
108 |             bc%bc_grid(p) = 0.d0
109 |         end do
110 |     end subroutine bc_init
111 |
112 |     subroutine bc_deinit(bc)
113 |         class(boundary_condition) bc
114 |         deallocate(bc%bc_grid)
115 |     end subroutine

```

```

111
112 subroutine point_set_cart(point, x, y, z)
113     class(point3d) :: point
114     double precision x, y, z
115     point%x = x
116     point%y = y
117     point%z = z
118     call point%cartesian_to_polar()
119 end subroutine point_set_cart
120
121 subroutine point_set_cyl(point, theta, r, z)
122     class(point3d) :: point
123     double precision theta, r, z
124     point%theta = theta
125     point%r = r
126     point%z = z
127     call point%polar_to_cartesian()
128 end subroutine point_set_cyl
129
130 subroutine polar_to_cartesian(point)
131     class(point3d) :: point
132     point%x = point%r*cos(point%theta)
133     point%y = point%r*sin(point%theta)
134 end subroutine polar_to_cartesian
135
136 subroutine cartesian_to_polar(point)
137     class(point3d) :: point
138     point%r = sqrt(point%x**2 + point%y**2)
139     point%theta = atan2(point%y, point%x)
140 end subroutine cartesian_to_polar
141
142 subroutine frond_set_shape(frond, fs, fr, ft)
143     class(frond_shape) frond
144     double precision fs, fr, ft
145     frond%fs = fs
146     frond%fr = fr
147     frond%ft = ft
148     call frond%calculate_angles()
149 end subroutine frond_set_shape
150
151 subroutine frond_calculate_angles(frond)
152     class(frond_shape) frond
153     frond%tan_alpha = 2.d0*frond%fs*frond%fr /
154         (1.d0 + frond%fs)
155     frond%alpha = atan(frond%tan_alpha)
156 end subroutine
157
158 subroutine iop_init(iops, grid)
159     class(optical_properties) iops
160     type(space_angle_grid) grid

```

```

160
161     iops%grid = grid
162
163     ! Assume that these are preallocated and
164     ! passed to function
165     ! Nevermind, don't assume this.
166     allocate(iops%abs_water(grid%z%num))
167
168     ! Assume that these must be allocated here
169     allocate(iops%vsf_angles(iops%num_vsf))
170     allocate(iops%vsf_vals(iops%num_vsf))
171     allocate(iops%vsf_cos(iops%num_vsf))
172     allocate(iops%vsf(grid%angles%nomega,grid%
173         angles%nomega))
174     allocate(iops%vsf_integral(grid%angles%
175         nomega,grid%angles%nomega))
176     allocate(iops%abs_grid(grid%x%num, grid%y%
177         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 -
189             iops%abs_water(k)) * p_kelp(:,:,k) +
190             iops%abs_water(k)
191     end do
192 end subroutine calculate_coef_grids
193
194 subroutine load_vsf(iops, filename, fmtstr)
195     class(optical_properties) :: iops
196     character(len=*) :: filename, fmtstr
197     double precision, dimension(:,:),
198         allocatable :: tmp_2d_arr
199     integer num_rows, num_cols, skiplines_in
200
201     ! First column is the angle at which the
202     ! measurement is taken
203     ! Second column is the value of the VSF at
204     ! that angle
205     num_rows = iops%num_vsf
206     num_cols = 2
207     skiplines_in = 1 ! Ignore comment on first
208         line

```

```

201
202     allocate(tmp_2d_arr(num_rows, num_cols))
203
204     tmp_2d_arr = read_array(filename, fmtstr,
205         num_rows, num_cols, skiplines_in)
206     iops%vsf_angles = tmp_2d_arr(:,1)
207     iops%vsf_vals = tmp_2d_arr(:,2)
208
209     ! write(*,*) 'vsf_angles = ', iops%
210         vsf_angles
211     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
212
213     ! Pre-evaluate for all pair of angles
214     call iops%calc_vsf_on_grid()
215 end subroutine load_vsf
216
217 function eval_vsf(iops, theta)
218     class(optical_properties) iops
219     double precision theta
220     double precision eval_vsf
221     ! No need to set vsf(0) = 0.
222     ! It's the area under the curve that matters
223     , not the value.
224     eval_vsf = interp(theta, iops%vsf_angles,
225         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

```

```

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

```

```

287     diff = angle_diff_2d(theta_f, theta_w)
288
289     call von_mises_pdf(diff, 0.d0, v_w, output)
290 end function angle_distribution_pdf
291
292 function angle_mod(theta) result(mod_theta)
293     ! Shift theta to the interval [-pi, pi]
294     ! which is where von_mises_pdf is defined.
295
296     double precision theta, mod_theta
297
298     mod_theta = mod(theta + pi, 2.d0*pi) - pi
299 end function angle_mod
300
301 function angle_diff_2d(theta1, theta2) result(
302     diff)
303     ! Shortest difference between two angles
304     ! which may be
305     ! in different periods.
306     double precision theta1, theta2, diff
307     double precision modt1, modt2
308
309     ! Shift to [0, 2*pi]
310     modt1 = mod(theta1, 2*pi)
311     modt2 = mod(theta2, 2*pi)
312
313     ! https://gamedev.stackexchange.com/
314     ! questions/4467/comparing-angles-and-
315     ! 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
321     , phi_prime) result(diff)
322     ! Angle between two vectors in spherical
323     ! coordinates
324     double precision theta, phi, theta_prime,
325     phi_prime
326     double precision alpha, diff
327
328     ! Faster, but produces lots of NaNs (at
329     ! least in Python)
330     ! alpha = sin(theta)*sin(theta_prime)*cos(
331     ! theta-theta_prime) + cos(phi)*cos(
332     ! phi_prime)
333
334     ! Slower, but more accurate
335     alpha = (sin(phi)*sin(phi_prime) &

```



```

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

```

```

373 | iops%vsf_vals(:) = iops%vsf_vals(:) / (2*pi
      | * iops%vsf_scatt_coef)
374 |
375 | ! write(*,*) 'norm = ', iops%vsf_scatt_coef
376 | ! write(*,*) 'now: ', trap_rule_uneven(iops%
      | vsf_cos, iops%vsf_vals, iops%num_vsf)
377 | ! write(*,*) 'cos: ', iops%vsf_cos
378 | ! write(*,*) 'vals: ', iops%vsf_vals
379 |
380 | do p=1, nomega
381 |     th = grid%angles%theta_p(p)
382 |     ph = grid%angles%phi_p(p)
383 |     do pp=1, nomega
384 |         thp = grid%angles%theta_p(pp)
385 |         php = grid%angles%phi_p(pp)
386 |         ! TODO: Might be better to calculate
      |         average scattering
387 |         ! from angular cell rather than only
      |         using center
388 |         iops%vsf(p, pp) = iops%eval_vsf(
      |         angle_diff_3d(th,ph,thp,php))
389 |     end do
390 |
391 |     ! Normalize each row of VSF (midpoint
      |     rule)
392 |     norm = sum(iops%vsf(p,:) * grid%angles%
      |     area_p(:))
393 |     iops%vsf(p,:) = iops%vsf(p,:) / norm
394 |
395 |     ! % / meter light scattered from cell pp
      |     into direction p.
396 |     ! TODO: Could integrate VSF instead of
      |     just using value at center
397 |     iops%vsf_integral(p, :) = iops%vsf(p, :)
      |     * grid%angles%area_p(:)
398 |     !write(*,*) 'vsf_integral (beta_pp)', p,
      |     ' = ', iops%vsf_integral(p, :)
399 | end do
400 |
401 | ! Normalize VSF on unit sphere w.r.t. north
      | pole
402 | !iops%vsf_scatt_coef = sum(iops%vsf(1,:) *
      | iops%grid%angles%area_p)
403 | !iops%vsf = iops%vsf / iops%vsf_scatt_coef
404 | !iops%vsf_integral = iops%vsf_integral /
      | iops%vsf_scatt_coef
405 | end subroutine calc_vsf_on_grid
406 |
407 | subroutine iop_deinit(iops)
408 |     class(optical_properties) iops

```

```

409 |     deallocate(iops%vsf_angles)
410 |     deallocate(iops%vsf_vals)
411 |     deallocate(iops%vsf_cos)
412 |     deallocate(iops%vsf)
413 |     deallocate(iops%vsf_integral)
414 |     deallocate(iops%abs_water)
415 |     deallocate(iops%abs_grid)
416 |
417 |     end subroutine iop_deinit
418 |
419 | 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(:,:,:),
9 |       allocatable :: irradiance
10 |     double precision, dimension(:,:,:,:),
11 |       allocatable :: radiance
12 |     type(space_angle_grid) :: grid
13 |     type(rte_mat) :: mat
14 |     contains
15 |       procedure :: init => light_init
16 |       procedure :: init_grid => light_init_grid
17 |       procedure :: calculate_radiance
18 |       procedure :: calculate_irradiance
19 |       procedure :: deinit => light_deinit
20 |       !procedure :: to_hdf => light_to_hdf
21 |     end type light_state
22 |
23 | contains
24 |
25 |   ! Init for use with mat
26 |   subroutine light_init(light, mat)
27 |     class(light_state) light
28 |     type(rte_mat) mat
29 |     integer nx, ny, nz, nomega
30 |
31 |     light%mat = mat
32 |     light%grid = mat%grid
33 |
34 |     nx = light%grid%x%num
35 |     ny = light%grid%y%num
36 |     nz = light%grid%z%num
37 |     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%
79                         radiance(i,j,k,p)
80                     index = index + 1
81                 end do
82             end do
83         end do
84     end do
85
86     !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%
98                         mat%sol(index)
99                     index = index + 1
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
110         nx = light%grid%x%num
111         ny = light%grid%y%num
112         nz = light%grid%z%num
113
114         do i=1, nx
115             do j=1, ny
116                 do k=1, nz
117                     light%irradiance(i,j,k) = light%
118                         grid%angles%integrate_points( &
119                             light%radiance(i,j,k,:))
120                 end do
121             end do
122         end do
123     end subroutine calculate_irradiance
124
125     ! subroutine light_to_hdf(light, radfile,
126     !     irrادfile)
127     !     class(light_state) light
128     !     character(len=*) radfile
129     !     character(len=*) irrادfile
130     !     call hdf_write_radiance(radfile, light%
131     !         radiance, light%grid)
132     !     call hdf_write_irradiance(irradfile, light%
133     !         irradiance, light%grid)

```

```

132 | ! end subroutine light_to_hdf
133 |
134 | subroutine light_deinit(light)
135 |     class(light_state) light
136 |
137 |     deallocate(light%irradiance)
138 |     deallocate(light%radiance)
139 | end subroutine light_deinit
140 | end module

```

asymptotics.f90

```

1 | module asymptotics
2 |     use kelp_context
3 |     !use rte_sparse_matrices
4 |     !use light_context
5 |     implicit none
6 |     contains
7 |
8 |     subroutine calculate_light_with_scattering(
9 |         grid, bc, iops, radiance, num_scatters)
10 |         type(space_angle_grid) grid
11 |         type(boundary_condition) bc
12 |         type(optical_properties) iops
13 |         double precision, dimension(:,:,:,::) ::
14 |             radiance
15 |         double precision, dimension(:,:,:,::),
16 |             allocatable :: source
17 |         integer num_scatters
18 |         integer nx, ny, nz, nomega
19 |         integer max_cells
20 |
21 |         double precision, dimension(:), allocatable
22 |             :: path_length, path_spacing, a_tilde, gn
23 |
24 |         nx = grid%x%num
25 |         ny = grid%y%num
26 |         nz = grid%z%num
27 |         nomega = grid%angles%nomega
28 |
29 |         max_cells = calculate_max_cells(grid)
30 |
31 |         allocate(path_length(max_cells+1))
32 |         allocate(path_spacing(max_cells))
33 |         allocate(a_tilde(max_cells))
34 |         allocate(gn(max_cells))
35 |         allocate(source(nx, ny, nz, nomega))
36 |
37 |         call calculate_light_before_scattering(grid,
38 |             bc, iops, source, radiance, path_length,
39 |             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,
39             path_spacing, &
40             a_tilde, gn)
41     end if
42
43     deallocate(path_length)
44     deallocate(path_spacing)
45     deallocate(a_tilde)
46     deallocate(gn)
47     deallocate(source)
48 end subroutine calculate_light_with_scattering
49
50 subroutine calculate_light_before_scattering(
51     grid, bc, iops, source, radiance,
52     path_length, path_spacing, a_tilde, gn)
53     type(space_angle_grid) grid
54     type(boundary_condition) bc
55     type(optical_properties) iops
56     double precision, dimension(:,:,:) ::
57         radiance, source
58     double precision, dimension(:) ::
59         path_length, path_spacing, a_tilde, gn
60     integer i, j, k, p
61     double precision surface_val
62
63     ! Downwelling light
64     do p=1, grid%angles%nomega/2
65         surface_val = bc%bc_grid(p)
66         do i=1, grid%x%num
67             do j=1, grid%y%num
68                 do k=1, grid%z%num
69                     call attenuate_light_from_surface
70                     (&
71                         grid, iops, source, i, j, k,
72                         p,&
73                         radiance, path_length,
74                         path_spacing,&
75                         a_tilde, gn, bc)
76                 end do
77             end do
78         end do
79     end do
80
81     ! No upwelling light before scattering
82     do p = grid%angles%nomega/2+1, grid%angles%
83         nomega
84         do i=1, grid%x%num

```

```

76         do j=1, grid%y%num
77             do k=1, grid%z%num
78                 radiance(i,j,k,p) = 0.d0
79             end do
80         end do
81     end do
82 end do
83 end subroutine
    calculate_light_before_scattering
84
85 subroutine attenuate_light_from_surface(grid,
86     iops, source, i, j, k, p,&
87     radiance, path_length, path_spacing,
88     a_tilde, gn, bc)
89     type(space_angle_grid) grid
90     type(boundary_condition) bc
91     type(optical_properties) iops
92     double precision, dimension(:,:,:,:) ::
93         radiance, source
94     double precision, dimension(:) ::
95         path_length, path_spacing, a_tilde, gn
96     integer i, j, k, p
97     integer num_cells
98     double precision atten
99
100     ! Don't need gn here, so just ignore it
101     call traverse_ray(grid, iops, source, i, j,
102         k, p, path_length, path_spacing, a_tilde,
103         gn, num_cells)
104
105     ! Start with surface bc and attenuate along
106     path
107     atten = sum(path_spacing(1:num_cells) *
108         a_tilde(1:num_cells))
109     ! Avoid underflow
110     if(atten .lt. 100.d0) then
111         radiance(i,j,k,p) = bc%bc_grid(p) * exp(-
112             atten)
113     else
114         radiance(i,j,k,p) = 0.d0
115     end if
116 end subroutine attenuate_light_from_surface
117
118 subroutine calculate_light_after_scattering(
119     grid, iops, source, radiance,&
120     num_scatters, path_length, path_spacing,
121     a_tilde, gn)
122     type(space_angle_grid) grid
123     type(optical_properties) iops

```



```

114 | double precision, dimension(:,:,:,) ::
      |     radiance, source
115 | integer num_scatters
116 | double precision, dimension(:) ::
      |     path_length, path_spacing, a_tilde, gn
117 | double precision, dimension(:,:,:,),
      |     allocatable :: rad_scatter
118 | integer n
119 | double precision bb
120 |
121 | allocate(rad_scatter(grid%x%num, grid%y%num,
      |     grid%z%num, grid%angles%nomega))
122 | rad_scatter = radiance
123 | bb = iops%scat
124 |
125 | do n=1, num_scatters
126 |     write(*,*) 'scatter #', n
127 |     call scatter(grid, iops, source,
      |         rad_scatter, path_length, path_spacing
      |         , a_tilde, gn)
128 |     radiance = radiance + bb*n * rad_scatter
129 | end do
130 |
131 | deallocate(rad_scatter)
132 | end subroutine
      | calculate_light_after_scattering
133 |
134 | ! Perform one scattering event
135 | subroutine scatter(grid, iops, source,
      |     rad_scatter, path_length, path_spacing,
      |     a_tilde, gn)
136 | type(space_angle_grid) grid
137 | type(optical_properties) iops
138 | double precision, dimension(:,:,:,) ::
      |     rad_scatter, source
139 | double precision, dimension(:,:,:,),
      |     allocatable :: scatter_integral
140 | double precision, dimension(:) ::
      |     path_length, path_spacing, a_tilde, gn
141 | integer nx, ny, nz, nomega
142 |
143 | nx = grid%x%num
144 | ny = grid%y%num
145 | nz = grid%z%num
146 | nomega = grid%angles%nomega
147 |
148 | allocate(scatter_integral(nx, ny, nz, nomega
      | ))
149 |
150 | call calculate_source(grid, iops,
      |     rad_scatter, source, scatter_integral)

```

```

151      call advect_light(grid, iops, source,
152                        rad_scatter, path_length, path_spacing,
153                        a_tilde, gn)
154  end subroutine scatter
155
156  ! Calculate source from no-scatter or previous
157    scattering layer
158  subroutine calculate_source(grid, iops,
159    rad_scatter, source, scatter_integral)
160    type(space_angle_grid) grid
161    type(optical_properties) iops
162    double precision, dimension(:,:,:,:) ::
163      rad_scatter, source, scatter_integral
164    type(index_list) indices
165    integer nx, ny, nz, nomega
166    integer i, j, k, p
167
168    nx = grid%x%num
169    ny = grid%y%num
170    nz = grid%z%num
171    nomega = grid%angles%nomega
172
173    !$OMP PARALLEL DO FIRSTPRIVATE(indices)
174    do k=1, nz
175      indices%k = k
176      do i=1, nx
177        indices%i = i
178        do j=1, ny
179          indices%j = j
180          do p=1, nomega
181            indices%p = p
182            call calculate_scatter_integral
183              (&
184                iops, rad_scatter,&
185                scatter_integral,&
186                indices)
187          end do
188        end do
189      end do
190    end do
191    !$OMP END PARALLEL DO
192
193    source(:,:,:,:) = -rad_scatter(:,:,:,:) +
194      scatter_integral(:,:,:,:)
195  end subroutine calculate_source
196
197  subroutine calculate_scatter_integral(iops,
198    rad_scatter, scatter_integral, indices)

```

```

194     type(optical_properties) iops
195     double precision, dimension(:,:,:,:) ::
196         rad_scatter, scatter_integral
197     type(index_list) indices
198
199     scatter_integral(indices%i,indices%j,indices
200         %k,indices%p) &
201         = sum(iops%vsf_integral(indices%p, :) &
202             * rad_scatter(&
203                 indices%i,&
204                 indices%j,&
205                 indices%k,:))
206
207 end subroutine calculate_scatter_integral
208
209 subroutine advect_light(grid, iops, source,
210     rad_scatter, path_length, path_spacing,
211     a_tilde, gn)
212     type(space_angle_grid) grid
213     type(optical_properties) iops
214     double precision, dimension(:,:,:,:) ::
215         rad_scatter, source
216     double precision, dimension(:) ::
217         path_length, path_spacing, a_tilde, gn
218
219     integer i, j, k, p
220     integer nx, ny, nz, nomega
221
222     nx = grid%x%num
223     ny = grid%y%num
224     nz = grid%z%num
225     nomega = grid%angles%nomomega
226
227     !$OMP PARALLEL DO FIRSTPRIVATE(i, j, p)
228     do k=1, nz
229         do i=1, nx
230             do j=1, ny
231                 do p=1, nomega
232                     call integrate_ray(grid, iops,
233                         source,&
234                         rad_scatter, path_length,
235                         path_spacing,&
236                         a_tilde, gn, i, j, k, p)
237                 end do
238             end do
239         end do
240     end do
241     !$OMP END PARALLEL DO
242
243 end subroutine advect_light

```

```

237 | ! New algorithm, double integral over
      | piecewise-constant 1d funcs
238 | subroutine integrate_ray(grid, iops, source,
      |   rad_scatter, path_length, path_spacing,
      |   a_tilde, gn, i, j, k, p)
239 |   type(space_angle_grid) grid
240 |   type(optical_properties) iops
241 |   double precision, dimension(:, :, :, :) ::
      |     source, rad_scatter
242 |   double precision, dimension(:) ::
      |     path_length, path_spacing, a_tilde, gn
243 |   integer i, j, k, p
244 |   integer num_cells
245 |
246 |   call traverse_ray(grid, iops, source, i, j,
      |     k, p, path_length, path_spacing, a_tilde,
      |     gn, num_cells)
247 |   rad_scatter(i, j, k, p) =
      |     calculate_ray_integral(num_cells,
      |       path_length, path_spacing, a_tilde, gn)
248 | end subroutine integrate_ray
249 |
250 | function calculate_ray_integral(num_cells, s,
      |   ds, a_tilde, gn) result(integral)
251 |   ! Double integral which accumulates all
      |     scattered light along the path
252 |   ! via an angular integral and attenuates it
      |     by integrating along the path
253 |   integer num_cells
254 |   double precision, dimension(num_cells) :: ds
      |     , a_tilde, gn
255 |   double precision, dimension(num_cells+1) ::
      |     s
256 |   double precision integral, bi, di
257 |   integer i, j
258 |
259 |   integral = 0
260 |   do i=1, num_cells
261 |     bi = -a_tilde(i)*s(i+1)
262 |     do j=i+1, num_cells
263 |       bi = bi - a_tilde(j)*ds(j)
264 |     end do
265 |
266 |     ! Careful: This will overflow if a_tilde
      |       is too large.
267 |     if(a_tilde(i) .eq. 0) then
268 |       di = ds(i)
269 |     else
270 |       di = (exp(a_tilde(i)*s(i+1))-exp(
      |         a_tilde(i)*s(i)))/a_tilde(i)
271 |     end if

```

```

272
273     integral = integral + gn(i)*di * exp(bi)
274 end do
275
276 end function calculate_ray_integral
277
278 ! Calculate maximum number of cells a path
    through the grid could take
279 ! This is a loose upper bound
280 function calculate_max_cells(grid) result(
    max_cells)
281     type(space_angle_grid) grid
282     integer max_cells
283     double precision dx, dy, zrange, phi_middle
284
285     ! Angle that will have the longest ray
286     phi_middle = grid%angles%phi(grid%angles%
        nphi/2)
287     dx = grid%x%spacing(1)
288     dy = grid%y%spacing(1)
289     zrange = grid%z%maxval - grid%z%minval
290
291     max_cells = grid%z%num + ceiling((1/dx+1/dy)
        *zrange*tan(phi_middle))
292 end function calculate_max_cells
293
294 ! Traverse from surface or bottom to point (xi
    , yj, zk)
295 ! in the direction omega_p, extracting path
    lengths (ds) and
296 ! function values (f) along the way,
297 ! as well as number of cells traversed (n).
298 subroutine traverse_ray(grid, iops, source, i,
    j, k, p, s_array, ds, a_tilde, gn,
    num_cells)
299     type(space_angle_grid) grid
300     type(optical_properties) iops
301     integer i, j, k, p
302     double precision, dimension(:,:,:) ::
        source
303     double precision, dimension(:), intent(out)
        :: s_array, ds, a_tilde, gn
304     integer t
305     integer, intent(out) :: num_cells
306     double precision p0x, p0y, p0z
307     double precision p1x, p1y, p1z
308     double precision z0
309     double precision s_tilde, s
310     integer dir_x, dir_y, dir_z
311     integer shift_x, shift_y, shift_z
312     integer cell_x, cell_y, cell_z

```

```

313 integer edge_x, edge_y, edge_z
314 integer first_x, last_x, first_y, last_y,
    last_z
315 double precision s_next_x, s_next_y,
    s_next_z, s_next
316 double precision x_factor, y_factor,
    z_factor
317 double precision ds_x, ds_y
318 double precision, dimension(grid%z%num) ::
    ds_z
319 double precision smx, smy
320
321 ! Divide by these numbers to get path
    separation
322 ! from separation in individual dimensions
323 x_factor = grid%angles%sin_phi_p(p) * grid%
    angles%cos_theta_p(p)
324 y_factor = grid%angles%sin_phi_p(p) * grid%
    angles%sin_theta_p(p)
325 z_factor = grid%angles%cos_phi_p(p)
326
327 ! Destination point
328 p1x = grid%x%vals(i)
329 p1y = grid%y%vals(j)
330 p1z = grid%z%vals(k)
331
332 !write(*,*) 'START PATH.'
333 !write(*,*) 'ijk = ', i, j, k
334
335 ! Direction
336 if(p .le. grid%angles%nomega/2) then
337     ! Downwelling light originates from
        surface
338     z0 = grid%z%minval
339     dir_z = 1
340 else
341     ! Upwelling light originates from bottom
342     z0 = grid%z%maxval
343     dir_z = -1
344 end if
345
346 ! Total path length from origin to
    destination
347 ! (sign is correct for upwelling and
    downwelling)
348 s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p
    )
349
350 ! Path spacings between edge intersections
    in each dimension
351 ! Set to 2*s_tilde if infinite in this
    dimension so that it's unreachable

```

```

352      ! Assume x & y spacings are uniform,
353      ! so it's okay to just use the first value.
354      if(x_factor .eq. 0) then
355          ds_x = 2*s_tilde
356      else
357          ds_x = abs(grid%x%spacing(1)/x_factor)
358      end if
359      if(y_factor .eq. 0) then
360          ds_y = 2*s_tilde
361      else
362          ds_y = abs(grid%y%spacing(1)/y_factor)
363      end if
364
365      ! This one is an array because z spacing can
366      ! vary
367      ! z_factor should never be 0, because the
368      ! ray will never
369      ! reach the surface or bottom.
370      ds_z(1:grid%z%num) = dir_z * grid%z%spacing
371      (1:grid%z%num)/z_factor
372
373      ! Origin point
374      p0x = p1x - s_tilde * x_factor
375      p0y = p1y - s_tilde * y_factor
376      p0z = p1z - s_tilde * z_factor
377
378      ! Direction of ray in each dimension. 1 =>
379      ! increasing. -1 => decreasing.
380      dir_x = int(sgn(p1x-p0x))
381      dir_y = int(sgn(p1y-p0y))
382
383      ! Shifts
384      ! Conversion from cell_inds to edge_inds
385      ! merge is fortran's ternary operator
386      shift_x = merge(1,0,dir_x>0)
387      shift_y = merge(1,0,dir_y>0)
388      shift_z = merge(1,0,dir_z>0)
389
390      ! Indices for cell containing origin point
391      cell_x = floor((p0x-grid%x%minval)/grid%x%
392      spacing(1)) + 1
393      cell_y = floor((p0y-grid%y%minval)/grid%y%
394      spacing(1)) + 1
395      ! x and y may be in periodic image, so shift
396      ! back.
397      cell_x = mod1(cell_x, grid%x%num)
398      cell_y = mod1(cell_y, grid%y%num)
399
400      ! z starts at top or bottom depending on
401      ! direction.
402      if(dir_z > 0) then
403          cell_z = 1

```

```

396     else
397         cell_z = grid%z%num
398     end if
399
400     ! Edge indices preceeding starting cells
401     edge_x = mod1(cell_x + shift_x, grid%x%num)
402     edge_y = mod1(cell_y + shift_y, grid%y%num)
403     edge_z = mod1(cell_z + shift_z, grid%z%num)
404
405     ! First and last cells given direction
406     if(dir_x .gt. 0) then
407         first_x = 1
408         last_x = grid%x%num
409     else
410         first_x = grid%x%num
411         last_x = 1
412     end if
413     if(dir_y .gt. 0) then
414         first_y = 1
415         last_y = grid%y%num
416     else
417         first_y = grid%y%num
418         last_y = 1
419     end if
420     if(dir_z .gt. 0) then
421         last_z = grid%z%num
422     else
423         last_z = 1
424     end if
425
426     ! Calculate periodic images
427     smx = shift_mod(p0x, grid%x%minval, grid%x%
428         maxval)
429     smy = shift_mod(p0y, grid%y%minval, grid%y%
430         maxval)
431
432     ! Path length to next edge plane in each
433     dimension
434     if(abs(x_factor) .lt. 1.d-10) then
435         ! Will never cross, so set above total
436         path length
437         s_next_x = 2*s_tilde
438     else if(cell_x .eq. last_x) then
439         ! If starts out at last cell, then
440         compare to periodic image
441         s_next_x = (grid%x%edges(first_x) + dir_x
442             * (grid%x%maxval - grid%x%minval)&
443             - smx) / x_factor
444     else
445         ! Otherwise, just compare to next cell

```



```

440         s_next_x = (grid%x%edges(edge_x) - smx) /
              x_factor
441     end if
442
443     ! Path length to next edge plane in each
              dimension
444     if(abs(y_factor) .lt. 1.d-10) then
445         ! Will never cross, so set above total
              path length
446         s_next_y = 2*s_tilde
447     else if(cell_y .eq. last_y) then
448         ! If starts out at last cell, then
              compare to periodic image
449         s_next_y = (grid%y%edges(first_y) + dir_y
              * (grid%y%maxval - grid%y%minval)&
              - smy) / y_factor
450     else
451         ! Otherwise, just compare to next cell
452         s_next_y = (grid%y%edges(edge_y) - smy) /
              y_factor
453     end if
454
455     s_next_z = ds_z(cell_z)
456
457     ! Initialize path
458     s = 0.d0
459     s_array(1) = 0.d0
460
461     ! Start with t=0 so that we can increment
              before storing,
462     ! so that t will be the number of grid cells
              at the end of the loop.
463     t=0
464
465     ! s is the beginning of the current cell,
466     ! s_next is the end of the current cell.
467     do while (s .lt. s_tilde)
468         ! Move cell counter
469         t = t + 1
470
471         ! Extract function values
472         a_tilde(t) = iops%abs_grid(cell_x, cell_y
              , cell_z)
473         gn(t) = source(cell_x, cell_y, cell_z, p)
474
475         !write(*,*) ''
476         !write(*,*) 's_next_x = ', s_next_x
477         !write(*,*) 's_next_y = ', s_next_y
478         !write(*,*) 's_next_z = ', s_next_z
479         !write(*,*) 'theta, phi =', grid%angles%
              theta_p(p)*180.d0/pi, grid%angles%
              phi_p(p)*180.d0/pi

```

```

481      !write(*,*) 's = ', s, '/', s_tilde
482      !write(*,*) 'cell_z =', cell_z, '/', grid
          %z%num
483      !write(*,*) 's_next_z =', s_next_z
484      !write(*,*) 'last_z =', last_z
485      !write(*,*) 'new'
486
487      ! Move to next cell in path
488      if(s_next_x .le. min(s_next_y, s_next_z))
          then
489          ! x edge is closest
490          s_next = s_next_x
491
492          ! Increment indices (periodic)
493          cell_x = mod1(cell_x + dir_x, grid%x%
          num)
494          edge_x = mod1(edge_x + dir_x, grid%x%
          num)
495
496          ! x intersection after the one at s=
          s_next
497          s_next_x = s_next + ds_x
498
499      else if (s_next_y .le. min(s_next_x,
          s_next_z)) then
500          ! y edge is closest
501          s_next = s_next_y
502
503          ! Increment indices (periodic)
504          cell_y = mod1(cell_y + dir_y, grid%y%
          num)
505          edge_y = mod1(edge_y + dir_y, grid%y%
          num)
506
507          ! y intersection after the one at s=
          s_next
508          s_next_y = s_next + ds_y
509
510      else if(s_next_z .le. min(s_next_x,
          s_next_y)) then
511          ! z edge is closest
512          s_next = s_next_z
513
514          ! Increment indices
515          cell_z = cell_z + dir_z
516          edge_z = edge_z + dir_z
517
518          !write(*,*) 'z edge, s_next =', s_next
519
520          ! z intersection after the one at s=
          s_next

```

```

521         if(cell_z .lt. last_z) then
522             ! Only look ahead if we aren't at
                    the end
523             s_next_z = s_next + ds_z(cell_z)
524         else
525             ! Otherwise, no need to continue.
526             ! this is our final destination.
527             ! exit
528             s_next_z = 2*s_tilde
529             !write(*,*) 'end. s_next_z =',
                    s_next_z
530         end if
531
532     end if
533
534     ! Cut off early if this is the end
535     ! This will be the last cell traversed if
        s_next >= s_tilde
536     s_next = min(s_tilde, s_next)
537
538     ! Store path length
539     s_array(t+1) = s_next
540     ! Extract path length from same cell as
        function vals
541     ds(t) = s_next - s
542
543     ! Update path length
544     s = s_next
545 end do
546
547     ! Return number of cells traversed
548     num_cells = t
549
550 end subroutine traverse_ray
551 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,
47                               nphi
48      ! Absorption and scattering coefficients
49      double precision, intent(in) ::
50          absorptance_kelp, scat
51      double precision, dimension(nz), intent(in)
52          :: abs_water
53      ! Volume scattering function
54      integer, intent(in) :: num_vsf
55      character(len=*) :: vsf_file
56      !double precision, dimension(num_vsf),
57          intent(int) :: vsf_angles
58      !double precision, dimension(num_vsf),
59          intent(int) :: vsf_vals
60
61      ! SUNLIGHT
62      double precision, intent(in) :: solar_zenith
63      double precision, intent(in) ::
64          solar_azimuthal
65      double precision, intent(in) ::
66          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
104 | :: pop_length_means, pop_length_stds
105 | ! Number of fronds in each depth layer
106 | double precision, dimension(:), allocatable
107 | :: num_fronds
108 | double precision, dimension(:,:,:),
109 | allocatable :: p_kelp
110 |
111 | write(*,*) 'Light calculation'
112 |
113 | allocate(pop_length_means(nz))
114 | allocate(pop_length_stds(nz))
115 | allocate(num_fronds(nz))
116 | allocate(p_kelp(nx, ny, nz))
117 |
118 | xmin = -rope_spacing/2
119 | xmax = rope_spacing/2
120 |
121 | ymin = -rope_spacing/2
122 | ymax = rope_spacing/2
123 |
124 | zmin = 0.d0
125 | zmax = sum(depth_spacing)
126 |
127 | write(*,*) 'Grid'
128 | call grid%set_bounds(xmin, xmax, ymin, ymax,
129 | zmin, zmax)
130 | call grid%set_num(nx, ny, nz, ntheta, nphi)
131 | call grid%init()
132 | !call grid%set_uniform_spacing_from_num()
133 | call grid%z%set_spacing_array(depth_spacing)
134 |
135 | call rope%init(grid)
136 |
137 | write(*,*) 'Rope'
138 | ! Calculate kelp distribution
139 | call calculate_length_dist_from_superinds( &
140 | nz, &
141 | num_si, &
142 | si_area, &
143 | si_ind, &

```

```

140     frond_aspect_ratio, &
141     num_fronnds, &
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_fronnds = num_fronnds
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_fronnds  =', rope%num_fronnds
154     write(*,*) 'water_speeds  =', rope%
        water_speeds
155     write(*,*) 'water_angles  =', rope%
        water_angles
156
157     write(*,*) 'Fronnd'
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_fronnds)
239     deallocate(p_kelp)
240
241     !write(*,*) 'done'
242 end subroutine full_light_calculations
243
244 subroutine
245     calculate_length_dist_from_superinds( &
246     nz, &
247     num_si, &
248     si_area, &
249     si_ind, &
250     frond_aspect_ratio, &
251     num_fronnds, &
252     pop_length_means, &
253     pop_length_stds)
254
255     implicit none
256
257     ! Number of depth levels
258     integer, intent(in) :: nz
259     ! Number of Superindividuals in each depth
260     level
261     integer, intent(in) :: num_si
262     ! si_area(i,j) = area of superindividual j
263     at depth i
264     double precision, dimension(nz, num_si),
265     intent(in) :: si_area
266     ! si_area(i,j) = number of inidividuals
267     represented by superindividual j at depth
268     i
269     double precision, dimension(nz, num_si),
270     intent(in) :: si_ind
271     double precision, intent(in) ::
272     frond_aspect_ratio
273

```

```

266      double precision, dimension(nz), intent(out)
267      :: num_fronnds
268      ! Population mean area at each depth level
269      double precision, dimension(nz), intent(out)
270      :: pop_length_means
271      ! Population area standard deviation at each
272      depth level
273      double precision, dimension(nz), intent(out)
274      :: pop_length_stds
275
276      !-----!
277
278      integer i, k
279      ! Numerators for mean and std
280      double precision mean_num, std_num
281      ! Convert area to length
282      double precision, dimension(num_si) ::
283      si_length
284
285      do k=1, nz
286          mean_num = 0.d0
287          std_num = 0.d0
288          num_fronnds(k) = 0
289
290          do i=1, num_si
291              si_length(i) = sqrt(2.d0*
292                  frond_aspect_ratio*si_area(k,i))
293              mean_num = mean_num + si_length(i)
294              num_fronnds(k) = num_fronnds(k) + si_ind
295                  (k,i)
296          end do
297
298          pop_length_means(k) = mean_num /
299              num_fronnds(k)
300
301          do i=1, num_si
302              std_num = std_num + (si_length(i) -
303                  pop_length_means(k))**2
304          end do
305
306          pop_length_stds(k) = std_num / (
307              num_fronnds(k) - 1)
308      end do
309
310      end subroutine
311      calculate_length_dist_from_superinds
312
313      subroutine calculate_average_irradiance(grid,
314          light, avg_irrad)
315      type(space_angle_grid) grid
316      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
315                               (:,:,k)) / nx / ny)
316     end do
317 end subroutine calculate_average_irradiance
318
319 subroutine calculate_perceived_irrad(grid,
320     p_kelp, &
321     perceived_irrad, irradiance)
322 type(space_angle_grid) grid
323 double precision, dimension(:,:,:) :: p_kelp
324 real, dimension(:) :: perceived_irrad
325 double precision, dimension(:,:,:) ::
326     irradiance
327
328 integer k
329
330 ! Calculate the average irradiance
331 ! experienced over the frond.
332 ! Has same units as irradiance.
333 do k=1, grid%z%num
334     perceived_irrad(k) = real( &
335         sum(p_kelp(:,:,k)*irradiance(:,:,k))
336         &
337         / sum(p_kelp(:,:,k)))
338     end do
339 end subroutine calculate_perceived_irrad
340
341 end module light_interface_module

```