

©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 . . . . .	4
1.3 Background on Radiative Transfer . . . . .	7
1.4 Overview of Thesis . . . . .	8
II. KELP MODEL . . . . .	10
2.1 Physical Setup . . . . .	10
2.2 Coordinate System . . . . .	11
2.3 Population Distributions . . . . .	13
2.4 Spatial Distribution . . . . .	17
III. LIGHT MODEL . . . . .	24
3.1 Optical Definitions . . . . .	24
3.2 The Radiative Transfer Equation . . . . .	26
3.3 Low-Scattering Approximation . . . . .	28

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 [10]. 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 $4 \times 4$ array of vertical kelp ropes . . . . .	11
2.2 Downward-facing right-handed coordinate system with radial distance $r$ from the origin, distance $s$ from the $z$ axis, zenith angle $\phi$ and azimuthal angle $\theta$ . . . . .	12
2.3 Simplified kite-shaped frond . . . . .	13
2.4 von Mises distribution for a variety of parameters . . . . .	16
2.5 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$ . . . . .	17
2.6 A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$ . . . . .	18
2.7 Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$ . . . . .	21
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 $(\theta, s) = (3\pi/4, 3/2)$ . . . . .	22
2.9 Contour plot of the probability of occupying sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$ . . . . .	23
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. In particular, food and fuel are clearly in high demand. Meanwhile, growing concern for the negative environmental impacts of petroleum-based fuel is generating a market for biofuel, especially corn-based ethanol. However, corn-based ethanol has been heavily criticized for diverting land usage away from food production. At the same time, a great deal of unutilized saltwater coastline is available for both food and fuel production through seaweed cultivation. Specifically, the sugar kelp *Saccharina Latissima* is known to be a viable source of food, both for direct human consumption and for fish cultivation, as well as for biofuel production.

Furthermore, nitrogen leakage into water bodies is a significant ecological danger, and is especially relevant near large conventional agriculture facilities due to run-off from nitrogen-based fertilizers, as well as near wastewater treatment plants. As a specific example, there is a wastewater treatment plant in Boothbay Harbor, Maine which is facing increasingly demanding EPA regulations limiting the concentration of certain nutrients permissible to be released into the ocean via wastewater

treatment outfalls. In order to adhere to these stricter requirements using conventional nutrient remediation, a significant quantity of specialized equipment would be necessary, which is not currently present in the Boothbay Harbor plant. Being surrounded on all sides by water and private property, the treatment plant lacks the necessary space for the additional equipment, and would therefore need to move their entire facility to a new location in order to conform to these new nutrient regulations. As an alternative to conventional nutrient remediation techniques, the cultivation of the macroalgae *Saccharina Latissima* (sugar kelp) near the outfall site has been proposed. The purpose of such an undertaking would be twofold: to prevent eutrophication of the surrounding ecosystem by sequestering the nutrients in question, and to reduce the need for nutrient input, which is one of the largest costs in macroalgae cultivation.

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

Industrial scale macroalgae cultivation has long existed in Eastern Asia due to the popularity of seaweed in Asian cuisine. More recently, kelp aquaculture has been developing in Scandinavia and in the Northeastern United States. For example, the MACROSEA project is a four year international research collaboration funded by

the Research Council of Norway targeting “successful and predictable production of high quality biomass thereby making significant steps towards industrial macroalgae cultivation in Norway.” The project includes both cultivators and scientists, working to develop a precise understanding of the full life cycle of kelp and its interaction with its environment. A fundamental aspect of this endeavor is the development of mathematical models to describe the growth of kelp. Work is underway at SINTEF, a private Norwegian research institution, to develop such models. Ole Jacob Broch is a mathematician at SINTEF, a research organization in Trondheim, Norway, who has been working to model the growth of *Saccharina Latissima* using SINMOD, a large-scale 3D hydrodynamical ocean model developed at SINTEF which generates data on water temperature, water velocity, light intensity, and phytoplankton concentrations among other valuable quantities [13].

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

by source code for a FORTRAN implementation of the solution. This model can be used independently, or in conjunction with a life cycle kelp model to determine the amount of light available for photosynthesis at a single time step.



Figure 1.1: *Saccharina Latissima* being harvested

## 1.2 Background on Kelp Models

Mathematical modeling of macroalgae growth is not a new topic, although it is a reemerging one. Several authors in the second half of the twentieth century were interested in describing the growth and composition of the macroalgae *Macrocystis pyrifera*, commonly known as “giant kelp,” which grows prolifically off the coast

of southern California. The first such mathematical model was developed by W.J. North for the Kelp Habitat Improvement Project at the California Institute of Technology in 1968 using seven variables. By 1974, Nick Anderson greatly expanded on North's work, and created the first comprehensive model of kelp growth which he programmed using FORTRAN [1]. In his model, he accounts for solar radiation intensity as a function of time of year and time of day, and refraction on the surface of the water. He uses a simple model for shading, simply specifying a single parameter which determines the percentage of light which is allowed to pass through the kelp canopy floating on the surface of the water. He also accounts for attenuation due to turbidity using Beer's Law. Using this data on the availability of light, he calculates the photosynthesis rates and the growth experienced by the kelp.

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

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

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

### 1.3 Background on Radiative Transfer

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

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



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

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

#### 1.4 Overview of Thesis

The remainder of this document is organized as follows. In Chapter 2, we develop a probabilistic model to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. We begin Chapter 3 with a survey of fundamental radiometric quantities and optical properties of matter. We use the spatial kelp distribution from Chapter 2 to determine optical properties of the combined water-kelp medium. We then present the Radiative Transfer Equation, an integro-partial differential equation which describes the the light field as a function of position and angle. An asymptotic expansion is explored for cases where absorption dominates scattering in the medium, such as in very clear water or high kelp den-

sity. In Chapter 4, details are given for the numerical solution of the equations from Chapters 2 and 3. Both the full finite difference solution and the asymptotic approximation are described. Next, in Chapter 5, we discuss the availability of necessary parameters in the literature. For those which are not readily available, we give rough estimates and briefly describe experimental methods for their determination. Then, in Chapter 6, we investigate the necessary grid resolution for adequate accuracy in the full finite difference solution and compare to the asymptotic approximation for a few parameter sets. Further, we determine the effect of varying a few key parameters on the light field predicted by the asymptotic approximation. Afterwards, we use the light model developed here in a full lifecycle simulation of kelp growth and compare the light field and biomass production to those predicted by a simpler 1D exponential decay light model. Finally, we conclude with Chapter 7 by giving a brief summary of the model, discuss and its performance, and suggest improvements and avenues for future work.

## CHAPTER II

### KELP MODEL

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

#### 2.1 Physical Setup

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

light on the surface and the nature of scattering in the water. In addition, light will be naturally absorbed by the water to varying degrees as determined by the clarity of the water.

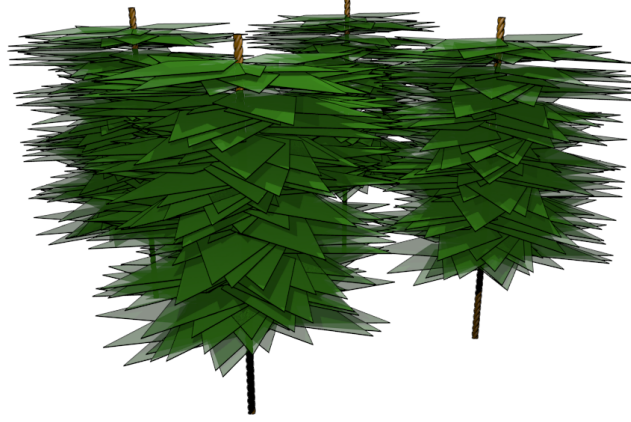


Figure 2.1:  $4 \times 4$  array of vertical kelp ropes

## 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

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

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

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

Then, calculating derivatives from (2.1) yields

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

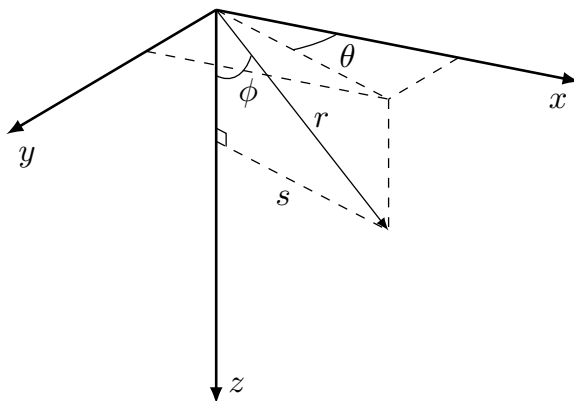


Figure 2.2: Downward-facing right-handed coordinate system with radial distance  $r$  from the origin, distance  $s$  from the  $z$  axis, zenith angle  $\phi$  and azimuthal angle  $\theta$

## 2.3 Population Distributions

### 2.3.1 Frond Shape

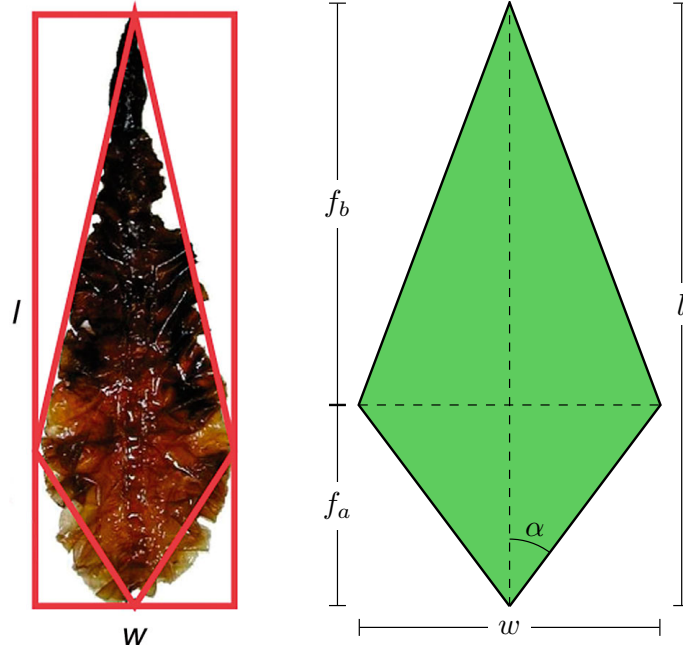


Figure 2.3: Simplified kite-shaped frond

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

$$f_a + f_b = l \quad (2.4)$$

When considering a whole population with varying sizes, it is more convenient to

specify ratios than absolute lengths. Let the following ratios be defined.

$$f_r = \frac{l}{w} \quad (2.5)$$

$$f_s = \frac{f_a}{f_b} \quad (2.6)$$

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

$$w = \frac{l}{f_r} \quad (2.7)$$

$$f_a = \frac{l f_s}{1 + f_s} \quad (2.8)$$

$$f_b = \frac{l}{1 + f_s} \quad (2.9)$$

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

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

The area of the frond is given by

$$A = \frac{lw}{2} = \frac{l^2}{2f_r}. \quad (2.11)$$

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

$$l = \sqrt{2A f_r} \quad (2.12)$$

### 2.3.2 Length and Angle Distributions

We assume that frond lengths are normally distributed with mean  $\mu_l$  and standard deviation  $\sigma_l$ . We assume the frond angle varies according to the von Mises distribution, which is the periodic analogue of the normal distribution, defined on  $[-\pi, \pi]$  rather than  $(-\infty, \infty)$ . The von Mises distribution has two parameters,  $\mu$  and  $\kappa$ , which shift and sharpen its peak respectively, as shown in Figure 2.4.  $\kappa$  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  $\theta_w$  and  $v_w$  vary over depth.

The PDF for this distribution is

$$P_{\theta_f}(\theta_f) = \frac{\exp(v_w \cos(\theta_f - \theta_w))}{2\pi I_0(v_w)} \quad (2.13)$$

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

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

### 2.3.3 Joint Length-Orientation Distribution

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



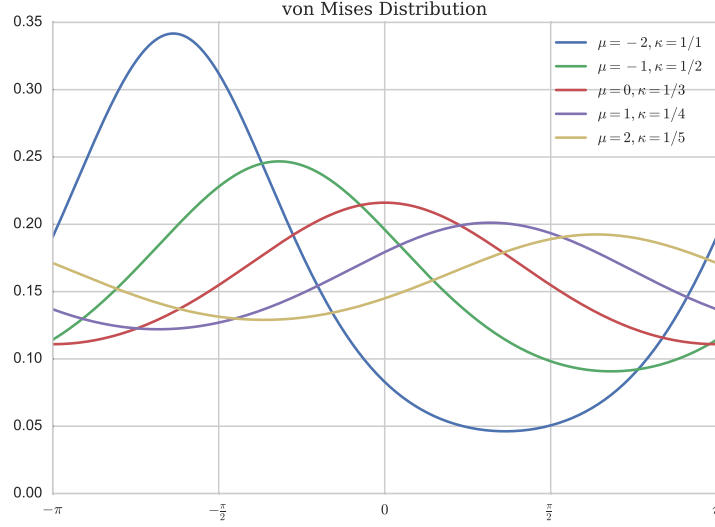


Figure 2.4: von Mises distribution for a variety of parameters

Given independent events  $A$  and  $B$ ,

$$P(A \cap B) = P(A)P(B) \quad (2.15)$$

Then the probability of frond length  $l$  and frond angle  $\theta_f$  coinciding is

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

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

It is important to note that if  $P_{\theta_f}$  were dependent on  $l$ , the above definition of  $P_{2D}$  would no longer be valid. For example, it might be more realistic to say that

larger fronds are less likely to bend towards the direction of the current. In this case, (2.15) would no longer hold, and it would be necessary to use the following more general relation.

$$P(A \cap B) = P(A)P(B|A) = P(B)P(B|A) \quad (2.17)$$

This is currently not taken into consideration in this model.

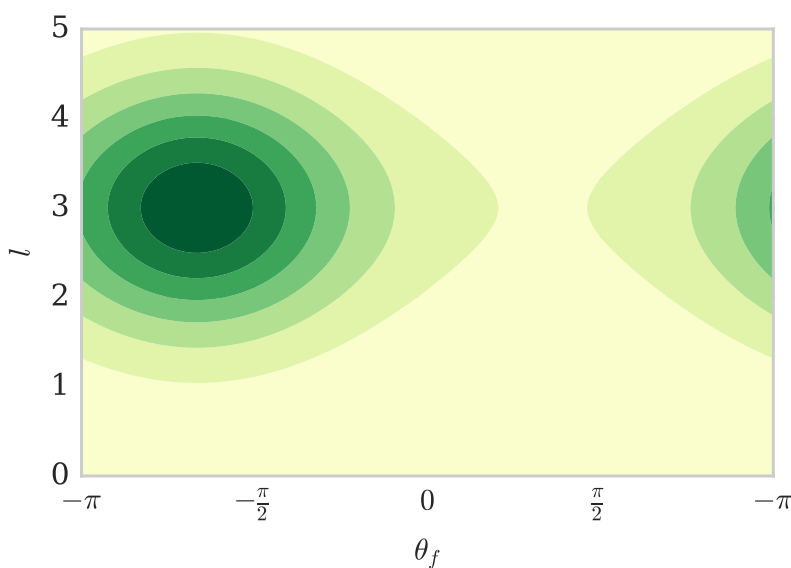


Figure 2.5: 2D length-angle probability distribution with  $\theta_w = 2\pi/3, v_w = 1$

## 2.4 Spatial Distribution

### 2.4.1 Rotated Coordinate System

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

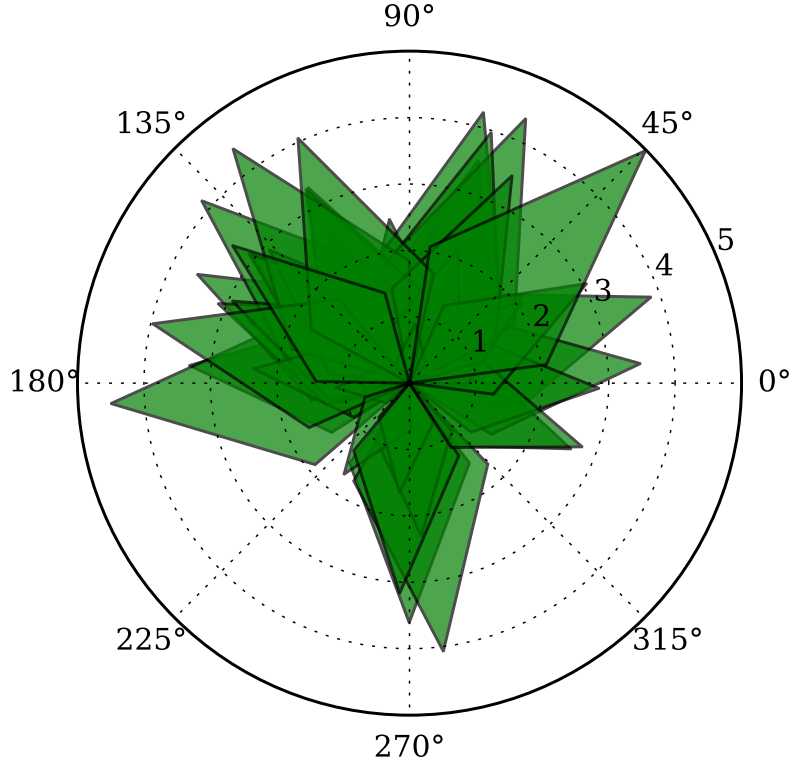


Figure 2.6: A sample of 50 kelp fronds with length and angle picked from the distribution above with  $f_s = 0.5$  and  $f_r = 2$ .

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

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

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

and

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

$$\theta' = \text{atan2}(y, x) \quad (2.21)$$

### 2.4.2 Functional Description of Frond Edge

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

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

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

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

where

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

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

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

### 2.4.3 Absolute Coordinates

To generalize to a frond pointed at an angle  $\theta_f$ , we will use the coordinate system  $(\theta, s)$  such that

$$\theta = \theta' + \theta_f - \frac{\pi}{2} \quad (2.26)$$

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

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

#### 2.4.4 Conditions for Occupancy

Consider a fixed frond of length  $l$  at an angle  $\theta_f$ . The point  $(\theta, s)$  is occupied by the frond if

$$|\theta_f - \theta| < \alpha \quad (2.28)$$

and

$$s < s_f(\theta) \quad (2.29)$$

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

$$\theta - \alpha < \theta_f < \theta + \alpha \quad (2.30)$$

and

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

where

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

Then, considering the point to be fixed, (2.30) and (2.31) define the spacial region  $R_s(\theta, s)$  called the “occupancy region for  $(\theta, s)$ ” 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  $\theta_f$  are shown in various colors. Any frond longer than these at the same angle will also occupy the point.

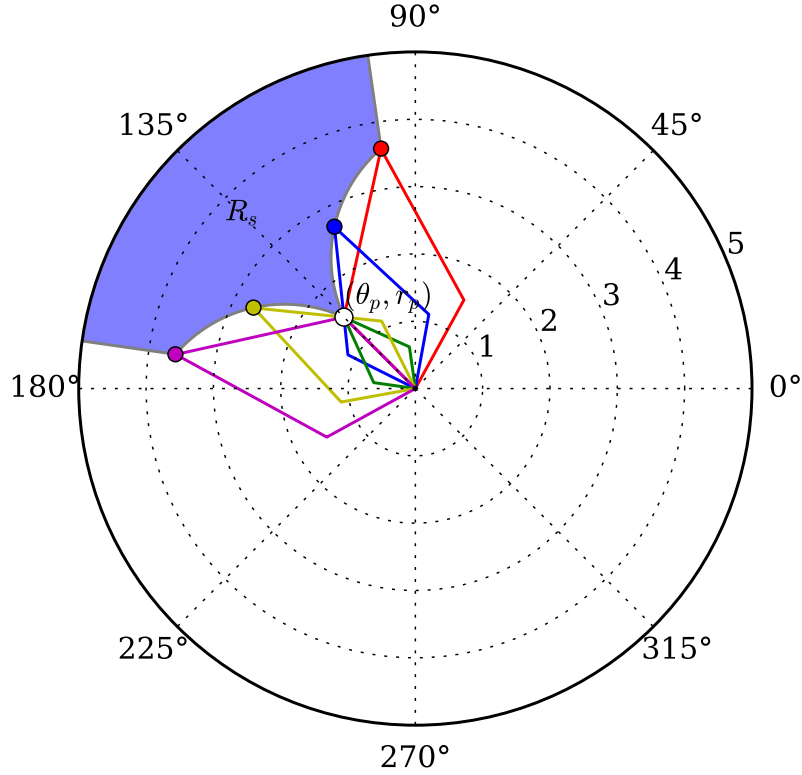


Figure 2.7: Outlines of minimum-length fronds for a variety of angles to occupy the point  $(\theta, s) = (3\pi/4, 3/2)$

#### 2.4.5 Probability of Occupancy

We are interested in the probability that, given a fixed point  $(\theta, s)$ , values of  $l$  and  $\theta_f$  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  $(\theta, s)$ , as depicted in figure 2.8.

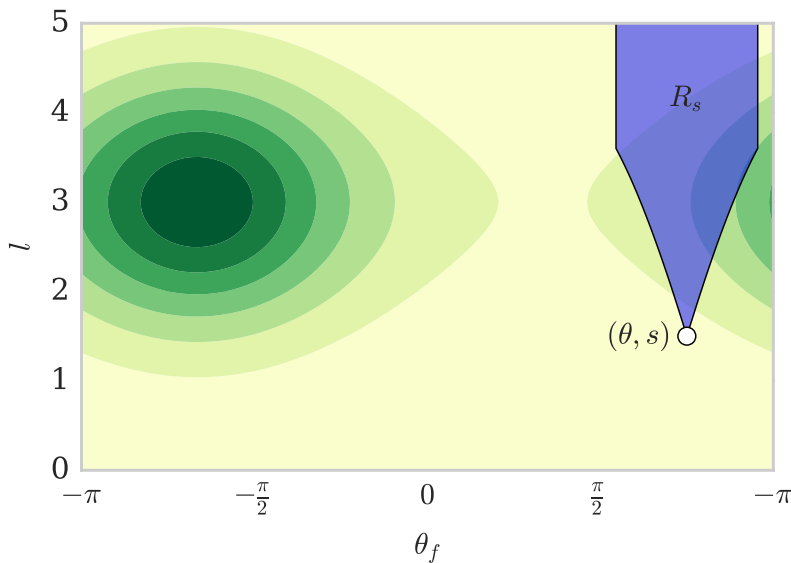


Figure 2.8: Contour plot of  $P_{2D}(\theta_f, l)$  overlaid with the region in the  $\theta_f - l$  plane which results in a frond occupying the point  $(\theta, s) = (3\pi/4, 3/2)$

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

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

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

the proportion of the grid cell occupied by kelp to be

$$P_k = \frac{nt}{dz} \tilde{P}. \quad (2.34)$$

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

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

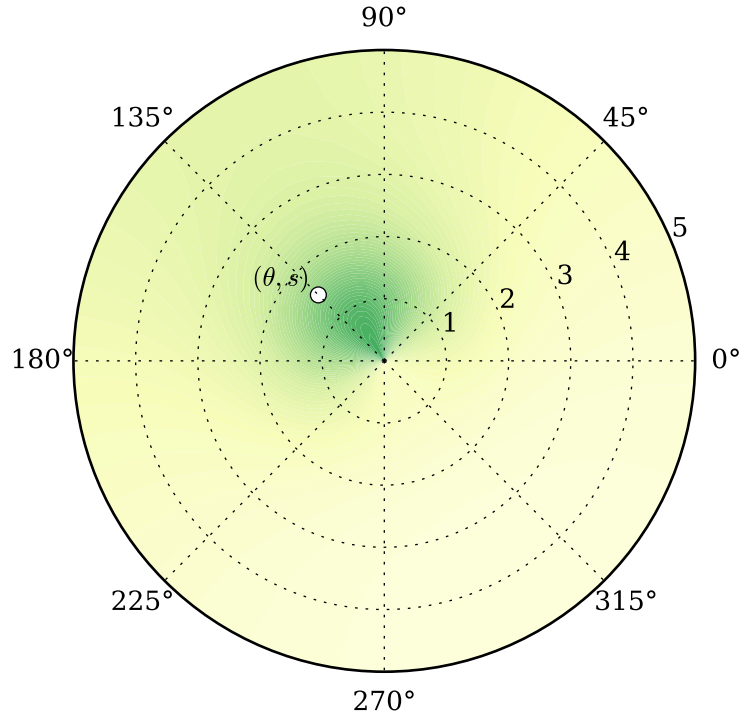


Figure 2.9: Contour plot of the probability of occupying sampled at 121 points using  $\theta_f = 2\pi/3, v_w = 1$



## CHAPTER III

### LIGHT MODEL

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

#### 3.1 Optical Definitions

##### 3.1.1 Radiometric Quantities

One of the most fundamental quantities in optics is radiant flux  $\Phi$ , 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,

$$L = \frac{d^2\Phi}{dA d\omega} \quad (3.1)$$

Once the radiance  $L$  is calculated everywhere, the irradiance is

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

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

#### 4.4.1

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

$$1 \text{ W/m}^2 = 4.2 \text{ } \mu\text{mol photons/s.} \quad (3.3)$$

### 3.1.2 Inherent Optical Properties

We must now define a few inherent optical properties (IOPs) which depend only on the medium of propagation. These phenomena are governed by three inherent optical properties (IOPs) of the medium. The absorption coefficient  $a(\vec{x})$  (units  $\text{m}^{-1}$ ) defines the proportional loss of radiance per unit length. The scattering coefficient  $b$  (units  $\text{m}^{-1}$ ), 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  $\text{sr}^{-1}$ ) defines the probability of light scattering at any given angle from its source. Formally, given two directions  $\vec{\omega}$  and  $\vec{\omega}'$ ,  $\beta(\vec{\omega} \cdot \vec{\omega}')$  is the probability density of light scattering from  $\vec{\omega}$  into  $\vec{\omega}'$  (or vice-versa). Of course, since a single direction subtends no solid angle, the probability of scattering occurring exactly from  $\vec{\omega}$  to  $\vec{\omega}'$  is 0. Rather, we say that the probability of radiance being scattered from a direction  $\omega$  into an element of solid angle  $\Omega$  is  $\int_{\Omega} \beta(\vec{\omega} \cdot \vec{\omega}') d\vec{\omega}'$ .

The VSF is normalized such that

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

so that for any  $\omega$ ,

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

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

## 3.2 The Radiative Transfer Equation

### 3.2.1 Ray Notation

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

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

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

Then,

$$\vec{l}(\vec{x}, \vec{\omega}, s) = \frac{1}{\tilde{s}}(s\vec{x} + (\tilde{s} - s)\vec{x}_0(\vec{x}, \vec{\omega})) \quad (3.7)$$

where

$$\vec{x}_0(\vec{x}, \vec{\omega}) = \vec{x} - \tilde{s}\vec{\omega} \quad (3.8)$$

is the origin of the ray, and

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

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

### 3.2.2 Colloquial Description

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

1. Radiance is decreased due to absorption.
2. Radiance is decreased due to scattering out of the path to other directions.
3. Radiance is increased due to scattering into the path from other directions.

### 3.2.3 Equation of Transfer

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

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

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

Now, we have

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

Then, the general form of the Radiative Transfer Equation is

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

or, equivalently,

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

### 3.2.4 Boundary Conditions

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

$$L((x_{\min}, y, z), \vec{\omega}) = L((x_{\max}, y, z), \vec{\omega}) \quad (3.13)$$

$$L((x, y_{\min}, z), \vec{\omega}) = L((x, y_{\max}, z), \vec{\omega}) \quad (3.14)$$

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

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

$$L(\vec{x}_b, \vec{\omega}) = 0 \text{ if } \vec{\omega} \cdot \hat{z} < 0 \quad (3.16)$$

### 3.3 Low-Scattering Approximation

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

#### 3.3.1 Asymptotic Expansion

Taking  $b$  to be small, we introduce the asymptotic series

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

Then, substituting the above into the RTE,

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

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

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

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

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

$\vdots$

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

Then,

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

which becomes

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

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

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

$\vdots$

### 3.3.2 Analytical Solution

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

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

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

Then,

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

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

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

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

Then,

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

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



## CHAPTER IV

### NUMERICAL SOLUTION

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

#### 4.1 Super-Individuals

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

##### 4.1.1 Frond Length Distribution

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

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

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

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

We then assume that frond lengths are normally distributed in each depth layer with mean  $\mu_k$  and standard deviation  $\sigma_k$ .

## 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  $n_x$ ,  $n_y$ ,  $n_z$ ,  $n_\theta$ , and  $n_\phi$ , with uniform spacings  $dx$ ,  $dy$ ,  $dz$ ,  $d\theta$ , and  $d\phi$  between adjacent grid points.

The following indices are assigned to each dimension:

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

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

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

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

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

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

#### 4.2.1 Spatial Grid

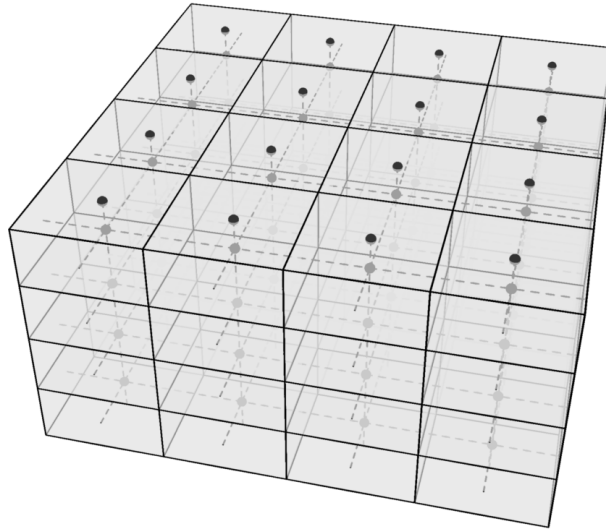


Figure 4.1: Spatial grid

$$dx = \frac{x_{\max} - x_{\min}}{n_x} \quad (4.8)$$

$$dy = \frac{y_{\max} - y_{\min}}{n_y} \quad (4.9)$$

$$dz = \frac{z_{\max} - z_{\min}}{n_z} \quad (4.10)$$

Denote the edges as

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

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

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

and the cell centers as

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

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

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

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

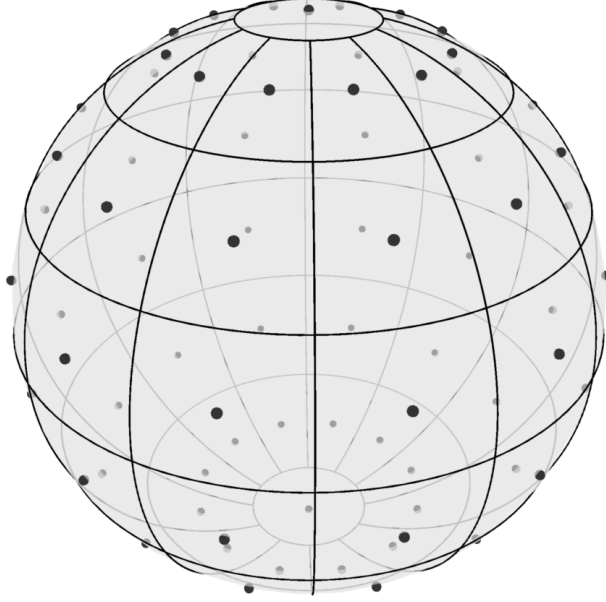


Figure 4.2: Angular grid at each point in space

Now, we define the azimuthal angle such that

$$\theta_l = (l - 1)d\theta. \quad (4.17)$$

For the sake of periodicity, we need

$$\theta_1 = 0, \quad (4.18)$$

$$\theta_{n_\theta} = 2\pi - d\theta, \quad (4.19)$$

which requires

$$d\theta = \frac{2\pi}{n_\theta}. \quad (4.20)$$

For the polar angle, we similarly let

$$\phi_m = (m - 1)d\phi \quad (4.21)$$

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

$$\phi_1 = 0, \quad (4.22)$$

$$\phi_{n_\phi} = \pi. \quad (4.23)$$

This gives us

$$d\phi = \frac{\pi}{n_\phi - 1}. \quad (4.24)$$

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

$$\theta_l^e = (l - 1/2)d\theta, \quad l = 1, \dots, n_\theta \quad (4.25)$$

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

Note that while  $\theta$  has its final edge following its final center, this is not the case for  $\phi$ .

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

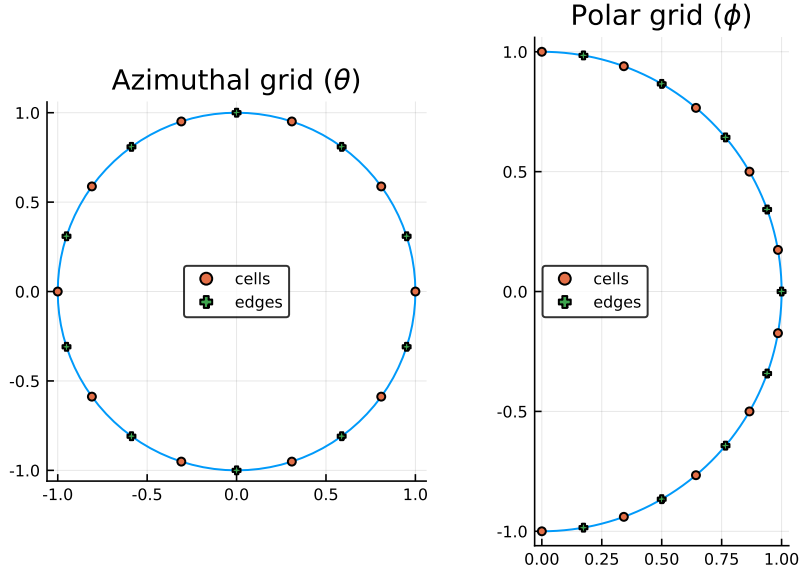


Figure 4.3: Angular grid

poles respectively. The following notation is used.

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

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

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

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

Thus, it follows that

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

Accordingly, define

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

Further, we refer to the angular grid cell centered at  $\vec{\omega}_p$  as  $\Omega_p$ , and the solid angle subtended by  $\Omega_p$  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

$$|\Omega_1| = |\Omega_{n_{\vec{\omega}}}| = \int_{\Omega_1} d\vec{\omega} \quad (4.33)$$

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

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

$$= 2\pi(1 - \cos(d\phi/2)) \quad (4.36)$$

And for all other angular grid cells,

$$|\Omega_p| = \int_{\Omega_p} d\vec{\omega} \quad (4.37)$$

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

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

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

### 4.2.3 Angular Quadrature

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



Define the *angular characteristic function*

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

$$\int_{4\pi} f(\vec{\omega}) d\vec{\omega} = \int_{4\pi} \sum_{p=1}^{n_{\vec{\omega}}} f_p \mathcal{X}_p^\Omega(\vec{\omega}) d\vec{\omega} \quad (4.42)$$

$$= \sum_{p=1}^{n_{\vec{\omega}}} f_p \int_{4\pi} \mathcal{X}_p^\Omega(\vec{\omega}) d\vec{\omega} \quad (4.43)$$

$$= \sum_{p=1}^{n_{\vec{\omega}}} f_p \int_{\Omega_p} d\vec{\omega} \quad (4.44)$$

$$= \sum_{p=1}^{n_{\vec{\omega}}} f_p |\Omega_p| \quad (4.45)$$

#### 4.2.4 Scattering Integral

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

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

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

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

$$\int_{\Omega} \int_{\Omega'} \beta(\vec{\omega} \cdot \vec{\omega}') L(\vec{x}, \vec{\omega}') d\vec{\omega}' d\vec{\omega} = L_{ijkp'} \int_{\Omega} \int_{\Omega_{p'}} \beta(\vec{\omega} \cdot \vec{\omega}') d\vec{\omega}' d\vec{\omega} \quad (4.47)$$

$$= \beta_{pp'} |\Omega| |\Omega'| L_{ijkp'}. \quad (4.48)$$

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

$$f(x+\varepsilon) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x)}{n!} \varepsilon^n \quad (4.49)$$

Truncating after the first few terms, we have

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

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

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

Rearranging (4.50) produces

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

Combining (4.51) with (4.52) gives

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

Then, dividing by  $\varepsilon$  gives

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

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

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

#### 4.3.2 Difference Equation

In general, we have

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

Then,

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

Interior:

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

Surface downwelling (BC):

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

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

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

Likewise for the bottom boundary condition, we have

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

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

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

Grouping  $L_{ijkp}$  terms gives

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

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

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

#### 4.3.3 Structure of Linear System

Describe layout of matrix.

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

Table 4.1: Breakdown of nonzero matrix elements by derivative case

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

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

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

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

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

Let

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

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

$$(4.65)$$

and

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

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

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

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

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

$$u_n(\tilde{s}) = \int_0^{\tilde{s}} g_n(s') \exp \left( - \int_{s''}^{s'} \tilde{a}(s'') ds'' \right) ds' \quad (4.67)$$

$$= \int_0^{s_N} \sum_{i=1}^{N-1} g_{ni} \mathcal{X}_i(s') \exp \left( - \int_{s''}^{s'} \sum_{j=1}^{N-1} \tilde{a}_j \mathcal{X}_j(s'') ds'' \right) ds' \quad (4.68)$$

$$= \sum_{i=1}^{N-1} g_{ni} \int_0^{s_N} \mathcal{X}_i(s') \exp \left( - \sum_{j=1}^{N-1} \tilde{a}_j \int_{s''}^{s'} \mathcal{X}_j(s'') ds'' \right) ds' \quad (4.69)$$

$$= \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp \left( - \tilde{a}_{\hat{i}(s')} \tilde{d}(s') - \sum_{j=\hat{i}(s')}^{N-1} \tilde{a}_j ds_j \right) ds' \quad (4.70)$$

$$= \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp \left( - \tilde{a}_i (s_{i+1} - s') - \sum_{j=i+1}^{N-1} \tilde{a}_j ds_j \right) ds' \quad (4.71)$$

Let

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

Then,

$$u_n(\tilde{s}) = \sum_{i=1}^{N-1} g_{ni} \int_{s_i}^{s_{i+1}} \exp (\tilde{a}_i s' + b_i) ds' \quad (4.73)$$

$$= \sum_{i=1}^{N-1} g_{ni} e^{b_i} \int_{s_i}^{s_{i+1}} \exp (\tilde{a}_i s') ds' \quad (4.74)$$



Let

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

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

Then,

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

#### 4.4.1 Perceived Irradiance

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

$$\tilde{I}_k = \frac{\sum_{ij} P_{ijk} I_{ijk}}{\sum_{ij} P_{ijk}} \quad (4.78)$$

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

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

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

## CHAPTER V

### PARAMETER VALUES

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

#### 5.1 Parameters from Literature

\* More to come

#### 5.2 Frond Distribution Parameters

##### 5.2.1 Rotation

##### 5.2.2 Lift

Parameter Name		Symbol	Value(s)	Citation	Notes
Kelp	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	[12]	Table 2, $b_{\lambda 0}$ , mean
VSF		$\beta$	tabulated	[10, 12],	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 <sup>-2</sup>	[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 [10]. I'll pull a few cases from here and point out when the asymptotic approximation will work.

## CHAPTER VI

### MODEL ANALYSIS

#### 6.1 Grid Study

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

#### 6.2 Asymptotic Convergence

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

#### 6.3 Sensitivity Analysis

Vary parameters and measure average differences in radiance for full grid, as well as average 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. Radiative Transfer in the Ocean. In *Encyclopedia of Ocean Sciences*, pages 2321–2330. Elsevier, 2001.
- [9] M. Nyman, M. Brown, M. Neushul, and J. A. Keogh. *Macrocystis pyrifera in New Zealand: testing two mathematical models for whole plant growth*, volume 2. Sept. 1990.
- [10] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.
- [11] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Ecological Modelling*, 80:161–170, Mar. 1994.
- [12] A. Sokolov, M. Chami, E. Dmitriev, and G. Khomenko. Parameterization of volume scattering function of coastal waters based on the statistical approach. *Optics express*, 18(5):4615–4636, 2010.
- [13] P. Wassmann, D. Slagstad, C. W. Riser, and M. Reigstad. Modelling the ecosystem dynamics of the Barents Sea including the marginal ice zone. *Journal of Marine Systems*, 59(1-2):1–24, Jan. 2006.

- [14] A. Yoshimori, T. Kono, and H. Iizumi. Mathematical models of population dynamics of the kelp *Laminaria religiosa*, with emphasis on temperature dependence. *Fisheries Oceanography*, 7(2):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  $\vec{p}_1 = (p_{1x}, p_{1y}, p_{1z})$  and a corresponding path  $\vec{l}(\vec{x}_1, \vec{\omega}, s)$ .

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

Given

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

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

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

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,  $\sigma$  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  $\vec{p}_0$ .

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 <
169      !   xx(i+1))
170      integer ii
171      ! Slope of liine between (xx(ii),yy(ii)) and
172      !   (xx(ii+1),yy(ii+1))
173      double precision mm
174
175      ! If out of bounds, then return endpoint
176      !   value
177      if (x0 < xx(1)) then
178          interp = yy(1)
179      else if (x0 > xx(nn)) then
180          interp = yy(nn)
181      else
182          ! Determine ii
183          do ii = 1, nn
184              if (xx(ii) > x0) then
185                  ! We've now gone one index too far
186                  exit
187              end if
188          end do
189
190          ! Determine whether we're on the right
191          !   endpoint
192          if(ii-1 < nn) then
193              ! If this is a legitimate
194              !   interpolation, then
195              ! subtract since we went one index too
196              !   far
197              ii = ii - 1
198
199              ! Calculate slope
200              mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
201              !   xx(ii))
202
203              ! Return interpolated value
204              interp = yy(ii) + mm * (x0 - xx(ii))
205          else
206              ! If we're actually interpolating the
207              !   right endpoint,
208              ! then just return it.
209              interp = yy(nn)
210          end if
211      end if
212
213 end function

```

```

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

```

```

215         p = angles%phat(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_frond_edge(theta_prime, L,
    | !     fs, fr)
182 | ! ! r_f'(\theta')
183 | !     double precision, intent(in) :: theta_prime,
    | !     L, fs, fr
184 | !     double precision, intent(out) ::
    | !     relative_frond_edge
185 | !
186 | !     relative_frond_edge = L / (sin(theta_prime)
    | !     + angular_sign(theta_prime * alpha(fs, fr) *
    | !     cos(theta_prime)))
187 | ! end function relative_frond_edge
188 |
189 | function angular_sign(theta_prime)
190 | ! S(\theta')
191 |     double precision, intent(in) :: theta_prime
192 |     double precision angular_sign
193 |
194 |     ! This seems to be incorrect in summary.pdf as
    |     ! of 9/9/18
195 |     ! In the report, it's written as sgn(
    |     ! theta_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 i, j, k, p

```

```

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

```

```

64 | end type rte_mat
65 |
66 | interface
67 |     ! Define interface for external procedure
68 |     ! https://stackoverflow.com/questions/8549415/how-to-declare-the-interface-section-for-a-procedure-argument-which-in-turn-ref
69 |     subroutine solver_interface(n_total, nonzero,
70 |         row, col, data, &
71 |         sol, rhs, maxiter_outer, maxiter_inner,
72 |         &
73 |         tol_abs, tol_rel)
74 |         integer :: n_total, nonzero
75 |         integer, dimension(nonzero) :: row, col
76 |         double precision, dimension(nonzero) ::
77 |             data
78 |         double precision, dimension(nonzero) :: sol
79 |         double precision, dimension(n_total) :: rhs
80 |         integer :: maxiter_outer, maxiter_inner
81 |         double precision :: tol_abs, tol_rel
82 |     end subroutine solver_interface
83 | end interface
84 |
85 | contains
86 |
87 |     subroutine mat_init(mat, grid, iops)
88 |         class(rte_mat) mat
89 |         type(space_angle_grid) grid
90 |         type(optical_properties) iops
91 |         integer nnz, n_total
92 |
93 |         mat%grid = grid
94 |         mat%iops = iops
95 |
96 |         call mat%calculate_size()
97 |
98 |         n_total = mat%n_total
99 |         nnz = mat%nonzero
100 |         allocate(mat%surface_vals(grid%angles%nomega
101 |             ))
102 |         allocate(mat%row(nnz))
103 |         allocate(mat%col(nnz))
104 |         allocate(mat%data(nnz))
105 |         allocate(mat%rhs(n_total))
106 |         allocate(mat%sol(n_total))
107 |
108 |         call zeros(mat%rhs, n_total)
109 |         call zeros(mat%sol, n_total)
110 |
111 |     end subroutine mat_init

```

```

108
109 subroutine mat_deinit(mat)
110     class(rte_mat) mat
111     deallocate(mat%row)
112     deallocate(mat%col)
113     deallocate(mat%data)
114     deallocate(mat%rhs)
115     deallocate(mat%sol)
116     deallocate(mat%surface_vals)
117 end subroutine mat_deinit
118
119 subroutine calculate_size(mat)
120     class(rte_mat) mat
121     integer nx, ny, nz, nomega
122
123     nx = mat%grid%x%num
124     ny = mat%grid%y%num
125     nz = mat%grid%z%num
126     nomega = mat%grid%angles%nomega
127
128     !mat%nonzero = nx * ny * ntheta * nphi * ( (
129         nz-1) * (6 + ntheta * nphi) + 1)
130     mat%nonzero = nx * ny * nomega * (nz * (
131         nomega + 6) - 1)
132     mat%n_total = nx * ny * nz * nomega
133
134     !mat%theta_block_size = 1
135     !mat%phi_block_size = mat%theta_block_size *
136         ntheta
137     mat%omega_block_size = 1
138     mat%y_block_size = mat%omega_block_size *
139         nomega
140     mat%x_block_size = mat%y_block_size * ny
141     mat%z_block_size = mat%x_block_size * nx
142
143 end subroutine calculate_size
144
145 ! subroutine mat_to_hdf(mat,filename)
146 !     class(rte_mat) mat
147 !     character(len=*) filename
148 !     call write_coo(filename, mat%row, mat%col,
149         mat%data, mat%nonzero)
150 ! end subroutine mat_to_hdf
151
152 subroutine mat_set_bc(mat, bc)
153     class(rte_mat) mat
154     class(boundary_condition) bc
155     integer p
156
157     do p=1, mat%grid%angles%nomega/2

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

422
423     call mat%assign(row_num,ent,-val,i,j-1,k,p)
424     call mat%assign(row_num,ent,val,i,1,k,p)
425 end subroutine y_cd2_last
426
427 subroutine z_cd2(mat, indices, row_num, ent)
428     class(rte_mat) mat
429     double precision val, dz
430     type(index_list) indices
431     integer i, j, k, p
432     integer row_num, ent
433
434     i = indices%i
435     j = indices%j
436     k = indices%k
437     p = indices%p
438
439     dz = mat%grid%z%spacing(indices%k)
440
441     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
442         dz)
443
444     call mat%assign(row_num,ent,-val,i,j,k-1,p)
445     call mat%assign(row_num,ent,val,i,j,k+1,p)
446 end subroutine z_cd2
447
448 subroutine z_fd2(mat, indices, row_num, ent,
449     repeat_ent)
450     ! Has to be called after angular_integral
451     ! Because they both write to the same matrix
452     ! entry
453     ! And adding here is more efficient than a
454     ! conditional
455     ! in the angular loop.
456     class(rte_mat) mat
457     double precision val, val1, val2, val3, dz
458     type(index_list) indices
459     integer i, j, k, p
460     integer row_num, ent, repeat_ent
461
462     i = indices%i
463     j = indices%j
464     k = indices%k
465     p = indices%p
466
467     dz = mat%grid%z%spacing(indices%k)
468
469     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
470         dz)
471
472     val1 = -3.d0 * val

```

```

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

```

```

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

```

```

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

```

rte3d.f90



```

1 module rte3d
2 use kelp_context
3 use rte_sparse_matrices
4 use light_context
5 implicit none
6
7 interface
8     subroutine deriv_interface(mat, indices,
9         row_num, ent)
10        use rte_sparse_matrices
11        class(rte_mat) mat
12        type(index_list) indices
13        integer row_num, ent
14    end subroutine deriv_interface
15    subroutine angle_loop_interface(mat, indices,
16        ddx, ddy)
17        use rte_sparse_matrices
18        import deriv_interface
19        type(space_angle_grid) grid
20        type(rte_mat) mat
21        type(index_list) indices
22        procedure(deriv_interface) :: ddx, ddy
23    end subroutine angle_loop_interface
24 end interface
25
26 contains
27
28 subroutine whole_space_loop(mat, indices)
29     type(rte_mat) mat
30     type(index_list) indices
31     integer i, j, k
32
33     procedure(deriv_interface), pointer :: ddx,
34         ddy
35     procedure(angle_loop_interface), pointer ::
36         angle_loop
37
38     !$ integer omp_get_num_procs
39     !$ integer num_threads_z, num_threads_x,
40         num_threads_y
41
42     ! Enable nested parallelism
43     !$ call omp_set_nested(.true.)
44
45     ! Use nz procs for outer loop,
46     ! or num_procs if num_procs < nz
47     ! Divide the rest of the tasks as appropriate
48
49     !$ num_threads_z = min(omp_get_num_procs(),
50         mat%grid%z%num)
51     !$ num_threads_x = min( &

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

kelp\_context.f90

```

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

```

```

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

```



```

68 | end type boundary_condition
69 |
70 | contains
71 |
72 |     function bc_gaussian(bc, theta, phi)
73 |         class(boundary_condition) bc
74 |         double precision theta, phi, diff
75 |         double precision bc_gaussian
76 |         diff = angle_diff_3d(theta, phi, bc%theta_s,
77 |             bc%phi_s)
78 |         bc_gaussian = exp(-bc%decay * diff)
79 |     end function bc_gaussian
80 |
81 |     subroutine bc_init(bc, grid, theta_s, phi_s,
82 |         decay, I0)
83 |         class(boundary_condition) bc
84 |         type(space_angle_grid) grid
85 |         double precision theta_s, phi_s, decay, I0
86 |         integer p
87 |         double precision theta, phi
88 |
89 |         allocate(bc%bc_grid(grid%angles%nomega))
90 |
91 |         bc%theta_s = theta_s
92 |         bc%phi_s = phi_s
93 |         bc%decay = decay
94 |         bc%I0 = I0
95 |
96 |         ! Only set BC for downwelling light
97 |         do p=1, grid%angles%nomega/2
98 |             theta = grid%angles%theta_p(p)
99 |             phi = grid%angles%phi_p(p)
100 |             bc%bc_grid(p) = bc%bc_gaussian(theta, phi
101 |                 )
102 |         end do
103 |         ! Zero upwelling light specified at surface
104 |         do p=grid%angles%nomega/2+1, grid%angles%
105 |             nomega
106 |             bc%bc_grid(p) = 0.d0
107 |         end do
108 |
109 |         ! Normalize
110 |         bc%bc_grid = bc%I0 * bc%bc_grid &
111 |             / grid%angles%integrate_points(bc%
112 |                 bc_grid)
113 |     end subroutine bc_init
114 |
115 |     subroutine bc_deinit(bc)
116 |         class(boundary_condition) bc
117 |         deallocate(bc%bc_grid)

```

```

114     end subroutine
115
116     subroutine point_set_cart(point, x, y, z)
117         class(point3d) :: point
118         double precision x, y, z
119         point%x = x
120         point%y = y
121         point%z = z
122         call point%cartesian_to_polar()
123     end subroutine point_set_cart
124
125     subroutine point_set_cyl(point, theta, r, z)
126         class(point3d) :: point
127         double precision theta, r, z
128         point%theta = theta
129         point%r = r
130         point%z = z
131         call point%polar_to_cartesian()
132     end subroutine point_set_cyl
133
134     subroutine polar_to_cartesian(point)
135         class(point3d) :: point
136         point%x = point%r*cos(point%theta)
137         point%y = point%r*sin(point%theta)
138     end subroutine polar_to_cartesian
139
140     subroutine cartesian_to_polar(point)
141         class(point3d) :: point
142         point%r = sqrt(point%x**2 + point%y**2)
143         point%theta = atan2(point%y, point%x)
144     end subroutine cartesian_to_polar
145
146     subroutine frond_set_shape(frond, fs, fr, ft)
147         class(frond_shape) frond
148         double precision fs, fr, ft
149         frond%fs = fs
150         frond%fr = fr
151         frond%ft = ft
152         call frond%calculate_angles()
153     end subroutine frond_set_shape
154
155     subroutine frond_calculate_angles(frond)
156         class(frond_shape) frond
157         frond%tan_alpha = 2.d0*frond%fs*frond%fr /
158             (1.d0 + frond%fs)
159         frond%alpha = atan(frond%tan_alpha)
160     end subroutine
161
162     subroutine iop_init(iops, grid)
163         class(optical_properties) iops

```

```

163     type(space_angle_grid) grid
164
165     iops%grid = grid
166
167     ! Assume that these are preallocated and
168     ! passed to function
169     ! Nevermind, don't assume this.
170     allocate(iops%abs_water(grid%z%num))
171
172     ! Assume that these must be allocated here
173     allocate(iops%vsf_angles(iops%num_vsf))
174     allocate(iops%vsf_vals(iops%num_vsf))
175     allocate(iops%vsf_cos(iops%num_vsf))
176     allocate(iops%vsf(grid%angles%nomega,grid%
177         angles%nomega))
178     allocate(iops%vsf_integral(grid%angles%
179         nomega,grid%angles%nomega))
180     allocate(iops%abs_grid(grid%x%num, grid%y%
181         num, grid%z%num))
182 end subroutine iop_init
183
184 subroutine calculate_coef_grids(iops, p_kelp)
185     class(optical_properties) iops
186     double precision, dimension(:,:,:) :: p_kelp
187
188     integer k
189
190     ! Allow water IOPs to vary over depth
191     do k=1, iops%grid%z%num
192         iops%abs_grid(:,:,k) = (iops%abs_kelp -
193             iops%abs_water(k)) * p_kelp(:,:,k) +
194             iops%abs_water(k)
195     end do
196 end subroutine calculate_coef_grids
197
198 subroutine load_vsf(iops, filename, fmtstr)
199     class(optical_properties) :: iops
200     character(len=*) :: filename, fmtstr
201     double precision, dimension(:,:),
202         allocatable :: tmp_2d_arr
203     integer num_rows, num_cols, skiplines_in
204
205     ! First column is the angle at which the
206     ! measurement is taken
207     ! Second column is the value of the VSF at
208     ! that angle
209     num_rows = iops%num_vsf
210     num_cols = 2

```

```

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

```

```

249
250     subroutine set_depth(depth, rope, grid,
251         depth_layer)
252     class(depth_state) depth
253     type(rope_state) rope
254     type(space_angle_grid) grid
255     integer depth_layer
256
257     depth%frond_length = rope%frond_lengths(
258         depth_layer)
259     depth%frond_std = rope%frond_stds(
260         depth_layer)
261     depth%water_speeds = rope%water_speeds(
262         depth_layer)
263     depth%water_angles = rope%water_angles(
264         depth_layer)
265     depth%num_fronds = rope%num_fronds(
266         depth_layer)
267     depth%depth_layer = depth_layer
268     depth%depth = grid%z%vals(depth_layer)
269 end subroutine set_depth
270
271 function length_distribution_cdf(depth, L)
272     result(output)
273     ! C_L(L)
274     class(depth_state) depth
275     double precision L, L_mean, L_std
276     double precision output
277
278     L_mean = depth%frond_length
279     L_std = depth%frond_std
280
281     call normal_cdf(L, L_mean, L_std, output)
282 end function length_distribution_cdf
283
284 function angle_distribution_pdf(depth, theta_f
285     ) result(output)
286     ! P_{\theta_f}(\theta_f)
287     class(depth_state) depth
288     double precision theta_f, v_w, theta_w
289     double precision output
290     double precision diff
291
292     v_w = depth%water_speeds
293     theta_w = depth%water_angles
294
295     ! von_mises_pdf is only defined on [-pi, pi]
296     ! So take difference of angles and input
297     into
298     ! von_mises dist. centered & x=0.

```

```

290
291     diff = angle_diff_2d(theta_f, theta_w)
292
293     call von_mises_pdf(diff, 0.d0, v_w, output)
294 end function angle_distribution_pdf
295
296 function angle_mod(theta) result(mod_theta)
297     ! Shift theta to the interval [-pi, pi]
298     ! which is where von_mises_pdf is defined.
299
300     double precision theta, mod_theta
301
302     mod_theta = mod(theta + pi, 2.d0*pi) - pi
303 end function angle_mod
304
305 function angle_diff_2d(theta1, theta2) result(
306     diff)
307     ! Shortest difference between two angles
308     ! which may be
309     ! in different periods.
310     double precision theta1, theta2, diff
311     double precision modt1, modt2
312
313     ! Shift to [0, 2*pi]
314     modt1 = mod(theta1, 2*pi)
315     modt2 = mod(theta2, 2*pi)
316
317     ! https://gamedev.stackexchange.com/
318     ! questions/4467/comparing-angles-and-
319     ! working-out-the-difference
320
321     diff = pi - abs(abs(modt1-modt2) - pi)
322 end function angle_diff_2d
323
324 function angle_diff_3d(theta, phi, theta_prime
325     , phi_prime) result(diff)
326     ! Angle between two vectors in spherical
327     ! coordinates
328     double precision theta, phi, theta_prime,
329     phi_prime
330     double precision alpha, diff
331
332     ! Faster, but produces lots of NaNs (at
333     ! least in Python)
334     ! alpha = sin(theta)*sin(theta_prime)*cos(
335     ! theta-theta_prime) + cos(phi)*cos(
336     ! phi_prime)
337
338     ! Slower, but more accurate
339     alpha = (sin(phi)*sin(phi_prime) &

```

```

331         * (cos(theta)*cos(theta_prime) + sin(theta
332           )*sin(theta_prime)) &
333         + cos(phi)*cos(phi_prime))
334     ! Avoid out-of-bounds errors due to rounding
335     alpha = min(1.d0, alpha)
336     alpha = max(-1.d0, alpha)
337
338     diff = acos(alpha)
339 end function angle_diff_3d
340
341 subroutine vsf_from_function(iops, func)
342     class(optical_properties) iops
343     double precision, external :: func
344     integer i
345     type(angle_dim) :: angle1d
346
347     call angle1d%set_bounds(-1.d0, 1.d0)
348     call angle1d%set_num(iops%num_vsf)
349     call angle1d%assign_legendre()
350
351     iops%vsf_angles(:) = acos(angle1d%vals(:))
352     do i=1, iops%num_vsf
353         iops%vsf_vals(i) = func(iops%vsf_angles(i
354           ))
355     end do
356
357     call iops%calc_vsf_on_grid()
358
359     call angle1d%deinit()
360 end subroutine vsf_from_function
361
362 subroutine calc_vsf_on_grid(iops)
363     class(optical_properties) iops
364     double precision th, ph, thp, php
365     integer p, pp
366     integer nomega
367     double precision norm
368
369     nomega = iops%grid%angles%nomega
370
371     ! Calculate cos VSF
372     iops%vsf_cos = cos(iops%vsf_angles)
373
374     ! Normalize cos VSF to 1/(2pi) on [-1, 1]
375     iops%vsf_scatter_coef = abs(trap_rule_uneven(
376         iops%vsf_cos, iops%vsf_vals, iops%num_vsf
377         ))
378     iops%vsf_vals(:) = iops%vsf_vals(:) / (2*pi
379         * iops%vsf_scatter_coef)

```

```

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

```



```

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

```

light\_context.f90

```

1 | module light_context
2 |   use sag
3 |   use rte_sparse_matrices
4 |   !use hdf5
5 |   implicit none
6 |
7 |   type light_state
8 |     double precision, dimension(:,:,:),
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 |   ! Init for use with mat
25 |   subroutine light_init(light, mat)
26 |     class(light_state) light
27 |     type(rte_mat) mat
28 |     integer nx, ny, nz, nomega
29 |
30 |     light%mat = mat
31 |     light%grid = mat%grid
32 |
33 |     nx = light%grid%x%num
34 |     ny = light%grid%y%num
35 |     nz = light%grid%z%num
36 |     nomega = light%grid%angles%nomega
37 |
38 |     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)
79                     index = index + 1
80                 end do
81             end do
82         end do
83     end do
84
85     !call light%mat%initial_guess()
86
87     ! Solve (MGMRES)

```

```

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

```

```

134 | !     call hdf_write_radiance(radfile, light%
      | radiance, light%grid)
135 | !     call hdf_write_irradiance(irradfile, light%
      | irradiance, light%grid)
136 | ! end subroutine light_to_hdf
137 |
138 | subroutine light_deinit(light)
139 |     class(light_state) light
140 |
141 |     deallocate(light%irradiance)
142 |     deallocate(light%radiance)
143 | end subroutine light_deinit
144 | 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(:,:,:,:) ::
      | radiance
14 |         double precision, dimension(:,:,:,:),
      | allocatable :: source
15 |         integer num_scatters
16 |         integer nx, ny, nz, nomega
17 |         integer max_cells
18 |
19 |         double precision, dimension(:), allocatable
      | :: path_length, path_spacing, a_tilde, gn
20 |
21 |         nx = grid%x%num
22 |         ny = grid%y%num
23 |         nz = grid%z%num
24 |         nomega = grid%angles%nomega
25 |
26 |         max_cells = calculate_max_cells(grid)
27 |
28 |         allocate(path_length(max_cells+1))
29 |         allocate(path_spacing(max_cells))
30 |         allocate(a_tilde(max_cells))
31 |         allocate(gn(max_cells))
32 |         allocate(source(nx, ny, nz, nomega))

```

```

32
33     call calculate_light_before_scattering(grid,
34         bc, iops, source, radiance, path_length,
35         path_spacing, a_tilde, gn)
36
37     if (num_scatters .gt. 0) then
38         call calculate_light_after_scattering(&
39             grid, iops, source, radiance, &
40             num_scatters, path_length,
41             path_spacing, &
42             a_tilde, gn)
43     end if
44
45     deallocate(path_length)
46     deallocate(path_spacing)
47     deallocate(a_tilde)
48     deallocate(gn)
49     deallocate(source)
50 end subroutine calculate_light_with_scattering
51
52 subroutine calculate_light_before_scattering(
53     grid, bc, iops, source, radiance,
54     path_length, path_spacing, a_tilde, gn)
55     type(space_angle_grid) grid
56     type(boundary_condition) bc
57     type(optical_properties) iops
58     double precision, dimension(:,:,:,) ::
59         radiance, source
60     double precision, dimension(:) ::
61         path_length, path_spacing, a_tilde, gn
62     integer i, j, k, p
63
64     ! !$ integer omp_get_num_procs
65     ! !$ integer num_threads_z, num_threads_x
66
67     ! ! Enable nested parallelism
68     ! !$ call omp_set_nested(.true.)
69
70     ! ! Use nz procs for outer loop,
71     ! ! or num_procs if num_procs < nz
72     ! ! Divide the rest of the tasks as
73         appropriate
74
75     ! !$ num_threads_z = min(omp_get_num_procs()
76         , grid%z%num)
77     ! !$ num_threads_x = min( &
78         omp_get_num_procs()/num_threads_z, &
79         grid%x%num)
80
81     ! !$omp parallel do default(none) private(i,
82         j,k,p) &

```

```

73 | ! !$omp shared(grid,iops,radiance,bc,
74 |   num_threads_x) &
75 | ! !$omp private(source,path_length,
76 |   path_spacing,a_tilde,gn) &
77 | ! !$omp num_threads(num_threads_z) if(
78 |   num_threads_z .gt. 1)
79 | do k=1, grid%z%num
80 |   ! !$omp parallel do default(none) private
81 |     (i,j,p) &
82 |     ! !$omp firstprivate(k) shared(grid,iops,
83 |       radiance,bc) &
84 |     ! !$omp private(source,path_length,
85 |       path_spacing,a_tilde,gn) &
86 |     ! !$omp num_threads(num_threads_x) if(
87 |       num_threads_x .gt. 1)
88 |     do i=1, grid%x%num
89 |       do j=1, grid%y%num
90 |         do p=1, grid%angles%nomega/2
91 |           ! Downwelling light
92 |           call
93 |             attenuate_light_from_surface
94 |             (&
95 |               grid, iops, source, i, j, k,
96 |               p,&
97 |               radiance, path_length,
98 |               path_spacing,&
99 |               a_tilde, gn, bc)
100 |           ! No upwelling light before
101 |             scattering
102 |           radiance(i,j,k,p+grid%angles%
103 |             nomega/2) = 0.d0
104 |         end do
105 |       end do
106 |     end do
107 |   ! !$omp end parallel do
108 | end do
109 | ! !$omp end parallel do
110 | end subroutine
111 | calculate_light_before_scattering
112 |
113 | subroutine attenuate_light_from_surface(&
114 |   grid, iops, source, i, j, k, p, radiance,
115 |   &
116 |   path_length, path_spacing, a_tilde, gn,
117 |   bc)
118 |   type(space_angle_grid) grid
119 |   type(boundary_condition) bc
120 |   type(optical_properties) iops

```

```

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

```

```

144         rad_scatter, path_length, path_spacing
145         , a_tilde, gn)
146     radiance = radiance + bb*n * rad_scatter
147 end do
148 deallocate(rad_scatter)
149 end subroutine
150 calculate_light_after_scattering
151 ! Perform one scattering event
152 subroutine scatter(grid, iops, source,
153     rad_scatter, path_length, path_spacing,
154     a_tilde, gn)
155     type(space_angle_grid) grid
156     type(optical_properties) iops
157     double precision, dimension(:,:,:,:) ::
158         rad_scatter, source
159     double precision, dimension(:,:,:,:),
160         allocatable :: scatter_integral
161     double precision, dimension(:) ::
162         path_length, path_spacing, a_tilde, gn
163     integer nx, ny, nz, nomega
164
165     nx = grid%x%num
166     ny = grid%y%num
167     nz = grid%z%num
168     nomega = grid%angles%nomega
169
170     allocate(scatter_integral(nx, ny, nz, nomega
171         ))
172
173     call calculate_source(grid, iops,
174         rad_scatter, source, scatter_integral)
175     call advect_light(grid, iops, source,
176         rad_scatter, path_length, path_spacing,
177         a_tilde, gn)
178
179     deallocate(scatter_integral)
180 end subroutine scatter
181 ! Calculate source from no-scatter or previous
182     scattering layer
183 subroutine calculate_source(grid, iops,
184     rad_scatter, source, scatter_integral)
185     type(space_angle_grid) grid
186     type(optical_properties) iops
187     double precision, dimension(:,:,:,:) ::
188         rad_scatter
189     double precision, dimension(:,:,:,:) ::
190         source

```



```

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

```

```

217         indices%j = j
218         do p=1, nomega
219             indices%p = p
220             call calculate_scatter_integral
221                 (&
222                     iops, rad_scatter,&
223                     scatter_integral,&
224                     indices)
225         end do
226     end do
227     !$omp end parallel do
228 end do
229 !$omp end parallel do
230
231 source(:,:,:) = -rad_scatter(:,:,:) +
232     scatter_integral(:,:,:)
233
234 write(*,*) 'source: ', sum(source)/size(
235     source), minval(source), maxval(source)
236
237 end subroutine calculate_source
238
239 subroutine calculate_scatter_integral(iops,
240     rad_scatter, scatter_integral, indices)
241     type(optical_properties) iops
242     double precision, dimension(:,:,:) ::
243         rad_scatter, scatter_integral
244     type(index_list) indices
245
246     scatter_integral(indices%i,indices%j,indices
247         %k,indices%p) &
248         = sum(iops%vsf_integral(indices%p, :) &
249             * rad_scatter(&
250                 indices%i,&
251                 indices%j,&
252                 indices%k,:))
253
254 end subroutine calculate_scatter_integral
255
256 subroutine advect_light(grid, iops, source,
257     rad_scatter, path_length, path_spacing,
258     a_tilde, gn)
259     type(space_angle_grid) grid
260     type(optical_properties) iops
261     double precision, dimension(:,:,:) ::
262         rad_scatter, source
263     double precision, dimension(:) ::
264         path_length, path_spacing, a_tilde, gn
265     integer i, j, k, p
266

```

```

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

```

```

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

```

```

332         di = ds(i)
333     else
334         di = (exp(a_tilde(i)*s(i+1))-exp(
335             a_tilde(i)*s(i)))/a_tilde(i)
336     end if
337     integral = integral + gn(i)*di * exp(bi)
338 end do
339
340 end function calculate_ray_integral
341
342 ! Calculate maximum number of cells a path
343 ! through the grid could take
344 ! This is a loose upper bound
345 function calculate_max_cells(grid) result(
346     max_cells)
347     type(space_angle_grid) :: grid
348     integer :: max_cells
349     double precision dx, dy, zrange, phi_middle
350
351     ! Angle that will have the longest ray
352     phi_middle = grid%angles%phi(grid%angles%
353         nphi/2)
354     dx = grid%x%spacing(1)
355     dy = grid%y%spacing(1)
356     zrange = grid%z%maxval - grid%z%minval
357
358     max_cells = grid%z%num + ceiling((1/dx+1/dy)
359         *zrange*tan(phi_middle))
360 end function calculate_max_cells
361
362 ! Traverse from surface or bottom to point (xi
363 ! , yj, zk)
364 ! in the direction omega_p, extracting path
365 ! lengths (ds) and
366 ! function values (f) along the way,
367 ! as well as number of cells traversed (n).
368 subroutine traverse_ray(grid, iops, source, i,
369     j, k, p, s_array, ds, a_tilde, gn,
370     num_cells)
371     type(space_angle_grid) :: grid
372     type(optical_properties) :: iops
373     double precision, dimension(:,:,:) ::
374         source
375     integer :: i, j, k, p
376     double precision, dimension(:) :: s_array,
377         ds, a_tilde, gn
378     integer :: num_cells
379
380     integer t
381     double precision p0x, p0y, p0z

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

light\_interface.f90

```

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

```

```

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

```

```

55
56      ! SUNLIGHT
57      double precision, intent(in) :: solar_zenith
58      double precision, intent(in) ::
59          solar_azimuthal
60
61      ! KELP
62      ! Number of Superindividuals in each depth
63          level
64      integer, intent(in) :: num_si
65      ! si_area(i,j) = area of superindividual j
66          at depth i
67      double precision, dimension(nz, num_si),
68          intent(in) :: si_area
69      ! si_ind(i,j) = number of inidividuals
70          represented by superindividual j at depth
71          i
72      double precision, dimension(nz, num_si),
73          intent(in) :: si_ind
74      ! Thickness of each frond
75      double precision, intent(in) ::
76          frond_thickness
77      ! Ratio of length to width (0,infty)
78      double precision, intent(in) ::
79          frond_aspect_ratio
80      ! Rescaled position of greatest width (0=
81          base, 1=tip)
82      double precision, intent(in) ::
83          frond_shape_ratio
84
85      ! WATER CURRENT
86      double precision, dimension(nz), intent(in)
87          :: current_speeds
88      double precision, dimension(nz), intent(in)
89          :: current_angles
90
91      ! SPACING
92      double precision, intent(in) :: rope_spacing
93      double precision, dimension(nz), intent(in)
94          :: depth_spacing
95      ! SOLVER PARAMETERS
96      integer, intent(in) :: num_scatters
97
98      ! FINAL RESULT
99      real, dimension(nz), intent(out) ::
100          avg_irrad, perceived_irrad
101
102      !-----!

```

```

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

```

```

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

```

```

178      call iops%calculate_coef_grids(p_kelp)
179
180      !write(*,*) 'BC'
181      max_rad = 1.d0 ! Doesn't matter because we'
182      ll rescale
183      decay = 1.d0 ! Does matter, but maybe not
184      much. Determines drop-off from angle
185      call bc%init(grid, solar_zenith,
186      solar_azimuthal, decay, max_rad)
187      ! Rescale surface radiance to match surface
188      irradiance
189      bc%bc_grid = bc%bc_grid * surface_irrad /
190      grid%angles%integrate_points(bc%bc_grid)
191
192      write(*,*) 'bc'
193      write(*,*) bc%bc_grid
194
195      ! write(*,*) 'bc'
196      ! do i=1, grid%y%num
197      !     write(*, '(10F15.3)') bc%bc_grid(i,:)
198      ! end do
199
200      call light%init_grid(grid)
201
202      write(*,*) 'Scatter'
203      call calculate_light_with_scattering(grid,
204      bc, iops, light%radiance, num_scatters)
205
206      write(*,*) 'Irrad'
207      call light%calculate_irradiance()
208
209      ! Calculate output variables
210      call calculate_average_irradiance(grid,
211      light, avg_irrad)
212      call calculate_perceived_irrad(grid, p_kelp,
213      &
214      perceived_irrad, light%irradiance)
215
216      !write(*,*) 'vsf_angles = ', iops%vsf_angles
217      !write(*,*) 'vsf_vals = ', iops%vsf_vals
218      !write(*,*) 'vsf norm = ', grid%
219      integrate_angle_2d(iops%vsf(1,1,:,:))
220
221      ! write(*,*) 'abs_water = ', abs_water
222      ! write(*,*) 'scat_water = ', scat_water
223      write(*,*) 'kelp '
224      write(*,*) p_kelp(:,:,:)
225      write(*,*) 'ft =', frond%ft
226
227      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

```

```

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

```

```

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

```