

©2018

OLIVER GRAHAM EVANS

ALL RIGHTS RESERVED

MODELLING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Oliver Graham Evans

May, 2018

MODELLING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

Approved:

Advisor
Dr. Kevin Kreider

Accepted:

Dean of the College
Dr. John Green

Co-Advisor
Dr. Curtis Clemons

Dean of the Graduate School
Dr. Chand Midha

Faculty Reader
Dr. Gerald Young

Date

Department Chair
Dr. Kevin Kreider

ABSTRACT

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

ACKNOWLEDGEMENTS

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

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

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Background on Kelp Models	3
1.3 Background on Radiative Transfer	6
1.4 Overview of Thesis	7
II. KELP MODEL	9
2.1 Physical Setup	9
2.2 Coordinate System	10
2.3 Population Distributions	12
2.4 Spatial Distribution	16
III. LIGHT MODEL	24
3.1 Optical Definitions	24
3.2 The Radiative Transfer Equation	26
3.3 Low-Scattering Approximation	29

IV. NUMERICAL SOLUTION	33
4.1 Super-Individuals	33
4.2 Discrete Grid	34
4.3 Quadrature Rules	37
4.4 Numerical Asymptotics	39
4.5 Finite Difference	42
V. PARAMETER VALUES	51
5.1 Simulation Parameters	51
5.2 Parameters from Literature	52
5.3 Frond Bending Coefficient	54
VI. MODEL ANALYSIS	56
6.1 Homogeneous medium	56
6.2 Grid Study	59
6.3 Asymptotic Convergence	63
6.4 Sensitivity Analysis	67
VII. CONCLUSION	72
APPENDICES	77
APPENDIX A. GRID DETAILS	78
APPENDIX B. RAY TRACING ALGORITHM	82
APPENDIX C. FORTRAN CODE	86

LIST OF TABLES

Table	Page
4.1 Breakdown of nonzero matrix elements by derivative case	49
5.1 Parameter values	53
5.2 Field measurement data of optical properties in the ocean[12]. Site names from the source are used.	53

LIST OF FIGURES

Figure	Page
1.1 <i>Saccharina latissima</i> being harvested	4
2.1 Rendering of four nearby vertical kelp ropes	10
2.2 Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ	11
2.3 Simplified kite-shaped frond	12
2.4 von Mises distribution for a variety of parameters	15
2.5 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$	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 $(\theta, s) = (3\pi/4, 3/2)$.	21
2.9 Colored plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3, v_w = 1$	23
4.1 Spatial grid	35
4.2 Angular grid at each point in space	36
6.1 Exact v.s. finite difference irradiance, linear scale	57

6.2	Exact v.s. finite difference irradiance, log scale	57
6.3	Exact v.s. finite difference irradiance, absolute error	58
6.4	Exact v.s. finite difference irradiance, relative error	58
6.5	Exact v.s. finite difference irradiance, relative error v.s. grid resolution	59
6.6	Grid study, n_s	61
6.7	Grid study, n_z	61
6.8	Grid study, n_a	62
6.9	Grid study, summary	62
6.10	Successive asymptotic approximations, irradiance: AUT8	64
6.11	Successive asymptotic approximations, relative error: AUT8	64
6.12	Successive asymptotic approximations, irradiance: HA011	65
6.13	Successive asymptotic approximations, relative error: HA011	65
6.14	Successive asymptotic approximations, irradiance: NUC2200	66
6.15	Successive asymptotic approximations, relative error: NUC2240	66
6.16	Comparison of asymptotic approximations for various waters.	67
6.17	<i>top-heavy</i> kelp distribution (left) and no-scattering irradiance profile (right)	68
6.18	<i>bottom-heavy</i> kelp distribution (left) and no-scattering irradiance profile (right)	68
6.19	<i>uniform</i> kelp distribution (left) and no-scattering irradiance profile (right)	69
6.20	Several kelp profiles	69
6.21	Several values of kelp absorptance	70
6.22	Several values of absorption coefficient of water	70

6.23 Several values of scattering coefficient	71
A.1 Angular grid	80

CHAPTER I

INTRODUCTION

1.1 Motivation

Given the global rise in population, efficient and innovative resource utilization is increasingly important. Future generations face major challenges regarding food, energy, and water security while addressing major issues associated with global climate change. Growing concern for the negative environmental impacts of petroleum-based fuel is generating a market for biofuel, especially corn-based ethanol. However, corn-based ethanol has been heavily criticized for diverting land usage away from food production, for increasing use of fertilizers that impair water quality, and for low return on energy investments for production. At the same time, a great deal of unutilized saltwater coastline is available for both food and fuel production through seaweed cultivation. Specifically, the sugar kelp *Saccharina latissima* is known to be a viable source of food, both for direct human consumption and biofuel production.

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

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

Large scale macroalgae cultivation has long existed in Eastern Asia due to the popularity of seaweed in Asian cuisine, and low labor costs that facilitate its manual seeding and harvest. More recently, less labor-intense and more industrialized kelp aquaculture has been developing in Scandinavia and in the Northeastern United States and Canada. For example, the MACROSEA project is a four year international research collaboration led by SINTEF, an independent research organization in Norway, and funded by the Research Council of Norway targeting “successful and predictable production of high quality biomass thereby making significant steps towards industrial macroalgae cultivation in Norway.” The project includes both cultivators and scientists, working to develop a precise understanding of the full life cycle of kelp and its interaction with its environment. A fundamental aspect of this endeavor is the development of mathematical models to describe the growth of kelp.

Recently, a growth model[3] for *S. latissima* has been produced and integrated into the SINMOD[18] hydrodynamic and ecosystem model of SINTEF. One aspect of the model which has yet to be fully developed is the availability of light, considering factors such as absorption and scattering by the aquatic medium, as well as by the kelp itself. This thesis contributes to this effort by developing a first-principles model of the light field in a kelp farming environment. As a first step, a model for the spatial distribution of kelp is developed. Radiative transfer theory is then applied to determine the effects of the kelp and water on the availability of light throughout the medium. A finite difference solution to the radiative transfer equation is developed, followed by asymptotic approximations that prove to be sufficiently accurate for clear water, and significantly less computationally intensive. A detailed description of the numerical solution of this model is presented, accompanied by source code for a FORTRAN implementation of the solution. This model can be used independently, or in conjunction with a kelp growth model to determine the amount of light available for photosynthesis at a single time step.

1.2 Background on Kelp Models

Mathematical modeling of macroalgae growth is not a new topic, although it is a reemerging one. Several authors in the second half of the twentieth century were interested in describing the growth and composition of the macroalgae *Macrocystis pyrifera*, commonly known as “giant kelp,” which grows prolifically off the coast of southern California. The first such mathematical model was developed by W.J. North



Figure 1.1: *Saccharina latissima* being harvested

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

photosynthesis rates and the growth experienced by the kelp.

Over a decade later in 1987, G.A. Jackson expanded on Anderson's model for *Macrocystis pyrifera*, with an emphasis on including more environmental parameters and a more complete description of the growth and decay of the kelp[7]. The author takes into account respiration, frond decay, and sub-canopy light attenuation due to self-shading. Light attenuation is represented with a simple exponential model, and self-shading appears as an added term in the decay coefficient. The author does not consider radial or angular dependence on shading. Jackson also expands Anderson's definition of canopy shading, treating the canopy not as a single layer, but as 0, 1, or 2 discrete layers, each composed of individual fronds. While this is a significant improvement over Anderson's light model, it is still rather simplistic.

Both Anderson's and Jackson's model were carried out by numerically solving a system of differential equations over small time intervals. In 1990, M.A. Burgman and V.A. Gerard developed a stochastic population model[4]. This approach is quite different, and functions by dividing kelp plants into groups based on size and age and generating random numbers to determine how the population distribution over these groups changes over time based on measured rates of growth, death, decay, light availability, etc. In the same year, Nyman et. al. published a similar model alongside a Markov chain model, and compared the results with experimental data collected in New Zealand[11].

In 1996 and 1998 respectively, P. Duarte and J.G. Ferreira used the size-class approach to create a more general model of macroalgae growth, and Yoshimori

et. al. created a differential equation model of *Laminaria religiosa* with specific emphasis on temperature dependence of growth rate[6, 19]. These were the some of the first models of kelp growth that did not specifically relate to *Macrocystis pyrifera* (“giant kelp”). Initially, there was a great deal of excitement about this species due to it’s incredible size and growth rate, but difficulties in harvesting and negative environmental impacts have caused scientists to investigate other kelp species.

1.3 Background on Radiative Transfer

In terms of optical quantities, of primary interest is in the radiance at each point from all directions, which affects the photosynthetic rate of the kelp, and therefore the total amount of biomass producible in a given area as well as the total nutrient remediation potential. The equation governing the radiance throughout the system is known as the radiative transfer equation (RTE), which has been largely unutilized in the fields of oceanography and aquaculture. The radiative transfer equation has been used primarily in stellar astrophysics; it’s application to marine biology is fairly recent[9]. In its full form, radiance is a function of 3 spatial dimensions, 2 angular dimensions, and frequency, making for an incredibly complex problem. In this work, frequency is ignored, only monochromatic radiation is considered. The RTE states that along a given path, radiance is decreased by absorption and scattering out of the path, while it is increased by emission and scattering into the path. In our situation, emission is negligible, owing only perhaps to some small luminescent phytoplankton or other anomaly, and can therefore be safely ignored.

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

1.4 Overview of Thesis

The remainder of this document is organized as follows. In Chapter 2, probabilistic model is developed to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. Chapter 3 begins with a survey of fundamental radiometric quantities and optical properties of matter. The spatial kelp distribution from Chapter 2 is used to determine optical properties of the combined water-kelp medium, and the radiative transfer equation, an integro-partial differential equation which describes the the light field as a function of position and angle, is discussed. An asymptotic expansion is explored for cases where absorption dominates scattering in the medium, such as in very clear water or high kelp density. In Chapter 4, details are given for the numerical solution of the equations from Chapters 2 and 3. Both the full finite difference solution and the asymptotic approx-

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

CHAPTER II

KELP MODEL

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

2.1 Physical Setup

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

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

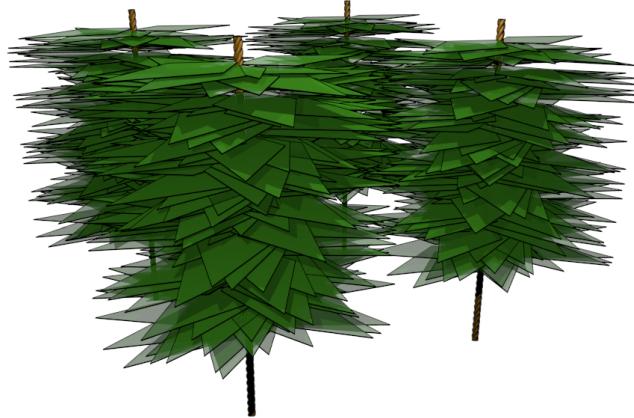


Figure 2.1: Rendering of four nearby vertical kelp ropes

2.2 Coordinate System

Consider the rectangular domain

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

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

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

For all three dimensional analysis, we use the absolute coordinate system defined in Figure 2.2. In the following sections, it is necessary to convert between Cartesian

and spherical coordinates, which we do using the relations

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

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

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

Then, calculating derivatives from (2.1) yields

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

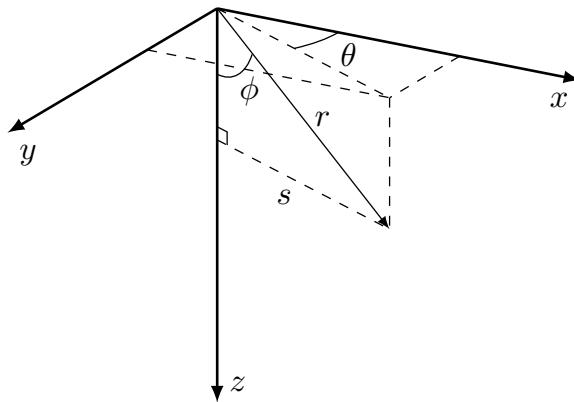


Figure 2.2: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ

2.3 Population Distributions

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

2.3.1 Frond Shape

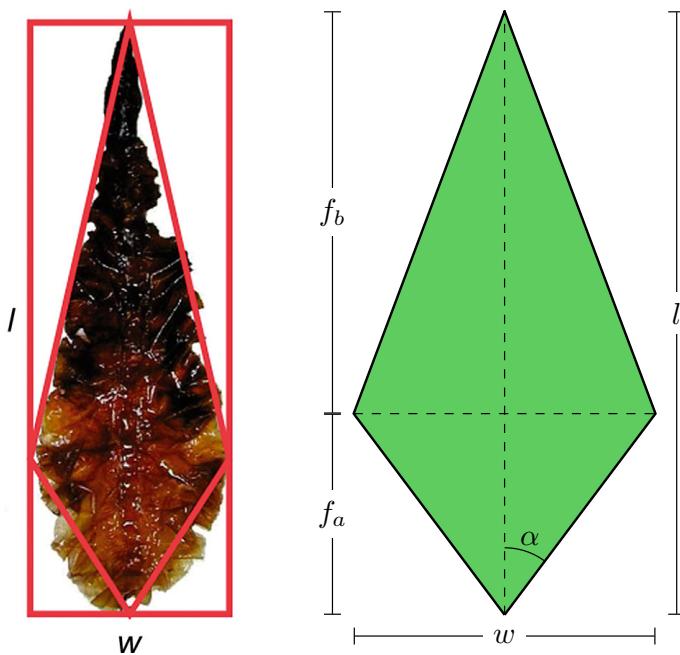


Figure 2.3: Simplified kite-shaped frond

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

We have

$$f_a + f_b = l$$

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

$$\begin{aligned} f_r &= \frac{l}{w} \\ f_s &= \frac{f_a}{f_b} \end{aligned}$$

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.

$$\begin{aligned} w &= \frac{l}{f_r} \\ f_a &= \frac{lf_s}{1+f_s} \\ f_b &= \frac{l}{1+f_s} \end{aligned}$$

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

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

The area of the frond is given by

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

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

$$l = \sqrt{2Af_r}. \quad (2.3)$$

2.3.2 Length and Angle Distributions

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

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

It is further assumed that frond angle varies according to the von Mises distribution, which is the periodic analogue of the normal distribution, defined on $[-\pi, \pi]$ rather than $(-\infty, \infty)$. The von Mises distribution has two parameters, μ and κ , which shift and sharpen its peak respectively, as shown in Figure 2.4. κ can be considered analogous to $1/\sigma$ in the normal distribution. That is, in the case of zero current velocity, the frond angles are be distributed uniformly, while as current velocity increases, they become increasingly likely to be pointing in the direction of the current, depending on the stiffness of the frond. Assuming a linear relationship between the current velocity and the steepness of the angular distribution, define the frond bending coefficient η , with units s m^{-1} . Then, in use $\mu = \theta_w$ and $\kappa = \eta v_w$ as the von Mises distribution parameters. Note that θ_w and v_w vary over depth, while η is assumed constant for the population. Then, the PDF for the von Mises frond angle distribution is

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

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

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

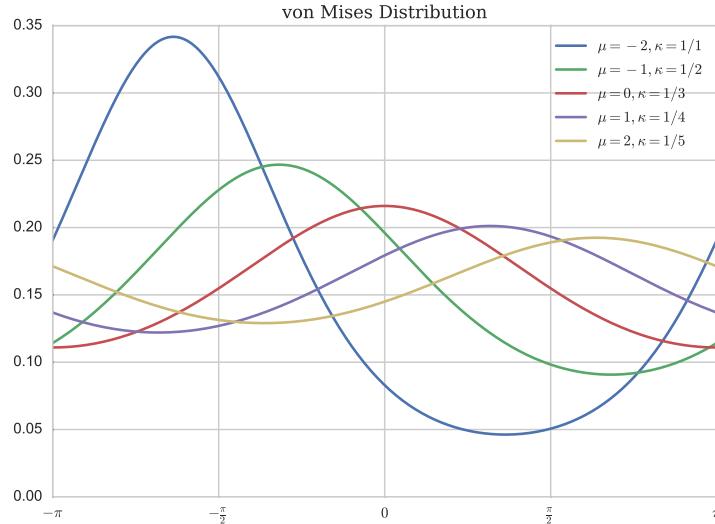


Figure 2.4: von Mises distribution for a variety of parameters

2.3.3 Joint Length-Orientation Distribution

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

Given independent events A and B ,

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

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

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

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

It is important to note that if P_{θ_f} were dependent on l , the above definition of P_{2D} would no longer be valid. For example, it might be more realistic to say that larger fronds are less likely to bend towards the direction of the current. In this case, (2.3.3) would no longer hold, and it would be necessary to use the more general relation

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

which is currently not taken into consideration in this model.

2.4 Spatial Distribution

2.4.1 Rotated Coordinate System

To determine under what conditions a frond will occupy a given point, we begin by describing the shape of the frond in Cartesian coordinates and then convert to

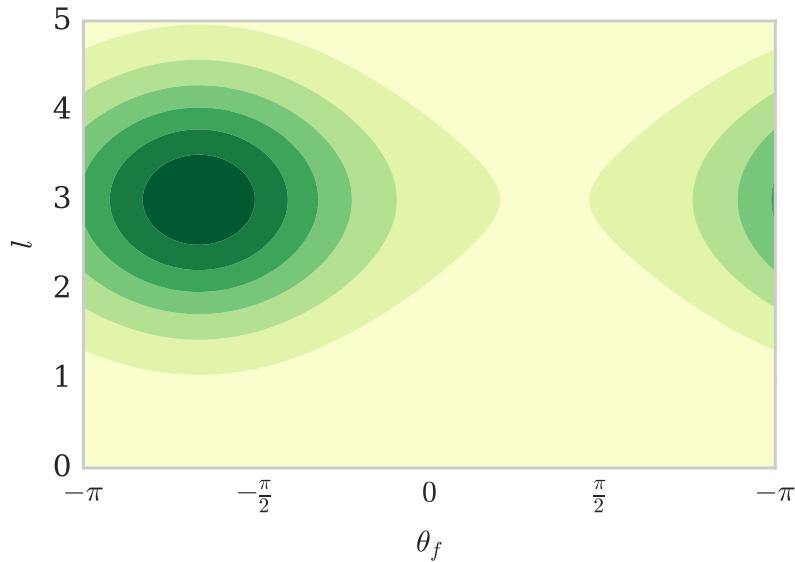


Figure 2.5: 2D length-angle probability distribution with $\theta_w = 2\pi/3, v_w = 1$

polar coordinates. Of primary interest are the edges connected to the frond tip. For convenience, we will use a rotated coordinate system (θ', s) such that the line connecting the base to the tip is vertical, with the base at $(0, 0)$. Denote the Cartesian analogue of this coordinate system as (x', y') which is related to (θ', s) by

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

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

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

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

2.4.2 Functional Description of Frond Edge

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

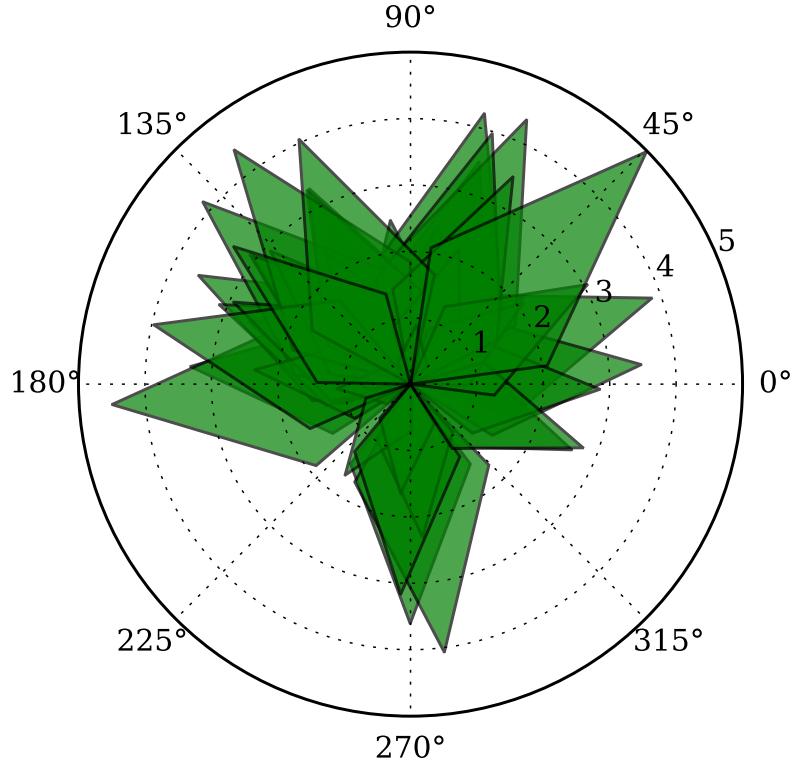


Figure 2.6: A sample of 50 kelp fronds with length and angle picked from the distribution above with $f_s = 0.5$ and $f_r = 2$.

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

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

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

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

where

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

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

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

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

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

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

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

2.4.3 Conditions for Occupancy

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

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

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

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

$$\theta - \alpha < \theta_f < \theta + \alpha, \quad (2.4)$$

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

where

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

Then, considering the point to be fixed, (2.4) and (2.5) define the spacial region $R_s(\theta, s)$ called the “occupancy region for (θ, s) ” with the property that if the tip of a frond lies within this region (i.e., $(\theta_f, l) \in R_s(\theta, s)$), then it occupies the point. $R_s(3\pi/4, 3/2)$ is shown in blue in Figure 2.7 and the smallest possible occupying fronds for several values of θ_f are shown in various colors. Any frond longer than these at the same angle will also occupy the point.

2.4.4 Probability of Occupancy

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

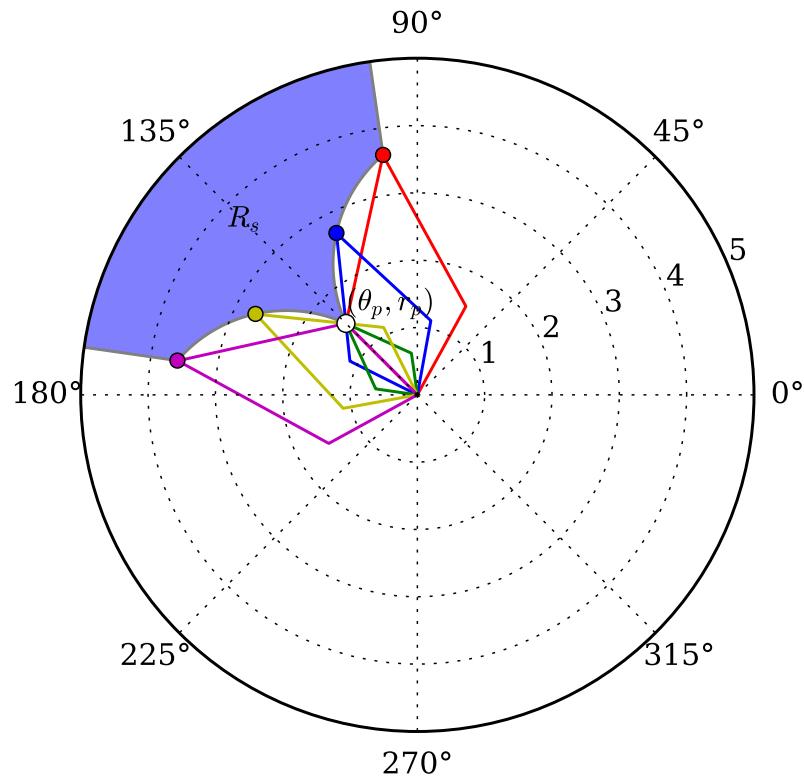


Figure 2.7: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$

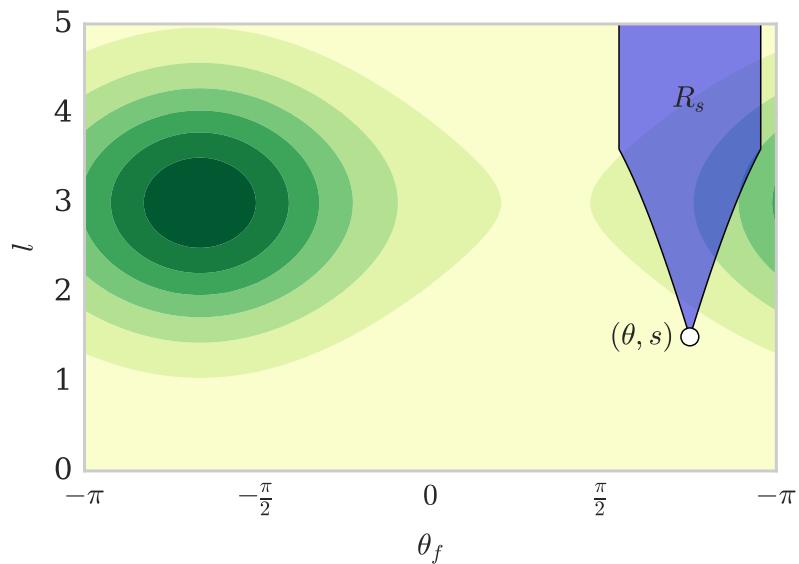


Figure 2.8: Contour plot of $P_{2D}(\theta_f, l)$ overlaid with the region in the $\theta_f - l$ plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$

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

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

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

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

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

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

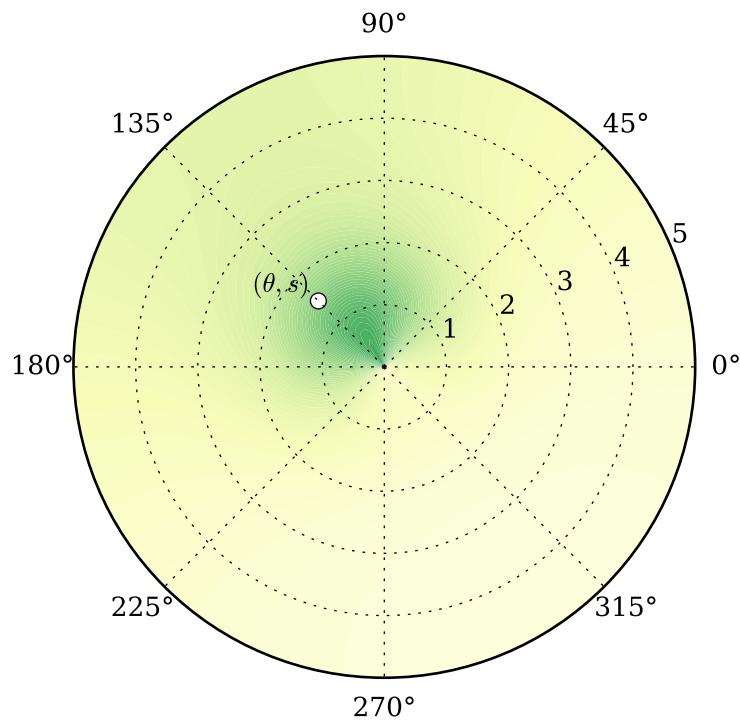


Figure 2.9: Colored plot of the probability of frond occupation sampled at 121 points

using $\theta_f = 2\pi/3, v_w = 1$

CHAPTER III

LIGHT MODEL

Now that we have formulated the distribution of kelp throughout the medium, we introduce the Radiative Transfer Equation, which is used to calculate the light field.

3.1 Optical Definitions

3.1.1 Radiometric Quantities

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

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

Once the radiance L is calculated everywhere, the irradiance is

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

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

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

$$1 \text{ W/m}^2 = 4.2 \mu\text{mol photons/s}. \quad (3.1)$$

3.1.2 Perceived Irradiance

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

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

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

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

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

3.1.3 Inherent Optical Properties

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

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

The VSF is normalized such that

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

so that for any ω ,

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

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

3.2 The Radiative Transfer Equation

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

3.2.1 Ray Notation

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

initial z coordinate given by

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

Hence, the ray path is parameterized as

$$\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s) = \frac{1}{\tilde{s}}(s\mathbf{x} + (\tilde{s} - s)\mathbf{x}_0(\mathbf{x}, \boldsymbol{\omega})), \quad (3.2)$$

where

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

is the origin of the ray, and

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

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

3.2.2 Colloquial Description

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

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

3.2.3 Equation of Transfer

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

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

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

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

which unveils the general form of the Radiative Transfer Equation,

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

or, equivalently,

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

3.2.4 Boundary Conditions

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

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

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

In the z direction, we specify a spatially uniform downwelling light just under the surface of the water by a function $f(\boldsymbol{\omega})$. Or if $z_{\min} > 0$, then the radiance at $z = z_{\min}$ should be specified instead (as opposed to the radiance at the first grid cell center). Further, we assume that no upwelling light enters the domain from the bottom, so

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

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

3.3 Low-Scattering Approximation

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

3.3.1 Asymptotic Expansion

Taking b to be small, we introduce the asymptotic series

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

Substituting the above into (3.4) yields

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

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

$$\boldsymbol{\omega} \cdot \nabla L_0(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_0(\mathbf{x}) = 0, \tag{3.5}$$

$$\boldsymbol{\omega} \cdot \nabla L_1(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_1(\mathbf{x}) = \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_0(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_0(\mathbf{x}, \boldsymbol{\omega}),$$

$$\boldsymbol{\omega} \cdot \nabla L_2(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_2(\mathbf{x}) = \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_1(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_1(\mathbf{x}, \boldsymbol{\omega}).$$

⋮

In general, for $n \geq 1$,

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

For boundary conditions, let \mathbf{x}_s be a point on the surface of the domain.

Then,

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

which can be decomposed as

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

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

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

⋮

In general, for $n \geq 1$,

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

3.3.2 Analytical Solution

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

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

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

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

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

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

Then, integrating both sides yields

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

Hence,

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

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

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

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

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

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

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

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

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

Integrating both sides yields

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

Hence,

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

which simplifies to

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

As before, the conversion back to spatial coordinates is

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

CHAPTER IV

NUMERICAL SOLUTION

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

4.1 Super-Individuals

Rather than model each kelp frond, a subset of the population, called super-individuals, are modeled explicitly, and are considered to represent many identical individuals, as in [14]. Specifically, at each depth k , there are n super-individuals, indexed by i . Super-individual i has a frond area A_{ki} and represents n_{ki} individual fronds.

From (2.3), the frond length of the super-individual is $l_{ki} = \sqrt{2A_{ki}f_r}$. Given the super-individual data, we calculate the mean μ and standard deviation σ frond

lengths using the formulas

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

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

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

4.2 Discrete Grid

The following is a description of the spatial-angular grid used in the numerical implementation of this model. It is assumed that all simulated quantities are constant over the interior of a grid cell. Other legitimate choices of grids exists; this one was chosen for its relative simplicity.

The domain of the radiative transfer equation is embedded in five dimensions: three spatial (x , y , and z) and two angular (azimuthal θ and polar ϕ). The number of grid cells in each dimension are denoted by n_x , n_y , n_z , n_θ , and n_ϕ , with uniform spacings dx , dy , dz , $d\theta$, and $d\phi$ between adjacent grid points.

The following indices are assigned to each dimension:

$$x \rightarrow i$$

$$y \rightarrow j$$

$$z \rightarrow k$$

$$\theta \rightarrow l$$

$$\phi \rightarrow m$$

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

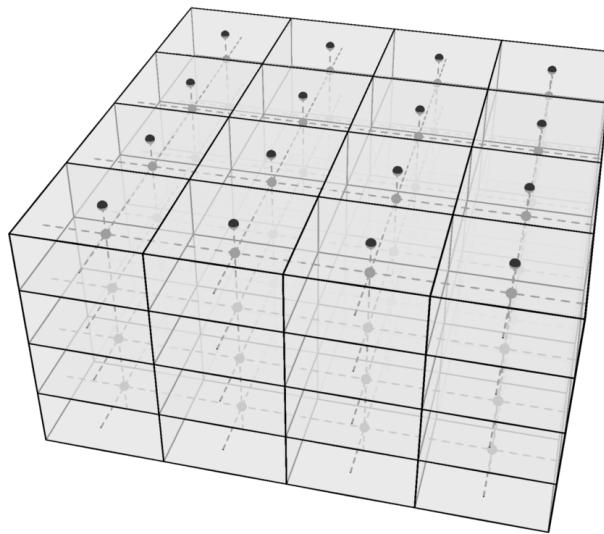


Figure 4.1: Spatial grid

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

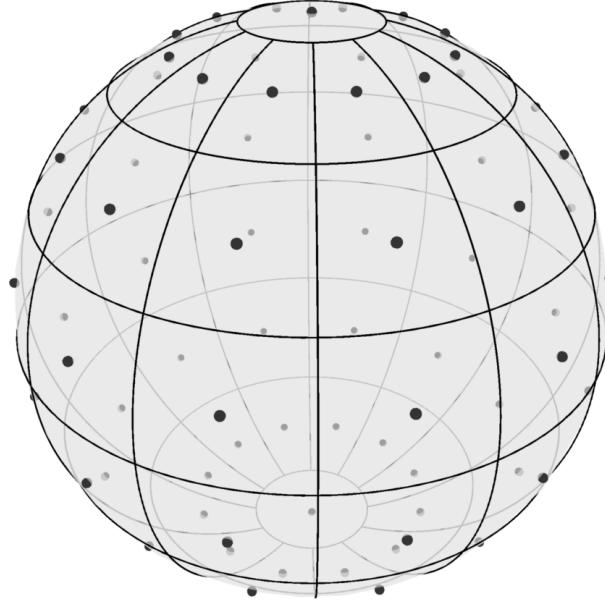


Figure 4.2: Angular grid at each point in space

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

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

4.3 Quadrature Rules

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

4.3.1 Spatial Quadrature

Define the *spatial characteristic function* as

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

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

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

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

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

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

4.3.2 Angular Quadrature

Define the *angular characteristic function* as

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

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

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

4.4 Numerical Asymptotics

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

4.4.1 Scattering Integral

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

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

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

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

Hence, the average radiance scattered from $\Omega_{p'}$ into some $\omega \in \Omega_p$ is $\beta_{pp'} |\Omega'| L_{ijkp'}$.

Therefore, the radiance gain due to scattering into ω_p from all other angles is

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

4.4.2 Ray Integral

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

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

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

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

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

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

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

Given s , the index next edge crossing occurs at

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

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

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

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

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

This integral is made straightforward by setting

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

which yields

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

Define the intermediate variable

$$d_\nu = \int_{s_\nu}^{s_{\nu+1}} \exp(\tilde{a}_\nu s') ds'$$

$$= \begin{cases} ds_\nu, & \tilde{a} = 0 \\ (\exp(\tilde{a}_\nu s_{\nu+1}) - \exp(\tilde{a}_\nu s_\nu)) / \tilde{a}_\nu, & \text{otherwise,} \end{cases}$$

which permits the simple formula

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

4.5 Finite Difference

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

4.5.1 Discretization

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

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

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

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

Truncating after the first few terms, we have

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

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

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

Rearranging (4.5) produces

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

Combining (4.6) with (4.7) gives

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

Then, dividing by ε gives

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

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

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

4.5.2 Difference Equations

In general, we have

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

Then,

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

Interior:

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

Surface downwelling (BC):

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

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

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

Likewise for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

4.5.3 Structure of Linear System

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

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

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

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

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

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

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

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

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

Table 4.1: Breakdown of nonzero matrix elements by derivative case

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

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

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

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

4.5.4 Iterative Solution

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

CHAPTER V

PARAMETER VALUES

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

5.1 Simulation Parameters

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

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

5.2 Parameters from Literature

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

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

Table 5.1: Parameter values

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

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

Site	$a(\text{m}^{-1})$	$b(\text{m}^{-1})$	$c(\text{m}^{-1})$	a/c	b/c
AUTEC 8	0.114	0.037	0.151	0.753	0.247
HAOCE 11	0.179	0.219	0.398	0.449	0.551
NUC 2200	0.337	1.583	1.92	0.176	0.824
NUC 2240	0.125	1.205	1.33	0.094	0.906

5.3 Frond Bending Coefficient

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

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

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

CHAPTER VI

MODEL ANALYSIS

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

6.1 Homogeneous medium

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

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

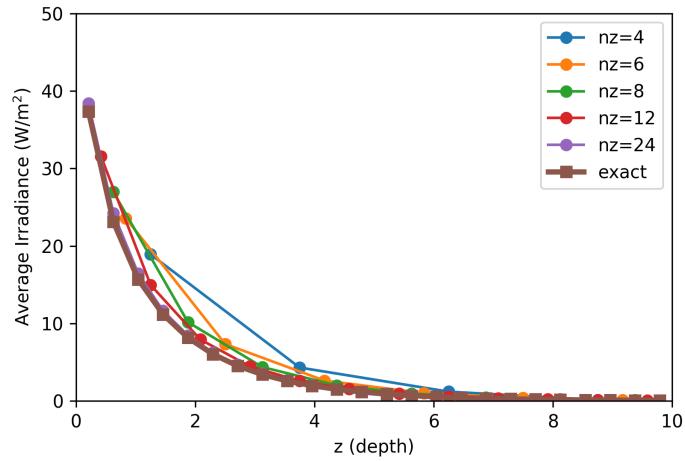


Figure 6.1: Exact v.s. finite difference irradiance, linear scale

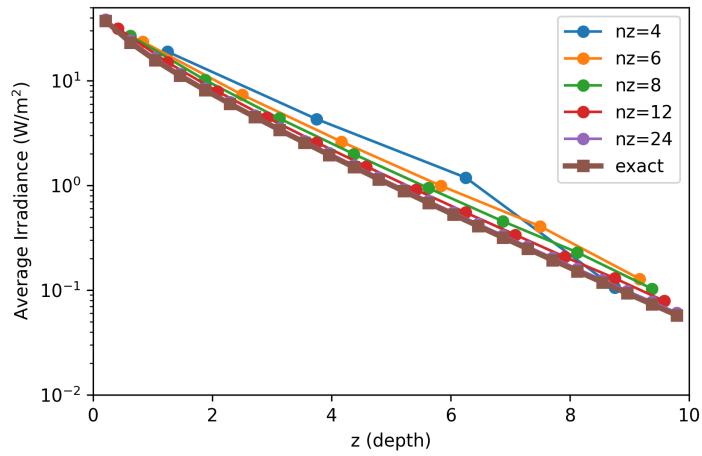


Figure 6.2: Exact v.s. finite difference irradiance, log scale

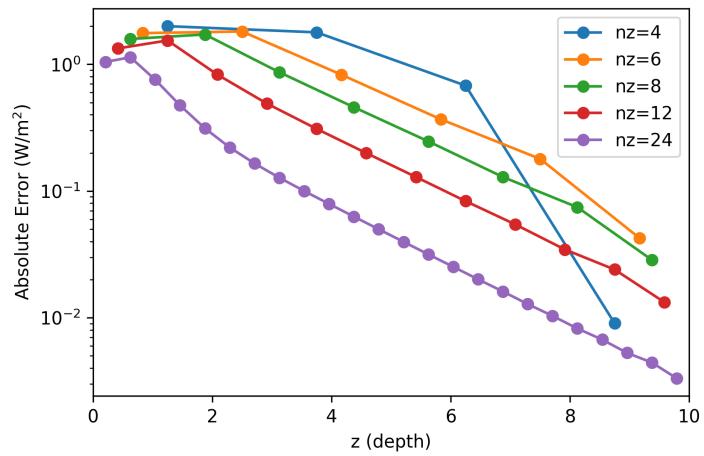


Figure 6.3: Exact v.s. finite difference irradiance, absolute error

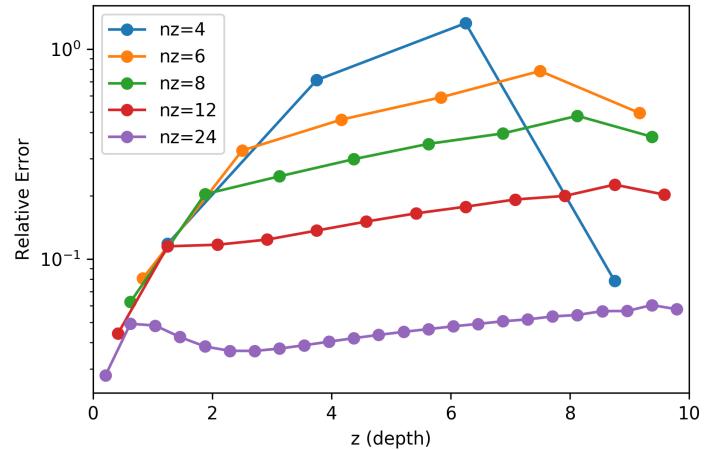


Figure 6.4: Exact v.s. finite difference irradiance, relative error

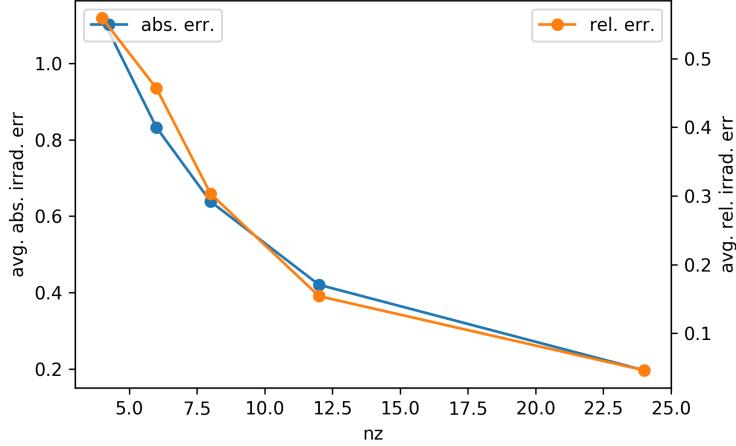


Figure 6.5: Exact v.s. finite difference irradiance, relative error v.s. grid resolution

6.2 Grid Study

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

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

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

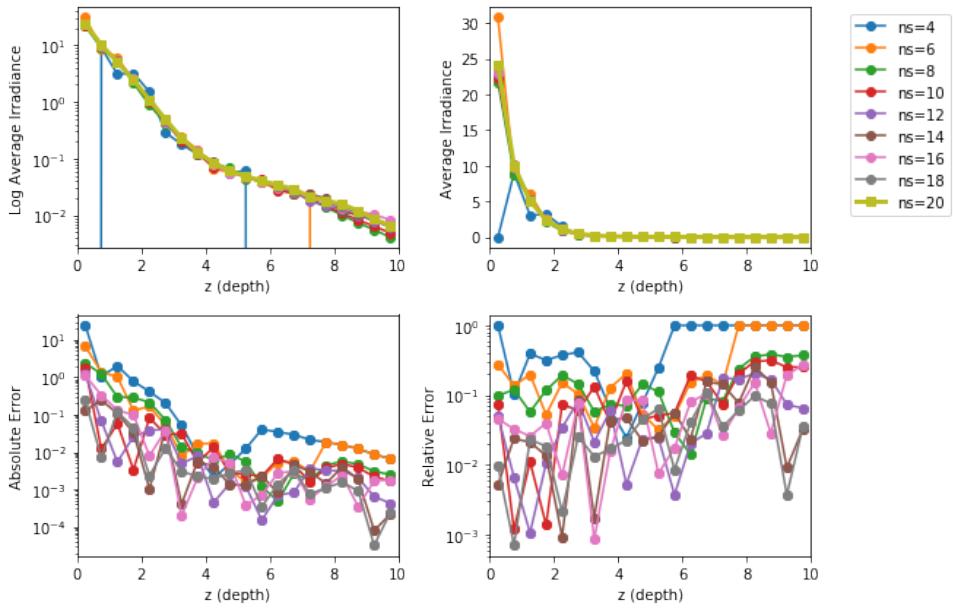


Figure 6.6: Grid study, n_s

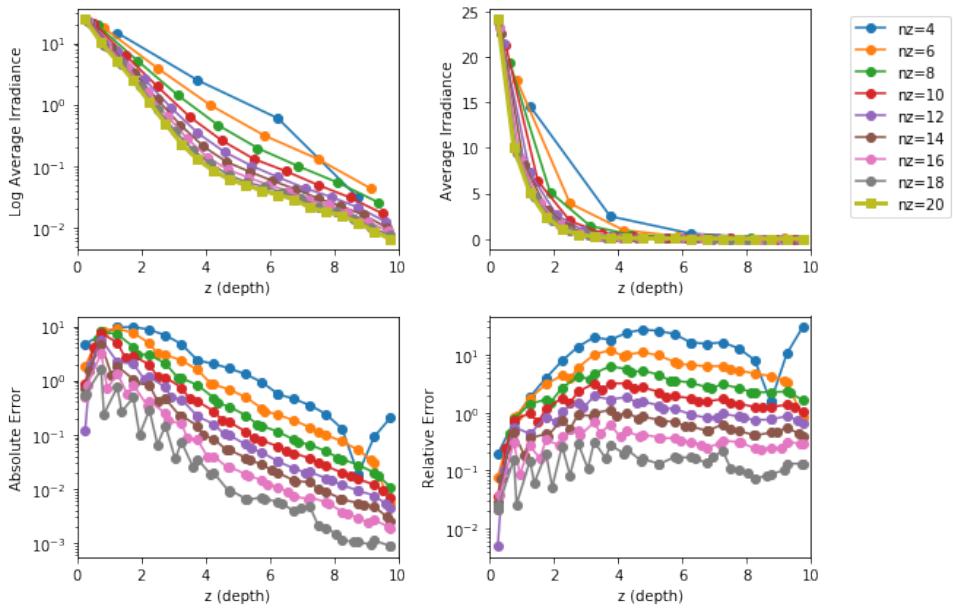


Figure 6.7: Grid study, n_z

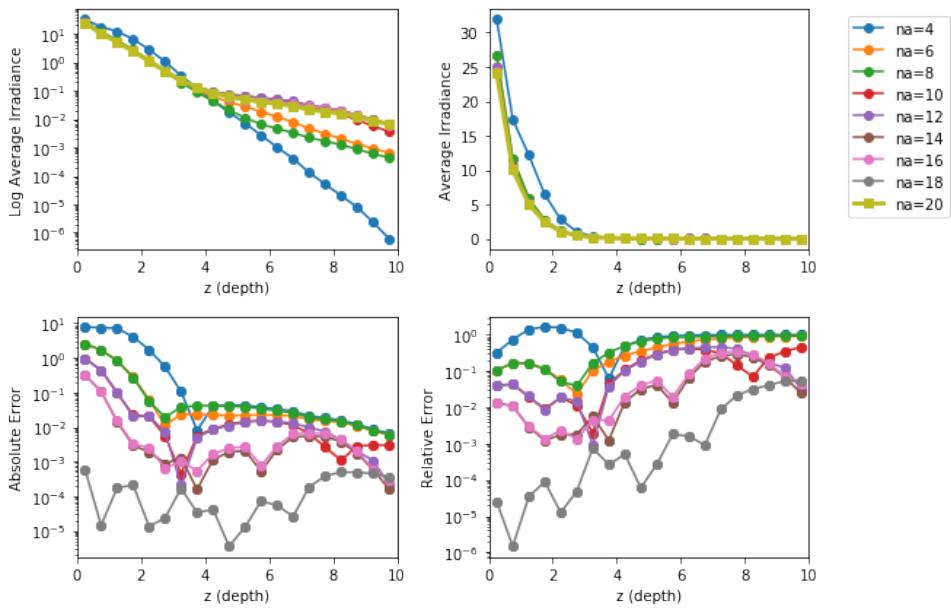


Figure 6.8: Grid study, n_a

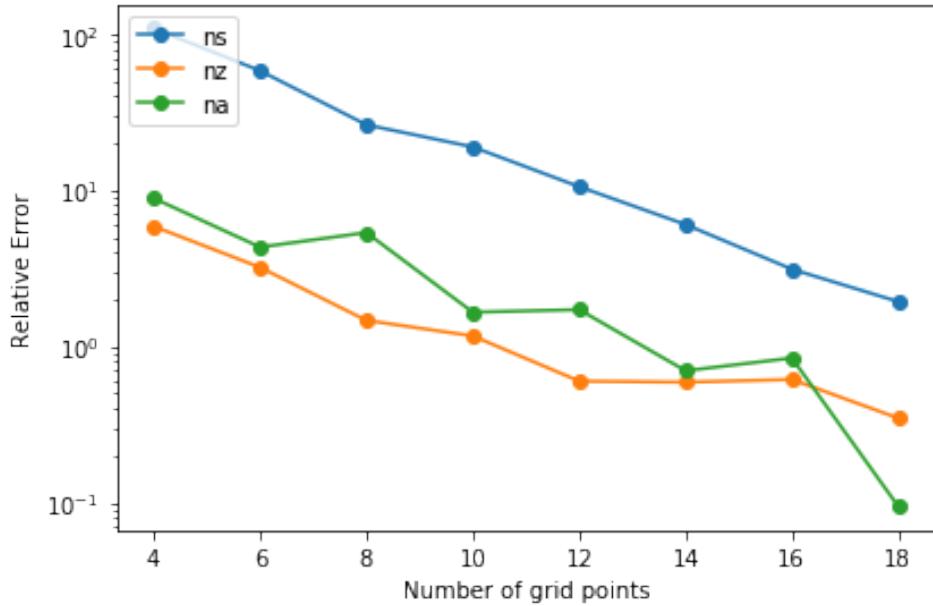


Figure 6.9: Grid study, summary

6.3 Asymptotic Convergence

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

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

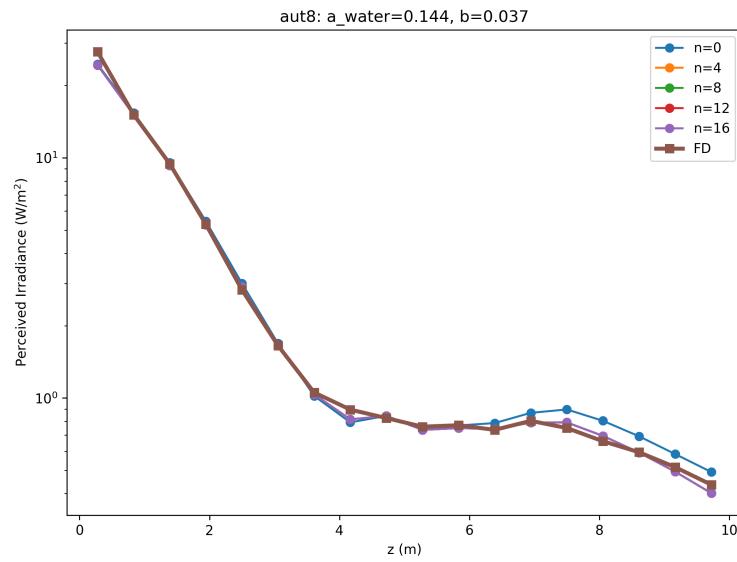


Figure 6.10: Successive asymptotic approximations, irradiance: AUT8

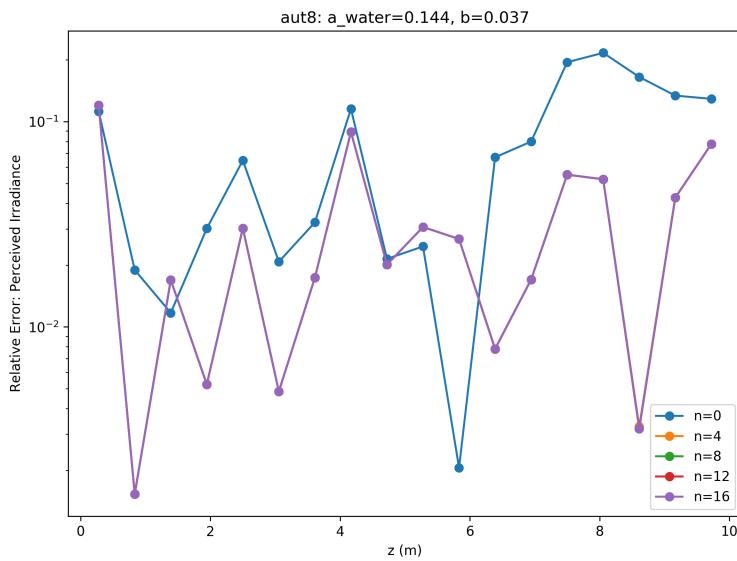


Figure 6.11: Successive asymptotic approximations, relative error: AUT8

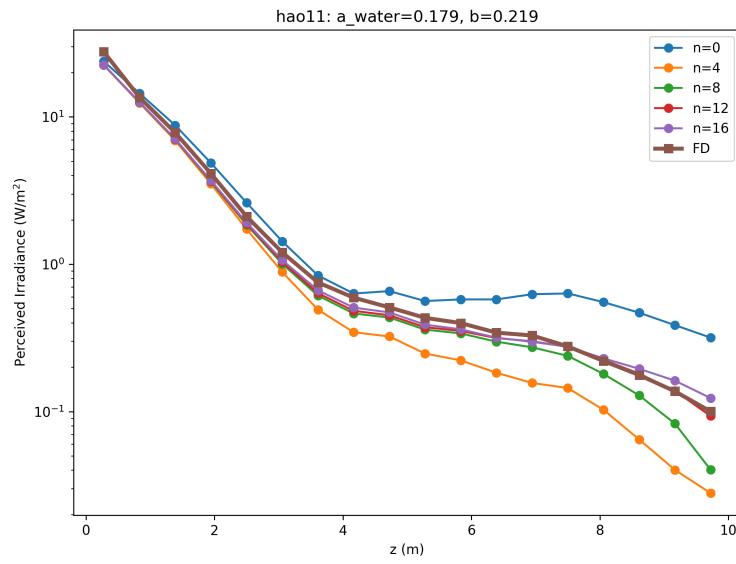


Figure 6.12: Successive asymptotic approximations, irradiance: HA011

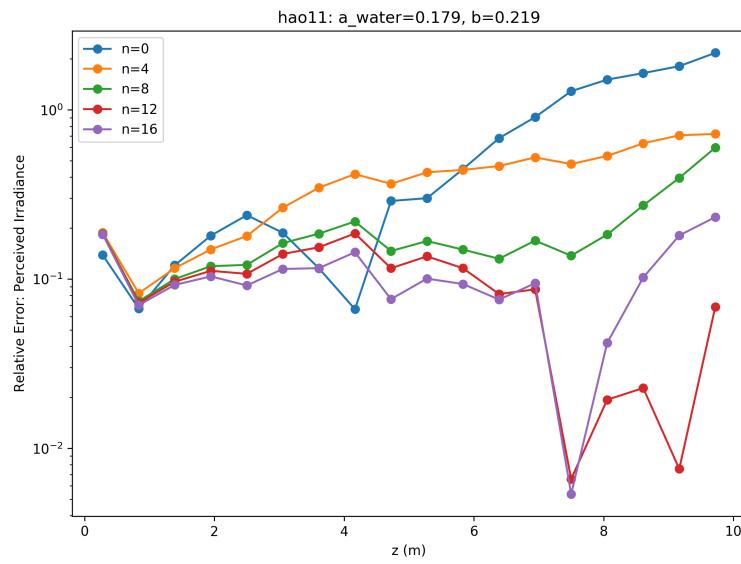


Figure 6.13: Successive asymptotic approximations, relative error: HA011

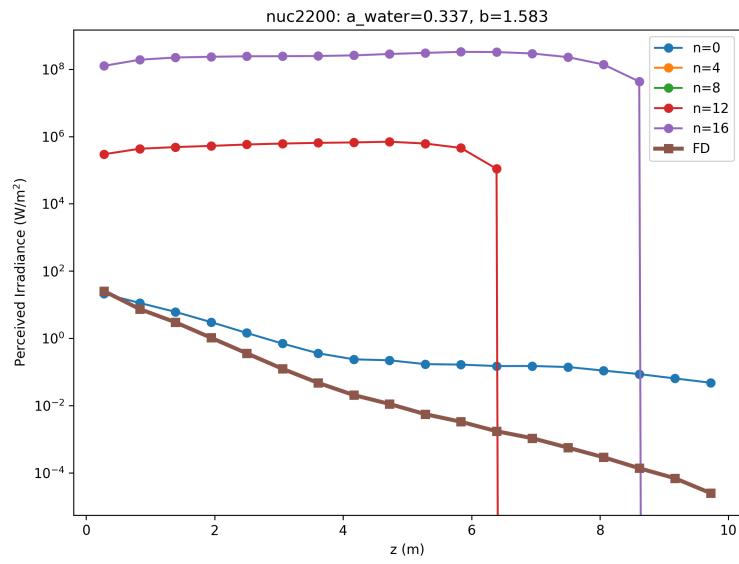


Figure 6.14: Successive asymptotic approximations, irradiance: NUC2200

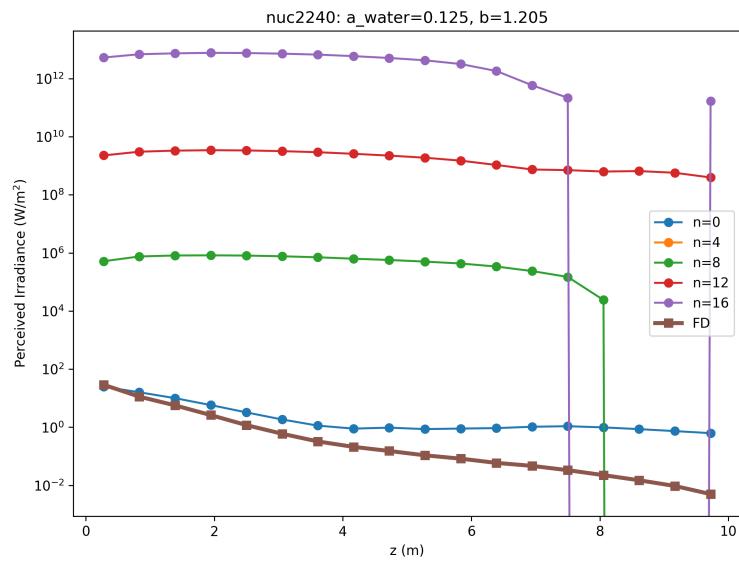


Figure 6.15: Successive asymptotic approximations, relative error: NUC2240

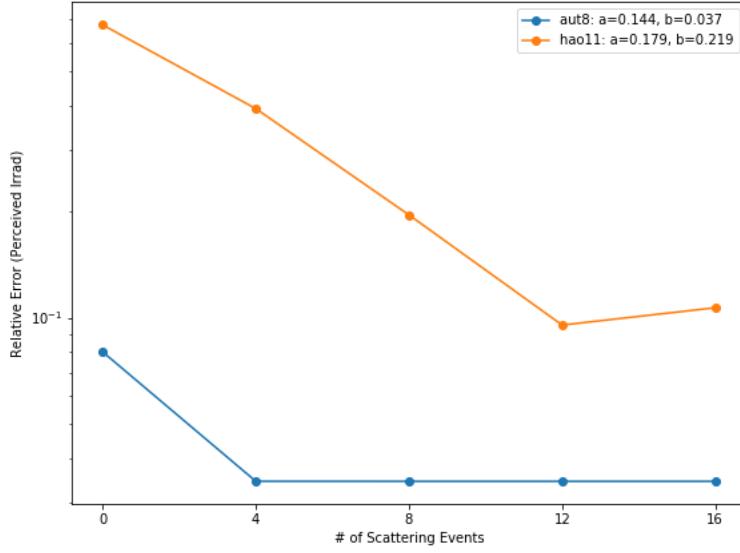


Figure 6.16: Comparison of asymptotic approximations for various waters.

6.4 Sensitivity Analysis

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

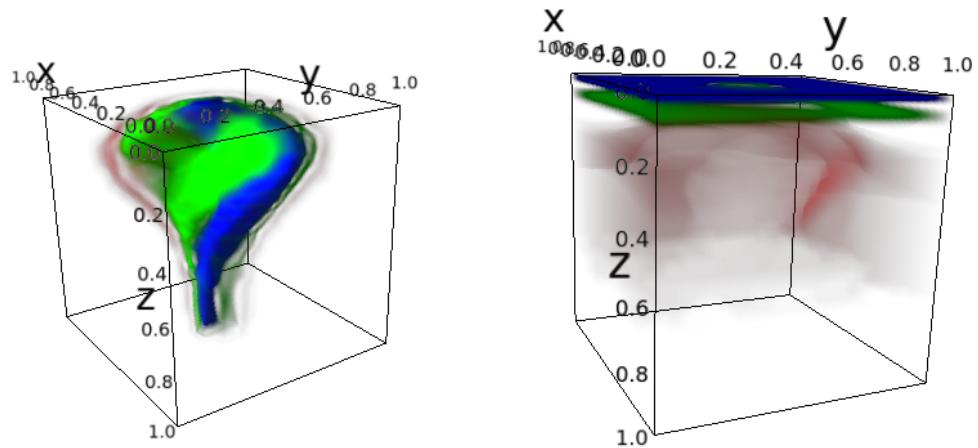


Figure 6.17: *top-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

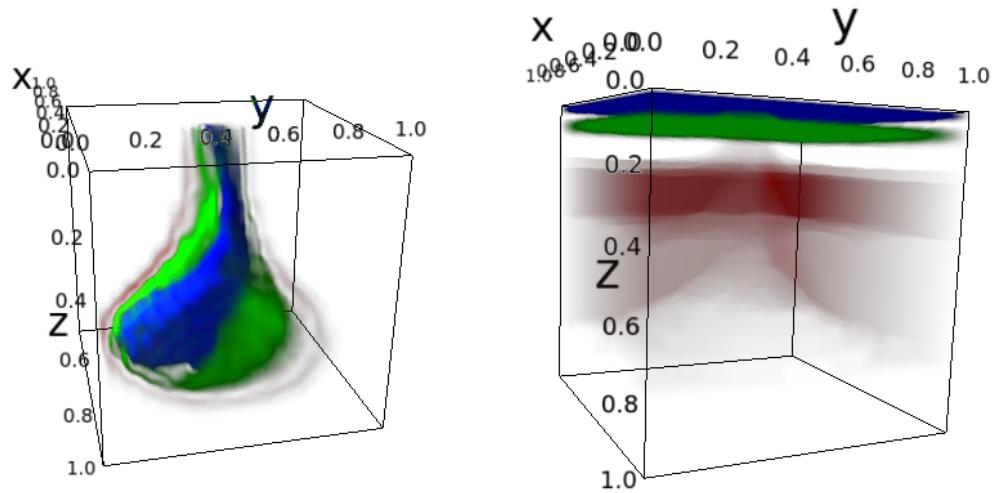


Figure 6.18: *bottom-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

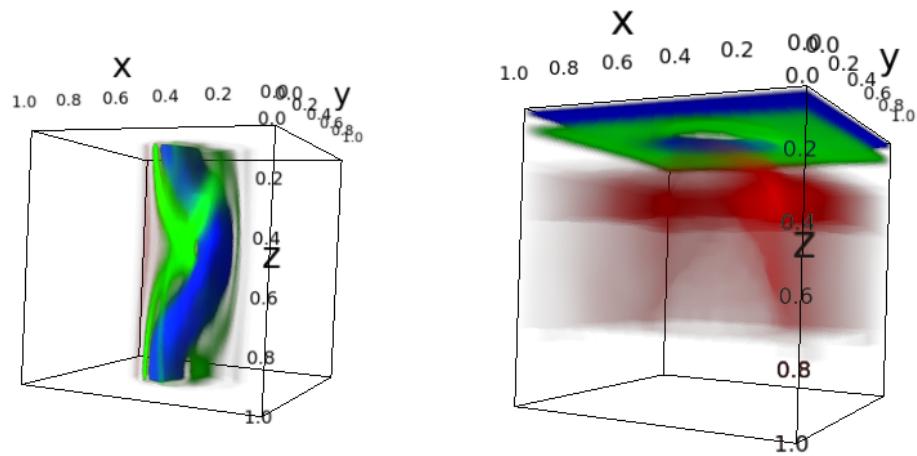


Figure 6.19: *uniform* kelp distribution (left) and no-scattering irradiance profile (right)

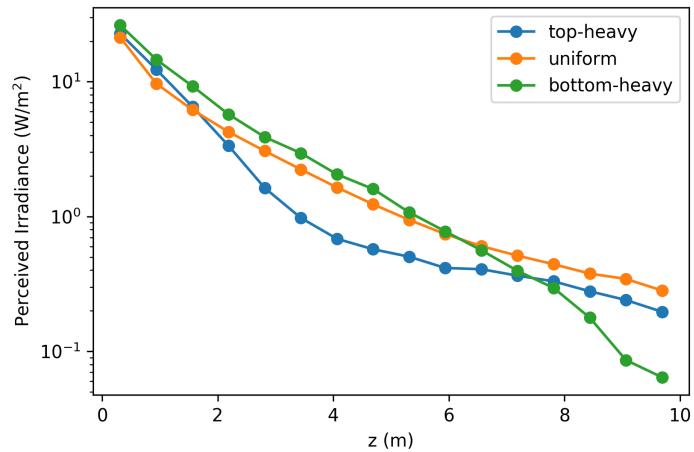


Figure 6.20: Several kelp profiles

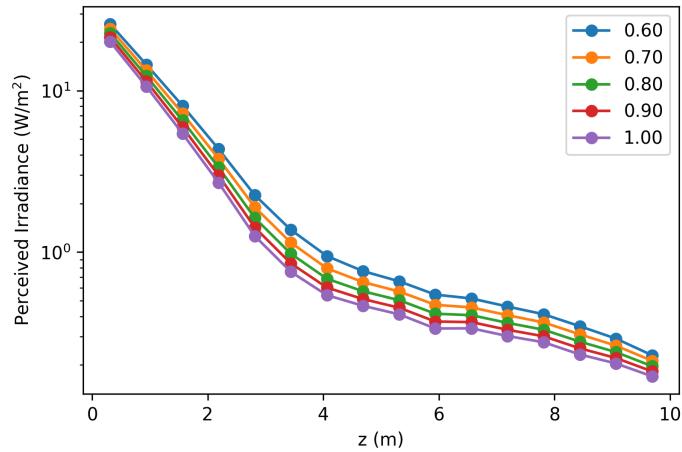


Figure 6.21: Several values of kelp absorptance

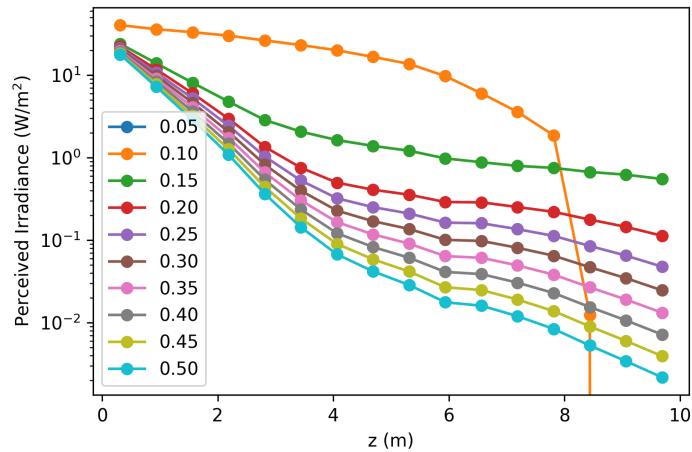


Figure 6.22: Several values of absorption coefficient of water

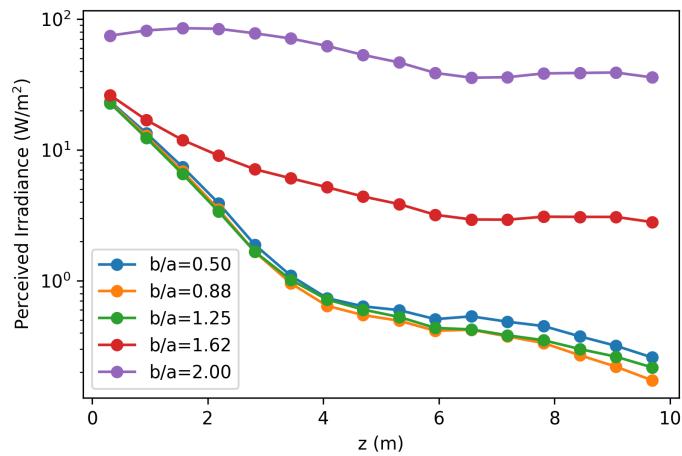


Figure 6.23: Several values of scattering coefficient

CHAPTER VII

CONCLUSION

We present a probabilistic model for the spatial distribution of kelp, and develop a first-principles model for the light field, considering absorption and scattering due to the water and kelp. A full finite difference solution is presented, and an asymptotic approximation based on discrete scattering events is subsequently developed. The asymptotic approximation is shown to converge to the finite difference solution in cases where the absorption coefficient is the same order of magnitude as the scattering coefficient or larger. Otherwise, the solution diverges.

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

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

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

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

BIBLIOGRAPHY

- [1] N. Anderson. A mathematical model for the growth of giant kelp. *Simulation*, 22(4):97–105, 1974.
- [2] A. H. Baker, E. R. Jessup, and T. Manteuffel. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, Jan. 2005.
- [3] O. J. Broch and D. Slagstad. Modelling seasonal growth and composition of the kelp *Saccharina latissima*. *Journal of Applied Phycology*, 24(4):759–776, Aug. 2012.
- [4] M. A. Burgman and V. A. Gerard. A stage-structured, stochastic population model for the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 105(1):15–23, 1990.
- [5] M. F. Colombo-Pallotta, E. Garca-Mendoza, and L. B. Ladah. Photosynthetic Performance, Light Absorption, and Pigment Composition of *Macrocystis Pyrifera* (laminariales, Phaeophyceae) Blades from Different Depths1. *Journal of Phycology*, 42(6):1225–1234, Dec. 2006.

- [6] P. Duarte and J. G. Ferreira. A model for the simulation of macroalgal population dynamics and productivity. *Ecological modelling*, 98(2-3):199–214, 1997.
- [7] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 95(4):611–624, 1987.
- [8] C. Mobley. *Light and Water: Radiative Transfer in Natural Waters*. Academic Press, 1994.
- [9] C. Mobley. Radiative Transfer in the Ocean. In *Encyclopedia of Ocean Sciences*, pages 2321–2330. Elsevier, 2001.
- [10] C. Norvik. Design of Artificial Seaweeds for Assessment of Hydrodynamic Properties of Seaweed Farms. 2017.
- [11] M. Nyman, M. Brown, M. Neushul, and J. A. Keogh. *Macrocystis pyrifera in New Zealand: testing two mathematical models for whole plant growth*, volume 2. Sept. 1990.
- [12] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.
- [13] Y. Saad and M. H. Schultz. GMRES: a Generalized Minimal Residual algorithm for solving nonsymmetric linear systems. Research Report YALEU/DCS/RR-254, Yale University, May 1985.

- [14] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Eco-logical Modelling*, 80:161–170, Mar. 1994.
- [15] A. Sokolov, M. Chami, E. Dmitriev, and G. Khomenko. Parameterization of volume scattering function of coastal waters based on the statistical approach. *Optics express*, 18(5):4615–4636, 2010.
- [16] P. Sonneveld and M. B. van Gijzen. IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. *SIAM Journal on Scientific Computing; Philadelphia*, 31(2):28, 2008.
- [17] H. Van Der Vorst. BI-CGSTAB: A Fast and Smoothly Converging Variant of BI-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, Mar. 1992.
- [18] P. Wassmann, D. Slagstad, C. W. Riser, and M. Reigstad. Modelling the ecosystem dynamics of the Barents Sea including the marginal ice zone. *Journal of Marine Systems*, 59(1-2):1–24, Jan. 2006.
- [19] A. Yoshimori, T. Kono, and H. Iizumi. Mathematical models of population dynamics of the kelp *Laminaria religiosa*, with emphasis on temperature dependence. *Fisheries Oceanography*, 7(2):136146, 1998.

APPENDICES

APPENDIX A

GRID DETAILS

The width of the spatial grid cells in each dimension are

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

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

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

Denote the edges as

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

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

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

and the cell centers as

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

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

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

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

Now, we define the azimuthal angle such that

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

For the sake of periodicity, we need

$$\theta_1 = 0,$$

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

which requires

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

For the polar angle, we similarly let

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

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

$$\phi_1 = 0,$$

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

This gives us

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

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

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

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

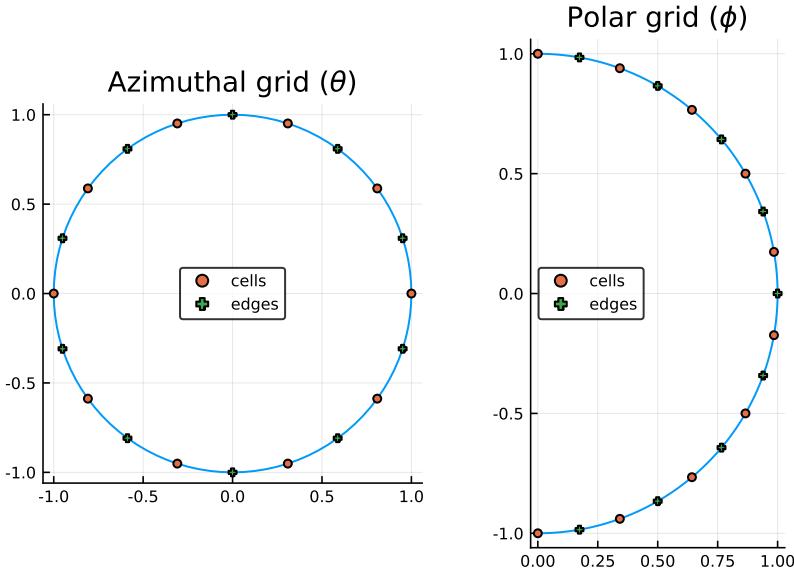


Figure A.1: Angular grid

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

The following notation is used.

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

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

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

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

Thus, it follows that

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

Accordingly, define

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

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

For the poles, we have

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

And for all other angular grid cells,

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

APPENDIX B

RAY TRACING ALGORITHM

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

Consider a grid center $\mathbf{p}_1 = (p_{1x}, p_{1y}, p_{1z})$ and a corresponding path $\mathbf{l}(\mathbf{x}_1, \omega, s)$. To find the location of discontinuities in the integrand, we first calculate the distance from its origin, $\mathbf{p}_0 = \mathbf{x}_0(\mathbf{p}_1, \omega) = (p_{0x}, p_{0y}, p_{0z})$ to grid edges in each dimension separately.

Given

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

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

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

we have

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

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

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

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

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

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

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

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

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

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

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

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

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

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

That is,

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

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

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

Then,

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

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

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

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

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

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

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

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

* We also track s , the path length.

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

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

* This doesn't quite make sense yet.

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

and

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

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

APPENDIX C

FORTRAN CODE

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

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

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

```
utils.f90
1 ! General utilities which might be useful in
2     other settings
3 module utils
4 implicit none
5
6 ! Constants
7 double precision, parameter :: pi = 4.D0 * datan
8     (1.D0)
9
10 contains
11
12 ! Determine base directory relative to current
13 ! directory
14 ! by looking for Makefile, which is in the base
15 ! dir
16 ! Assuming that this is executed from within the
17 ! git repo.
18 function getbasedir()
19     implicit none
20
21     ! INPUTS:
22     ! Number of paths to check
23     integer, parameter :: numpaths = 3
24     ! Maximum length of path names
25     integer, parameter :: maxlenlength = numpaths *
26         2 - 1
27     ! Paths to check for Makefile
28     character(len=maxlength), parameter,
29         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
70 ! maximum
71 function bnd2max(xmin,xmax,dx)
72     implicit none
73
74     ! INPUTS:
75     ! xmin - minimum x value in array
76     ! xmax - maximum x value in array (inclusive
77     ! )
78     ! dx - step size
79     double precision, intent(in) :: xmin, xmax,
80             dx
81
82     ! OUTPUT:
83     ! step2max - maximum index of array
84     integer bnd2max
85
86     ! Calculate array size
87     bnd2max = int(ceiling((xmax-xmin)/dx))
88 end function
89
90 ! Create array from bounds and number of
91 ! elements
92 ! xmax is not included in array
93 function bnd2arr(xmin,xmax,imax)
94     implicit none
95
96     ! INPUTS:
97     ! xmin - minimum x value in array
98     ! xmax - maximum x value in array (exclusive
99     ! )
100    double precision, intent(in) :: xmin, xmax
101    ! imax - number of elements in array
102    integer imax
103
104    ! OUTPUT:
105    ! bnd2arr - array to generate
106    double precision, dimension(imax) :: bnd2arr
107
108    ! BODY:
109
110    ! Counter
111    integer ii
112    ! Step size
113    double precision dx
114
115    ! Calculate step size
116    dx = (xmax - xmin) / imax
117
118    ! Generate array
119    do ii = 1, imax
120        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
167

```

```

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
183     ! Determine ii
184     do ii = 1, nn
185         if (xx(ii) > x0) then
186             ! We've now gone one index too far
187             .
188             exit
189         end if
190     end do
191
192     ! Determine whether we're on the right
193     ! endpoint
194     if(ii-1 < nn) then
195         ! If this is a legitimate
196         ! interpolation , then
197         ! subtract since we went one index too
198         ! far
199         ii = ii - 1
200
201         ! Calculate slope
202         mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
203             xx(ii))
204
205         ! Return interpolated value
206         interp = yy(ii) + mm * (x0 - xx(ii))
207     else
208         ! If we're actually interpolating the
209         ! right endpoint ,
210         ! then just return it.
211         interp = yy(nn)
212     end if
213
214 end if
215
216
217 end function
218
```

```

209 ! Calculate unshifted position of periodic image
210 ! Assuming xmin, xmax are extreme attainable
211 ! values of x
212 function shift_mod(x, xmin, xmax)
213     double precision x, xmin, xmax
214     double precision mod_part, shift_mod
215     mod_part = mod(x-xmin, xmax-xmin)
216     if(mod_part .ge. 0) then
217         ! In this case, mod_part is distance
218         ! between image & lower bound
219         shift_mod = xmin + mod_part
220     else
221         ! In this case, mod_part is distance
222         ! between image & upper bound
223         shift_mod = xmax + mod_part
224     endif
225 end function shift_mod
226
227 ! Bilinear interpolation on evenly spaced 2D
228 ! grid
229 ! Assume upper endpoint is not included and is
230 ! identical
231 ! to the lower endpoint, which is included.
232 function bilinear_array_periodic(x, y, nx, ny,
233     x_vals, y_vals, fun_vals)
234     implicit none
235     double precision x, y
236     integer nx, ny
237     double precision, dimension(:) :: x_vals,
238             y_vals
239     double precision, dimension(:, :) :: fun_vals
240
241     double precision dx, dy, xmin, ymin
242     integer i0, j0, i1, j1
243     double precision x0, x1, y0, y1
244     double precision z00, z10, z01, z11
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,
276 |                                         y0, x1, y1, z00, z01, z10, z11)
277 | end function bilinear_array_periodic
278 |
279 ! Bilinear interpolation on evenly spaced 2D
280 ! grid
281 ! Assume upper and lower endpoints are included
282 function bilinear_array(x, y, x_vals, y_vals,
283                         fun_vals)
284 implicit none
285 double precision x, y
286 double precision, dimension(:) :: x_vals,
287                         y_vals
288 double precision, dimension(:, :) :: fun_vals
289
290 double precision dx, dy, xmin, ymin
291 integer i0, j0, i1, j1
292 double precision x0, x1, y0, y1
293 double precision z00, z10, z01, z11
294
295 double precision bilinear_array
296
297 xmin = x_vals(1)
298 ymin = y_vals(1)
299 dx = x_vals(2) - x_vals(1)
300 dy = y_vals(2) - y_vals(1)
301
302 ! Add 1 for one-indexing
303 i0 = int(floor((x-xmin)/dx))+1
304 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
342 function bilinear(x, y, x0, y0, x1, y1, z00, z01
343                 , z10, z11)
344 implicit none
345 double precision x, y
346 double precision x0, y0, x1, y1, z00, z01, z10
347                 , z11
348 double precision a, b, c, d
349 double precision bilinear
350
351 a = (x-x0)*(y-y0)
352 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) / (
350     a + b + c + d)
350 end function bilinear
351
352 ! Integrate using left endpoint rule
353 ! Assuming the right endpoint is not included in
353 ! 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 *
397             (arr(ii) + arr(ii+1))
398 end do
399
400 end function trap_rule_dx
401
402 ! Integrate using trapezoid rule
403 ! Assuming both endpoints are included in arr
404 function trap_rule_uneven(xx, yy, nn)
405 implicit none
406 double precision, dimension(nn) :: xx
407 double precision, dimension(nn) :: yy
408 integer ii, nn
409 double precision trap_rule_uneven
410
411 trap_rule_uneven = 0.0d0
412
413 do ii=1, nn-1
414     trap_rule_uneven = trap_rule_uneven + 0.5d0
415         * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii
416             +1))
417 end do
418
419 end function trap_rule_uneven
420
421 function trap_rule_dx_uneven(dx, yy, nn)
422 implicit none
423 double precision, dimension(nn-1) :: dx
424 double precision, dimension(nn) :: yy
425 integer ii, nn
426 double precision trap_rule_dx_uneven
427
428 trap_rule_dx_uneven = 0.0d0
429
430 do ii=1, nn-1
431     trap_rule_dx_uneven = trap_rule_dx_uneven +
432         0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
433 end do
434
435 end function trap_rule_dx_uneven
436
437 ! Integrate using midpoint rule
438 ! First and last bins, only use inner half
439 function midpoint_rule_halfends(dx, yy, nn)
440     result(integral)
441 implicit none
442 integer ii, nn
443 double precision, dimension(nn) :: dx, yy
444 double precision integral
445
446 if(nn > 1) then
447     integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(
448         nn))
449
450 end function midpoint_rule_halfends

```

```

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

```

```

490 ! OUTPUT:
491 ! normalize - integral before normalization (
492     left endpoint rule)
493 double precision norm
494
495 ! BODY:
496
497 ! Calculate integral
498 ! PERHAPS WE SHOULD USE TRAPEZOID RULE
499 norm = trap_rule_uneven(xx, yy, nn)
500
501 ! Normalize array
502 yy(:) = yy(:) / norm
503
504 end function normalize_uneven
505
506 ! Read 2D array from file
507 function read_array(filename,fmtstr,nn,mm,
508     skiplines_in)
509     implicit none
510
511     ! INPUTS:
512     ! filename - path to file to be read
513     ! fmtstr - input format (no parentheses, don
514         't specify columns)
515     ! e.g. 'E10.2', not '(2E10.2)'
516     character(len=*), intent(in) :: filename,
517         fmtstr
518     ! nn - Number of data rows in file
519     ! mm - number of data columns in file
520     integer, intent(in) :: nn, mm
521     ! skiplines - optional - number of lines to
522         skip from header
523     integer, optional :: skiplines_in
524     integer skiplines
525
526     ! OUTPUT:
527     double precision, dimension(nn,mm) :: read_array
528
529     ! BODY:
530
531     ! Row counter
532     integer ii
533     ! File unit number
534     integer, parameter :: un = 10
535     ! Final format to use
536     character(len=256) finfmt
537
538     ! Generate final format string
539     write(finfmt,'(A,I1,A,A)') '(', mm, fmtstr,
540         ')'

```

```

535      ! Print message
536      !write(*,*) 'Reading data from '' , trim(
537          filename) , ''
538      !write(*,*) 'using format '' , trim(finfmt) ,
539          ''
540
541      ! Open file
542      open(unit=un, file=trim(filename), status='
543          old', form='formatted')
544
545      ! Skip lines if desired
546      if(present(skiplines_in)) then
547          skip.lines = skip.lines_in
548          do ii = 1, skip.lines
549              ! Read without variable ignores the
550                  line
551              read(un, *)
552          end do
553      else
554          skip.lines = 0
555      end if
556
557      ! Loop through lines
558      do ii = 1, nn
559          ! Read one row at a time
560          read(unit=un, fmt=trim(finfmt))
561          read_array(ii,:)
562      end do
563
564      ! Close file
565      close(unit=un)
566
567  end function
568
569  ! Print 2D array to stdout
570  subroutine print_int_array(arr,nn,mm,fmtstr_in)
571      implicit none
572
573      ! INPUTS:
574      ! arr - array to print
575      integer, dimension(nn,mm), intent(in) :: arr
576      ! nn - number of data rows in file
577      ! nn - number of data columns in file
578      integer, intent(in) :: nn, mm
579      ! fmtstr - output format (no parentheses, don' t specify columns)
580      ! e.g. 'E10.2', not '(2E10.2)'
581      character(len=*), optional :: fmtstr_in
582      character(len=256) fmtstr
583
584      ! NO OUTPUTS

```

```

581      ! BODY
582
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(finfo, '(A,I4,A,A)') '(', mm, trim(
598                                     fmtstr), ')'
599
600      ! Loop through rows
601      do ii = 1, nn
602          ! Print one row at a time
603          write(*,finfo) arr(ii,:)
604      end do
605
606      ! Print blank line after
607      write(*,*) ''
608
609      end subroutine print_int_array
610
611      subroutine print_array(arr,nn,mm,fmtstr_in)
612          implicit none
613
614          ! INPUTS:
615          ! arr - array to print
616          double precision, dimension (nn,mm), intent(
617              in) :: arr
618          ! nn - number of data rows in file
619          ! nn - number of data columns in file
620          integer, intent(in) :: nn, mm
621          ! fmtstr - output format (no parentheses,
622          !         don't specify columns)
623          ! e.g. 'E10.2', not '(2E10.2)'
624          character(len=*), optional :: fmtstr_in
625          character(len=256) fmtstr
626
627          ! NO OUTPUTS
628
629          ! BODY
630
631          ! Row counter
632          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(
642             fmtstr), ')'
643
644 ! Loop through rows
645 do ii = 1, nn
646     ! Include row number
647     !write(*,'(I10)', advance='no') ii
648     ! Print one row at a time
649     write(*,finfmt) arr(ii,:)
650 end do
651
652 ! Print blank line after
653 write(*,*) ''
654
655 end subroutine
656
657 ! Write 2D array to file
658 subroutine write_array(arr,nn,mm,filename,
659                         fmtstr_in)
660     implicit none
661
662     ! INPUTS:
663     ! arr - array to print
664     double precision, dimension (nn,mm), intent(
665         in) :: arr
666     ! nn - number of data rows in file
667     ! nn - number of data columns in file
668     integer, intent(in) :: nn, mm
669     ! filename - file to write to
670     character(len=*) filename
671     ! fmtstr - output format (no parentheses,
672     !          don't specify columns)
673     ! e.g. 'E10.2', not '(2E10.2)'
674     character(len=*), optional :: fmtstr_in
675     character(len=256) fmtstr
676
677     ! NO OUTPUTS
678
679     ! BODY
680
681     ! 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='
686         replace', form='formatted')
687
688     ! Determine string format
689     if(present(fmtstr_in)) then
690         fmtstr = fmtstr_in
691     else
692         fmtstr = 'E10.2'
693     end if
694
695     ! Generate final format string
696     write(finfmt,'(A,I4,A,A)') '(', mm, trim(
697         fmtstr), ')'
698
699     ! Loop through rows
700     do ii = 1, nn
701         ! Print one row at a time
702         write(un,finfmt) arr(ii,:)
703     end do
704
705     ! Close file
706     close(unit=un)
707
708 end subroutine
709
710 subroutine zeros(x, n)
711     implicit none
712     integer n, i
713     double precision, dimension(n) :: x
714
715     do i=1, n
716         x(i) = 0
717     end do
718 end subroutine zeros
719
720 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 ! )
15 ! call grid%init()
16 ! ...
17 ! call grid%deinit()
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
50         :: vals, weights, sin, cos
51 contains
52     procedure :: set_bounds => angle_set_bounds
53     procedure :: set_num => angle1d_set_num
54     procedure :: deinit => angle1d_deinit
55     procedure :: integrate_points =>
56         angle1d_integrate_points
57     procedure :: integrate_func =>
58         angle1d_integrate_func
59     procedure :: assign_linspace =>
60         angle1d_assign_linspace
61     procedure :: assign_legendre
62 end type angle_dim
63
64 type space_dim
65     integer num
66     double precision minval, maxval
67     double precision, dimension(:), allocatable
68         :: vals, edges, spacing
69 contains
70     procedure :: integrate_points =>
71         space_integrate_points
72     procedure :: trapezoid_rule
73     procedure :: set_bounds => space_set_bounds
74     procedure :: set_num => space_set_num
75     procedure :: set_uniform_spacing =>
76         space_set_uniform_spacing
77 !procedure :: set_num_from_spacing
78     procedure :: set_uniform_spacing_from_num
79     procedure :: set_spacing_array =>
80         space_set_spacing_array
81     procedure :: deinit => space_deinit
82     procedure :: assign_linspace
83 end type space_dim
84
85 type space_angle_grid !(sag)
86     type(space_dim) :: x, y, z
87     type(angle2d) :: angles
88     double precision, dimension(:), allocatable :: 
89         x_factor, y_factor
90 contains
91     procedure :: set_bounds => sag_set_bounds
92     procedure :: set_num => sag_set_num
93     procedure :: init => sag_init
94     procedure :: deinit => sag_deinit

```

```

86 !procedure :: set_num_from_spacing =>
87   sag_set_num_from_spacing
88 procedure :: set_uniform_spacing_from_num =>
89   sag_set_uniform_spacing_from_num
90 procedure :: calculate_factors =>
91   sag_calculate_factors
92 end type space_angle_grid
93 contains
94 subroutine index_list_init(indices)
95   class(index_list) indices
96   indices%i = 1
97   indices%j = 1
98   indices%k = 1
99   indices%p = 1
100 end subroutine
101 subroutine index_list_print(indices)
102   class(index_list) indices
103
104   write(*,*) 'i, j, k, p =', indices%i,
105     indices%j, indices%k, indices%p
106 end subroutine index_list_print
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 + l + 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%
180 |             nomega))
180 |     allocate(angles%sin_theta_edge_p(angles%
181 |             nomega))
181 |     allocate(angles%cos_phi_edge_p(angles%nomega
182 |             -1))
182 |     allocate(angles%sin_phi_edge_p(angles%nomega
183 |             -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)*
195 |         angles%dtheta
195 |     angles%cos_theta_edge(l) = cos(angles%
196 |         theta_edge(l))
196 |     angles%sin_theta_edge(l) = sin(angles%
196 |         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)*
204 |             angles%dphi
205 |         angles%cos_phi_edge(m) = cos(angles%
205 |             phi_edge(m))
206 |         angles%sin_phi_edge(m) = sin(angles%
206 |             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
213 |                 %cos_phi_edge(m))
214 |     do l=1, angles%ntheta

```

```

215      p = angles%phat(l, m)
216
217      angles%theta_p(p) = angles%theta(l)
218      angles%phi_p(p) = angles%phi(m)
219      angles%theta_edge_p(p) = angles%
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(l)
254      angles%theta_edge_p(p) = angles%theta_edge(l
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%
253 |         theta_edge_p(p))
254 |     angles%sin_theta_edge_p(p) = sin(angles%
255 |         theta_edge_p(p))
256 |     angles%cos_phi_edge_p(p) = cos(angles%
257 |         phi_edge_p(p))
258 |     angles%sin_phi_edge_p(p) = sin(angles%
259 |         phi_edge_p(p))
260 |     angles%area_p(p) = area
261 |
262 | ! South Pole
263 | p = angles%nomega
264 | m = angles%nphi
265 | angles%theta_p(p) = angles%theta(l)
266 | angles%theta_edge_p(p) = angles%theta_edge(l
267 | )
268 | angles%phi_p(p) = angles%phi(m)
269 | angles%cos_theta_p(p) = cos(angles%theta_p(p
270 | ))
271 | angles%sin_theta_p(p) = sin(angles%theta_p(p
272 | ))
273 | angles%cos_phi_p(p) = cos(angles%phi_p(p))
274 | angles%sin_phi_p(p) = sin(angles%phi_p(p))
275 | angles%area_p(p) = area
276 | end subroutine angle_init
277 |
278 | ! Integrate function given function values at
279 | grid cells
280 | function angle_integrate_points(angles,
281 |     func_vals) result(integral)
282 |     class(angle2d) :: angles
283 |     double precision, dimension(angles%nomega)
284 |         :: func_vals
285 |     double precision integral
286 |     integer p
287 |
288 |     integral = 0.d0
289 |
290 |     do p=1, angles%nomega
291 |         integral = integral + angles%area_p(p) *
292 |             func_vals(p)
293 |     end do
294 |
295 | end function angle_integrate_points
296 |
297 | function angle_integrate_func(angles,
298 |     func_callable) result(integral)
299 |     class(angle2d) :: angles
300 |     double precision, external :: func_callable

```

```

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

```

```

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 |
390 | end subroutine assign_legendre
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 |   end do
408 |
409 |   integral = angle%integrate_points(func_vals)
410 |
411 |   deallocate(func_vals)
412 | end function angle1d_integrate_func
413 |
414 | ! Integrate function given function values
415 |   sampled at legendre theta values
416 | function angle1d_integrate_points(angle,
417 |                                     func_vals) result(integral)
418 |   class(angle_dim) :: angle
419 |   double precision, dimension(angle%num) ::
420 |     func_vals
421 |   double precision integral
422 |
423 |   integral = angle%prefactor * sum(angle%
424 |                                         weights * func_vals)
425 | end function angle1d_integrate_points
426 |
427 | subroutine angle1d_deinit(angle)
428 |   class(angle_dim) :: angle
429 |   deallocate(angle%vals)
430 |   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
432 ! sampled at even grid points
433 function space_integrate_points(space,
434     func_vals) result(integral)
435     class(space_dim) :: space
436     double precision, dimension(space%num) :: func_vals
437     double precision integral
438
439 ! Encapsulate actual method for easy
440 ! switching
441     integral = space%trapezoid_rule(func_vals)
442 end function space_integrate_points
443
444 function trapezoid_rule(space, func_vals)
445     result(integral)
446     class(space_dim) :: space
447     double precision, dimension(space%num) :: func_vals
448     double precision integral
449
450     integral = 0.5d0 * sum(func_vals * space%
451 !         spacing)
452 end function
453
454 subroutine space_set_bounds(space, minval,
455 !     maxval)
456     class(space_dim) :: space
457     double precision minval, maxval
458     space%minval = minval
459     space%maxval = maxval
460 end subroutine space_set_bounds
461
462 subroutine space_set_num(space, num)
463     class(space_dim) :: space
464     integer num
465     space%num = num
466 end subroutine space_set_num
467
468 subroutine space_set_uniform_spacing(space,
469 !     spacing)
470     class(space_dim) :: space
471     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) :: space
476      spacing = space%spacing
477  end subroutine space_set_spacing_array
478
479  subroutine assign_linspace(space)
480      class(space_dim) :: space
481      double precision spacing
482      integer i
483
484      allocate(space%vals(space%num))
485      allocate(space%edges(space%num))
486      allocate(space%spacing(space%num))
487
488      spacing = spacing_from_num(space%minval,
489          space%maxval, space%num)
490      call space%set_uniform_spacing(spacing)
491
492      do i=1, space%num
493          space%edges(i) = space%minval + dble(i-1)
494              * space%spacing(i)
495          space%vals(i) = space%minval + dble(i-0.5
496              d0) * space%spacing(i)
497      end do
498
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
510  end subroutine set_uniform_spacing_from_num
511
512  ! subroutine set_num_from_spacing(space)
513  !     class(space_dim) :: space

```

```

509   !      !space%num = num_from_spacing(space%minval
510   , space%maxval, space%spacing)
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,
523   ymin, ymax, zmin, zmax)
524   class(space_angle_grid) :: grid
525   double precision xmin, xmax, ymin, ymax,
526   zmin, zmax
527   call grid%x%set_bounds(xmin, xmax)
528   call grid%y%set_bounds(ymin, ymax)
529   call grid%z%set_bounds(zmin, zmax)
530 end subroutine sag_set_bounds
531
532 subroutine sag_set_uniform_spacing(grid, dx,
533   dy, dz)
534   class(space_angle_grid) :: grid
535   double precision dx, dy, dz
536   call grid%x%set_uniform_spacing(dx)
537   call grid%y%set_uniform_spacing(dy)
538   call grid%z%set_uniform_spacing(dz)
539 end subroutine sag_set_uniform_spacing
540
541 subroutine sag_set_num(grid, nx, ny, nz,
542   ntheta, nphi)
543   class(space_angle_grid) :: grid
544   integer nx, ny, nz, ntheta, nphi
545   call grid%x%set_num(nx)
546   call grid%y%set_num(ny)
547   call grid%z%set_num(nz)
548   call grid%angles%set_num(ntheta, nphi)
549 end subroutine sag_set_num
550
551 subroutine sag_init(grid)
552   class(space_angle_grid) :: grid
553
554   call grid%x%assign_linspace()
555   call grid%y%assign_linspace()
556   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
593 end subroutine
594 sag_set_uniform_spacing_from_num
595
596 ! subroutine sag_set_num_from_spacing(grid)
597 !     class(space_angle_grid) :: grid
598 !     call grid%x%set_num_from_spacing()
599 !     call grid%y%set_num_from_spacing()
600 !     call grid%z%set_num_from_spacing()
601
602 ! end subroutine sag_set_num_from_spacing
603
604 subroutine sag_deinit(grid)
605 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]
610 subroutine affine_transform(x, xmin, xmax,
611      ymin, ymax)
611     double precision x, xmin, xmax, ymin, ymax
612     x = ymin + (ymax-ymin)/(xmax-xmin) * (x-xmin
613      )
613 end subroutine affine_transform
614
615 function num_from_spacing(xmin, xmax, dx)
616     result(n)
616     double precision xmin, xmax, dx
617     integer n
618     n = floor( (xmax - xmin) / dx )
619 end function num_from_spacing
620
621 function spacing_from_num(xmin, xmax, nx)
621     result(dx)
622     double precision xmin, xmax, dx
623     integer nx
624     dx = (xmax - xmin) / dble(nx)
625 end function spacing_from_num
626
end module sag

```

```

kelp3d.f90
1 ! Kelp 3D
2 ! Oliver Evans
3 ! 8/31/2017
4
5 ! Given superindividual/water current data at
5 ! each depth, generate kelp distribution at
5 ! each point in 3D space
6
7 module kelp3d
8
9 use kelp_context
10
11 implicit none
12
13 contains
14

```

```

15 subroutine generate_grid(xmin, xmax, nx, ymin,
16   ymax, ny, zmin, zmax, nz, ntheta, nphi, grid,
17   p_kelp)
18   double precision xmin, xmax, ymin, ymax, zmin,
19   zmax
20   integer nx, ny, nz, ntheta, nphi
21   type(space_angle_grid) grid
22   double precision, dimension(:,:,:),
23   allocatable :: p_kelp
24
25   call grid%set_bounds(xmin, xmax, ymin, ymax,
26     zmin, zmax)
27   call grid%set_num(nx, ny, nz, ntheta, nphi)
28
29   allocate(p_kelp(nx,ny,nz))
30
31 end subroutine generate_grid
32
33 subroutine kelp3d_deinit(grid, rope, p_kelp)
34   type(space_angle_grid) grid
35   type(rope_state) rope
36   double precision, dimension(:,:,:),
37   allocatable :: p_kelp
38   call rope%deinit()
39   call grid%deinit()
40   deallocate(p_kelp)
41 end subroutine kelp3d_deinit
42
43 subroutine calculate_kelp_on_grid(grid, p_kelp,
44   frond, rope, quadrature_degree)
45   type(space_angle_grid), intent(in) :: grid
46   type(frond_shape), intent(in) :: frond
47   type(rope_state), intent(in) :: rope
48   type(point3d) point
49   integer, intent(in) :: quadrature_degree
50   double precision, dimension(grid%x%num, grid%y
51     %num, grid%z%num) :: p_kelp
52   type(depth_state) depth
53
54   integer i, j, k, nx, ny, nz
55   double precision x, y, z
56
57   nx = grid%x%num
58   ny = grid%y%num
59   nz = grid%z%num
60
61   do k=1, nz
62     z = grid%z%vals(k)
63     call depth%set_depth(rope, grid, k)
64     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,
113   theta_high_lim, quadrature_degree, point,
114   frond, depth) result(integral)
115 type(point3d), intent(in) :: point
116 type(frond_shape), intent(in) :: frond
117 double precision, intent(in) :: theta_low_lim,
     theta_high_lim
118 integer, intent(in) :: quadrature_degree
119 type(depth_state), intent(in) :: depth
120 double precision integral
121 double precision, dimension(:), allocatable :: integrand_vals
122 integer i
123
124 type(angle_dim) :: theta_f
125 call theta_f%set_bounds(theta_low_lim,
     theta_high_lim)
126 call theta_f%set_num(quadrature_degree)
127 call theta_f%assign_legendre()
128
129 allocate(intrand_vals(theta_f%num))
130 do i=1, theta_f%num
131   integrand_vals(i) = ps_integrand(theta_f%
     vals(i), point, frond, depth)
132 end do
133
134 integral = theta_f%integrate_points(
     integrand_vals)
135 deallocate(intrand_vals)

```

```

136   call theta_f%deinit()
137
138 end function integrate_ps
139
140 function ps_integrand(theta_f, point, frond,
141   depth)
142   type(point3d), intent(in) :: point
143   type(frond_shape), intent(in) :: frond
144   type(depth_state), intent(in) :: depth
145   double precision theta_f, l_min
146   double precision angular_part, length_part
147   double precision ps_integrand
148
149   l_min = min_shading_length(theta_f, point,
150     frond)
151
152   angular_part = depth%angle_distribution_pdf(
153     theta_f)
154   length_part = 1 - depth%
155     length_distribution_cdf(l_min)
156
157   ps_integrand = angular_part * length_part
158 end function ps_integrand
159
160
161 function min_shading_length(theta_f, point,
162   frond) result(l_min)
163 ! L_min(\theta)
164   type(point3d), intent(in) :: point
165   type(frond_shape), intent(in) :: frond
166   double precision, intent(in) :: theta_f
167   double precision l_min
168   double precision tpp
169   double precision frond_frac
170
171   ! tpp === theta_p_prime
172   tpp = point%theta - theta_f + pi / 2.d0
173   frond_frac = 2.d0 * frond%fr / (1.d0 + frond%
174     fs)
175   l_min = point%r * (sin(tpp) + angular_sign(tpp)
176     ) * frond_frac * cos(tpp))
177 end function min_shading_length
178
179 ! function frond_edge(theta, theta_f, L, fs, fr)
180 ! ! r_f(\theta)
181 !   double precision, intent(in) :: theta,
182     theta_f, L, fs, fr
183 !   double precision, intent(out) :: frond_edge
184 !
185 !   frond_edge = relative_frond_edge(theta -
186     theta_f + pi/2.d0)

```

```

178 ! end function frond_edge
179 !
180 !
181 ! function relative_frond_edge(theta_prime, L,
182 !   fs, fr)
182 ! ! r_f'(\theta')
183 !   double precision, intent(in) :: theta_prime,
183 !   L, fs, fr
184 !   double precision, intent(out) :::
184 !   relative_frond_edge
185 !
186 !   relative_frond_edge = L / (sin(theta_prime)
186 !     + angular_sign(theta_prime * alpha(fs, fr) *
186 !       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
194 ! of 9/9/18
195 ! In the report, it's written as sgn(
195 !   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,
21      z_block_size, omega_block_size
22   double precision, dimension(:), allocatable
23      :: surface_vals
24   ! A stored in coordinate form in row, col,
25      data
26   integer, dimension(:), allocatable :: row,
27      col
28   double precision, dimension(:), allocatable
29      :: data
30   ! b and x stored in rhs in full form
31   double precision, dimension(:), allocatable
32      :: rhs, sol
33
34   ! Pointer to solver subroutine
35   ! Set to mgmres by default
36   procedure(solver_interface), pointer, nopass
37      :: solver => mgmres_st
38
39 contains
40   procedure :: init => mat_init
41   procedure ::_deinit => mat_deinit
42   procedure :: calculate_size
43   procedure :: set_solver_params =>
44      mat_set_solver_params
45   procedure :: assign => mat_assign
46   procedure :: add => mat_add
47   procedure :: assign_rhs => mat_assign_rhs
48   !procedure :: store_index => mat_store_index
49   !procedure :: find_index => mat_find_index
50   procedure :: set_bc => mat_set_bc
51   procedure :: solve => mat_solve
52   procedure :: ind => mat_ind
53   !procedure :: to_hdf => mat_to_hdf
54   procedure attenuate
55   procedure angular_integral
56
57   ! Derivative subroutines
58   procedure x_cd2
59   procedure x_cd2_first
60   procedure x_cd2_last
61   procedure y_cd2
62   procedure y_cd2_first
63   procedure y_cd2_last
64   procedure z_cd2
65   procedure z_fd2
66   procedure z_bd2
67   procedure z_surface_bc
68   procedure z_bottom_bc

```

```

64  end type rte_mat
65
66 interface
67   ! Define interface for external procedure
68   ! https://stackoverflow.com/questions/
69   ! /8549415/how-to-declare-the-interface-
70   ! section-for-a-procedure-argument-which-in-
71   ! turn-ref
72   subroutine solver_interface(n_total, nonzero,
73     row, col, data, &
74     sol, rhs, maxiter_outer, maxiter_inner,
75     &
76     tol_abs, tol_rel)
77   integer :: n_total, nonzero
78   integer, dimension(nonzero) :: row, col
79   double precision, dimension(nonzero) :: data
80   double precision, dimension(nonzero) :: sol
81   double precision, dimension(n_total) :: rhs
82   integer :: maxiter_outer, maxiter_inner
83   double precision :: tol_abs, tol_rel
84   end subroutine solver_interface
85 end interface
86
87 contains
88
89 subroutine mat_init(mat, grid, iops)
90   class(rte_mat) mat
91   type(space_angle_grid) grid
92   type(optical_properties) iops
93   integer nnz, n_total
94
95   mat%grid = grid
96   mat%iops = iops
97
98   call mat%calculate_size()
99
100  n_total = mat%n_total
101  nnz = mat%nonzero
102  allocate(mat%surface_vals(grid%angles%omega
103    ))
104  allocate(mat%row(nnz))
105  allocate(mat%col(nnz))
106  allocate(mat%data(nnz))
107  allocate(mat%rhs(n_total))
108  allocate(mat%sol(n_total))
109
110  call zeros(mat%rhs, n_total)
111  call zeros(mat%sol, n_total)
112
113 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   !mat%omega_block_size = 1
137   !mat%y_block_size = mat%omega_block_size *
138   !mat%x_block_size = mat%y_block_size * ny
139   !mat%z_block_size = mat%x_block_size * nx
140
141 ! subroutine mat_to_hdf(mat,filename)
142 !   class(rte_mat) mat
143 !   character(len=*) filename
144 !   call write_coo(filename, mat%row, mat%col,
145 !     mat%data, mat%nonzero)
146 ! end subroutine mat_to_hdf
147
148 subroutine mat_set_bc(mat, bc)
149   class(rte_mat) mat
150   class(boundary_condition) bc
151   integer p
152
153   do p=1, mat%grid%angles%nomega/2

```

```

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

```

```

196    end subroutine mat_solve
197
198
199 subroutine mat_set_solver_params(mat ,
200     maxiter_outer , &
201         maxiter_inner , tol_abs , tol_rel)
202     class(rte_mat) mat
203     integer maxiter_outer , maxiter_inner
204     double precision tol_abs , tol_rel
205
206     mat%params%maxiter_outer = maxiter_outer
207     mat%params%maxiter_inner = maxiter_inner
208     mat%params%tol_abs = tol_abs
209     mat%params%tol_rel = tol_rel
210 end subroutine mat_set_solver_params
211
212 function mat_ind(mat , i , j , k , p) result(ind)
213     ! Assuming var ordering: z , x , y , omega
214     class(rte_mat) mat
215     integer i , j , k , p
216     integer ind
217
218     ind = (i-1) * mat%x_block_size + (j-1) * mat%
219         %y_block_size + &
220             (k-1) * mat%z_block_size + p * mat%
221                 omega_block_size
222 end function mat_ind
223
224 subroutine mat_assign(mat , row_num , ent , val ,
225     i , j , k , p)
226     ! It's assumed that this is the only time
227         this entry is defined
228     class(rte_mat) mat
229     double precision val
230     integer i , j , k , p
231     integer row_num , ent
232
233     mat%row(ent) = row_num
234     mat%col(ent) = mat%ind(i , j , k , p)
235     mat%data(ent) = val
236
237     ent = ent + 1
238 end subroutine mat_assign
239
240 subroutine mat_add(mat , repeat_ent , val)
241     ! Use this when you know that this entry has
242         already been assigned
243     ! and you'd like to add this value to the
244         existing value.
245
246     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)
245      + val
246  end subroutine mat_add
247
248 subroutine mat_assign_rhs(mat, row_num, data)
249   class(rte_mat) mat
250   double precision data
251   integer row_num
252
253   mat%rhs(row_num) = data
254 end subroutine mat_assign_rhs
255
256 ! subroutine mat_store_index(mat, row_num,
257 !   col_num)
258 !   ! Remember where we stored information for
259 !   ! this matrix element
260 !   ! class(rte_mat) mat
261 !   ! integer row_num, col_num
262 !   ! mat%index_map(row_num, col_num) = mat%ent
263 ! end subroutine
264
265 ! function mat_find_index(mat, row_num,
266 !   col_num) result(index)
267 !   ! Find the position in row, col, data
268 !   ! where this entry
269 !   ! is defined.
270 !   ! class(rte_mat) mat
271 !   ! integer row_num, col_num, index
272
273 !   index = mat%index_map(row_num, col_num)
274
275 !   ! This took up 95% of execution time.
276 !   ! Only search up to most recently assigned
277 !   ! index
278 !   ! do index=1, mat%ent-1
279 !   !   if( (mat%row(index) .eq. row_num) .
280 !   and. (mat%col(index) .eq. col_num)) then
281 !   !       exit
282 !   !   end if
283 !   ! end do
284 ! end function mat_find_index
285
286 subroutine attenuate(mat, indices, repeat_ent)
287   ! Has to be called after angular_integral
288   ! Because they both write to the same matrix
289   ! entry
290   ! And adding here is more efficient than a
291   ! 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,
291     indices%k)
292 bb = mat%iops%scat
293 attenuation = aa + bb
294
295 call mat%add(repeat_ent, attenuation)
296 end subroutine attenuate
297
298 subroutine x_cd2(mat, indices, row_num, ent)
299 class(rte_mat) mat
300 double precision val, dx
301 type(index_list) indices
302 integer i, j, k, p
303 integer row_num, ent
304
305 i = indices%i
306 j = indices%j
307 k = indices%k
308 p = indices%p
309
310 dx = mat%grid%x%spacing(1)
311
312 val = mat%grid%angles%sin_phi_p(p) &
313     * mat%grid%angles%cos_theta_p(p) / (2.
314         d0 * dx)
315
316 call mat%assign(row_num, ent, -val, i-1, j, k, p)
317 call mat%assign(row_num, ent, val, i+1, j, k, p)
318 end subroutine x_cd2
319
320 subroutine x_cd2_first(mat, indices, row_num,
321     ent)
322 class(rte_mat) mat
323 double precision val, dx
324 integer nx
325 type(index_list) indices
326 integer i, j, k, p
327 integer row_num, ent
328
329 i = indices%i
330 j = indices%j
331 k = indices%k
332 p = indices%p

```

```

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.
336       d0 * dx)
337
338   call mat%assign(row_num,ent,-val,nx,j,k,p)
339   call mat%assign(row_num,ent,val,i+1,j,k,p)
340 end subroutine x_cd2_first
341
342 subroutine x_cd2_last(mat, indices, row_num,
343   ent)
344   class(rte_mat) mat
345   double precision val, dx
346   type(index_list) indices
347   integer i, j, k, p
348   integer row_num, ent
349
350   i = indices%i
351   j = indices%j
352   k = indices%k
353   p = indices%p
354
355   dx = mat%grid%x%spacing(1)
356
357   val = mat%grid%angles%sin_phi_p(p) &
358     * mat%grid%angles%cos_theta_p(p) / (2.
359       d0 * dx)
360
361   call mat%assign(row_num,ent,-val,i-1,j,k,p)
362   call mat%assign(row_num,ent,val,1,j,k,p)
363 end subroutine x_cd2_last
364
365 subroutine y_cd2(mat, indices, row_num, ent)
366   class(rte_mat) mat
367   double precision val, dy
368   type(index_list) indices
369   integer i, j, k, p
370   integer row_num, ent
371
372   i = indices%i
373   j = indices%j
374   k = indices%k
375   p = indices%p
376
377   dy = mat%grid%y%spacing(1)
378
379   val = mat%grid%angles%sin_phi_p(p) &

```

```

377      * mat%grid%angles%sin_theta_p(p) / (2.
378      d0 * dy)
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,
384   ent)
385   class(rte_mat) mat
386   double precision val, dy
387   integer ny
388   type(index_list) indices
389   integer i, j, k, p
390   integer row_num, ent
391
392   i = indices%i
393   j = indices%j
394   k = indices%k
395   p = indices%p
396
397   dy = mat%grid%y%spacing(1)
398   ny = mat%grid%y%num
399
400   val = mat%grid%angles%sin_phi_p(p) &
401       * mat%grid%angles%sin_theta_p(p) / (2.
402       d0 * dy)
403
404   call mat%assign(row_num,ent,-val,i,ny,k,p)
405   call mat%assign(row_num,ent,val,i,j+1,k,p)
406 end subroutine y_cd2_first
407
408 subroutine y_cd2_last(mat, indices, row_num,
409   ent)
410   class(rte_mat) mat
411   double precision val, dy
412   type(index_list) indices
413   integer i, j, k, p
414   integer row_num, ent
415
416   i = indices%i
417   j = indices%j
418   k = indices%k
419   p = indices%p
420
421   dy = mat%grid%y%spacing(1)
422
423   val = mat%grid%angles%sin_phi_p(p) &
424       * mat%grid%angles%sin_theta_p(p) / (2.
425       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,
477   repeat_ent)
478 ! Has to be called after angular_integral
479 ! Because they both write to the same matrix
480 ! entry
481 ! And adding here is more efficient than a
482 ! conditional
483 ! in the angular loop.
484 class(rte_mat) mat
485 double precision val, val1, val2, val3, dz
486 type(index_list) indices
487 integer i, j, k, p
488 integer row_num, ent, repeat_ent
489
490 i = indices%i
491 j = indices%j
492 k = indices%k
493 p = indices%p
494
495 dz = mat%grid%z%spacing(indices%k)
496
497 val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
498   dz)
499
500 val1 = 3.d0 * val
501 val2 = -4.d0 * val
502 val3 = val
503
504 call mat%add(repeat_ent, val1)
505 call mat%assign(row_num, ent, val2, i, j, k-1, p)
506 call mat%assign(row_num, ent, val3, i, j, k-2, p)
507 end subroutine z_bd2
508
509 subroutine angular_integral(mat, indices,
510   row_num, ent)
511 class(rte_mat) mat
512 ! Primed angular integration variables
513 integer pp
514 double precision val
515 type(index_list) indices
516 integer row_num, ent
517
518 ! Interior

```

```

514   do pp=1, mat%grid%angles%nomega
515     ! TODO: Make sure I don't have p and pp
516     ! backwards
517     val = -mat%iops%scat * mat%iops%
518       vsf_integral(indices%p, pp)
519     call mat%assign(row_num, ent, val,
520                   indices%i, indices%j, indices%k, pp)
521   end do
522 end subroutine angular_integral

523 subroutine z_surface_bc(mat, indices, row_num,
524   ent, repeat_ent)
525   class(rte_mat) mat
526   double precision bc_val
527   type(index_list) indices
528   double precision val1, val2, dz
529   integer row_num, ent, repeat_ent
530
531   dz = mat%grid%z%spacing(1)
532
533   val1 = mat%grid%angles%cos_phi_p(indices%p)
534     / (5.d0 * dz)
535   val2 = 7.d0 * val1
536   bc_val = 8.d0 * val1 * mat%surface_vals(
537     indices%p)
538
539   call mat%assign(row_num, ent, val1, indices%i,
540     indices%j, 2, indices%p)
541   call mat%add(repeat_ent, val2)
542   call mat%assign_rhs(row_num, bc_val)
543
544 end subroutine z_surface_bc

545 subroutine z_bottom_bc(mat, indices, row_num
546   , ent, repeat_ent)
547   class(rte_mat) mat
548   type(index_list) indices
549   double precision val1, val2, dz
550   integer nz
551   integer row_num, ent, repeat_ent
552
553   dz = mat%grid%z%spacing(1)
554   nz = mat%grid%z%num
555
556   val1 = -mat%grid%angles%cos_phi_p(indices%p)
557     / (5.d0 * dz)
558   val2 = 7.d0 * val1
559
560   call mat%assign(row_num, ent, val1, indices%i,
561     indices%j, nz-1, indices%p)
562   call mat%add(repeat_ent, val2)

```

```

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

```

```

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

```

```

85      !$omp parallel do default(none) shared(
86          mat) private(j) &
87          !$omp firstprivate(indices,ddx,ddy,
88              angle_loop, i, k) &
89          !$omp shared(num_threads_x,num_threads_y
90              ,num_threads_z) &
91          !$omp num_threads(num_threads_y) if(
92              num_threads_y .gt. 1)
93      do j=1, mat%grid%y%num
94          indices%j = j
95          if(indices%j .eq. 1) then
96              ddy => y_cd2_first
97          else if(indices%j .eq. mat%grid%y%num
98                  ) then
99              ddy => y_cd2_last
100         else
101             ddy => y_cd2
102         end if
103
104         call angle_loop(mat, indices, ddx,
105                         ddy)
106     end do
107
108     !$omp end parallel do
109   end do
110
111   !$omp end parallel do
112 end do
113
114   !$omp end parallel do
115 end subroutine whole_space_loop
116
117
118
119
120
121
122
123

```

```

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,
209 |         repeat_ent)
210 |     call mat%attenuate(indices, repeat_ent)
211 |     row_num = row_num + 1
212 |   end do
213 | end subroutine surface_angle_loop
214 subroutine bottom_angle_loop(mat, indices, ddx,
215 |     ddy)
216 type(rte_mat) mat
217 type(index_list) indices
218 integer p
219 integer row_num, ent, repeat_ent
220 procedure(deriv_interface) :: ddx, ddy
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,
225 |     indices%k, indices%p)
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,
231 |         ent)
232 |     call ddx(mat, indices, row_num, ent)
233 |     call ddy(mat, indices, row_num, ent)
234 |     call mat%z_bd2(indices, row_num, ent,
235 |         repeat_ent)
236 |     call mat%attenuate(indices, repeat_ent)
237 |     row_num = row_num + 1
238 ! Upwelling
239 do p=mat%grid%angles%nomega/2+1, mat%grid%
240 |     angles%nomega
241 |     indices%p = p
242 |     repeat_ent = calculate_repeat_ent(ent, p)
243 |     call mat%angular_integral(indices, row_num,
244 |         ent)
245 |     call ddx(mat, indices, row_num, ent)
246 |     call ddy(mat, indices, row_num, ent)
247 |     call mat%z_bottom_bc(indices, row_num, ent,
248 |         repeat_ent)
249 |     call mat%attenuate(indices, repeat_ent)
250 |     row_num = row_num + 1
251 end do
252 end subroutine bottom_angle_loop

```

```

249 | subroutine gen_matrix(mat)
250 |   type(rte_mat) mat
251 |   type(index_list) indices
252 |
253 |   call indices%init()
254 |
255 |   call whole_space_loop(mat, indices)
256 |     ! call surface_space_loop(mat, indices)
257 |     ! call interior_space_loop(mat, indices)
258 |     ! call bottom_space_loop(mat, indices)
259 |
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
271 end subroutine
272
273 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
26          :: frond_lengths, frond_stds, num_fronds,
27          water_speeds, water_angles
28 contains
29     procedure :: init => rope_init
30     procedure :: deinit => rope_deinit
31 end type rope_state
32
33 type depth_state
34     double precision frond_length, frond_std,
35         num_fronds, water_speeds, water_angles,
36         depth
37     integer depth_layer
38 contains
39     procedure :: set_depth
40     procedure :: length_distribution_cdf
41     procedure :: angle_distribution_pdf
42 end type depth_state
43
44 type optical_properties
45     integer num_vsf
46     type(space_angle_grid) grid
47     double precision, dimension(:), allocatable
48         :: vsf_angles, vsf_vals, vsf_cos
49     double precision, dimension(:), allocatable
50         :: abs_water
51     double precision abs_kelp, vsf_scat_coef,
52         scat
53     ! On x, y, z grid - including water & kelp.
54     double precision, dimension(:,:,:,:),
55         allocatable :: abs_grid
56     ! On theta, phi, theta_prime, phi_prime grid
57     double precision, dimension(:,:,,:), allocatable
58         :: vsf, vsf_integral
59 contains
60     procedure :: init => iop_init
61     procedure :: calculate_coef_grids
62     procedure :: load_vsf
63     procedure :: eval_vsf
64     procedure :: calc_vsf_on_grid
65     procedure :: deinit => iop_deinit
66     procedure :: vsf_from_function
67 end type optical_properties
68
69 type boundary_condition
70     double precision I0, decay, theta_s, phi_s
71     type(space_angle_grid) grid
72     double precision, dimension(:), allocatable
73         :: bc_grid
74 contains
75     procedure :: bc_gaussian
76     procedure :: init => bc_init
77     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 |    end do
102 |    ! Zero upwelling light specified at surface
103 |    do p=grid%angles%nomega/2+1, grid%angles%
104 |      nomega
105 |        bc%bc_grid(p) = 0.d0
106 |    end do
107 |
108 |    ! Normalize
109 |    bc%bc_grid = bc%I0 * bc%bc_grid &
110 |                  / grid%angles%integrate_points(bc%
111 |                                              bc_grid)
112 |
113 |  end subroutine bc_init

```

```

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
197 end subroutine calculate_coef_grids
198
199
200 subroutine load_vsf(iops, filename, fmtstr)
201   class(optical_properties) :: iops
202   character(len=*) :: filename, fmtstr
203   double precision, dimension(:,:),
204     allocatable :: tmp_2d_arr
205   integer num_rows, num_cols, skip_lines_in
206
207   ! First column is the angle at which the
208   ! measurement is taken
209   ! Second column is the value of the VSF at
210   ! that angle
211   num_rows = iops%num_vsf
212   num_cols = 2

```

```

204     skiplines_in = 1 ! Ignore comment on first
205     line
206
207     allocate(tmp_2d_arr(num_rows, num_cols))
208
209     tmp_2d_arr = read_array(filename, fmtstr,
210                             num_rows, num_cols, skiplines_in)
211     iops%vsf_angles = tmp_2d_arr(:,1)
212     iops%vsf_vals = tmp_2d_arr(:,2)
213
214     ! write(*,*) 'vsf_angles = ', iops%
215     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
216
217     ! Pre-evaluate for all pair of angles
218     call iops%calc_vsf_on_grid()
219   end subroutine load_vsf
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248

```

`skiplines_in = 1 ! Ignore comment on first
line

allocate(tmp_2d_arr(num_rows, num_cols))

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

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

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

end subroutine load_vsf

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

end function eval_vsf

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

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

end subroutine rope_init

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

end subroutine rope_deinit`

```

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
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289

```

```

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

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

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

    L_mean = depth%frond_length
    L_std = depth%frond_std

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

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

    v_w = depth%water_speeds
    theta_w = depth%water_angles

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

```

```

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
335 ! Avoid out-of-bounds errors due to rounding
336 alpha = min(1.d0, alpha)
337 alpha = max(-1.d0, alpha)
338
339 diff = acos(alpha)
340 end function angle_diff_3d
341
342 subroutine vsf_from_function(iops, func)
343   class(optical_properties) iops
344   double precision, external :: func
345   integer i
346   type(angle_dim) :: angle1d
347
348   call angle1d%set_bounds(-1.d0, 1.d0)
349   call angle1d%set_num(iops%num_vsf)
350   call angle1d%assign_legendre()
351
352   iops%vsf_angles(:) = acos(angle1d%vals(:))
353   do i=1, iops%num_vsf
354     iops%vsf_vals(i) = func(iops%vsf_angles(i))
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_scat_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_scat_coef)
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2598
259
```

```

377      ! write(*,*) 'norm = ', iops%vsf_scat_coef
378      ! write(*,*) 'now: ', trap_rule_uneven(iops%
379          & vsf_cos, iops%vsf_vals, iops%num_vsf)
380      ! write(*,*) 'cos: ', iops%vsf_cos
381      ! write(*,*) 'vals: ', iops%vsf_vals
382
383      do p=1, nomega
384          th = iops%grid%angles%theta_p(p)
385          ph = iops%grid%angles%phi_p(p)
386          do pp=1, nomega
387              thp = iops%grid%angles%theta_p(pp)
388              php = iops%grid%angles%phi_p(pp)
389              ! TODO: Might be better to calculate
390                  average scattering
391              ! from angular cell rather than only
392                  using center
393              iops%vsf(p, pp) = iops%eval_vsf(
394                  angle_diff_3d(th,ph,thp,php))
395
396      end do
397
398      ! Normalize each row of VSF (midpoint
399          & rule)
400      norm = sum(iops%vsf(p,:) * iops%grid%
401          & angles%area_p(:))
402      iops%vsf(p,:) = iops%vsf(p,:) / norm
403
404      ! % / meter light scattered from cell pp
405          & into direction p.
406      ! TODO: Could integrate VSF instead of
407          & just using value at center
408      iops%vsf_integral(p, :) = iops%vsf(p, :)
409          &
410          * iops%grid%angles%area_p(:)
411      ! write(*,*) 'vsf_integral (beta_pp)', p,
412          & ' = ', iops%vsf_integral(p, :)
413
414      ! Normalize VSF on unit sphere w.r.t. north
415          & pole
416      ! iops%vsf_scat_coef = sum(iops%vsf(1,:) *
417          & iops%grid%angles%area_p)
418      ! iops%vsf = iops%vsf / iops%vsf_scat_coef
419      ! iops%vsf_integral = iops%vsf_integral /
420          & iops%vsf_scat_coef
421
422      end subroutine calc_vsf_on_grid
423
424      subroutine iop_deinit(iops)
425          class(optical_properties) iops
426          deallocate(iops%vsf_angles)
427          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 ! Init for use with mat
24 subroutine light_init(light, mat)
25   class(light_state) light
26   type(rte_mat) mat
27   integer nx, ny, nz, nomega
28
29   light%mat = mat
30   light%grid = mat%grid
31
32   nx = light%grid%x%num
33   ny = light%grid%y%num
34   nz = light%grid%z%num
35   nomega = light%grid%angles%nomega
36
37   allocate(light%irradiance(nx, ny, nz))

```

```

38     allocate(light%radiance(nx, ny, nz, nomega))
39 end subroutine light_init
40
41 ! Init for use without mat
42 subroutine light_init_grid(light, grid)
43   class(light_state) light
44   type(space_angle_grid) grid
45   integer nx, ny, nz, nomega
46
47   light%grid = grid
48
49   nx = light%grid%x%num
50   ny = light%grid%y%num
51   nz = light%grid%z%num
52   nomega = light%grid%angles%nomega
53
54   allocate(light%irradiance(nx, ny, nz))
55   allocate(light%radiance(nx, ny, nz, nomega))
56 end subroutine light_init_grid
57
58 subroutine calculate_radiance(light)
59   class(light_state) light
60   integer i, j, k, p
61   integer nx, ny, nz, nomega
62   integer index
63
64   nx = light%grid%x%num
65   ny = light%grid%y%num
66   nz = light%grid%z%num
67   nomega = light%grid%angles%nomega
68
69   index = 1
70
71 ! Set initial guess from provided radiance
72 ! Traverse solution vector in order
73 ! so as to avoid calculating index
74 do k=1, nz
75   do i=1, nx
76     do j=1, ny
77       do p=1, nomega
78         light%mat%sol(index) = light%
79             radiance(i,j,k,p)
80         index = index + 1
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%
98                          mat%sol(index)
99                      index = index + 1
100                 end do
101            end do
102        end do
103    end subroutine calculate_radiance
104
105 subroutine calculate_irradiance(light)
106     class(light_state) light
107     integer i, j, k
108     integer nx, ny, nz
109     double precision, dimension(light%grid%
110         angles%nomega) :: tmp_rad
111
112     nx = light%grid%x%num
113     ny = light%grid%y%num
114     nz = light%grid%z%num
115
116     do i=1, nx
117         do j=1, ny
118             do k=1, nz
119                 ! Use temporary array to avoid
120                 ! creating one
121                 ! implicitly at every spatial grid
122                 ! point
123                 tmp_rad = light%radiance(i,j,k,:)
124                 light%irradiance(i,j,k) = &
125                     light%grid%angles%
126                     integrate_points(tmp_rad)
127             end do
128         end do
129     end do
130
131 end subroutine calculate_irradiance
132
133 ! subroutine light_to_hdf(light, radfile,
134 !     irradfile)
135 !     class(light_state) light
136 !     character(len=*) radfile
137 !     character(len=*) irradfile
138 !

```

```

134 !      call hdf_write_radiance(radfile, light%
135 !          radiance, light%grid)
136 !      call hdf_write_irradiance(irradfile, light%
137 !          irradiance, light%grid)
138 !  end subroutine light_to_hdf
139
140 subroutine light_deinit(light)
141     class(light_state) light
142
143     deallocate(light%irradiance)
144     deallocate(light%radiance)
145 end subroutine light_deinit
146 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(:,:,:,:,:) ,
15          allocatable :: source
16      integer num_scatters
17      integer nx, ny, nz, nomega
18      integer max_cells
19
20      double precision, dimension(:), allocatable
21          :: path_length, path_spacing, a_tilde, gn
22
23      nx = grid%x%num
24      ny = grid%y%num
25      nz = grid%z%num
26      nomega = grid%angles%nomega
27
28      max_cells = calculate_max_cells(grid)
29
30      allocate(path_length(max_cells+1))
31      allocate(path_spacing(max_cells))
32      allocate(a_tilde(max_cells))
33      allocate(gn(max_cells))
34      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
51 end subroutine calculate_light_with_scattering
52
53 subroutine calculate_light_before_scattering(
54     grid, bc, iops, source, radiance,
55     path_length, path_spacing, a_tilde, gn)
56     type(space_angle_grid) grid
57     type(boundary_condition) bc
58     type(optical_properties) iops
59     double precision, dimension(:,:,:,:,:) :: :
60         radiance, source
61     double precision, dimension(:) :: :
62         path_length, path_spacing, a_tilde, gn
63     integer i, j, k, p
64
65 ! !$ integer omp_get_num_procs
66 ! !$ integer num_threads_z, num_threads_x
67
68 ! ! Enable nested parallelism
69 ! !$ call omp_set_nested(.true.)
70
71 ! ! Use nz procs for outer loop,
72 ! ! or num_procs if num_procs < nz
73 ! ! Divide the rest of the tasks as
74 ! ! appropriate
75
76 ! !$ num_threads_z = min(omp_get_num_procs()
77 ! , grid%z%num)
78 ! !$ num_threads_x = min( &
79 ! !$     omp_get_num_procs()/num_threads_z, &
80 ! !$     grid%x%num)
81
82 ! !$omp parallel do default(none) private(i,
83 ! , 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          ! !$omp num_threads(num_threads_x) if(
87              num_threads_x .gt. 1)
88          do i=1, grid%x%num
89              do j=1, grid%y%num
90                  do p=1, grid%angles%nomega/2
91                      ! Downwelling light
92                      call
93                          attenuate_light_from_surface
94                          (&
95                              grid, iops, source, i, j, k,
96                              p,&
97                              radiance, path_length,
98                              path_spacing,&
99                              a_tilde, gn, bc)
100
101                      ! No upwelling light before
102                      scattering
103                      radiance(i,j,k,p+grid%angles%
104                          nomega/2) = 0.d0
105
106                  end do
107              end do
108          end do
109          ! !$omp end parallel do
110      end do
111      ! !$omp end parallel do
112  end subroutine
113      calculate_light_before_scattering
114
115  subroutine attenuate_light_from_surface(&
116      grid, iops, source, i, j, k, p, radiance,
117      &
118      path_length, path_spacing, a_tilde, gn,
119      bc)
120      type(space_angle_grid) grid
121      type(boundary_condition) bc
122      type(optical_properties) iops

```

```

106   double precision, dimension(:,:,:,:,:) ::  

107     radiance, source  

108   double precision, dimension(:) ::  

109     path_length, path_spacing, a_tilde, gn  

110   integer i, j, k, p  

111   integer num_cells  

112   double precision atten  

113  

114 ! Don't need gn here, so just ignore it  

115 call traverse_ray(grid, iops, source, i, j,  

116   k, p, path_length, path_spacing, a_tilde,  

117   gn, num_cells)  

118  

119 ! Start with surface bc and attenuate along  

120 ! path  

121 atten = sum(path_spacing(1:num_cells) *  

122   a_tilde(1:num_cells))  

123 ! Avoid underflow  

124 if(atten .lt. 100.d0) then  

125   radiance(i,j,k,p) = bc%bc_grid(p) * exp(-  

126     atten)  

127 else  

128   radiance(i,j,k,p) = 0.d0  

129 end if  

130  

131 end subroutine attenuate_light_from_surface  

132  

133 subroutine calculate_light_after_scattering(  

134   grid, iops, source, radiance,&  

135   num_scatters, path_length, path_spacing,  

136   a_tilde, gn)  

137 type(space_angle_grid) grid  

138 type(optical_properties) iops  

139 double precision, dimension(:,:,:,:,:) ::  

140   radiance, source  

141   integer num_scatters  

142   double precision, dimension(:) ::  

143     path_length, path_spacing, a_tilde, gn  

144   double precision, dimension(:,:,:,:),  

145     allocatable :: rad_scatter  

146   integer n  

147   double precision bb  

148  

149   allocate(rad_scatter(grid%x%num, grid%y%num,  

150     grid%z%num, grid%angles%nomega))  

151   rad_scatter = radiance  

152   bb = iops%scat  

153  

154   do n=1, num_scatters  

155     write(*,*) 'scatter #', n  

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

```

```

178 |     double precision, dimension(:,:,:,:,:) :: 
179 |         scatter_integral
180 | type(index_list) indices
181 | integer nx, ny, nz, nomega
182 | integer i, j, k, p
183 |
184 | !$ integer omp_get_num_procs
185 | !$ integer num_threads_z, num_threads_x
186 |
187 | nx = grid%x%num
188 | ny = grid%y%num
189 | nz = grid%z%num
190 | nomega = grid%angles%nomega
191 |
192 | ! Enable nested parallelism
193 | !$ call omp_set_nested(.true.)
194 |
195 | ! Use nz procs for outer loop,
196 | ! or num_procs if num_procs < nz
197 | ! Divide the rest of the tasks as
198 |     appropriate
199 |
200 | !$ num_threads_z = min(omp_get_num_procs(),
201 |     grid%z%num)
202 |
203 | !$ num_threads_x = min( &
204 |     omp_get_num_procs()/num_threads_z, &
205 |     grid%x%num)
206 |
207 | !$omp parallel do default(none) private(
208 |     indices) &
209 | !$omp private(i,j,k,p) shared(nx,ny,nz,
210 |     nomega) &
211 | !$omp shared(iops, rad_scatter,
212 |     scatter_integral) &
213 | !$omp shared(num_threads_x) &
214 | !$omp num_threads(num_threads_z) if(
215 |     num_threads_z .gt. 1)
216 | do k=1, nz
217 |     indices%k = k
218 |     !$omp parallel do default(none)
219 |         firstprivate(indices,k) &
220 |     !$omp private(i,j,p) shared(nx,ny,nz,
221 |         nomega) &
222 |     !$omp shared(iops, rad_scatter,
223 |         scatter_integral) &
224 |     !$omp num_threads(num_threads_x) if(
225 |         num_threads_x .gt. 1)
226 |     do i=1, nx
227 |         indices%i = i
228 |         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 write(*,*) 'source: ', sum(source)/size(
234     source), minval(source), maxval(source)
235 end subroutine calculate_source
236
237 subroutine calculate_scatter_integral(iops,
238     rad_scatter, scatter_integral, indices)
239 type(optical_properties) iops
240 double precision, dimension(:,:,:,:) :::
241     rad_scatter, scatter_integral
242 type(index_list) indices
243
244 scatter_integral(indices%i,indices%j,indices
245     %k,indices%p) &
246     = sum(iops%vsf_integral(indices%p, :) &
247         * rad_scatter(&
248             indices%i,&
249             indices%j,&
250                 indices%k,:))
251
252 end subroutine calculate_scatter_integral
253
254 subroutine advect_light(grid, iops, source,
255     rad_scatter, path_length, path_spacing,
256     a_tilde, gn)
257 type(space_angle_grid) grid
258 type(optical_properties) iops
259 double precision, dimension(:,:,:,:) :::
260     rad_scatter, source
261 double precision, dimension(:) :::
262     path_length, path_spacing, a_tilde, gn
263 integer i, j, k, p

```

```

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
267      ! appropriate
268
269      !$ num_threads_z = min(omp_get_num_procs(), grid%z%num)
270      !$ num_threads_x = min( &
271      !$     omp_get_num_procs()/num_threads_z, &
272      !$     grid%x%num)
273
274      !$omp parallel do default(none) &
275      !$omp private(i,j,k,p) &
276      !$omp shared(rad_scatter,source,grid,iops,
277      !     num_threads_x) &
278      !$omp private(path_length,path_spacing,
279      !     a_tilde,gn) &
280      !$omp num_threads(num_threads_z) if(
281      !     num_threads_z .gt. 1)
282      do k=1, grid%z%num
283          !$omp parallel do default(none) &
284          !$omp firstprivate(k) private(i,j,p) &
285          !$omp shared(rad_scatter,source,grid,iops
286          !     ) &
287          !$omp private(path_length,path_spacing,
288          !     a_tilde,gn) &
289          !$omp num_threads(num_threads_x) if(
290          !     num_threads_x .gt. 1)
291          do i=1, grid%x%num
292              do j=1, grid%y%num
293                  do p=1, grid%angles%nomega
294                      call integrate_ray(grid, iops,
295                      !     source,&
296                      !     rad_scatter, path_length,
297                      !     path_spacing,&
298                      !     a_tilde, gn, i, j, k, p)
299                  end do
300              end do
301          end do
302          !$omp end parallel do
303      end do
304      !$omp end parallel do
305  end subroutine advect_light

```

```

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
      bi = -a_tilde(i)*s(i+1)
325      do j=i+1, num_cells
            bi = bi - a_tilde(j)*ds(j)
326      end do
327
328      ! WARNING: This will overflow if a_tilde
      is too large.
329      if(a_tilde(i) .eq. 0) then
330
```

```

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
338   integral = integral + gn(i)*di * exp(bi)
339 end do
340
341 end function calculate_ray_integral
342
343 ! Calculate maximum number of cells a path
344 ! through the grid could take
345 ! This is a loose upper bound
346 function calculate_max_cells(grid) result(
347   max_cells)
348   type(space_angle_grid) :: grid
349   integer :: max_cells
350   double precision dx, dy, zrange, phi_middle
351
352   ! Angle that will have the longest ray
353   phi_middle = grid%angles%phi(grid%angles%
354     nphi/2)
355   dx = grid%x%spacing(1)
356   dy = grid%y%spacing(1)
357   zrange = grid%z%maxval - grid%z%minval
358
359   max_cells = grid%z%num + ceiling((1/dx+1/dy)
360     *zrange*tan(phi_middle))
361 end function calculate_max_cells
362
363 ! Traverse from surface or bottom to point (xi
364   , yj, zk)
365 ! in the direction omega_p, extracting path
366 ! lengths (ds) and
367 ! function values (f) along the way,
368 ! as well as number of cells traversed (n).
369 subroutine traverse_ray(grid, iops, source, i,
370   j, k, p, s_array, ds, a_tilde, gn,
371   num_cells)
372   type(space_angle_grid) :: grid
373   type(optical_properties) :: iops
374   double precision, dimension(:,:,:,:,:) :: 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
        ! Upwelling light originates from bottom
406       z0 = grid%z%maxval
407       dir_z = -1
408   end if
409
410   ! Total path length from origin to
      destination
411   ! (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
416 |     ! in each dimension
417 |     ! Set to 2*s_tilde if infinite in this
418 |     ! dimension so that it's unreachable
419 |     ! Assume x & y spacings are uniform,
420 |     ! so it's okay to just use the first value.
421 |     if(x_factor .eq. 0) then
422 |         ds_x = 2*s_tilde
423 |     else
424 |         ds_x = abs(grid%x%spacing(1)/x_factor)
425 |     end if
426 |     if(y_factor .eq. 0) then
427 |         ds_y = 2*s_tilde
428 |     else
429 |         ds_y = abs(grid%y%spacing(1)/y_factor)
430 |     end if
431 |
432 |     ! This one is an array because z spacing can
433 |     ! vary
434 |     ! z_factor should never be 0, because the
435 |     ! ray will never
436 |     ! reach the surface or bottom.
437 |     ds_z(1:grid%z%num) = dir_z * grid%z%spacing
438 |                         (1:grid%z%num)/z_factor
439 |
440 |     ! Origin point
441 |     p0x = p1x - s_tilde * x_factor
442 |     p0y = p1y - s_tilde * y_factor
443 |     p0z = p1z - s_tilde * z_factor
444 |
445 |     ! Direction of ray in each dimension. 1 =>
446 |     ! increasing. -1 => decreasing.
447 |     dir_x = int(sgn(p1x-p0x))
448 |     dir_y = int(sgn(p1y-p0y))
449 |
450 |     ! Shifts
451 |     ! Conversion from cell_inds to edge_inds
452 |     ! merge is fortran's ternary operator
453 |     shift_x = merge(1,0,dir_x>0)
454 |     shift_y = merge(1,0,dir_y>0)
455 |     shift_z = merge(1,0,dir_z>0)
456 |
457 |     ! Indices for cell containing origin point
458 |     cell_x = floor((p0x-grid%x%minval)/grid%x%
459 |                     spacing(1)) + 1
460 |     cell_y = floor((p0y-grid%y%minval)/grid%y%
461 |                     spacing(1)) + 1
462 |     ! x and y may be in periodic image, so shift
463 |     ! 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 preceding 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
501      ! compare to periodic image
502      s_next_x = (grid%x%edges(first_x) + dir_x
503          * (grid%x%maxval - grid%x%minval)&
504          - smx) / x_factor
505  else
506      ! Otherwise, just compare to next cell
507      s_next_x = (grid%x%edges(edge_x) - smx) /
508          x_factor
509  end if
510
511      ! Path length to next edge plane in each
512      ! dimension
513  if(abs(y_factor) .lt. 1.d-10) then
514      ! Will never cross, so set above total
515      ! path length
516      s_next_y = 2*s_tilde
517  else if(cell_y .eq. last_y) then
518      ! If starts out at last cell, then
519      ! compare to periodic image
520      s_next_y = (grid%y%edges(first_y) + dir_y
521          * (grid%y%maxval - grid%y%minval)&
522          - smy) / y_factor
523  else
524      ! Otherwise, just compare to next cell
525      s_next_y = (grid%y%edges(edge_y) - smy) /
526          y_factor
527  end if
528
529      s_next_z = ds_z(cell_z)
530
531      ! Initialize path
532      s = 0.d0
533      s_array(1) = 0.d0
534
535      ! Start with t=0 so that we can increment
536      ! before storing,
537      ! so that t will be the number of grid cells
538      ! at the end of the loop.
539  t=0
540
541      ! s is the beginning of the current cell,
542      ! s_next is the end of the current cell.
543  do while (s .lt. s_tilde)
544      ! Move cell counter
545      t = t + 1
546
547      ! Extract function values
548      a_tilde(t) = iops%abs_grid(cell_x, cell_y
549          , cell_z)
550      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%
546          theta_p(p)*180.d0/pi, grid%angles%
547          phi_p(p)*180.d0/pi
548      !write(*,*) 's = ', s, '/', s_tilde
549      !write(*,*) 'cell_z =', cell_z, '/', grid
550          %z%num
551      !write(*,*) 's_next_z =', s_next_z
552      !write(*,*) 'last_z =', last_z
553      !write(*,*) 'new'
554
555      ! Move to next cell in path
556      if(s_next_x .le. min(s_next_y, s_next_z))
557          then
558              ! x edge is closest
559              s_next = s_next_x
560
561              ! Increment indices (periodic)
562              cell_x = mod1(cell_x + dir_x, grid%x%
563                  num)
564              edge_x = mod1(edge_x + dir_x, grid%x%
565                  num)
566
567              ! x intersection after the one at s=
568              s_next
569              s_next_x = s_next + ds_x
570
571      else if (s_next_y .le. min(s_next_x,
572          s_next_z)) then
573          ! y edge is closest
574          s_next = s_next_y
575
576          ! Increment indices (periodic)
577          cell_y = mod1(cell_y + dir_y, grid%y%
578              num)
579          edge_y = mod1(edge_y + dir_y, grid%y%
580              num)
581
582          ! y intersection after the one at s=
583          s_next
584          s_next_y = s_next + ds_y
585
586      else if(s_next_z .le. min(s_next_x,
587          s_next_y)) then
588          ! z edge is closest
589          s_next = s_next_z
590
591          ! 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=
586     ! s_next
587     if(cell_z .lt. last_z) then
588         ! Only look ahead if we aren't at
589         ! the end
590         s_next_z = s_next + ds_z(cell_z)
591     else
592         ! Otherwise, no need to continue.
593         ! this is our final destination.
594         ! exit
595         s_next_z = 2*s_tilde
596         !write(*,*) 'end. s_next_z =',
597         ! s_next_z
598     end if
599
600     ! Cut off early if this is the end
601     ! This will be the last cell traversed if
602     ! s_next >= s_tilde
603     s_next = min(s_tilde, s_next)
604
605     ! Store path length
606     s_array(t+1) = s_next
607     ! Extract path length from same cell as
608     ! function vals
609     ds(t) = s_next - s
610
611     ! Update path length
612     s = s_next
613   end do
614
615   ! Return number of cells traversed
616   num_cells = t
617
618 end subroutine traverse_ray
619 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
11         ABSORPTION COEFFICIENT
12     abs_water, &
13     scat, &
14     num_vsf, &
15     vsf_file, &
16     ! SUNLIGHT
17     solar_zenith, &
18     solar_azimuthal, &
19     surface_irrad, &
20     ! KELP &
21     num_si, &
22     si_area, &
23     si_ind, &
24     frond_thickness, &
25     frond_aspect_ratio, &
26     frond_shape_ratio, &
27     ! WATER CURRENT
28     current_speeds, &
29     current_angles, &
30     ! SPACING
31     rope_spacing, &
32     depth_spacing, &
33     ! SOLVER PARAMETERS
34     nx, &
35     ny, &
36     nz, &
37     ntheta, &
38     nphi, &
39     num_scatters, &
40     ! FINAL RESULTS
41     perceived_irrad, &
42     avg_irrad)
43
44     implicit none
45
46     ! OPTICAL PROPERTIES
47     integer, intent(in) :: nx, ny, nz, ntheta,
48                     nphi
49     ! Absorption and scattering coefficients
50     double precision, intent(in) :::
51             absorptance_kelp, scat
52     double precision, dimension(nz), intent(in)
53             :: abs_water
54     ! Volume scattering function
55     integer, intent(in) :: num_vsf
56     character(len=*) :: vsf_file
57     !double precision, dimension(num_vsf),
58             intent(int) :: vsf_angles
59     !double precision, dimension(num_vsf),
60             intent(int) :: vsf_vals

```

```

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

```

```

90   double precision xmin, xmax, ymin, ymax,
91   zmin, zmax
92   character(len=5), parameter :: fmtstr = 'E13
93   .4'
94   !double precision, dimension(num_vsf) :::
95   vsf_angles, vsf_vals
96   double precision max_rad, decay
97   integer quadrature_degree
98
99   type(space_angle_grid) grid
100  type(optical_properties) iops
101  type(light_state) light
102  type(rope_state) rope
103  type(frond_shape) frond
104  type(boundary_condition) bc
105
106  double precision, dimension(:, :, :, :),
107    :: pop_length_means, pop_length_stds
108  ! Number of fronds in each depth layer
109  double precision, dimension(:, :, :, :),
110    :: num_fronds
111  double precision, dimension(:, :, :, :),
112    :: allocatable :: p_kelp
113
114  write(*,*) 'Light calculation'
115
116  allocate(pop_length_means(nz))
117  allocate(pop_length_stds(nz))
118  allocate(num_fronds(nz))
119  allocate(p_kelp(nx, ny, nz))
120
121  xmin = -rope_spacing/2
122  xmax = rope_spacing/2
123
124  ymin = -rope_spacing/2
125  ymax = rope_spacing/2
126
127  zmin = 0.d0
128  zmax = sum(depth_spacing)
129
130  write(*,*) 'Grid'
131  call grid%set_bounds(xmin, xmax, ymin, ymax,
132    zmin, zmax)
133  call grid%set_num(nx, ny, nz, ntheta, nphi)
134  call grid%init()
135  !call grid%set_uniform_spacing_from_num()
136  call grid%z%set_spacing_array(depth_spacing)
137
138  call rope%init(grid)

```

```

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%
152 |     frond_lengths
153 | write(*,*) 'frond_stds = ', rope%frond_stds
154 | write(*,*) 'num_fronds = ', rope%num_fronds
155 | write(*,*) 'water_speeds = ', rope%
156 |     water_speeds
157 | write(*,*) 'water_angles = ', rope%
158 |     water_angles
159 |
160 | write(*,*) 'Frond'
161 | ! INIT FROND
162 | call frond%set_shape(frond_shape_ratio,
163 |     frond_aspect_ratio, frond_thickness)
164 | ! CALCULATE KELP
165 | quadrature_degree = 5
166 | call calculate_kelp_on_grid(grid, p_kelp,
167 |     frond, rope, quadrature_degree)
168 | ! INIT IOPS
169 | iops%num_vsf = num_vsf
170 | call iops%init(grid)
171 | write(*,*) 'IOPs'
172 | iops%abs_kelp = absorptance_kelp /
173 |     frond_thickness
174 | iops%abs_water = abs_water
175 | iops%scat = scat
176 | !write(*,*) 'iop init'
177 | !iops%vsf_angles = vsf_angles
178 | !iops%vsf_vals = vsf_vals
179 | call iops%load_vsf(vsf_file, fmtstr)
180 |
181 | ! load_vsf already calls calc_vsf_on_grid
182 | !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 |             11 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 = ',
223          perceived_irrad
224
225      write(*,*) 'deinit'
226      call bc%deinit()
227      !write(*,*) 'a'
228      call iops%deinit()
229      !write(*,*) 'b'
230      call light%deinit()
231      !write(*,*) 'c'
232      call rope%deinit()
233      !write(*,*) 'd'
234      call grid%deinit()
235      !write(*,*) 'e'
236
237      deallocate(pop_length_means)
238      deallocate(pop_length_stds)
239      deallocate(num_fronds)
240      deallocate(p_kelp)
241
242      !write(*,*) 'done'
243  end subroutine full_light_calculations
244
245  subroutine
246      calculate_length_dist_from_superinds( &
247      nz, &
248      num_si, &
249      si_area, &
250      si_ind, &
251      frond_aspect_ratio, &
252      num_fronds, &
253      pop_length_means, &
254      pop_length_stds)
255
256      implicit none
257
258      ! Number of depth levels
259      integer, intent(in) :: nz
260      ! Number of Superindividuals in each depth
261      ! level
262      integer, intent(in) :: num_si
263      ! si_area(i,j) = area of superindividual j
264      ! at depth i
265      double precision, dimension(nz, num_si),
266          intent(in) :: si_area
267      ! si_area(i,j) = number of individuals
268      ! represented by superindividual j at depth
269          i

```

```

263 |     double precision, dimension(nz, num_si),
264 |         intent(in) :: si_ind
265 |     double precision, intent(in) :: frond_aspect_ratio
266 |
267 |     double precision, dimension(nz), intent(out)
268 |         :: num_fronds
269 | ! Population mean area at each depth level
270 |     double precision, dimension(nz), intent(out)
271 |         :: pop_length_means
272 | ! Population area standard deviation at each
273 |     depth level
274 |     double precision, dimension(nz), intent(out)
275 |         :: pop_length_stds
276 |
277 | ! -----
278 |
279 |     integer i, k
280 | ! Numerators for mean and std
281 |     double precision mean_num, std_num
282 | ! Convert area to length
283 |     double precision, dimension(num_si) :: si_length
284 |
285 |     do k=1, nz
286 |         mean_num = 0.d0
287 |         std_num = 0.d0
288 |         num_fronds(k) = 0
289 |
290 |         do i=1, num_si
291 |             si_length(i) = sqrt(2.d0*
292 |                 frond_aspect_ratio*si_area(k,i))
293 |             mean_num = mean_num + si_length(i)
294 |             num_fronds(k) = num_fronds(k) + si_ind
295 |                 (k,i)
296 |         end do
297 |
298 |         pop_length_means(k) = mean_num /
299 |             num_fronds(k)
300 |
301 |         do i=1, num_si
302 |             std_num = std_num + (si_length(i) -
303 |                 pop_length_means(k))**2
304 |         end do
305 |
306 |         pop_length_stds(k) = std_num / (
307 |             num_fronds(k) - 1)
308 |
309 |     end do
310 |
311 | end subroutine
312 | 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
341 end subroutine calculate_perceived_irrad
342
343 end module light_interface_module

```