

©2018

OLIVER GRAHAM EVANS

ALL RIGHTS RESERVED

MODELING 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

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

Approved:

Advisor

Dr. Kevin Kreider

Accepted:

Dean of the College

Dr. Linda Subich

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 mathematical model is developed to describe the light field in vertical line seaweed cultivation to determine the degree to which the seaweed shades itself and limits the amount of light available for photosynthesis. A probabilistic description of the spatial distribution of kelp is formulated using simplifying assumptions about frond geometry and orientation. An integro-partial differential equation called the radiative transfer equation is used to describe the light field as a function of position and angle. A finite difference solution is implemented, providing robustness and accuracy at the cost of large CPU and memory requirements, and a less computationally intensive asymptotic approximation is explored for the case of low scattering. Conditions for applicability of the asymptotic approximation are discussed, and depth-dependent light availability is compared to the predictions of simpler light models.

ACKNOWLEDGEMENTS

I'd like to express my deep gratitude to my professors, friends and family for putting up with me during the last few months of completing this work. This endeavor has been a great exercise in reconciling my idealistic vision of perfection with the constraints imposed by finite time and energy. I'm grateful to my mentors for having allowed me enough creative freedom in this process to explore the concepts and techniques that most interested me at the time. I've learned so much both about myself and about the process of working effectively as an applied mathematician over the course of the process of writing this thesis. To name some topics that come to my mind: mathematical modeling, numerical methods, fortran, debugging, profiling, parallel computing, HPC resource orchestration, version control, build automation, verification of codes and calculations, routine, sleep, diet, time management, interpersonal communication, the importance of social interaction, and deciding when to stop working. Without the particular challenges and opportunities afforded me during this project, I may have avoided many uncomfortable and important lessons I've learned lately.

In particular, in addition to my advisors, Dr. Kevin Kreider and Dr. Curtis Clemons, I'd like to thank Dr. Shane Rogers at Clarkson University in Potsdam,

New York and Dr. Ole Jacob Broch at SINTEF Ocean in Trondheim, Norway, with whom I began this project in Summer 2016. Also, thanks to Dr. Jutta Luettmer-Strathmann for staying up late with me one evening in the physics department helping me to prepare for my defense. The ongoing support of these people and many others has been invaluable, in practical advice and especially in emotional encouragement.

I would also like to express my gratitude to the kind folks at the NSF Pacific Research Platform including my uncle, John Graham, for allowing me access to Nautilus, a distributed, heterogenous HPC cluster which I used for many thousands of CPU hours to run simulations for this thesis.

And finally, thanks to my parents for feeding and housing me at the beginning and end of my studies. I feel extraordinarily blessed to have been born into a stable, loving family that has never stopped urging me forward.

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.

TABLE OF CONTENTS

	Page
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Background on Kelp Models	4
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	11
2.3 Population Distributions	13
2.4 Spatial Distribution	17
2.5 Rope Position Distribution	24
III. LIGHT MODEL	25
3.1 Optical Definitions	25
3.2 The Radiative Transfer Equation	27

3.3	Low-Scattering Approximation	30
IV.	NUMERICAL SOLUTION	35
4.1	Super-Individuals	35
4.2	Discrete Grid	36
4.3	Quadrature Rules	39
4.4	Numerical Asymptotics	41
4.5	Finite Difference	44
V.	PARAMETER VALUES	53
5.1	Simulation Parameters	53
5.2	Parameters from Literature	54
5.3	Frond Alignment Coefficient	56
VI.	NUMERICAL ANALYSIS	58
6.1	Verification and Validation of Computational Codes	58
6.2	Computational Resource Requirements	70
6.3	Rules of Thumb	70
6.4	Comparison to Other Light Models	70
6.5	Sensitivity Analysis	70
VII.	CONCLUSION	75
APPENDICES	82
APPENDIX A.	GRID DETAILS	83
APPENDIX B.	RAY TRACING ALGORITHM	87

APPENDIX C. FORTRAN CODE	90
------------------------------------	----

LIST OF TABLES

Table	Page
4.1 Breakdown of nonzero matrix elements by derivative case.	51
5.1 Parameter values.	55
5.2 Field measurement data of optical properties in the ocean [20]. The site names used in the original paper are used: AUTEC – Bahamas, HAOCE – Coastal southern California, NUC – San Diego Harbor. Absorption, scattering, and attenuation coefficients (a, b, c) are given, and their ratios.	55

LIST OF FIGURES

Figure	Page
1.1 <i>Saccharina latissima</i> being harvested	3
2.1 <i>Saccharina latissima</i> innoculated onto a thread wrapped around a rope on which it is to be grown.	10
2.2 Rendering of four nearby vertical kelp ropes as represented in the spatial distribution model. Note the kite-shaped fronds and horizontal orientation.	11
2.3 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	12
2.4 Simplified kite-shaped frond.	13
2.5 von Mises distribution for a variety of parameters.	16
2.6 2D length-angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$	18
2.7 A sample of 50 kelp fronds with shape parameters $f_s = 0.5$ and $f_r = 2$ whose lengths are picked from a normal distribution and whose angles are picked from a von Mises distribution.	19
2.8 Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$	22
2.9 Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the θ_f - l plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$	23
2.10 Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3$, $\eta v_w = 1$	24

4.1	Spatial grid.	37
4.2	Angular grid at each point in space.	38
6.1	Code verification for the finite difference solution via the Method of Manufactured Solutions. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. The average absolute difference between the analytical and numerical solutions is shown. A slope of $m = 2$ on a log-log scale demonstrates second order convergence, as expected. This demonstrates the correctness of the code.	66
6.2	<i>top-heavy</i> kelp distribution (left) and no-scattering irradiance profile (right)	71
6.3	<i>bottom-heavy</i> kelp distribution (left) and no-scattering irradiance profile (right)	71
6.4	<i>uniform</i> kelp distribution (left) and no-scattering irradiance profile (right)	72
6.5	Several kelp profiles	72
6.6	Several values of kelp absorptance	73
6.7	Several values of absorption coefficient of water	73
6.8	Several values of scattering coefficient	74
A.1	Angular grid	85

CHAPTER I

INTRODUCTION

1.1 Motivation

Given the consistent global increase in population, efficient and innovative resource utilization is increasingly important. Our generation faces major challenges regarding food, energy, and water and must confront major issues associated with global climate change. Growing concern for the negative environmental impacts of petroleum-based fuel has generated 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 and pesticides that impair water quality, and for the high carbon footprint involved in its development [13, 23]. Meanwhile, a great deal of unutilized coastline is available for both food and fuel production through seaweed cultivation. Specifically, the sugar kelp *Saccharina latissima* has been demonstrated to be a viable source of food, both for direct human consumption and biofuel production, especially in conjunction with other aquatic species in *integrated multi-trophic aquaculture* (IMTA) [3, 6, 9, 10].

Furthermore, seaweed cultivation has been proposed as a nutrient remediation technique for natural waters [14]. Nitrogen leakage into water bodies is a

significant ecological problem, and is especially relevant in close proximity to large conventional agriculture facilities and wastewater treatment plants. Waste water treatment plants (WWTPs) in particular are facing increasingly stringent regulation 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 [28]. 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”. Figure 1.1 shows seaweed being



Figure 1.1: *Saccharina latissima* being harvested

harvested onboard a SINTEF research vessel. 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. The development of mathematical models enables insight into a system which would be otherwise difficult or impossible to obtain. For example, imagine that a company is interested in a new IMTA site, and is looking for a suitable location. Running simulations to predict the potential productivity of each area would be of great assistance in choosing the best site. Similarly,

if a new cultivation technique is under consideration, simulation can estimate its viability without having to deploy it on a large scale and risk failure or inefficiency.

Recently, a growth model [?] for *S. latissima* has been produced and integrated into the SINMOD [27] hydrodynamic and ecosystem model of SINTEF. This kelp model considers factors such as temperature, nutrient concentration, light availability, and water current. The amount of light available is informed by spatially varying attenuation coefficients from SINMOD, which considers optical properties of the water as well as concentrations of various organic and inorganic constituents. However, it does not consider the effect of the kelp itself on the light field. This is an important consideration, as high kelp densities should lead to low light levels which would inhibit further growth. However, without accounting for self-shading, the kelp is not adequately penalized for growing too densely, which is expected to cause overpredictions in the total biomass production. The purpose of this thesis is to develop a first principles light model which adequately predicts the effects of self-shading.

1.2 Background on Kelp Models

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

for the Kelp Habitat Improvement Project at the California Institute of Technology in 1968 using seven variables. By 1974, Nick Anderson greatly expanded on North's work, and created the first comprehensive model of kelp growth which he programmed using FORTRAN [1]. In his model, he accounts for solar radiation intensity as a function of time of year and time of day, and refraction on the surface of the water. He uses a simple model for shading, simply specifying a single parameter which determines the percentage of light 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* [12], with an emphasis on including more environmental parameters and a more complete description of the growth and decay of the kelp. 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 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. [18] published a similar model alongside a Markov chain model, and compared the results with experimental data collected in New Zealand.

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 [8, 29]. These were some of the first models of kelp growth that did not specifically relate to *Macrocystis pyrifera* (“giant kelp”).

1.3 Background on Radiative Transfer

In terms of optical quantities, of primary interest is the radiance at each point in 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 extensively in stellar astrophysics [5, 19]; its application to marine biology is fairly recent [16]. In its full form, radiance is a function of 3 spatial dimensions, 2

angular dimensions, and frequency, making for a formidable 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 the case of macroalgae cultivation, emission is negligible, owing only perhaps to some small luminescent phytoplankton or other anomaly, and can therefore be safely ignored.

1.4 Overview of Thesis

The remainder of this document is organized as follows. In Chapter 2, a 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 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 approximation are described. Next, in Chapter 5, the availability of necessary parameters in the literature is discussed. For those which are not readily available, rough estimates are given or 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, and a comparison is given between the light fields predicted with and without considering self-shading by the kelp. Finally, Chapter 7 concludes the thesis by giving a brief summary of the model, discusses and its performance, and suggests 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 larger rope as in Figure 2.1, which is hung from buoys in the ocean. The two primary configurations are vertical and horizontal or “long” lines. In the case of vertical lines, the seaweed rope hangs straight down from a single buoy, and is either weighted or anchored. In the case of long lines, the rope is strung from one buoy to another. Long lines allow more light to reach the seaweed since it grows closer to the surface, but more vertical lines can be set up in a given area, which may be advantageous for IMTA.



Figure 2.1: *Saccharina latissima* innoculated onto a thread wrapped around a rope on which it is to be grown.

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 at any given 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. A computer rendering of this scenario is shown in Figure 2.2.



Figure 2.2: Rendering of four nearby vertical kelp ropes as represented in the spatial distribution model. Note the kite-shaped fronds and horizontal orientation.

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 2.3. 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

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

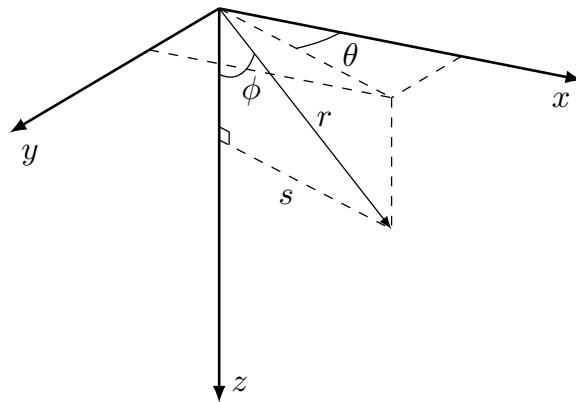


Figure 2.3: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ .

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. Since it is assumed that fronds extend perfectly horizontally, no angular elevation distribution is required.

2.3.1 Frond Shape

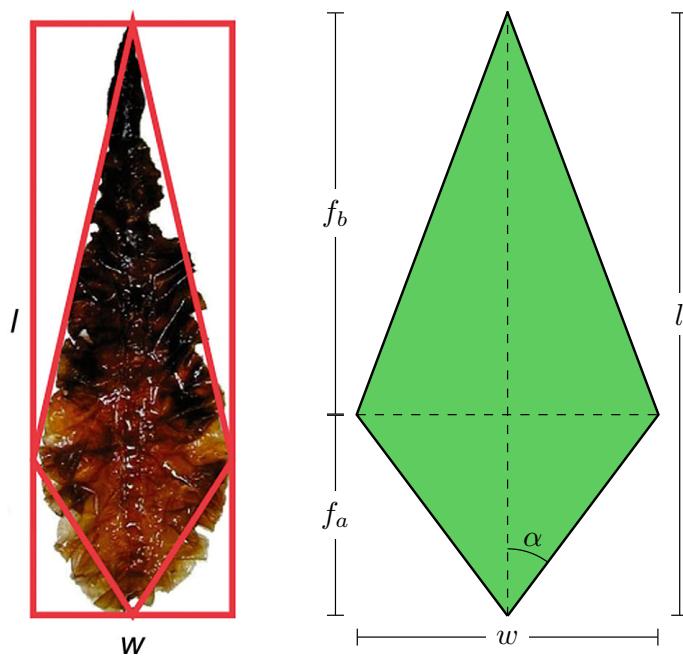


Figure 2.4: Simplified kite-shaped frond.

The frond is assumed to be kite-shaped with length l from base to tip, and width w from left to right. In Figure 2.4, 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 .

It is therefore clear that

$$f_a + f_b = l.$$

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

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

These ratios are assumed to be constant among the entire population, so that all fronds are geometrically similar. Thus, the shape of the frond can be fully specified by l , f_r , and f_s ; it is possible to redefine w , f_a and f_b from the preceding formulas as

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

The angle α , half of the angle at the base corner, is also noteworthy. From the above equations, it follows that

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

It is useful to convert between frond length and surface area, which can be done via the relations

$$A = \frac{lw}{2} = \frac{l^2}{2f_r}, \tag{2.3}$$

$$l = \sqrt{2Af_r}. \tag{2.4}$$

2.3.2 Length and Angle Distributions

The distribution of frond lengths is assumed to be normal, with mean μ_l and standard deviation σ_l . That is, it 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 2.5. κ is analogous to $1/\sigma$ in the normal distribution. In the absence of current, the frond angles are distributed uniformly, while as current velocity increases, they become increasingly likely to align in the current direction, 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 alignment coefficient* η , with units of inverse velocity (s m^{-1}). Then, 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

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

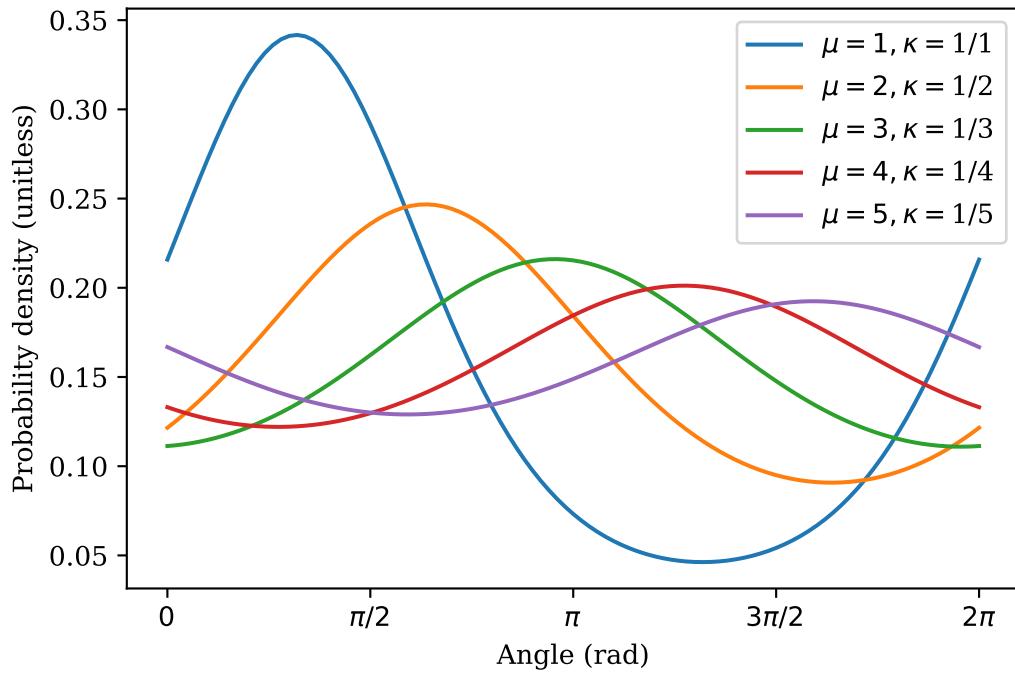


Figure 2.5: von Mises distribution for a variety of parameters.

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 , the probability of their intersection is the product of their individual probabilities. That is,

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

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 2.6, where probability is represented by color in the 2D plane. Darker green represents higher probability, while lighter beige represents lower probability. In Figure 2.7, 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

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

which is currently not taken into consideration in this model.

2.4 Spatial Distribution

In this section, the population length and angle distributions from the previous section are used to construct a spatial distribution of kelp. This is made possible by the simple kite-shape fronds, and would be considerably more difficult with more general frond shapes.

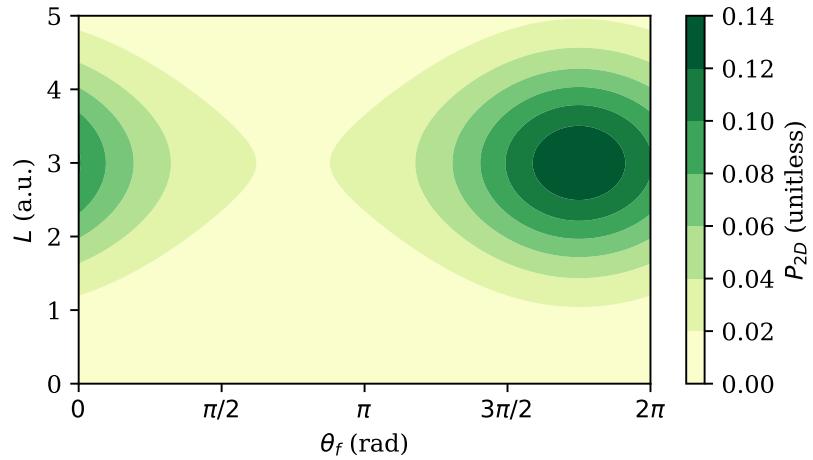


Figure 2.6: 2D length-angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$.

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 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 points in the $+y$ direction ($\theta = \pi/2$), with the base at $(0, 0)$. Denote the Cartesian analogue of this coordinate system as (x', y') which

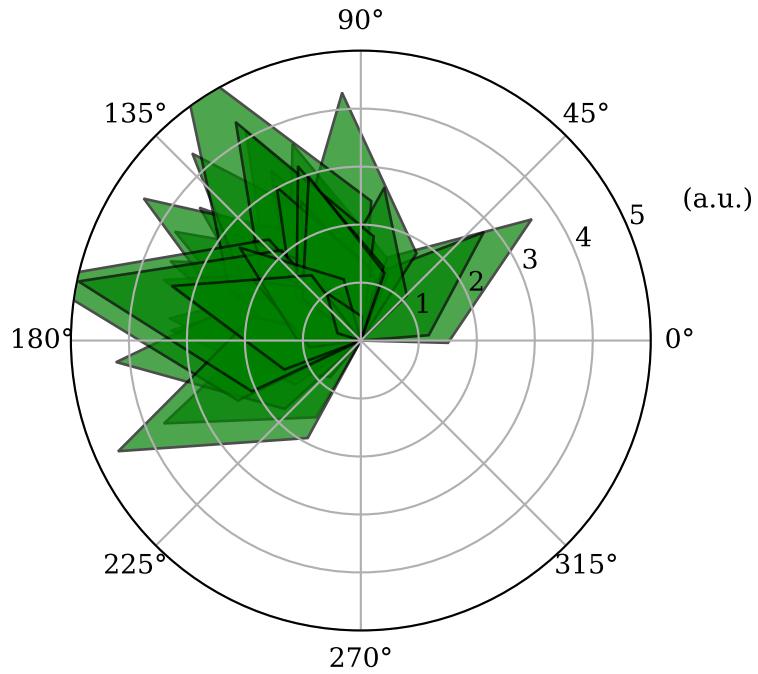


Figure 2.7: A sample of 50 kelp fronds with shape parameters $f_s = 0.5$ and $f_r = 2$ whose lengths are picked from a normal distribution and whose angles are picked from a von Mises distribution.

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 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

$$S(\theta') = \text{sign}(\cos \theta').$$

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

$$\theta - \alpha < \theta_f < \theta + \alpha, \quad (2.5)$$

$$l > l_{min}(\theta, s), \quad (2.6)$$

where

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

Then, considering the point to be fixed, (2.5) and (2.6) 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. $R_s(3\pi/4, 1.5)$ is shown in blue in Figure 2.8 and the smallest possible occupying fronds for several values of θ_f are shown in various colors. Any frond longer than these at the same angle will also occupy the point.

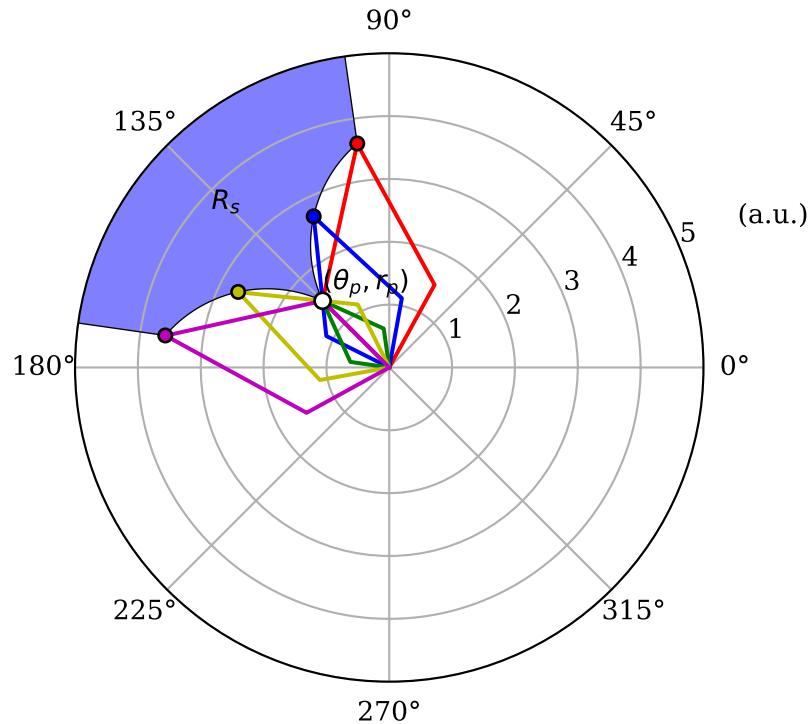


Figure 2.8: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$.

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) .

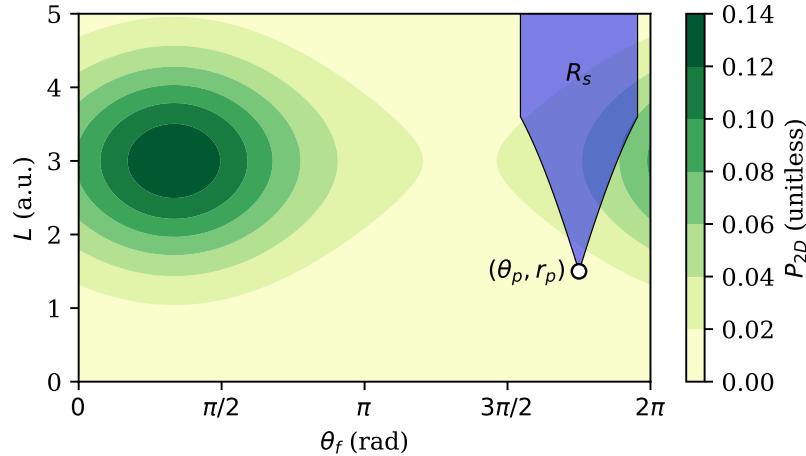


Figure 2.9: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the θ_f - l plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$.

Integrating $P_{2D}(\theta_f, l)$ over $R_s(\theta, s)$ as depicted in Figure 2.9 yields the proportion of the population in the depth layer occupying the point (θ, s) ,

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

Assuming that the depth layer has thickness dz and contains n fronds of thickness t , the proportion of the depth layer occupied by kelp at any horizontal position can be calculated as

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

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

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

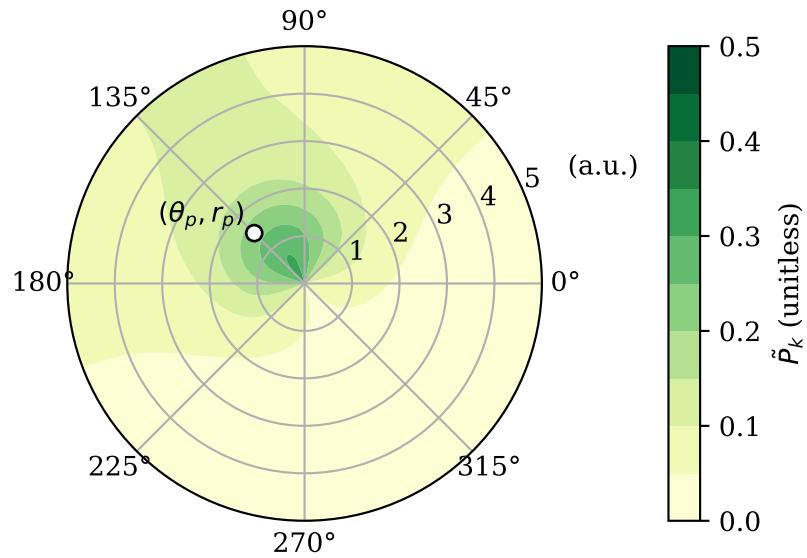


Figure 2.10: Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3$, $\eta v_w = 1$.

2.5 Rope Position Distribution

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

Before introducing the radiative transfer equation, it is necessary to discuss some basic radiometric quantities of interest which characterize the light field, as well as inherent optical properties which describe the medium of propagation.

3.1.1 Radiometric Quantities

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

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

where ω is an element of solid angle, and A is an element of projected surface area.

Once the radiance L is calculated everywhere, the irradiance is

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

Irradiance is sometimes given in units of moles of photons (a mole of photons is also called an Einstein) per second, with the conversion [15] given by

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

3.1.2 Perceived Irradiance

Assuming that the irradiance $I(\mathbf{x})$ is known, the average irradiance at each depth can be calculated as

$$\bar{I}(z) = \frac{\iint I(x, y, z) dx dy}{\iint 1 dx dy}.$$

More relevant, however, is the average irradiance perceived by the kelp. To calculate this value, we simply take a weight the irradiance by the normalized spatial kelp distribution before taking the mean. Then, the average perceived irradiance at each depth is

$$\tilde{I}(z) = \frac{\iint P_k(x, y, z) I(x, y, z) dx dy}{\iint P_k(x, y, z) dx dy}.$$

The irradiance perceived by the kelp is expected to be lower than the average irradiance, 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 now define a few inherent optical properties (IOPs) which depend only on the medium of propagation. The absorption coefficient $a(\mathbf{x})$ (units m^{-1}) defines the proportional loss of radiance per unit length due to absorption by the medium. For

example, this includes radiant energy which is converted to heat. The scattering coefficient b (units m^{-1}), defines the proportional loss of radiance per unit length due to scattering, and is assumed to be constant over space. Scattered light is not lost from the light field, it simply changes direction.

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). Now, 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.

TODO:

3.2 The Radiative Transfer Equation

We are now prepared to present the full details of Radiative Transfer Equation, whose solution is the radiance in the medium as a function of position x and angle ω .

3.2.1 Ray Notation

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

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

Hence, the ray path is parameterized as

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

where

$$\mathbf{x}_0(\mathbf{x}, \boldsymbol{\omega}) = \mathbf{x} - \tilde{s}\boldsymbol{\omega} \quad (3.3)$$

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 four 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.
4. Radiance is increased or decreased due to light sources or sinks

3.2.3 Equation of Transfer

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

$$\frac{dL}{ds}(\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}) = -(a(\mathbf{x}) + b)L(\mathbf{x}, \boldsymbol{\omega}) + b \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L(\mathbf{x}) d\boldsymbol{\omega}' + \sigma(\mathbf{x}, \boldsymbol{\omega}), \quad (3.4)$$

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 reveals the vector form of the radiative transfer equation,

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

or equivalently,

$$\boldsymbol{\omega} \cdot \nabla L(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L(\mathbf{x}, \boldsymbol{\omega}) = b \left(\int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L(\mathbf{x}, \boldsymbol{\omega}) \right) + \sigma(\mathbf{x}, \boldsymbol{\omega}). \quad (3.5)$$

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 $f(\omega)$. 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

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

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

3.3 Low-Scattering Approximation

In waters where absorption dominates scattering, an asymptotic series in terms of the scattering coefficient b can be constructed. The physical interpretation of the asymptotic series is that each term represents a discrete scattering event. With the addition of each term, light from the previous term is scattered and attenuated from each point along the ray path. In reality, the scattering cannot be considered to occur in discrete events, but rather all scattering occurs simultaneously (on a macroscopic timescale).

Since this is only an approximation, it is important to note that while the asymptotic series converges as $b \rightarrow 0$, it is not necessary that the series converges as the number of terms increases, although it may occur in certain cases. Especially in cases of large scattering, the asymptotic series diverges rapidly. The convergence properties will be expanded upon in -TODO-

3.3.1 Asymptotic Expansion

Taking b to be small, we introduce the asymptotic series

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

Since the source, σ may also depend on b , it deserves a similar expansion,

$$\sigma(\mathbf{x}, \boldsymbol{\omega}) = \sigma_0(\mathbf{x}, \boldsymbol{\omega}) + b\sigma_1(\mathbf{x}, \boldsymbol{\omega}) + b^2\sigma_2(\mathbf{x}, \boldsymbol{\omega}) + \dots.$$

Substituting the above into (3.5) yields

$$\begin{aligned} & \boldsymbol{\omega} \cdot \nabla [L_0(\mathbf{x}, \boldsymbol{\omega}) + bL_1(\mathbf{x}, \boldsymbol{\omega}) + b^2L_2(\mathbf{x}, \boldsymbol{\omega}) + \dots] \\ & + a(\mathbf{x}) [L_0(\mathbf{x}, \boldsymbol{\omega}) + bL_1(\mathbf{x}, \boldsymbol{\omega}) + b^2L_2(\mathbf{x}, \boldsymbol{\omega}) + \dots] \\ & = b \left(\int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') [L_0(\mathbf{x}, \boldsymbol{\omega}') + bL_1(\mathbf{x}, \boldsymbol{\omega}') + b^2L_2(\mathbf{x}, \boldsymbol{\omega}') + \dots] d\boldsymbol{\omega}' \right. \\ & \quad \left. - [L_0(\mathbf{x}, \boldsymbol{\omega}) + bL_1(\mathbf{x}, \boldsymbol{\omega}) + b^2L_2(\mathbf{x}, \boldsymbol{\omega}) + \dots] \right) \\ & + [\sigma_0(\mathbf{x}, \boldsymbol{\omega}) + b\sigma_1(\mathbf{x}, \boldsymbol{\omega}) + b^2\sigma_2(\mathbf{x}, \boldsymbol{\omega}) + \dots]. \end{aligned}$$

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

$$\boldsymbol{\omega} \cdot \nabla L_0(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_0(\mathbf{x}) = \sigma_0(\mathbf{x}, \boldsymbol{\omega}), \quad (3.6)$$

$$\begin{aligned} \boldsymbol{\omega} \cdot \nabla L_1(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_1(\mathbf{x}) &= \sigma_1(\mathbf{x}, \boldsymbol{\omega}) + \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_0(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_0(\mathbf{x}, \boldsymbol{\omega}), \\ \boldsymbol{\omega} \cdot \nabla L_2(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_2(\mathbf{x}) &= \sigma_2(\mathbf{x}, \boldsymbol{\omega}) + \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_1(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_1(\mathbf{x}, \boldsymbol{\omega}). \\ &\vdots \end{aligned}$$

In general, for $n \geq 1$,

$$\boldsymbol{\omega} \cdot \nabla L_n(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_n(\mathbf{x}) = \sigma_n(\mathbf{x}, \boldsymbol{\omega}) + \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_{n-1}(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_{n-1}(\mathbf{x}, \boldsymbol{\omega}). \quad (3.7)$$

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

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

Grouping by powers of b , we have

$$\begin{cases} L_0(\mathbf{x}_s, \boldsymbol{\omega}) = f(\boldsymbol{\omega}) & \text{if } \hat{z} \cdot \boldsymbol{\omega} > 0, \\ L_0(\mathbf{x}_b, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} < 0, \end{cases} \quad (3.8)$$

for the first term, and

$$\begin{cases} L_n(\mathbf{x}_s, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} > 0, \\ L_n(\mathbf{x}_b, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} < 0, \end{cases} \quad (3.9)$$

for $n > 0$.

3.3.2 Analytical Solution

Given $\mathbf{x}, \boldsymbol{\omega}$, consider the path $\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s)$ from (3.2) for $s \in [0, \tilde{s}]$. Denote the radiance, absorption coefficient, and source term along the path by

$$\tilde{u}_0(s) = L(\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}),$$

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

$$\tilde{\sigma}_0(s) = \sigma_0(\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}).$$

Then, the first equation from the asymptotic expansion, (3.6) and its associated boundary condition, (3.8), can be rewritten as the first order linear ordinary differ-

ential equation

$$\begin{cases} \tilde{\sigma}_0(s) = \frac{du_0}{ds}(s) + \tilde{a}(s)u_0(s) \\ u_0(0) = f(\boldsymbol{\omega})H(\boldsymbol{\omega} \cdot \hat{z}), \end{cases} \quad (3.10)$$

where $H(x)$ is the Heaviside step function. This equation is solved by multiplying by the appropriate integrating factor, as follows.

$$\begin{aligned} \exp\left(\int_0^s \tilde{a}(s') ds'\right) \tilde{\sigma}_0(s) &= \exp\left(\int_0^s \tilde{a}(s') ds'\right) \frac{du_0}{ds} + \exp\left(\int_0^s \tilde{a}(s') ds'\right) \tilde{a}(s)u_0(s) \\ &= \frac{d}{ds} \left[\exp\left(\int_0^s \tilde{a}(s') ds'\right) u_0(s) \right]. \end{aligned}$$

Then, integrating both sides yields

$$\begin{aligned} \int_0^s \exp\left(\int_0^{s'} \tilde{a}(s'') ds''\right) \tilde{\sigma}_0(s') ds' &= \int_0^s \frac{d}{ds} \left[\exp\left(\int_0^{s'} \tilde{a}(s'') ds''\right) u_0(s') \right] ds' \\ &= \exp\left(\int_0^s \tilde{a}(s') ds'\right) u_0(s) - f(\boldsymbol{\omega})H(\boldsymbol{\omega} \cdot \hat{z}). \end{aligned}$$

Hence,

$$\begin{aligned} u_0(s) &= \left[f(\boldsymbol{\omega})H(\boldsymbol{\omega} \cdot \hat{z}) + \int_0^s \exp\left(\int_0^{s'} \tilde{a}(s'') ds''\right) \tilde{\sigma}_0(s') ds' \right] \exp\left(-\int_0^s \tilde{a}(s') ds'\right) \\ &= \exp\left(-\int_0^s \tilde{a}(s') ds'\right) f(\boldsymbol{\omega})H(\boldsymbol{\omega} \cdot \hat{z}) \\ &\quad + \int_0^s \exp\left(-\int_{s'}^s \tilde{a}(s'') ds''\right) \tilde{\sigma}_0(s') ds' \end{aligned} \quad (3.11)$$

Then, we convert back from path length s to the spatial coordinate \boldsymbol{x} by evaluating the one-dimensional solution at the end of the ray path. That is,

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

In addition to the explicit source term, the $n \geq 1$ equations also have a scattering term, which is an integral of the previous term in the series. The sum of

these two sources is called the effective source,

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

This can be similarly extracted along a ray path as

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

Then, since \tilde{g}_n depends only on L_{n-1} and is therefore independent of u_n , (3.7) and its boundary condition (3.9) can be written as the first order linear ordinary differential equation along the ray path,

$$\begin{cases} \tilde{g}_n(s) = \frac{du_n}{ds}(s) + \tilde{a}(s)u_n(s) \\ u_n(0) = 0 \end{cases} \quad (3.12)$$

This is exactly (3.10) with $\tilde{\sigma}_0 \rightarrow \tilde{g}_n$ and $f(\boldsymbol{\omega}) \rightarrow 0$. Hence,

$$u_n(s) = \int_0^s \tilde{g}_n(s') \exp\left(-\int_{s''}^{s'} \tilde{a}(s'') ds''\right) ds'. \quad (3.13)$$

As before, the conversion back to full spatial and angular coordinates is

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

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, subsets of the population, called super-individuals, are modeled explicitly, and are considered to represent many identical individuals, as in [22]. 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 (2.4), 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.

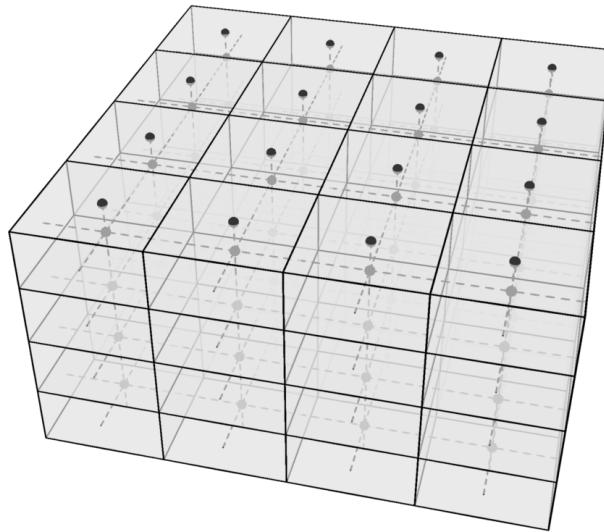


Figure 4.1: Spatial grid.

Each spatial grid cell is the Cartesian product of x , y , and z intervals of width dx , dy , and dz respectively, as shown in Figure 4.1. 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.

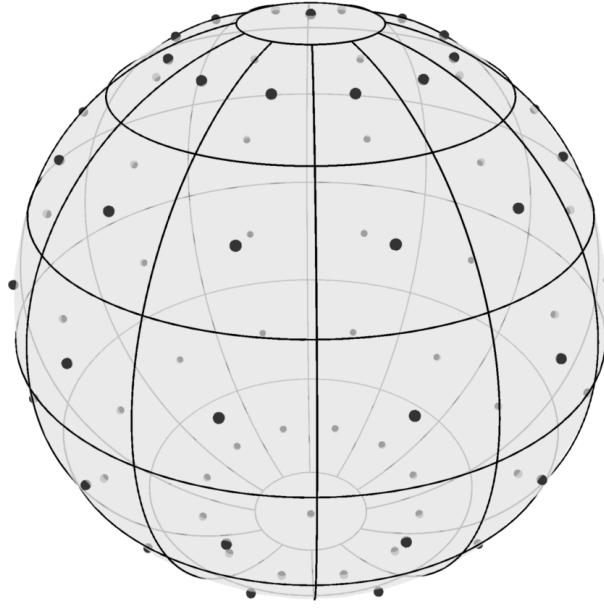


Figure 4.2: Angular grid at each point in space.

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 $d\phi$. 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 (3.11) and (3.13) to the radiative transfer equation (3.5).

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

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

4.4.2 Ray Integral

Given a position \mathbf{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 (wholly or partially) 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 $\tilde{g}_n(s)$ from Section 3.3.2 are

$$\begin{aligned} \tilde{g}_n(s) &= \sum_{\nu=1}^{N-1} \tilde{g}_{n\nu} \mathcal{X}_\nu^l(s), \\ \tilde{a}(s) &= \sum_{\nu=1}^{N-1} \tilde{a}_\nu \mathcal{X}_\nu^l(s). \end{aligned}$$

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 $s_N = \tilde{s}$. This implies that edge ν precedes cell ν . $\{s_\nu\}$ 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, the path

lengths through the grid cells, indexed by $\nu = 1, \dots, N - 1$, are

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

Given s , the index of the next edge crossing is

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

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

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

Then, evaluating (3.13) at $s = \tilde{s}$ is calculated as

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

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

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

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

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

When $n = 0$, the boundary condition must be included for downwelling light, and the effective source $\tilde{g}_{n\nu}$ is reduced to the explicit source $\tilde{\sigma}_{0\nu}$ for lack of a scattering term. Thus, the numerical solution to (3.11) is given by

$$u_0(\tilde{s}) = f(\boldsymbol{\omega}) H(\boldsymbol{\omega} \cdot \hat{z}) \exp\left(-\sum_{j=1}^{N-1} \tilde{a}_j ds_j\right) + \sum_{\nu=1}^{N-1} \tilde{\sigma}_{0\nu} d_\nu e^{b_\nu}. \quad (4.5)$$

4.5 Finite Difference

While the asymptotic solution is valid in the 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.

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

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

When applying the PDE on the upper or lower boundary, we use the forward and backward difference (FD2 and BD2) formulas respectively. The forward difference is given by

$$f'(x) = \frac{-3f(x) + 4f(x + dx) - f(x + 2dx)}{2dx} + \mathcal{O}(\varepsilon^3),$$

and the backward difference by

$$f'(x) = \frac{3f(x) - 4f(x - dx) + f(x - 2dx)}{2dx} + \mathcal{O}(\varepsilon^3).$$

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

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

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

$$f'(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \quad (4.7)$$

Rearranging (4.6) produces

$$f''(x)\varepsilon^2 = 2f(x + \varepsilon) - 2f(x) - 2f'(x)\varepsilon + \mathcal{O}(\varepsilon^3). \quad (4.8)$$

Combining (4.7) with (4.8) 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

For every spatial grid cell, the scattering integral is discretized as described in Section 4.4.1, as

$$\boldsymbol{\omega} \cdot \nabla L_p = -(a_{ijk} + b)L_p + \sum_{p'=1}^{n_\omega} \beta_{pp'} L_{p'} + \sigma_{ijkp},$$

or equivalently,

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

On the interior of the spatial domain, we apply the central difference formula in each dimension, which yields

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

Note that since periodic boundary conditions are used in x and y , the subscript $i+1$ should actually read $(i+1) \bmod 1 n_x$, where $\bmod 1$ is the one-indexed modulus. The same idea applies for $i-1$, $j+1$, and $j-1$. For the sake of readability, this is omitted from the equations in this section.

For downwelling light at the surface, we apply the asymmetric second order difference approximation (4.7) using the surface radiance value, which gives

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

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

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

Likewise for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

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 \mathbf{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 \mathbf{b} .

Also relevant is the total size of the system and of the sparse matrices necessary to store. The sizes of A , \mathbf{x} , and \mathbf{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.

Table 4.1: Breakdown of nonzero matrix elements by derivative case.

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	$n_\omega + 6$	$n_x n_y n_\omega / 2$

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

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

Also, note that \mathbf{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

Because of the large number of dimensions (three spatial, two angular), the matrix can easily have upwards of millions of nonzero elements, even for modest grid sizes. Direct methods such as Gaussian elimination, QR factorization, and singular value decomposition are therefore infeasible due to memory requirements. We therefore turn to iterative solvers. Many such solvers are available, including GMRES [21], LGMRES [2], IDR [25], and BI-CGSTAB [26]. 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 of the 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 [?], and an ocean model, as in [27]. Both models are assumed to use the same spatial grid, with n_z discrete depth layers of thickness dz_k for $k = 1, \dots, n_z$. 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 specimens 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 [?] was given in terms of photons per second, and was converted to W m^{-2} according to (3.1). No data in the literature exist for the frond thickness, so a best estimate is provided.

In [20], 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 Tongue 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	[7]
Water absorption coefficient	a_w	See Table 5.2	[20]
Scattering coefficient	b	See Table 5.2	[20]
Volume scattering function	β	tabulated	[20, 24],
Frond thickness	t	0.4 mm	estimated
Surface solar irradiance	I_0	50 W m^{-2}	[?]

Table 5.2: Field measurement data of optical properties in the ocean [20]. The site names used in the original paper are used: AUTEC – Bahamas, HAOCE – Coastal southern California, NUC – San Diego Harbor. Absorption, scattering, and attenuation coefficients (a, b, c) are given, and their ratios.

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 Alignment Coefficient

The *frond alignment 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. [17] 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 alignment 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 alignment 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 determined.

CHAPTER VI

NUMERICAL ANALYSIS

In this chapter, both numerical implementations of the model are probed. We begin by verifying that the methods are mistake-free by looking at convergence orders. Then, the two methods are compared, to determine the conditions under which the asymptotic approximation is valid. Finally, results are compared to simpler kelp models.

6.1 Verification and Validation of Computational Codes

As Roache explains, there are two aspects to confidence building for numerical codes, specifically those which solve PDEs: verification deals with *solving the equations right*, while validation deals with *solving the right equations*. Validation involves comparison with experimental evidence in order to determine that governing equations accurately describe physical phenomenon, and that their solutions match observation. Validation is an ongoing process. As new experimental data becomes available, the equations can be solved in an attempt to replicate the observation. Verification, however, is a purely mathematical exercise, and has nothing to do with the physical system being modeled. It deals only with the agreement between an equation and its numerical solution produced by a particular implementation of an

algorithm. Unlike validation, verification is something to be started and finished. If a set of parameters to the model can be chosen to exercise all terms in the equation, comparison between the numerical and exact solution is sufficient to demonstrate the correctness of a computational code, and the process need not be repeated unless the code is modified.

Due to lack of sufficient experimental data, rigorous validation of the model is left as future work. However, verification of both the finite difference and numerical asymptotics algorithms is presented here. There are two forms of verification. The first is *verification of codes*, where a whole implementation of an algorithm is tested, and the the difference between the numerical and analytical solutions is *explicitly measured* at every point in the numerical solution. The same calculation is repeated for several grid sizes, and it is checked that the convergence order as the grid spacing approaches zero matches the theoretical convergence order of the algorithm. The explicit measurement of errors requires that the analytical solution be known, which is generally only possible for some unrealistic or uninteresting set of parameters. If the analycal solution were available for the real, interesting case, then it would probably not have been necessary to implement a numerical solution in the first place.

Therefore, the second phase of verification is *verification of calculations*, where a single specific calculation is performed, and the error is *estimated* since it cannot be measured explicitly. This is generally done by repeating the calculation for several grid sizes, as above, then using a technique called *Richardson Extrapolation*.

lation to estimate the limiting solution as the grid spacing approaches zero. This estimated limiting solution is then compared to the actual numerical solutions. Error estimation for a specific calculation can be used to calculate error bars to display on a plot to represent discretization error in the solution.

6.1.1 Method of Manufactured Solutions

The most obvious way to obtain an analytical solution to compare to a numerical solution is by choosing a simple case where the PDE can be solved explicitly, perhaps through separation of variables or by reducing it to an ODE. This is referred to as the Method of Exact Solutions. However, such simple cases usually result in such a loss of generality as to make them nearly useless. In order to verify that a code will work in an interesting case, every term in the equation must be exercised during the verification process. An alternative process, the Method of Manufactured Solutions, retains arbitrary generality in the equations while making analytical solutions readily available. Of course, there is a trade-off: the solutions are not physically realistic. However, this is not an issue. As stated previously, *verification is a purely mathematical endeavor*. Determining that a code solves an equation correctly is unrelated to physical realism.

Consider a differential equation

$$Du(\mathbf{x}) = \sigma(\mathbf{x}), \quad (6.1)$$

$$u(\mathbf{x}) = f(\mathbf{x}) \text{ for } \mathbf{x} \in \Sigma, \quad (6.2)$$

where D is a differential operator, u is the solution, and σ is a source term, f is the boundary condition function, and Σ is the set of boundary points at which the boundary condition is applied. In general, D , σ , and f are known, and solving for u involves determining D^{-1} and calculating $u = D^{-1}\sigma$ while satisfying (6.2).

The Method of Manufactured Solutions reverses the normal procedure. Here, u is hand-picked at the outset, all parameters and coefficient functions in D are chosen to be nonzero, and the source term σ which produces the desired solution is calculated. Similarly, the boundary condition is determined from the chosen solution. In essence, rather than solving $u = D^{-1}\sigma$ subject to $u = f$, it suffices to compute $\sigma = Du$ and evaluate $f = u$ at the boundary. Whereas *inverting* a differential operator analytically is impossible for many equations and often requires ingenuity when it is, *applying* onw is a plug-and-chug application of algebra and calculus. Of course, it is necessary to construct u and any coefficient functions in D from simple, differentiable and integrable functions.

Also, u must satisfy any constraints imposed by the algorithm such as hard-coded boundary conditions or requirements for positive coefficients. Finally, the chosen functions should have small derivatives so that convergence can be achieved for reasonable grid sizes. Since these functions may need to be fairly complicated in order to achieve full generality while meeting the necessary constraints, it is advisable to use a *Computer Algebra System* (CAS) to perform the symbolic manipulations involved in computing σ .

Application

In order to verify the numerical implementations described in the previous chapters, it is necessary to first choose a manufactured solution $L(\mathbf{x}, \boldsymbol{\omega})$ to the radiative transfer equation, as well as coefficient functions for the absorption coefficient $a(\mathbf{x})$ and volume scattering function $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$. For the particular implementations developed in this thesis, the following conditions must be met by the manufactured solution:

1. Periodic solution and absorption coefficient in x and y
2. Positive solution and absorption coefficient
3. Position-independent surface downwelling boundary condition
4. Zero upwelling radiance at the bottom boundary
5. Properly normalized VSF $\beta(\Delta)$, as described in Section 3.1.3

The chosen functions are

$$\begin{aligned} L(x, y, z, \theta, \phi) &= \alpha (\sin(\phi + \theta) + 1) \\ &\cdot \left(z \left(\sin\left(\frac{2\pi x}{\alpha}\right) + \sin\left(\frac{2\pi y}{\alpha}\right) \right) + 1 \right) \\ &\cdot \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right), \end{aligned} \quad (6.3)$$

$$a(x, y, z) = \sin\left(\frac{2\pi x}{\alpha}\right) + \sin\left(\frac{2\pi y}{\alpha}\right) + \tanh(-\gamma + z) + 5, \quad (6.4)$$

$$\beta(\Delta) = \frac{\Delta + 1}{4\pi}, \quad (6.5)$$

where $\alpha = x_{max} - x_{min} = y_{max} - y_{min}$ is the domain width, and $\gamma = z_{max} - z_{min}$ is the domain depth. Using the python package Sympy, the boundary conditions and

source function are calculated to be

$$f(\theta, \phi) = \alpha (-\gamma + 1) (\sin(\phi + \theta) + 1), \quad (6.6)$$

$$\begin{aligned} \sigma(x, y, z, \theta, \phi) = & \alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) (\sin(\phi + \theta) + 1) \\ & \cdot \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right) \left(b + \sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) + \tanh(-\gamma + z) + 5 \right) \\ & - b \left[\frac{\alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) \left(\frac{\sin(\phi)\sin(\theta)}{3} + \frac{\cos(\phi)}{3} \right) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right)}{4\pi} \right. \\ & - \frac{\alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right)}{4\pi} \\ & \cdot \left(-\frac{\pi \sin(\phi)\sin(\theta)}{2} - \frac{\sin(\phi)\sin(\theta)}{3} - \frac{\cos(\phi)}{3} \right) \\ & - \frac{\alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right)}{4\pi} \\ & \cdot \left(\frac{\sin(\phi)\sin(\theta)}{3} - \frac{2\pi \sin(\phi)\cos(\theta)}{3} + \frac{\cos(\phi)}{3} - 2\pi \right) \\ & + \frac{\alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right)}{4\pi} \\ & \cdot \left(-\frac{\pi \sin(\phi)\sin(\theta)}{2} - \frac{\sin(\phi)\sin(\theta)}{3} + \frac{2\pi \sin(\phi)\cos(\theta)}{3} - \frac{\cos(\phi)}{3} + 2\pi \right) \Big] \\ & + 2\pi z (\sin(\phi + \theta) + 1) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right) \sin(\phi)\sin(\theta)\cos\left(\frac{2\pi y}{\alpha}\right) \\ & + 2\pi z (\sin(\phi + \theta) + 1) \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right) \sin(\phi)\cos(\theta)\cos\left(\frac{2\pi x}{\alpha}\right) \\ & + \left[\alpha \left(z \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) + 1 \right) \right. \\ & \cdot \left(\frac{(-b-1)(-\tanh^2((b+1)(\gamma-z))+1)}{\tanh(\gamma(b+1))} + 1 \right) (\sin(\phi + \theta) + 1) \\ & + \alpha \left(\sin \left(\frac{2\pi x}{\alpha} \right) + \sin \left(\frac{2\pi y}{\alpha} \right) \right) (\sin(\phi + \theta) + 1) \\ & \cdot \left. \left(-\gamma + z + \frac{\tanh((b+1)(\gamma-z))}{\tanh(\gamma(b+1))} \right) \right] \cos(\phi) \end{aligned} \quad (6.7)$$

Finite Difference

The only source of error in the finite difference method is discretization error due to the truncation of terms in the Taylor Series during the construction of the finite difference formulas in Section 4.5.1. Each of the derivatives in this implementation are approximated by two terms in the Taylor series: the leading order and first order terms. Therefore, the error should be second order in grid spacing. That is, when the grid spacing is sufficiently small, decreasing it further should result in the discretization error approaching zero quadratically. Of course, if large grid spacings are used, then quadratic convergence may not be observed even in a correct implementation since higher order terms in the Taylor series error will dominate the quadratic term. But for small grid spacings, the second order term dominates. The range of small grid spacings for which the lowest order term dominates is known as the *asymptotic range*. Whenever verification is performed, it is essential that the asymptotic range is achieved, either by using finer grids or using a solution and variable coefficients with smaller derivatives.

In the particular case of the radiative transfer equation in three spatial dimensions, there are five discretized variables (x, y, z, θ, ϕ) . A five dimensional resolution space is nontrivial to characterize, so we define generic spatial and angular resolutions for the sake of reducing dimensionality. Let $n_s = n_x = n_y = n_z$ and $n_a = n_\theta = n_\phi$. Then, we use the geometric mean to describe the spatial and angular

resolution, as

$$ds = (dx dy dz)^{1/3}, \quad (6.8)$$

$$da = (d\theta d\phi)^{1/3}. \quad (6.9)$$

This reduces the dimensionality of the resolution space to two, but big oh convergence is really only well-defined for a single variable. Therefore, the finite difference verification is performed by holding $n_a = 8$, and varying n_s between 4 and 64. As shown in Figure 6.1, second order convergence is observed, demonstrating that the code works without order-of-accuracy mistakes.

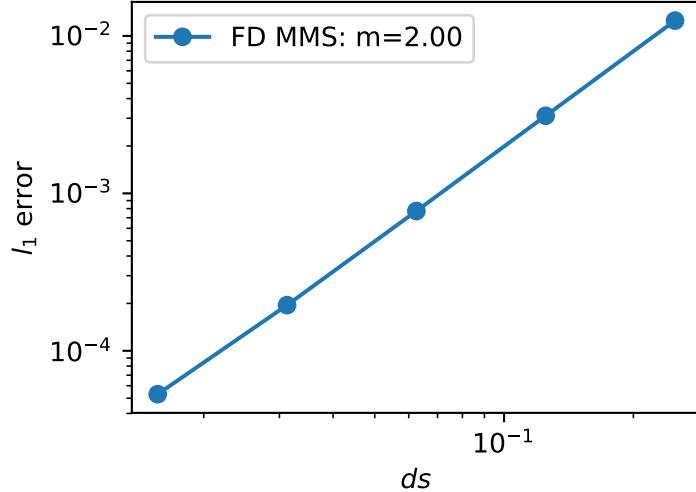


Figure 6.1: Code verification for the finite difference solution via the Method of Manufactured Solutions. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. The average absolute difference between the analytical and numerical solutions is shown. A slope of $m = 2$ on a log-log scale demonstrates second order convergence, as expected. This demonstrates the correctness of the code.

The quantity that we compare is *perceived irradiance*, which is different than the simple mean 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

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

Numerical Asymptotics

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 ??, 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.

6.1.2 Verification of Calculations

Richardson Extrapolation

Richardson extrapolation, is a technique for estimating the continuum value of a scalar functional derived from a solution to a differential equation by using values of the scalar obtained from numerical solutions on several different grids. The technique was developed by Richardson in 1912 with an application paper related to a stresses on a dam, and is also known as h^2 extrapolation since it was originally applied to a second order method.

Let the scalar of interest be called ϕ and the grid spacing h . Denote the exact solution as

$$\phi_e = \lim_{h \rightarrow 0} \phi(h), \quad (6.10)$$

The crux of the technique is to assume that discretization error can be written as a linear combination of powers of h , as in a Taylor series. That is,

$$\phi - \phi_e = g_0 + g_1 h + g_2 h^2 + g_3 h^3 + \dots . \quad (6.11)$$

Assuming that a second order numerical method is used, the first two terms on the right hand side are zero. For a first order method, only the first term is necessarily zero. Of course, in a “zeroth order” method, the absolute error is bounded from below by $|g_0|$, and so does not approach zero as the grid is refined. “Zeroth order” methods are also known as “incorrect.”

The original techniques involves numerical solutions on two grids with spacings $h_1 < h_2$ (i.e., grid 1 is finer), from which scalars ϕ_1 and ϕ_2 are calculated respectively. The ratio $r = h_2/h_1$ is called the grid refinement ratio.

Then,

$$\phi_1 = \phi_e + g_2 h_1^2 + O(h^3),$$

$$\phi_2 = \phi_e + g_2 h_2^2 + O(h^3).$$

Solving for g_2 yields

$$g_2 = \frac{\phi_1 - \phi_e}{h_1^2} + O(h^3),$$

so

$$\begin{aligned}\phi_1 &= \phi_e + \frac{h_2^2}{h_1^2}(\phi_1 - \phi_e) + O(h^3) \\ &= \phi_e + r^2(\phi_1 - \phi_e) + O(h^3) \\ &= \phi_e(1 - r^2) + \phi_1 r^2 + O(h^3).\end{aligned}$$

Hence, the approximate continuum solution is

$$\phi_e = \frac{\phi_2 - \phi_1 r^2}{1 - r^2} + O(h^3).$$

In essence, Richardson extrapolation allows for ϕ values from two solutions of order h^p to be combined to produce an approximation of order h^{p+1} to the continuum value of ϕ .

6.1.3 Modified Richardson Extrapolation

Of course, the above equations are only approximations. In reality, higher order terms introduce noise which may distort the extrapolated value when only two grid sizes are used. In order to reduce this noise, the concept can be easily generalized to incorporate more than two numerical solutions, as follows.

From

$$\phi \approx \phi_e + g_2 h^2,$$

it is clear that ϕ is linear in h^2 . Therefore, a simple linear fit through multiple points in (h^2, ϕ) space yields ϕ_e as the y-intercept, and the slope, g_2 , can be discarded. If

significant noise due to outliers still distorts the extrapolated values during fitting, a robust fitting algorithm such as Huber [30] or Ridge [11] regression can help reduce the influence of outliers.

Systematic Grid Convergence Study

6.2 Computational Resource Requirements

6.2.1 Finite Difference

Numerical Asymptotics

6.3 Rules of Thumb

6.3.1 What grid resolution to use?

Finite Difference

Numerical Asymptotics

6.3.2 Where is asymptotics valid

6.3.3 How many scattering events should be used?

6.4 Comparison to Other Light Models

6.5 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 6.7 and Figure 6.8, the solution is shown to diverge when the ratio b/a is too large, as in Section ??.

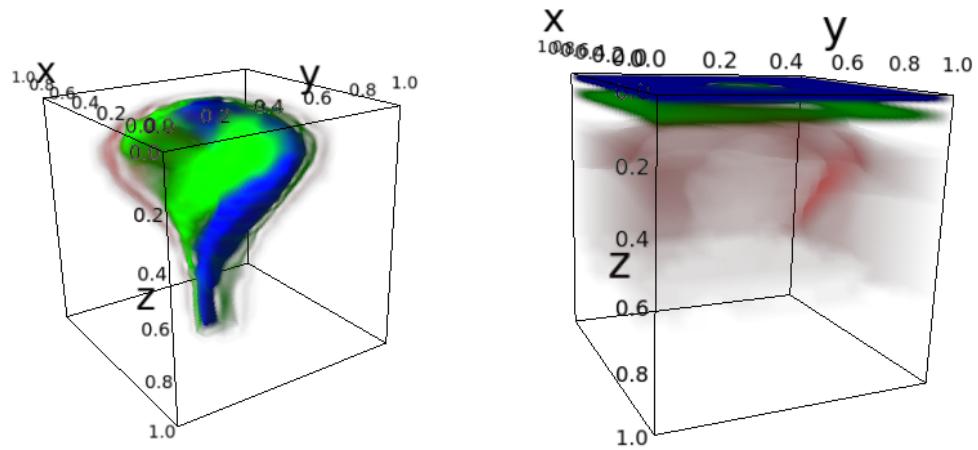


Figure 6.2: *top-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

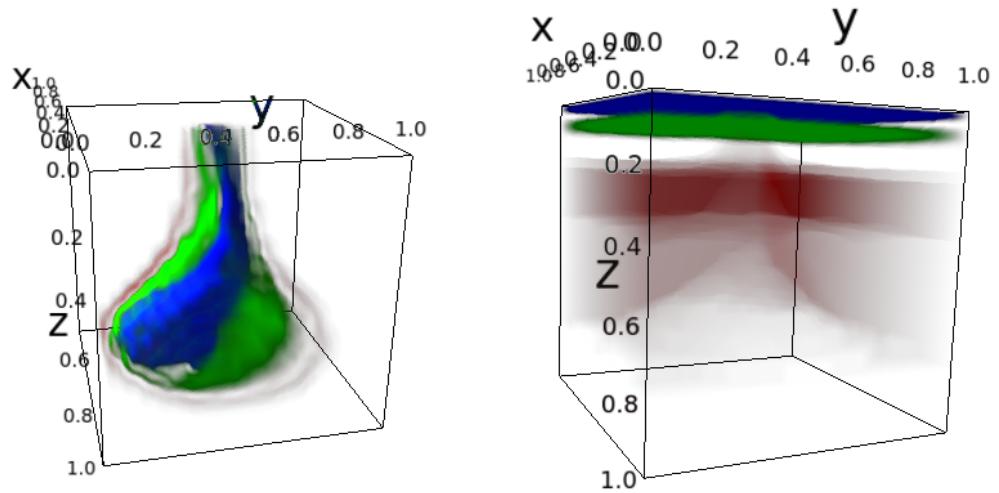


Figure 6.3: *bottom-heavy* kelp distribution (left) and no-scattering irradiance profile (right)

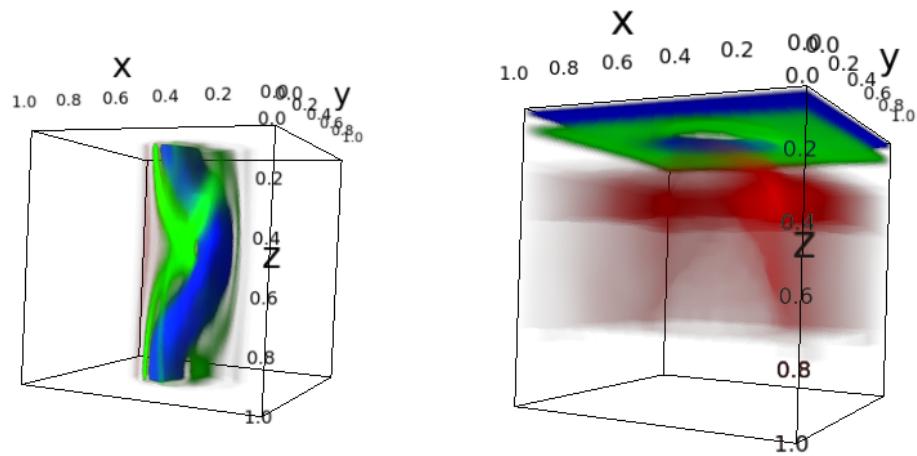


Figure 6.4: *uniform* kelp distribution (left) and no-scattering irradiance profile (right)

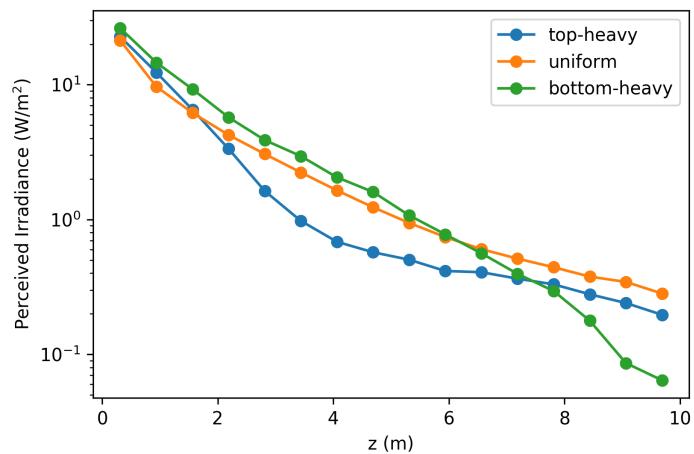


Figure 6.5: Several kelp profiles

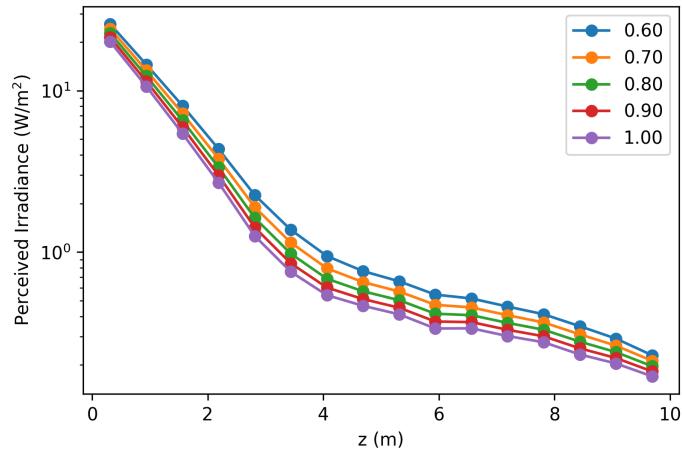


Figure 6.6: Several values of kelp absorptance

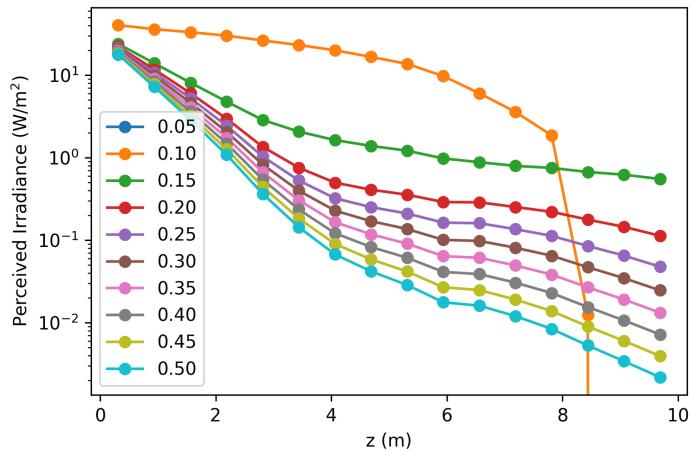


Figure 6.7: Several values of absorption coefficient of water

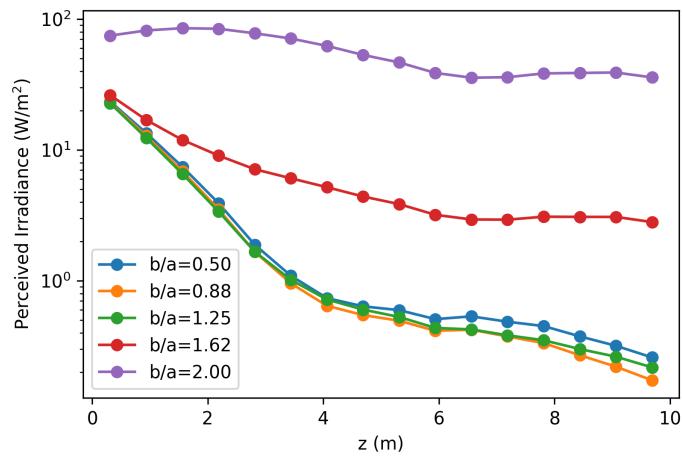


Figure 6.8: Several values of scattering coefficient

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 [17]. 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] V. Brzeski and G. Newkirk. Integrated coastal food production systems a review of current literature. page 17, July 1996.
- [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] S. Chandrasekhar. *Radiative Transfer*. Dover, 1960.
- [6] T. Chopin, A. H. Buschmann, C. Halling, M. Troell, N. Kautsky, A. Neori, G. P. Kraemer, J. A. Zertuche-Gonzalez, C. Yarish, and C. Neefus. Integrating Seaweeds into Marine Aquaculture Systems: a Key Toward Sustainability. *Journal of Phycology*, 37(6):975–986, Dec. 2001.

- [7] M. F. Colombo-Pallotta, E. García-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.
- [8] 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.
- [9] S. Hadley, K. Wild-Allen, C. Johnson, and C. Macleod. Modeling macroalgae growth and nutrient dynamics for integrated multi-trophic aquaculture. *Journal of Applied Phycology*, 27(2):901–916, Apr. 2015.
- [10] A. Handå, S. Forbord, X. Wang, O. J. Broch, S. W. Dahle, T. R. Størseth, K. I. Reitan, Y. Olsen, and J. Skjermo. Seasonal and depth-dependent growth of cultivated kelp (*Saccharina latissima*) in close proximity to salmon (*Salmo salar*) aquaculture in Norway. *Aquaculture*, 414-415:191–201, Nov. 2013.
- [11] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, Feb. 1970.
- [12] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 95(4):611–624, 1987.
- [13] D. G. Jones. Corn-Based Ethanol Production in the United States and the Propensity for Pesticide Use. page 47.

- [14] J. K. Kim, G. P. Kraemer, and C. Yarish. Field scale evaluation of seaweed aquaculture as a nutrient bioextraction strategy in Long Island Sound and the Bronx River Estuary. *Aquaculture*, 433:148–156, Sept. 2014.
- [15] C. Mobley. *Light and Water: Radiative Transfer in Natural Waters*. Academic Press, 1994.
- [16] C. Mobley. Radiative Transfer in the Ocean. In *Encyclopedia of Ocean Sciences*, pages 2321–2330. Elsevier, 2001.
- [17] C. Norvik. Design of Artificial Seaweeds for Assessment of Hydrodynamic Properties of Seaweed Farms. 2017.
- [18] 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.
- [19] M. Petkova and V. Springel. A novel approach for accurate radiative transfer in cosmological hydrodynamic simulations. *Monthly Notices of the Royal Astronomical Society*, 415(4):3731–3749, Aug. 2011.
- [20] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.
- [21] 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.

- [22] 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.
- [23] T. Searchinger, R. Heimlich, R. A. Houghton, F. Dong, A. Elobeid, J. Fabiosa, S. Tokgoz, D. Hayes, and T.-H. Yu. Use of U.S. Croplands for Biofuels Increases Greenhouse Gases Through Emissions from Land-Use Change. *Science*, 319(5867):1238–1240, Feb. 2008.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] 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.

- [28] Y. Yang. *Kelp Farming for Nutrient Bioextraction and Bioenergy Recovery from Ocean Outfalls of Publically-owned Treatment Works: A Thesis*. PhD Thesis, Clarkson University, 2015.
- [29] A. Yoshimori, T. Kono, and H. Iizumi. Mathematical models of population dynamics of the kelp *Laminaria religiosa*, with emphasis on temperature dependence. *Fisheries Oceanography*, 7(2):136–146, 1998.
- [30] C. Yu, W. Yao, and X. Bai. Robust Linear Regression: A Review and Comparison. *arXiv:1404.6274 [stat]*, Apr. 2014. arXiv: 1404.6274.

APPENDICES

APPENDIX A

GRID DETAILS

The width of the spatial grid cells in each dimension are

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

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

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

and the cell centers as

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

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

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

Denote the edges as

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

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

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

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

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

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

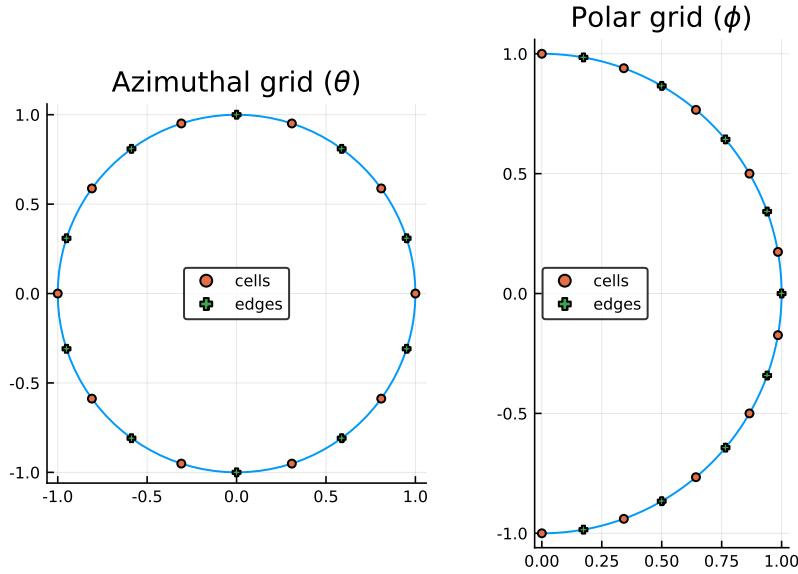


Figure A.1: Angular grid

Note that while θ has its final edge following its final center, this is not the case for ϕ , as seen in Figure A.1.

Because angles are indexed by a single integer p , there is a one-to-one relationship between an integer p and a pair (l, m) . The relationships are as follows:

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

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

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

Accordingly, define

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

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

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

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

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

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 discrete 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, $\mathbf{p}_0 = \mathbf{x}_0(\mathbf{p}_1, \omega) = (p_{0x}, p_{0y}, p_{0z})$ (as in (3.3)) to grid edges in each dimension separately. Given

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

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

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

the path lengths at which the ray intersects with edges in each dimension are calculated to be

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

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

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

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

$$\delta_x = \text{sign}(p_{0x} - p_{1x}), \quad (\text{B.7})$$

$$\delta_y = \text{sign}(p_{0y} - p_{1y}), \quad (\text{B.8})$$

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

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

$$\sigma_x = (\delta_x + 1)/2 \quad (\text{B.10})$$

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

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

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,

$$i_0 = \text{ceil} \left(\frac{p_{0x} - x_{\min}}{dx} \right) \quad (\text{B.13})$$

$$j_0 = \text{ceil} \left(\frac{p_{0y} - y_{\min}}{dy} \right) \quad (\text{B.14})$$

$$k_0 = \text{ceil} \left(\frac{p_{0z} - z_{\min}}{dz} \right) \quad (\text{B.15})$$

Then,

$$i_0^e = i_0 + \sigma_x \quad (\text{B.16})$$

$$j_0^e = j_0 + \sigma_y \quad (\text{B.17})$$

$$k_0^e = k_0 + \sigma_z \quad (\text{B.18})$$

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

$$s_i^x = \hat{s} \frac{x_i^e - p_{0x}}{p_{1x} - p_{0x}} \quad (\text{B.19})$$

$$s_j^y = \hat{s} \frac{y_j^e - p_{0y}}{p_{1y} - p_{0y}} \quad (\text{B.20})$$

$$s_k^z = \hat{s} \frac{z_k^e - p_{0z}}{p_{1z} - p_{0z}} \quad (\text{B.21})$$

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 whichever dimension is crossed soonest.

As each cell is traversed, the absorption coefficient and effective source are saved for use in the ray integral for the numerical calculation of the asymptotic approximation. For full implementation details, see the `traverse_ray` subroutine in `asymptotics.f90` in Appendix C.

APPENDIX C

FORTRAN CODE

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

<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 1D array to file
658 subroutine write_vec(arr,nn,filename,fmtstr_in)
659     implicit none
660
661     ! INPUTS:
662     ! arr - array to print
663     double precision, dimension (nn), intent(in)
664             :: arr
665     ! nn - number of data rows in file
666     ! nn - number of data columns in file
667     integer, intent(in) :: nn
668     ! filename - file to write to
669     character(len=*) filename
670     ! fmtstr - output format (no parentheses,
671             ! don't specify columns)
672     ! e.g. 'E10.2', not '(2E10.2)'
673     character(len=*), optional :: fmtstr_in
674     character(len=256) fmtstr
675
676     ! NO OUTPUTS
677
678     ! BODY
679
680     ! Row counter
681     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,A,A)') '(', trim(fmtstr), '
697      ! Loop through rows
698      do ii = 1, nn
699          ! Print entry per row
700          write(un,finfmt) arr(ii)
701      end do
702
703      ! Close file
704      close(unit=un)
705
706  end subroutine
707
708  ! Write 2D array to file
709  subroutine write_array(arr,nn,mm,filename,
710      fmtstr_in)
711      implicit none
712
713      ! INPUTS:
714      ! arr - array to print
715      double precision, dimension (nn,mm), intent(
716          in) :: arr
717      ! nn - number of data rows in file
718      ! nn - number of data columns in file
719      integer, intent(in) :: nn, mm
720      ! filename - file to write to
721      character(len=*) filename
722      ! fmtstr - output format (no parentheses,
723          ! don't specify columns)
724      ! e.g. 'E10.2', not '(2E10.2)'
725      character(len=*), optional :: fmtstr_in
726      character(len=256) fmtstr
727
728      ! NO OUTPUTS

```

```

726      ! BODY
727
728      ! Row counter
729      integer ii
730      ! Final format to use
731      character(len=256) finfmt
732      ! Dummy file unit to use
733      integer, parameter :: un = 20
734
735      ! Open file for writing
736      open(unit=un, file=trim(filename), status='
737          replace', form='formatted')
738
739      ! Determine string format
740      if(present(fmtstr_in)) then
741          fmtstr = fmtstr_in
742      else
743          fmtstr = 'E10.2'
744      end if
745
746      ! Generate final format string
747      write(finfmt,'(A,I4,A,A)') '(', mm, trim(
748          fmtstr), ')'
749
750      ! Loop through rows
751      do ii = 1, nn
752          ! Print one row at a time
753          write(un,finfmt) arr(ii,:)
754      end do
755
756      ! Close file
757      close(unit=un)
758
759  end subroutine
760
761  subroutine zeros(x, n)
762      implicit none
763      integer n, i
764      double precision, dimension(n) :: x
765
766      do i=1, n
767          x(i) = 0
768      end do
769  end subroutine zeros
770
771 end module

```

sag.f90

```

1 | module sag
2 | use utils

```

```

3 | use fastgl
4 |
5 | implicit none
6 |
7 | ! Spatial grids do not include upper endpoints.
8 | ! Angular grids do include upper endpoints.
9 | ! Both include lower endpoints.
10|
11| ! To use:
12| ! call grid%set_bounds(...)
13| ! call grid%set_num(...) (or set_uniform_spacing
|   )
14| ! call grid%init()
15| ! ...
16| ! call grid%deinit()
17|
18|!integer, parameter :: pi = 3.141592653589793D
|  +00
19|
20|type index_list
21|  integer i, j, k, p
22|contains
23|  procedure :: init => index_list_init
24|  procedure :: print => index_list_print
25|end type index_list
26|
27|type angle2d
28|  integer ntheta, nphi, nomega
29|  double precision dtheta, dphi
30|  double precision, dimension(:), allocatable
|    :: theta, phi, theta_edge, phi_edge
31|  double precision, dimension(:), allocatable
|    :: theta_p, phi_p, theta_edge_p,
|      phi_edge_p
32|  double precision, dimension(:), allocatable
|    :: cos_theta, sin_theta, cos_phi, sin_phi
33|  double precision, dimension(:), allocatable
|    :: cos_theta_edge, sin_theta_edge,
|      cos_phi_edge, sin_phi_edge
34|  double precision, dimension(:), allocatable
|    :: cos_theta_p, sin_theta_p, cos_phi_p,
|      sin_phi_p
35|  double precision, dimension(:), allocatable
|    :: cos_theta_edge_p, sin_theta_edge_p,
|      cos_phi_edge_p, sin_phi_edge_p
36|  double precision, dimension(:), allocatable
|    :: area_p
37|contains
38|  procedure :: set_num => angle_set_num
39|  procedure :: phat, lhat, mhat
40|  procedure :: init => angle_init ! Call after
|    set_num

```

```

41   procedure :: integrate_points =>
42     angle_integrate_points
43   procedure :: integrate_func =>
44     angle_integrate_func
45   procedure :: deinit => angle_deinit
46 end type angle2d
47
48 type angle_dim
49   integer num
50   double precision minval, maxval, prefactor
51   double precision, dimension(:), allocatable
52     :: vals, weights, sin, cos
53 contains
54   procedure :: set_bounds => angle_set_bounds
55   procedure :: set_num => angle1d_set_num
56   procedure :: deinit => angle1d_deinit
57   procedure :: integrate_points =>
58     angle1d_integrate_points
59   procedure :: integrate_func =>
60     angle1d_integrate_func
61   procedure :: assign_linspace =>
62     angle1d_assign_linspace
63   procedure :: assign_legendre
64 end type angle_dim
65
66 type space_dim
67   integer num
68   double precision minval, maxval
69   double precision, dimension(:), allocatable
70     :: vals, edges, spacing
71 contains
72   procedure :: integrate_points =>
73     space_integrate_points
74   procedure :: trapezoid_rule
75   procedure :: set_bounds => space_set_bounds
76   procedure :: set_num => space_set_num
77   procedure :: set_uniform_spacing =>
78     space_set_uniform_spacing
79   !procedure :: set_num_from_spacing
80   procedure :: set_uniform_spacing_from_num
81   procedure :: set_spacing_array =>
82     space_set_spacing_array
83   procedure :: deinit => space_deinit
84   procedure :: assign_linspace
85 end type space_dim
86
87 type space_angle_grid !(sag)
88   type(space_dim) :: x, y, z
89   type(angle2d) :: angles
90   double precision, dimension(:), allocatable :: 
91     x_factor, y_factor
92 contains
93   procedure :: set_bounds => sag_set_bounds

```

```

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

```

```

130 |     if(p .eq. 1) then
131 |         m = 1
132 |     else if(p .eq. angles%nomega) then
133 |         m = angles%nphi
134 |     else
135 |         m = ceiling(dble(p-1)/dble(angles%ntheta)
136 |                         ) + 1
137 |     end if
138 | end function mhat
139 |
140 | function phat(angles, l, m) result(p)
141 |     class(angle2d) :: angles
142 |     integer l, m, p
143 |
144 |     if(m .eq. 1) then
145 |         p = 1
146 |     else if(m .eq. angles%nphi) then
147 |         p = angles%nomega
148 |     else
149 |         p = (m-2)*angles%ntheta + 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 |                         ))

```

```

177 |     allocate(angles%cos_phi_edge(angles%nphi-1))
178 |     allocate(angles%sin_phi_edge(angles%nphi-1))
179 |     allocate(angles%cos_theta_edge_p(angles%
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 |         )
193 |     angles%sin_theta(l) = sin(angles%theta(l)
194 |         )
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)*
205 |             angles%dphi
205 |         angles%cos_phi_edge(m) = cos(angles%
206 |             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
211 |     area = angles%dtheta &

```

```

213      * (angles%cos_phi_edge(m-1) - angles
214          %cos_phi_edge(m))
215  do l=1, angles%ntheta
216      p = angles%phat(l, m)
217
218      angles%theta_p(p) = angles%theta(1)
219      angles%phi_p(p) = angles%phi(m)
220      angles%theta_edge_p(p) = angles%
221          theta_edge(1)
222      angles%phi_edge_p(p) = angles%phi_edge
223          (m)
224
225      angles%cos_theta_p(p) = cos(angles%
226          theta_p(p))
227      angles%sin_theta_p(p) = sin(angles%
228          theta_p(p))
229      angles%cos_phi_p(p) = cos(angles%phi_p
230          (p))
231      angles%sin_phi_p(p) = sin(angles%phi_p
232          (p))
233
234      angles%cos_theta_edge_p(p) = cos(
235          angles%theta_edge_p(p))
236      angles%sin_theta_edge_p(p) = sin(
237          angles%theta_edge_p(p))
238      angles%cos_phi_edge_p(p) = cos(angles%
239          phi_edge_p(p))
240      angles%sin_phi_edge_p(p) = sin(angles%
241          phi_edge_p(p))
242
243      angles%area_p(p) = area
244  end do
245  end do
246
247 ! Poles
248 l=1
249 area = 2.d0*pi*(1.d0-cos(angles%dphi/2.d0))
250
251 ! North Pole
252 p = 1
253 m=1
254 angles%theta_p(p) = angles%theta(1)
255 angles%theta_edge_p(p) = angles%theta_edge(1
256 )
257 angles%phi_p(p) = angles%phi(m)
258 ! phi_edge_p only defined up to nphi-1.
259 angles%phi_edge_p(p) = angles%phi_edge(m)
260 angles%cos_theta_p(p) = cos(angles%theta_p(p
261 )))

```

```

249 |     angles%sin_theta_p(p) = sin(angles%theta_p(p
250 |         ))
250 |     angles%cos_phi_p(p) = cos(angles%phi_p(p))
251 |     angles%sin_phi_p(p) = sin(angles%phi_p(p))
252 |     angles%cos_theta_edge_p(p) = cos(angles%
252 |         theta_edge_p(p))
253 |     angles%sin_theta_edge_p(p) = sin(angles%
253 |         theta_edge_p(p))
254 |     angles%cos_phi_edge_p(p) = cos(angles%
254 |         phi_edge_p(p))
255 |     angles%sin_phi_edge_p(p) = sin(angles%
255 |         phi_edge_p(p))
256 |     angles%area_p(p) = area
257 |
258 | ! South Pole
259 | p = angles%nomega
260 | m = angles%nphi
261 | angles%theta_p(p) = angles%theta(l)
262 | angles%theta_edge_p(p) = angles%theta_edge(l
262 |     )
263 | angles%phi_p(p) = angles%phi(m)
264 | angles%cos_theta_p(p) = cos(angles%theta_p(p
264 |     ))
265 | angles%sin_theta_p(p) = sin(angles%theta_p(p
265 |     ))
266 | angles%cos_phi_p(p) = cos(angles%phi_p(p))
267 | angles%sin_phi_p(p) = sin(angles%phi_p(p))
268 | angles%area_p(p) = area
269 | end subroutine angle_init
270 |
271 | ! Integrate function given function values at
271 |     grid cells
272 | function angle_integrate_points(angles ,
272 |     func_vals) result(integral)
273 | class(angle2d) :: angles
274 | double precision , dimension(angles%nomega)
274 |     :: func_vals
275 | double precision integral
276 | integer p
277 |
278 | integral = 0.d0
279 |
280 | do p=1 , angles%nomega
281 |     integral = integral + angles%area_p(p) *
281 |         func_vals(p)
282 | end do
283 |
284 | end function angle_integrate_points
285 |

```

```

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

```

```

333      deallocate(angles%area_p)
334  end subroutine angle_deinit
335
336
337  !!! ANGLE 1D !!!
338
339  subroutine angle_set_bounds(angle, minval,
340      maxval)
341      class(angle_dim) :: angle
342      double precision minval, maxval
343      angle%minval = minval
344      angle%maxval = maxval
345  end subroutine angle_set_bounds
346
347  subroutine angle1d_set_num(angle, num)
348      class(angle_dim) :: angle
349      integer num
350      angle%num = num
351  end subroutine angle1d_set_num
352
353  subroutine angle1d_assign_linspace(angle)
354      class(angle_dim) :: angle
355      double precision spacing
356      integer i
357
358      spacing = (angle%maxval - angle%minval) /
359          dble(angle%num)
360      do i=1, angle%num
361          angle%vals(i) = (i-1) * spacing
362      end do
363  end subroutine angle1d_assign_linspace
364
365  ! To calculate  $\int_{xmin}^{xmax} f(x) dx$  :
366  ! int = prefactor * sum(weights * f(roots))
367  subroutine assign_legendre(angle)
368      class(angle_dim) :: angle
369      double precision root, weight, theta
370      integer i
371      ! glpair produces both x and theta, where x=
372          cos(theta). We'll throw out theta.
373
374      allocate(angle%vals(angle%num))
375      allocate(angle%weights(angle%num))
376      allocate(angle%sin(angle%num))
377      allocate(angle%cos(angle%num))
378
379      ! Prefactor for integration
380      ! From change of variables
381      angle%prefactor = (angle%maxval - angle%
382          minval) / 2.d0

```

```

380 |     do i = 1, angle%num
381 |         call glpair(angle%num, i, theta, weight,
382 |                         root)
383 |         call affine_transform(root, -1.d0, 1.d0,
384 |                         angle%minval, angle%maxval)
385 |         angle%vals(i) = root
386 |         angle%weights(i) = weight
387 |         angle%sin(i) = sin(root)
388 |         angle%cos(i) = cos(root)
389 |     end do
390 |
391 | end subroutine assign_legendre
392 |
393 ! Integrate callable function over angle via
394 ! Gauss-Legendre quadrature
395 |
396 function angle1d_integrate_func(angle,
397     func_callable) result(integral)
398 class(angle_dim) :: angle
399 double precision, external :: func_callable
400 double precision, dimension(:), allocatable
401 :: func_vals
402 double precision integral
403 integer i
404 |
405 allocate(func_vals(angle%num))
406 |
407 do i=1, angle%num
408     func_vals(i) = func_callable(angle%vals(i))
409 end do
410 |
411 integral = angle%integrate_points(func_vals)
412 |
413 deallocate(func_vals)
414 end function angle1d_integrate_func
415 |
416 ! Integrate function given function values
417 ! sampled at legendre theta values
418 function angle1d_integrate_points(angle,
419     func_vals) result(integral)
420 class(angle_dim) :: angle
421 double precision, dimension(angle%num) ::
422     func_vals
423 double precision integral
424 |
425 integral = angle%prefactor * sum(angle%
426     weights * func_vals)
427 end function angle1d_integrate_points
428 |
429 subroutine angle1d_deinit(angle)

```

```

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

```

463  subroutine space_set_uniform_spacing(space ,
464      spacing)
465      class(space_dim) :: space
466      double precision spacing
467      integer k
468      do k=1, space%num
469          space%spacing(k) = spacing
470      end do
471  end subroutine space_set_uniform_spacing
472
473  subroutine space_set_spacing_array(space ,
474      spacing)
475      class(space_dim) :: space
476      double precision , dimension(space%num) :: :
477          spacing
478      space%spacing = spacing
479  end subroutine space_set_spacing_array
480
481  subroutine assign_linspace(space)
482      class(space_dim) :: space
483      double precision spacing
484      integer i
485
486      allocate(space%vals(space%num))
487      allocate(space%edges(space%num))
488      allocate(space%spacing(space%num))
489
490      spacing = spacing_from_num(space%minval ,
491          space%maxval , space%num)
492      call space%set_uniform_spacing(spacing)
493
494      do i=1, space%num
495          space%edges(i) = space%minval + dble(i-1)
496              * space%spacing(i)
497          space%vals(i) = space%minval + dble(i-0.5
498              d0) * space%spacing(i)
499      end do
500
501  end subroutine assign_linspace
502
503  subroutine set_uniform_spacing_from_num(space)
504      ! Create evenly spaced grid (linspace)
505      class(space_dim) :: space
506      double precision spacing
507
508      spacing = spacing_from_num(space%minval ,
509          space%maxval , space%num)
510      call space%set_uniform_spacing(spacing)
511
512  end subroutine set_uniform_spacing_from_num

```

```

507 ! subroutine set_num_from_spacing(space)
508 !   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
528   call grid%x%set_bounds(xmin, xmax)
529   call grid%y%set_bounds(ymin, ymax)
530   call grid%z%set_bounds(zmin, zmax)
531 end subroutine sag_set_bounds
532
533 subroutine sag_set_uniform_spacing(grid, dx,
534   dy, dz)
535   class(space_angle_grid) :: grid
536   double precision dx, dy, dz
537
538   call grid%x%set_uniform_spacing(dx)
539   call grid%y%set_uniform_spacing(dy)
540   call grid%z%set_uniform_spacing(dz)
541
542 end subroutine sag_set_uniform_spacing
543
544 subroutine sag_set_num(grid, nx, ny, nz,
545   ntheta, nphi)
546   class(space_angle_grid) :: grid
547   integer nx, ny, nz, ntheta, nphi
548
549   call grid%x%set_num(nx)
550   call grid%y%set_num(ny)
551   call grid%z%set_num(nz)
552
553   call grid%angles%set_num(ntheta, nphi)
554
555 end subroutine sag_set_num
556
557 subroutine sag_init(grid)
558   class(space_angle_grid) :: grid
559
560   call grid%x%assign_linspace()

```

```

552 |     call grid%y%assign_linspace()
553 |     call grid%z%assign_linspace()
554 |
555 |     call grid%angles%init()
556 |     call grid%calculate_factors()
557 |
558 end subroutine sag_init
559
560 subroutine sag_calculate_factors(grid)
561 ! Factors by which depth difference is
562 ! multiplied
563 ! in order to calculate distance traveled in
564 ! the
565 ! (x, y) direction along a ray in the (theta
566 ! , phi)
567 ! direction
568 class(space_angle_grid) :: grid
569 integer p, nomega
570 double precision theta, phi
571
572 nomega = grid%angles%nomega
573
574 allocate(grid%x_factor(nomega))
575 allocate(grid%y_factor(nomega))
576
577 do p=1, nomega
578     theta = grid%angles%theta_p(p)
579     phi = grid%angles%phi_p(p)
580     grid%x_factor(p) = tan(phi) * cos(theta)
581     grid%y_factor(p) = tan(phi) * sin(theta)
582 end do
583
584 end subroutine sag_calculate_factors
585
586 subroutine sag_set_uniform_spacing_from_num(
587     grid)
588 class(space_angle_grid) :: grid
589 call grid%x%set_uniform_spacing_from_num()
590 call grid%y%set_uniform_spacing_from_num()
591 call grid%z%set_uniform_spacing_from_num()
592 end subroutine
593 sag_set_uniform_spacing_from_num
594
595 ! subroutine sag_set_num_from_spacing(grid)
596 !     class(space_angle_grid) :: grid
597 !     call grid%x%set_num_from_spacing()
598 !     call grid%y%set_num_from_spacing()
599 !     call grid%z%set_num_from_spacing()
600
601 ! end subroutine sag_set_num_from_spacing

```

```

598  subroutine sag_deinit(grid)
599      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)
622     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
6 ! each depth, generate kelp distribution at
7 ! each point in 3D space
8
9 module kelp3d
10
11 use kelp_context
12 implicit none

```

```

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, n_images,
45 |   num_threads)
46 |   type(space_angle_grid), intent(in) :: grid
47 |   type(frond_shape), intent(in) :: frond
48 |   type(rope_state), intent(in) :: rope
49 |   type(point3d) point
50 |   integer, intent(in) :: quadrature_degree
51 |   integer, optional :: n_images
52 |   double precision, dimension(grid%x%num, grid%y
53 |     %num, grid%z%num) :: p_kelp

```

```

54 ! n_images=2 => 5x5 meta-grid (only necessary
55     for very dense kelp ropes)
56 integer im_i, im_j
57 double precision x_width, y_width
58 x_width = grid%x%maxval - grid%x%minval
59 y_width = grid%y%maxval - grid%y%minval
60
61 if(.not. present(n_images)) then
62     n_images = 1
63 end if
64
65 nx = grid%x%num
66 ny = grid%y%num
67 nz = grid%z%num
68
69 p_kelp(:,:,:,:) = 0
70
71 !$omp parallel do default(shared) private(x,y,
72     z) &
73 !$omp firstprivate(point,depth) &
74 !$omp private(i,j,k,im_i,im_j) shared(nx,ny,nz
75     ,n_images) &
76 !$omp shared(frond,rope,grid,quadrature_degree
77     ) &
78 !$omp shared(p_kelp,x_width,y_width) &
79 !$omp num_threads(num_threads) collapse(3) &
80 !$omp schedule(dynamic, 10) ! 10 grid points
81     per thread
82 do k=1, nz
83     do i=1, nx
84         do j=1, ny
85             z = grid%z%vals(k)
86             call depth%set_depth(rope, grid, k)
87             do im_i=-n_images, n_images
88                 x = im_i*x_width + grid%x%vals(i)
89                 do im_j=-n_images, n_images
90                     y = im_j*y_width + grid%y%vals(j)
91                     call point%set_cart(x, y, z)
92                     p_kelp(i, j, k) = p_kelp(i,j,k) +
93                         kelp_proportion(point, frond
94                             , grid, depth,
95                             quadrature_degree)
96                 end do
97             end do
98         end do
99     end do
100 !$omp end do
101 end subroutine calculate_kelp_on_grid

```

```

98 | subroutine shading_region_limits(theta_low_lim,
99 |   theta_high_lim, point, frond)
100| type(point3d), intent(in) :: point
101| type(frond_shape), intent(in) :: frond
102| double precision, intent(out) :: theta_low_lim
103|   , theta_high_lim
104|
105| theta_low_lim = point%theta - frond%alpha
106| theta_high_lim = point%theta + frond%alpha
107| end subroutine shading_region_limits
108|
109| function prob_kelp(point, frond, depth,
110|   quadrature_degree)
111| ! P_s(theta_p, r_p) - This is the proportion of
112|   the population of this depth layer which can
113|   be found in this Cartesian grid cell.
114| type(point3d), intent(in) :: point
115| type(frond_shape), intent(in) :: frond
116| type(depth_state), intent(in) :: depth
117| integer, intent(in) :: quadrature_degree
118| double precision prob_kelp
119| double precision theta_low_lim, theta_high_lim
120|
121| call shading_region_limits(theta_low_lim,
122|   theta_high_lim, point, frond)
123| prob_kelp = integrate_ps(theta_low_lim,
124|   theta_high_lim, quadrature_degree, point,
125|   frond, depth)
126| end function prob_kelp
127|
128| function kelp_proportion(point, frond, grid,
129|   depth, quadrature_degree)
130| ! This is the proportion of the volume of the
131|   Cartesian grid cell occupied by kelp
132| type(point3d), intent(in) :: point
133| type(frond_shape), intent(in) :: frond
134| type(depth_state), intent(in) :: depth
135| type(space_angle_grid), intent(in) :: grid
136| integer, intent(in) :: quadrature_degree
137| double precision p_k, n, t, dz
138| double precision kelp_proportion
139|
140| n = depth%num_fronds
141| dz = grid%z%spacing(depth%depth_layer)
142| t = frond%ft
143| !write(*,*) 'KELP PROPORTION'
144| !write(*,*) 'n=', n
145| !write(*,*) 'dz=', dz
146| !write(*,*) 't=', t
147| !write(*,*) 'coef=', n*t/dz

```

```

138     p_k = prob_kelp(point, frond, depth,
139                       quadrature_degree)
140     kelp_proportion = n*t/dz * p_k
141   end function kelp_proportion
142
142   function integrate_ps(theta_low_lim,
143                         theta_high_lim, quadrature_degree, point,
144                         frond, depth) result(integral)
143     type(point3d), intent(in) :: point
144     type(frond_shape), intent(in) :: frond
145     double precision, intent(in) :: theta_low_lim,
146                               theta_high_lim
146     integer, intent(in) :: quadrature_degree
147     type(depth_state), intent(in) :: depth
148     double precision integral
149     double precision, dimension(:), allocatable :: integrand_vals
150     integer i
151
152     type(angle_dim) :: theta_f
153     call theta_f%set_bounds(theta_low_lim,
154                               theta_high_lim)
154     call theta_f%set_num(quadrature_degree)
155     call theta_f%assign_legendre()
156
157     allocate(intrand_vals(theta_f%num))
158
159     do i=1, theta_f%num
160       integrand_vals(i) = ps_integrand(theta_f%
161                                         vals(i), point, frond, depth)
161     end do
162
163     integral = theta_f%integrate_points(
164                           integrand_vals)
164
165     deallocate(intrand_vals)
166     call theta_f%deinit()
167
168   end function integrate_ps
169
170   function ps_integrand(theta_f, point, frond,
171                         depth)
171     type(point3d), intent(in) :: point
172     type(frond_shape), intent(in) :: frond
173     type(depth_state), intent(in) :: depth
174     double precision theta_f, l_min
175     double precision angular_part, length_part
176     double precision ps_integrand
177

```

```

178     l_min = min_shading_length(theta_f, point,
179                               frond)
180
180     angular_part = depth%angle_distribution_pdf(
181                               theta_f)
181     length_part = 1 - depth%
181                               length_distribution_cdf(l_min)
182
183     ps_integrand = angular_part * length_part
184 end function ps_integrand
185
186 function min_shading_length(theta_f, point,
187                           frond) result(l_min)
187 ! L_min(\theta)
188 type(point3d), intent(in) :: point
189 type(frond_shape), intent(in) :: frond
190 double precision, intent(in) :: theta_f
191 double precision l_min
192 double precision tpp
193 double precision frond_frac
194
195 ! tpp === theta_p_prime
196 tpp = point%theta - theta_f + pi / 2.d0
197 frond_frac = 2.d0 * frond%fr / (1.d0 + frond%
197                               fs)
198 l_min = point%r * (sin(tpp) + angular_sign(tpp)
198                               ) * frond_frac * cos(tpp))
199 end function min_shading_length
200
201 ! function frond_edge(theta, theta_f, L, fs, fr)
202 ! ! r_f(\theta)
203 !   double precision, intent(in) :: theta,
203     theta_f, L, fs, fr
204 !   double precision, intent(out) :: frond_edge
205 !
206 !   frond_edge = relative_frond_edge(theta -
206     theta_f + pi/2.d0)
207 !
208 end function frond_edge
209 !
210 ! function relative_frond_edge(theta_prime, L,
210   fs, fr)
211 ! ! r_f'(\theta')
212 !   double precision, intent(in) :: theta_prime,
212     L, fs, fr
213 !   double precision, intent(out) ::
213     relative_frond_edge
214 !

```

```

215 !   relative_frond_edge = L / (sin(theta_prime)
216 !     + angular_sign(theta_prime * alpha(fs, fr) *
217 !       cos(theta_prime)))
218 ! end function relative_frond_edge
219
220 function angular_sign(theta_prime)
221 ! S(\theta') = sign(x')
222 double precision, intent(in) :: theta_prime
223 double precision angular_sign
224 angular_sign = sgn(cos(theta_prime))
225 end function angular_sign
226
227 subroutine gaussian_blur_2d(A, sigma, dx, dy, nk
228 , num_threads)
229 ! 2D Gaussian blur (periodic BC) with std
230 !   sigma
231 ! with kernel radius of nk (full size (2*nk+1)
232 !   x(2*nk+1))
233 ! applied to matrix A with element spacings dx
234 !   and dy.
235 double precision, intent(inout), dimension(:, :
236 ) :: A
237 double precision, intent(in) :: sigma, dx, dy
238 ! kernel half width
239 integer, intent(in) :: nk
240 ! kernel full width
241 integer kw
242 integer num_threads
243
244 ! A matrix size
245 integer nx, ny
246
247 ! indices
248 integer i1, j1
249 integer i2, j2
250 integer i, j
251 ! kernel
252 double precision, dimension(:, :, :), allocatable
253 !   :: k
254 ! output matrix
255 double precision, dimension(:, :, :), allocatable
256 !   :: B
257 ! kernel independent variables
258 double precision x, y
259
260 if(sigma > 0) then
261     nx = size(A, 1)
262     ny = size(A, 2)
263
264     kw = 2*nk + 1

```

```

257 |     allocate(B(nx, ny))
258 |     allocate(k(kw, kw))
259 |     write(*,*) 'creating kernel', sigma, nk
260 | ! Create kernel
261 | do i1=-nk, nk
262 |   x = i1*dx
263 |   i = i1+nk+1
264 |   do j1=-nk, nk
265 |     y = j1*dy
266 |     j = j1+nk+1
267 |     k(i,j) = exp(-(x**2+y**2)/(2*sigma**2))
268 |   end do
269 |
270 | end do
271 | ! normalize kernel
272 | k = k / sum(k)
273 |
274 | write(*,*) 'convolving'
275 | ! convolve
276 | !$omp parallel do default(private) private(x
277 |   ,y) &
278 |   !$omp private(i,j,i1,j1,i2,j2) shared(nx,ny,
279 |   nk,kw) &
280 |   !$omp shared(A,B,k) &
281 |   !$omp num_threads(num_threads) collapse(2) &
282 |   !$omp schedule(dynamic, 10) ! 10 grid points
283 |   per thread
284 | do i1=1, nx
285 |   do j1=1, ny
286 |     B(i1, j1) = 0
287 |     do i2=1, kw
288 |       do j2=1, kw
289 |         i = mod1(i1 - nk + i2 - 1, nx)
290 |         j = mod1(j1 - nk + j2 - 1, ny)
291 |         B(i1, j1) = B(i1, j1) + k(i2, j2)
292 |           * A(i, j)
293 |       end do
294 |     end do
295 |   end do
296 |   !$omp end parallel do
297 |   write(*,*) 'done convolving'
298 |
299 |   ! Update original matrix
300 |   A(:,:) = B(:,:)
301 |   deallocate(k)
302 |   deallocate(B)
303 |   write(*,*) 'gb2d done.'
304 | end if
305 end subroutine gaussian_blur_2d

```

```

303 | end module kelp3d

rte_sparse_matrices.f90

1 module rte_sparse_matrices
2 use sag
3 use kelp_context
4 use mgmres
5 use type_consts
6 !use hdf5_utils
7 #include "lisf.h"
8 implicit none
9
10 type solver_opts
11   integer maxiter_inner, maxiter_outer
12   double precision tol_abs, tol_rel
13 end type solver_opts
14
15 type rte_mat
16   type(space_angle_grid) grid
17   type(optical_properties) iops
18   type(solver_opts) params
19   integer nx, ny, nz, nomega
20   integer i, j, k, p
21   integer(index_kind) nonzero, n_total
22   integer x_block_size, y_block_size,
23         z_block_size, omega_block_size
24   double precision, dimension(:), allocatable
25         :: surface_vals
26
27   ! CSR format
28   ! http://www.scipy-lectures.org/advanced/
29   !     scipy_sparse/csr_matrix.html
30   ! with LIS method 2 (LIS manual, p.19)
31   integer(index_kind), dimension(:),
32         allocatable :: ptr, col
33   double precision, dimension(:), allocatable
34         :: data
35
36   ! Lis Matrix and vectors
37   LIS_MATRIX A
38   LIS_VECTOR b, x
39   LIS_SOLVER solver
40   LIS_INTEGER ierr
41   character(len=256) solver_opts
42   logical initx_zeros
43
44   ! Pointer to solver subroutine
45   ! Set to mgmres by default

```

```

42 ! procedure(solver_interface), pointer, nopass
43     :: solver => mgmres_st
44
45 contains
46     procedure :: init => mat_init
47     procedure :: deinit => mat_deinit
48     procedure :: calculate_size
49     procedure :: set_solver_opts =>
50         mat_set_solver_opts
51     procedure :: set_row => mat_set_row
52     procedure :: assign => mat_assign
53     procedure :: add => mat_add
54     procedure :: assign_rhs => mat_assign_rhs
55     procedure :: add_rhs => mat_add_rhs
56     !procedure :: store_index => mat_store_index
57     !procedure :: find_index => mat_find_index
58     procedure :: set_bc => mat_set_bc
59     procedure :: solve => mat_solve
60     procedure :: get_solver_stats
61     procedure :: ind => mat_ind
62     !procedure :: to_hdf => mat_to_hdf
63     procedure :: attenuate
64     procedure :: angular_integral
65     procedure :: add_source
66
67     ! Derivative subroutines
68     procedure x_cd2
69     procedure x_cd2_first
70     procedure x_cd2_last
71     procedure y_cd2
72     procedure y_cd2_first
73     procedure y_cd2_last
74     procedure z_cd2
75     procedure z_fd2
76     procedure z_bd2
77     procedure z_surface_bc
78     procedure z_bottom_bc
79
80 end type rte_mat
81
82 interface
83     ! Define interface for external procedure
84     ! https://stackoverflow.com/questions/8549415/how-to-declare-the-interface-section-for-a-procedure-argument-which-in-turn-ref
85     subroutine solver_interface(n_total, nonzero,
86         row, col, data, &
87         sol, rhs, maxiter_outer, maxiter_inner,
88         &
89         tol_abs, tol_rel)
90     use type_consts
91     integer(index_kind) :: n_total, nonzero

```

```

88      integer, dimension(nonzero) :: row, col
89      double precision, dimension(nonzero) :: 
90          data
91      double precision, dimension(nonzero) :: sol
92      double precision, dimension(n_total) :: rhs
93      integer :: maxiter_outer, maxiter_inner
94      double precision :: tol_abs, tol_rel
95  end subroutine solver_interface
96 end interface
97 contains
98
99  subroutine mat_init(mat, grid, iops)
100     class(rte_mat) mat
101     type(space_angle_grid) grid
102     type(optical_properties) iops
103     integer(index_kind) nnz, n_total
104
105     LIS_INTEGER comm_world
106
107     comm_world = LIS_COMM_WORLD
108
109     mat%grid = grid
110     mat%iops = iops
111
112     call mat%calculate_size()
113
114     mat%solver_opts = ''
115     mat% ierr = 0
116
117     n_total = mat%n_total
118     nnz = mat%nonzero
119
120     call lis_initialize(mat% ierr)
121
122     call lis_solver_create(mat%solver, mat% ierr)
123
124     call lis_matrix_create(comm_world, mat%A,
125                           mat% ierr)
125     call lis_vector_create(comm_world, mat%b,
126                           mat% ierr)
126     call lis_vector_create(comm_world, mat%x,
127                           mat% ierr)
128
128     call lis_matrix_set_size(mat%A, 0, n_total,
129                             mat% ierr)
129     call lis_vector_set_size(mat%b, 0, n_total,
130                             mat% ierr)
130     call lis_vector_set_size(mat%x, 0, n_total,
131                             mat% ierr)

```

```

131
132     call lis_vector_set_all(0.0d0, mat%x, mat%
133         ierr)
134     call lis_vector_set_all(0.0d0, mat%b, mat%
135         ierr)
136     if(mat% ierr .ne. 0) then
137         write(*,*) 'INIT ERR: ', mat% ierr
138         call exit(1)
139     end if
140
141     ! CSR Format
142     ! http://www.scipy-lectures.org/advanced/
143     !      scipy_sparse/csr_matrix.html
144     allocate(mat%ptr(n_total+1))
145     allocate(mat%col(nnz))
146     allocate(mat%data(nnz))
147     allocate(mat%surface_vals(grid%angles%nomega
148         ))
149
150     mat%ptr(n_total+1) = nnz
151 end subroutine mat_init
152
153 subroutine mat_deinit(mat)
154     class(rte_mat) mat
155
156     call lis_matrix_destroy(mat%A, mat% ierr)
157     call lis_vector_destroy(mat%b, mat% ierr)
158     call lis_vector_destroy(mat%x, mat% ierr)
159     call lis_solver_destroy(mat%solver, mat% ierr
160         )
161     call lis_finalize(mat% ierr)
162
163     if(mat% ierr .ne. 0) then
164         write(*,*) 'DEINIT ERR: ', mat% ierr
165         call exit(1)
166     end if
167
168     deallocate(mat%ptr)
169     deallocate(mat%col)
170     deallocate(mat%data)
171     deallocate(mat%surface_vals)
172 end subroutine mat_deinit
173
174 subroutine calculate_size(mat)
175     class(rte_mat) mat
176     integer(index_kind) nx, ny, nz, nomega
177
178     nx = mat%grid%x%num
179     ny = mat%grid%y%num

```

```

176     nz = mat%grid%z%num
177     nomega = mat%grid%angles%nomega
178
179     !mat%nonzero = nx * ny * ntheta * nphi * ( (
180         nz-1) * (6 + ntheta * nphi) + 1)
180     mat%nonzero = nx * ny * nomega * (nz * (
181         nomega + 6) - 1)
181     mat%n_total = nx * ny * nz * nomega
182     write(*,*) 'nnz = ', mat%nonzero
183     write(*,*) 'n_total = ', mat%n_total
184
185     !mat%theta_block_size = 1
186     !mat%phi_block_size = mat%theta_block_size *
187         ntheta
187     mat%omega_block_size = 1
188     mat%y_block_size = int(mat%omega_block_size *
189         nomega)
189     mat%x_block_size = int(mat%y_block_size * ny
190         )
190     mat%z_block_size = int(mat%x_block_size * nx
191         )
191
192 end subroutine calculate_size
193
194 ! subroutine mat_to_hdf(mat,filename)
195 !     class(rte_mat) mat
196 !     character(len=*) filename
197 !     call write_coo(filename, mat%row, mat%col,
198 !         mat%data, mat%nonzero)
198 ! end subroutine mat_to_hdf
199
200 subroutine mat_set_bc(mat, bc)
201     class(rte_mat) mat
202     class(boundary_condition) bc
203     integer p
204
205     do p=1, mat%grid%angles%nomega/2
206         mat%surface_vals(p) = bc%bc_grid(p)
207     end do
208 end subroutine mat_set_bc
209
210 subroutine mat_solve(mat)
211     class(rte_mat) mat
212     character(len=64) init_opt
213
214     ! write(*,*) 'mat%n_total = ', mat%n_total
215     ! write(*,*) 'mat%nonzero = ', mat%nonzero
216     ! open(unit=1, file='ptr.txt')
217     ! open(unit=2, file='col.txt')
218     ! open(unit=3, file='data.txt')

```

```

219 ! write(1,*) mat%ptr
220 ! write(2,*) mat%col
221 ! write(3,*) mat%data
222 ! close(1)
223 ! close(2)
224 ! close(3)
225
226 ! Create matrix
227 call lis_matrix_set_csr(mat%nonzero, mat%ptr
228   , mat%col, mat%data, mat%A, mat% ierr)
229 call lis_matrix_assemble(mat%A, mat% ierr)
230
231 ! Set solver options
232 if(mat%initx_zeros) then
233   init_opt = "-initx_zeros true -print out"
234 else
235   init_opt = "-initx_zeros false -print out"
236 end if
237
238 call lis_solver_set_option(init_opt, mat%
239   solver, mat% ierr)
240 if(len(trim(mat%solver_opts)) .gt. 0) then
241   call lis_solver_set_option(mat%
242     solver_opts, mat%solver, mat% ierr)
243 end if
244
245 ! Solve
246 call lis_solve(mat%A, mat%b, mat%x, mat%
247   solver, mat% ierr)
248
249 end subroutine mat_solve
250
251 subroutine get_solver_stats(mat, lis_iter,
252   lis_time, lis_resid)
253   class(rte_mat) mat
254   integer lis_iter
255   double precision lis_time
256   double precision lis_resid
257
258   call lis_solver_get_iter(mat%solver,
259     lis_iter, mat% ierr)
260   call lis_solver_get_time(mat%solver,
261     lis_time, mat% ierr)
262   call lis_solver_get_residualnorm(mat%solver,
263     lis_resid, mat% ierr)
264 end subroutine get_solver_stats
265
266 subroutine mat_set_solver_opts(mat,
267   solver_opts)

```

```

259 |     class(rte_mat) mat
260 |     character(len=*) solver_opts
261 |     write(*,*) "Setting solver opts: '',
262 |                 solver_opts, ''"
263 |     mat%solver_opts = solver_opts
264 | end subroutine mat_set_solver_opts
265 |
266 | function mat_ind(mat, i, j, k, p) result(ind)
267 |   ! Assuming var ordering: z, x, y, omega
268 |   class(rte_mat) mat
269 |   integer i, j, k, p
270 |   integer(index_kind) ind
271 |   ind = (i-1) * mat%x_block_size + (j-1) * mat%
272 |         %y_block_size + &
273 |         (k-1) * mat%z_block_size + p * mat%
274 |         omega_block_size
275 | end function mat_ind
276 |
277 | subroutine mat_set_row(mat, ent, row_num)
278 |   ! Start new row for CSR format
279 |   class(rte_mat) mat
280 |   integer(index_kind) ent, row_num
281 |   ! 0-indexing for LIS
282 |   mat%ptr(row_num) = ent - 1
283 | end subroutine mat_set_row
284 |
285 | subroutine mat_assign(mat, ent, val, i, j, k,
286 |                      p)
287 |   ! It's assumed that this is the first time
288 |   ! this entry is defined
289 |   class(rte_mat) mat
290 |   double precision val
291 |   integer i, j, k, p
292 |   integer(index_kind) ent
293 |
294 |   ! LIS method 2 (LIS manual, p. 19) requires
295 |   ! 0-indexing
296 |   mat%col(ent) = mat%ind(i, j, k, p) - 1
297 |   mat%data(ent) = val
298 |
299 |   ent = ent + 1
300 | end subroutine mat_assign
301 |
302 | subroutine mat_add(mat, repeat_ent, val)
303 |   ! Use this when you know that this entry has
304 |   ! already been assigned
305 |   ! and you'd like to add this value to the
306 |   ! existing value.
307 |
308 |   class(rte_mat) mat

```

```

302     double precision val
303     integer(index_kind) repeat_ent
304
305     ! Entry number where value is already stored
306     mat%data(repeat_ent) = mat%data(repeat_ent)
307     + val
308   end subroutine mat_add
309
310   subroutine mat_assign_rhs(mat, row_num, data)
311     class(rte_mat) mat
312     double precision data
313     integer(index_kind) row_num
314
315     call lis_vector_set_value(LIS_INS_VALUE,
316                               row_num, data, mat%b, mat% ierr)
317     if(mat% ierr .ne. 0) then
318       write(*,*) 'RHS ERR: ', mat% ierr
319       call exit(1)
320     end if
321   end subroutine mat_assign_rhs
322
323   subroutine mat_add_rhs(mat, row_num, data)
324     class(rte_mat) mat
325     double precision data
326     integer(index_kind) row_num
327
328     call lis_vector_set_value(LIS_ADD_VALUE,
329                               row_num, data, mat%b, mat% ierr)
330     if(mat% ierr .ne. 0) then
331       write(*,*) 'RHS ERR: ', mat% ierr
332       call exit(1)
333     end if
334   end subroutine mat_add_rhs
335
336   ! subroutine mat_store_index(mat, row_num,
337   !                           col_num)
338   !   ! Remember where we stored information for
339   !   ! this matrix element
340   !   ! class(rte_mat) mat
341   !   ! integer row_num, col_num
342   !   ! mat%index_map(row_num, col_num) = mat%ent
343   ! end subroutine
344
345   ! function mat_find_index(mat, row_num,
346   !                           col_num) result(index)
347   !   ! Find the position in row, col, data
348   !   ! where this entry
349   !   ! is defined.
350   !   ! class(rte_mat) mat
351   !   ! integer row_num, col_num, index

```

```

346 !     index = mat%index_map(row_num, col_num)
347 !     ! This took up 95% of execution time.
348 !     ! Only search up to most recently assigned
349 !     ! index
350 !     ! do index=1, mat%ent-1
351 !         ! if( (mat%row(index) .eq. row_num) .
352 !             and. (mat%col(index) .eq. col_num)) then
353 !             !       exit
354 !             ! end if
355 !         ! end do
356 !     end function mat_find_index
357
358 subroutine attenuate(mat, indices, repeat_ent)
359 ! Has to be called after angular_integral
360 ! Because they both write to the same matrix
361 ! entry
362 ! And adding here is more efficient than a
363 ! conditional
364 ! in the angular loop.
365 class(rte_mat) mat
366 double precision attenuation
367 type(index_list) indices
368 double precision aa, bb
369 integer(index_kind) repeat_ent
370
371 aa = mat%iops%abs_grid(indices%i, indices%j,
372 !           indices%k)
373 bb = mat%iops%scat
374 attenuation = aa + bb
375
376 call mat%add(repeat_ent, attenuation)
377 end subroutine attenuate
378
379 subroutine add_source(mat, indices, row_num)
380 ! Has to be called after angular_integral
381 ! Because they both write to the same matrix
382 ! entry
383 ! And adding here is more efficient than a
384 ! conditional
385 ! in the angular loop.
386 class(rte_mat) mat
387 type(index_list) indices
388 integer(index_kind) row_num
389 double precision source_val
390
391 source_val = mat%iops%source_grid(indices%i,
392 !           indices%j, indices%k, indices%p)
393
394 call mat%add_rhs(row_num, source_val)
395 end subroutine add_source

```

```

390 | subroutine x_cd2(mat, indices, ent)
391 |   class(rte_mat) mat
392 |   double precision val, dx
393 |   type(index_list) indices
394 |   integer i, j, k, p
395 |   integer(index_kind) ent
396 |
397 |   i = indices%i
398 |   j = indices%j
399 |   k = indices%k
400 |   p = indices%p
401 |
402 |   dx = mat%grid%x%spacing(1)
403 |
404 |   val = mat%grid%angles%sin_phi_p(p) &
405 |         * mat%grid%angles%cos_theta_p(p) / (2.
406 |                                         d0 * dx)
407 |
408 |   call mat%assign(ent,-val,i-1,j,k,p)
409 |   call mat%assign(ent,val,i+1,j,k,p)
410 | end subroutine x_cd2
411 |
412 | subroutine x_cd2_first(mat, indices, ent)
413 |   class(rte_mat) mat
414 |   double precision val, dx
415 |   integer nx
416 |   type(index_list) indices
417 |   integer i, j, k, p
418 |   integer(index_kind) ent
419 |
420 |   i = indices%i
421 |   j = indices%j
422 |   k = indices%k
423 |   p = indices%p
424 |
425 |   dx = mat%grid%x%spacing(1)
426 |   nx = mat%grid%x%num
427 |
428 |   val = mat%grid%angles%sin_phi_p(p) &
429 |         * mat%grid%angles%cos_theta_p(p) / (2.
430 |                                         d0 * dx)
431 |
432 |   call mat%assign(ent,-val,nx,j,k,p)
433 |   call mat%assign(ent,val,i+1,j,k,p)
434 | end subroutine x_cd2_first
435 |
436 | subroutine x_cd2_last(mat, indices, ent)
437 |   class(rte_mat) mat

```

```

438 |     integer i, j, k, p
439 |     integer(index_kind) ent
440 |
441 |     i = indices%i
442 |     j = indices%j
443 |     k = indices%k
444 |     p = indices%p
445 |
446 |     dx = mat%grid%x%spacing(1)
447 |
448 |     val = mat%grid%angles%sin_phi_p(p) &
449 |            * mat%grid%angles%cos_theta_p(p) / (2.
450 |                d0 * dx)
451 |
452 |     call mat%assign(ent,-val,i-1,j,k,p)
453 |     call mat%assign(ent,val,1,j,k,p)
454 | end subroutine x_cd2_last
455 |
456 subroutine y_cd2(mat, indices, ent)
457     class(rte_mat) mat
458     double precision val, dy
459     type(index_list) indices
460     integer i, j, k, p
461     integer(index_kind) ent
462 |
463 |     i = indices%i
464 |     j = indices%j
465 |     k = indices%k
466 |     p = indices%p
467 |
468 |     dy = mat%grid%y%spacing(1)
469 |
470 |     val = mat%grid%angles%sin_phi_p(p) &
471 |            * mat%grid%angles%sin_theta_p(p) / (2.
472 |                d0 * dy)
473 |
474 |     call mat%assign(ent,-val,i,j-1,k,p)
475 |     call mat%assign(ent,val,i,j+1,k,p)
476 | end subroutine y_cd2
477 |
478 subroutine y_cd2_first(mat, indices, ent)
479     class(rte_mat) mat
480     double precision val, dy
481     integer ny
482     type(index_list) indices
483     integer i, j, k, p
484     integer(index_kind) ent
485 |
486 |     i = indices%i
487 |     j = indices%j

```

```

486   k = indices%k
487   p = indices%p
488
489   dy = mat%grid%y%spacing(1)
490   ny = mat%grid%y%num
491
492   val = mat%grid%angles%sin_phi_p(p) &
493       * mat%grid%angles%sin_theta_p(p) / (2.
494           d0 * dy)
495
496   call mat%assign(ent,-val,i,ny,k,p)
497   call mat%assign(ent,val,i,j+1,k,p)
498 end subroutine y_cd2_first
499
500 subroutine y_cd2_last(mat, indices, ent)
501     class(rte_mat) mat
502     double precision val, dy
503     type(index_list) indices
504     integer i, j, k, p
505     integer(index_kind) ent
506
507     i = indices%i
508     j = indices%j
509     k = indices%k
510     p = indices%p
511
512     dy = mat%grid%y%spacing(1)
513
514     val = mat%grid%angles%sin_phi_p(p) &
515         * mat%grid%angles%sin_theta_p(p) / (2.
516             d0 * dy)
517
518     call mat%assign(ent,-val,i,j-1,k,p)
519     call mat%assign(ent,val,i,1,k,p)
520 end subroutine y_cd2_last
521
522 subroutine z_cd2(mat, indices, ent)
523     class(rte_mat) mat
524     double precision val, dz
525     type(index_list) indices
526     integer i, j, k, p
527     integer(index_kind) ent
528
529     i = indices%i
530     j = indices%j
531     k = indices%k
532     p = indices%p
533
534     dz = mat%grid%z%spacing(indices%k)

```

```

534     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
535         dz)
536     call mat%assign(ent,-val,i,j,k-1,p)
537     call mat%assign(ent,val,i,j,k+1,p)
538 end subroutine z_cd2
539
540 subroutine z_fd2(mat, indices, ent, repeat_ent
541 )
542 ! Has to be called after angular_integral
543 ! Because they both write to the same matrix
544 ! entry
545 ! And adding here is more efficient than a
546 ! conditional
547 ! in the angular loop.
548 class(rte_mat) mat
549 double precision val, val1, val2, val3, dz
550 type(index_list) indices
551 integer i, j, k, p
552 integer(index_kind) ent, repeat_ent
553
554 i = indices%i
555 j = indices%j
556 k = indices%k
557 p = indices%p
558
559 dz = mat%grid%z%spacing(indices%k)
560
561 val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
562     dz)
563
564 val1 = -3.d0 * val
565 val2 = 4.d0 * val
566 val3 = -val
567
568 call mat%add(repeat_ent, val1)
569 call mat%assign(ent, val2, i, j, k+1, p)
570 call mat%assign(ent, val3, i, j, k+2, p)
571 end subroutine z_fd2
572
573 subroutine z_bd2(mat, indices, ent, repeat_ent
574 )
575 ! Has to be called after angular_integral
576 ! Because they both write to the same matrix
577 ! entry
578 ! And adding here is more efficient than a
579 ! conditional
580 ! in the angular loop.
581 class(rte_mat) mat
582 double precision val, val1, val2, val3, dz
583 type(index_list) indices

```

```

577 |     integer i, j, k, p
578 |     integer(index_kind) ent, repeat_ent
579 |
580 |     i = indices%i
581 |     j = indices%j
582 |     k = indices%k
583 |     p = indices%p
584 |
585 |     dz = mat%grid%z%spacing(indices%k)
586 |
587 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
588 |                                              dz)
589 |     val1 = 3.d0 * val
590 |     val2 = -4.d0 * val
591 |     val3 = val
592 |
593 |     call mat%add(repeat_ent, val1)
594 |     call mat%assign(ent, val2, i, j, k-1, p)
595 |     call mat%assign(ent, val3, i, j, k-2, p)
596 end subroutine z_bd2
597
598 subroutine angular_integral(mat, indices, ent)
599   class(rte_mat) mat
600   ! Primed angular integration variables
601   integer pp
602   double precision val
603   type(index_list) indices
604   integer(index_kind) ent
605
606   do pp=1, mat%grid%angles%nomega
607     val = -mat%iops%scat * mat%iops%
608       vsf_integral(indices%p, pp)
609     call mat%assign(ent, val, indices%i,
610                   indices%j, indices%k, pp)
611   end do
612 end subroutine angular_integral
613
614 subroutine z_surface_bc(mat, indices, row_num,
615                         ent, repeat_ent)
616   class(rte_mat) mat
617   double precision bc_val
618   type(index_list) indices
619   double precision val1, val2, dz
620   integer(index_kind) row_num, ent, repeat_ent
621
622   dz = mat%grid%z%spacing(1)
623
624   val1 = mat%grid%angles%cos_phi_p(indices%p)
625     / (5.d0 * dz)
626   val2 = 7.d0 * val1

```

```

623     bc_val = 8.d0 * val1 * mat%surface_vals(
624         indices%p)
625     call mat%assign(ent, val1, indices%i, indices%j
626         , 2, indices%p)
627     call mat%add(repeat_ent, val2)
628     call mat%assign_rhs(row_num, bc_val)
629 end subroutine z_surface_bc
630
631     subroutine z_bottom_bc(mat, indices, ent,
632         repeat_ent)
633     class(rte_mat) mat
634     type(index_list) indices
635     double precision val1, val2, dz
636     integer nz
637     integer(index_kind) ent, repeat_ent
638
639     dz = mat%grid%z%spacing(1)
640     nz = mat%grid%z%num
641
642     val1 = -mat%grid%angles%cos_phi_p(indices%p)
643         / (5.d0 * dz)
644     val2 = 7.d0 * val1
645
646     call mat%assign(ent, val1, indices%i, indices%j
647         , nz-1, indices%p)
648     call mat%add(repeat_ent, val2)
649 end subroutine z_bottom_bc
650
651 end module rte_sparse_matrices

```

rte3d.f90

```

1 module rte3d
2 use kelp_context
3 use rte_sparse_matrices
4 use light_context
5 use type_consts
6 implicit none
7
8 interface
9     subroutine deriv_interface(mat, indices, ent)
10        use rte_sparse_matrices
11        use type_consts
12        class(rte_mat) mat
13        type(index_list) indices
14        integer(index_kind) ent
15    end subroutine deriv_interface
16    subroutine angle_loop_interface(mat, indices,
17        ddx, ddy)
18        use rte_sparse_matrices
19    end subroutine angle_loop_interface
20
21 end interface
22
23 end module rte3d

```

```

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   num_threads)
30   type(rte_mat) mat
31   type(index_list) indices
32   integer i, j, k
33   integer num_threads
34
35   procedure(deriv_interface), pointer :: ddx,
36     ddy
37   procedure(angle_loop_interface), pointer :::
38     angle_loop
39
40 !$omp parallel do default(none) shared(mat) &
41 !$omp private(ddx,ddy,angle_loop, k, i, j)
42   private(indices) &
43 !$omp num_threads(num_threads) collapse(3)
44 do k=1, mat%grid%z%num
45   do i=1, mat%grid%x%num
46     do j=1, mat%grid%y%num
47       indices%k = k
48       if(k .eq. 1) then
49         angle_loop => surface_angle_loop
50       else if(k .eq. mat%grid%z%num) then
51         angle_loop => bottom_angle_loop
52       else
53         angle_loop => interior_angle_loop
54       end if
55
56       indices%i = i
57       if(indices%i .eq. 1) then
58         ddx => x_cd2_first
59       else if(indices%i .eq. mat%grid%x%num
60 ) then
61         ddx => x_cd2_last
62       else
63         ddx => x_cd2
64       end if
65
66       indices%j = j
67       if(indices%j .eq. 1) then
68         ddy => y_cd2_first

```

```

64      else if(indices%j .eq. mat%grid%y%num
65          ) then
66          ddy => y_cd2_last
67      else
68          ddy => y_cd2
69      end if
70
71      call angle_loop(mat, indices, ddx,
72                      ddy)
73  end do
74  !$omp end parallel do
75 end subroutine whole_space_loop
76
77 function calculate_start_ent(grid, indices)
78     result(ent)
79     type(space_angle_grid) grid
80     type(index_list) indices
81     integer(index_kind) ent
82     integer(index_kind) boundary_nnz, interior_nnz
83     integer(index_kind) num_boundary, num_interior
84     integer(index_kind) num_this_x, num_this_z
85
86     ! Nonzero matrix entries for an surface or
87     ! bottom spatial grid cell
88     ! Definitely an integer since nomega is even
89     boundary_nnz = grid%angles%nomega * (2 * grid%
90         angles%nomega + 11) / 2
91     ! Nonzero matrix entries for an interior
92     ! spatial grid cell
93     interior_nnz = grid%angles%nomega * (grid%
94         angles%nomega + 6)
95
96     ! Order: z, x, y, omega
97     ! Total number traversed so far in each
98     ! spatial category
99     ! row
100    num_this_x = indices%j - 1
101    ! depth layer
102    num_this_z = (indices%i - 1) * grid%y%num +
103        num_this_x
104
105    ! Calculate number of spatial grid cells of
106    ! each type which have
107    ! already been traversed up to this point
108    if(indices%k .eq. 1) then
109        num_boundary = num_this_z
110        num_interior = 0
111    else if(indices%k .eq. grid%z%num) then

```

```

104     num_boundary = (grid%x%num * grid%y%num) +
105         num_this_z
106     num_interior = (grid%z%num-2) * grid%x%num
107         * grid%y%num
108 else
109     num_boundary = grid%x%num * grid%y%num
110     num_interior = num_this_z + (indices%k-2) *
111         grid%x%num * grid%y%num
112 end if
113
114 ent = num_boundary * boundary_nnz +
115     num_interior * interior_nnz + 1
116
117 end function calculate_start_ent
118
119 function calculate_repeat_ent(ent, p) result(
120     repeat_ent)
121     integer p
122     integer(index_kind) ent, repeat_ent
123 ! Entry number for row=mat%ind(i,j,k,p), col=
124     mat%ind(i,j,k,p),
125 ! which will be modified multiple times in
126     this matrix row
127     repeat_ent = ent + p - 1
128
129 end function calculate_repeat_ent
130
131 subroutine interior_angle_loop(mat, indices, ddx
132 , ddy)
133     type(rte_mat) mat
134     type(index_list) indices
135     procedure(deriv_interface) :: ddx, ddy
136     integer p
137     integer(index_kind) ent, repeat_ent
138     integer(index_kind) row_num
139
140 ! Determine which matrix row to start at
141 ent = calculate_start_ent(mat%grid, indices)
142 indices%p = 1
143 row_num = mat%ind(indices%i, indices%j,
144     indices%k, indices%p)
145
146 do p=1, mat%grid%angles%nomega
147     indices%p = p
148     repeat_ent = calculate_repeat_ent(ent, p)
149     call mat%set_row(ent, row_num)
150     call mat%angular_integral(indices, ent)
151     call ddx(mat, indices, ent)
152     call ddy(mat, indices, ent)
153     call mat%z_cd2(indices, ent)
154     call mat%attenuate(indices, repeat_ent)
155     call mat%add_source(indices, row_num)

```

```

145      row_num = row_num + 1
146    end do
147 end subroutine
148
149 subroutine surface_angle_loop(mat, indices, ddx,
150   ddy)
150   type(rte_mat) mat
151   type(index_list) indices
152   integer p
153   procedure(deriv_interface) :: ddx, ddy
154   integer(index_kind) ent, repeat_ent
155   integer(index_kind) row_num
156
157   ! Determine which matrix row to start at
158   ent = calculate_start_ent(mat%grid, indices)
159   indices%p = 1
160   row_num = mat%ind(indices%i, indices%j,
161     indices%k, indices%p)
162
163   ! Downwelling
164   do p=1, mat%grid%angles%nomega / 2
165     indices%p = p
166     repeat_ent = calculate_repeat_ent(ent, p)
167     call mat%set_row(ent, row_num)
168     call mat%angular_integral(indices, ent)
169     call ddx(mat, indices, ent)
170     call ddy(mat, indices, ent)
171     call mat%z_surface_bc(indices, row_num, ent
172       , repeat_ent)
173     call mat%attenuate(indices, repeat_ent)
174     call mat%add_source(indices, row_num)
175     row_num = row_num + 1
176   end do
177   ! Upwelling
178   do p=mat%grid%angles%nomega/2+1, mat%grid%
179     angles%nomega
180     indices%p = p
181     repeat_ent = calculate_repeat_ent(ent, p)
182     call mat%set_row(ent, row_num)
183     call mat%angular_integral(indices, ent)
184     call ddx(mat, indices, ent)
185     call ddy(mat, indices, ent)
186     call mat%z_fd2(indices, ent, repeat_ent)
187     call mat%attenuate(indices, repeat_ent)
188     call mat%add_source(indices, row_num)
189     row_num = row_num + 1
190   end do
191 end subroutine surface_angle_loop

```

```

190 | subroutine bottom_angle_loop(mat, indices, ddx,
191 |   ddy)
192 |   type(rte_mat) mat
193 |   type(index_list) indices
194 |   integer p
195 |   integer(index_kind) row_num, ent, repeat_ent
196 |   procedure(deriv_interface) :: ddx, ddy
197 |
198 |   ! Determine which matrix row to start at
199 |   ent = calculate_start_ent(mat%grid, indices)
200 |   indices%p = 1
201 |   row_num = mat%ind(indices%i, indices%j,
202 |     indices%k, indices%p)
203 |
204 |   ! Downwelling
205 |   do p=1, mat%grid%angles%nomega/2
206 |     indices%p = p
207 |     repeat_ent = calculate_repeat_ent(ent, p)
208 |     call mat%set_row(ent, row_num)
209 |     call mat%angular_integral(indices, ent)
210 |     call ddx(mat, indices, ent)
211 |     call ddy(mat, indices, ent)
212 |     call mat%z_bd2(indices, ent, repeat_ent)
213 |     call mat%attenuate(indices, repeat_ent)
214 |     call mat%add_source(indices, row_num)
215 |     row_num = row_num + 1
216 |   end do
217 |   ! Upwelling
218 |   do p=mat%grid%angles%nomega/2+1, mat%grid%
219 |     angles%nomega
220 |     indices%p = p
221 |     repeat_ent = calculate_repeat_ent(ent, p)
222 |     call mat%set_row(ent, row_num)
223 |     call mat%angular_integral(indices, ent)
224 |     call ddx(mat, indices, ent)
225 |     call ddy(mat, indices, ent)
226 |     call mat%z_bottom_bc(indices, ent,
227 |       repeat_ent)
228 |     call mat%attenuate(indices, repeat_ent)
229 |     call mat%add_source(indices, row_num)
230 |     row_num = row_num + 1
231 |   end do
232 | end subroutine bottom_angle_loop
233 |
234 |
235 | subroutine gen_matrix(mat, num_threads)
236 |   type(rte_mat) mat
237 |   type(index_list) indices
238 |   integer num_threads
239 |
240 |   call indices%init()

```

```

236
237     call whole_space_loop(mat, indices,
238         num_threads)
238     ! call surface_space_loop(mat, indices)
239     ! call interior_space_loop(mat, indices)
240     ! call bottom_space_loop(mat, indices)
241 end subroutine gen_matrix
242
243 subroutine rte3d_deinit(mat, iops, light)
244     type(rte_mat) mat
245     type(optical_properties) iops
246     type(light_state) light
247
248     call mat%deinit()
249     call iops%deinit()
250     call light%deinit()
251 end subroutine
252
253 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
21 end type frond_shape
22
23 type rope_state
24     integer nz
25     double precision, dimension(:), allocatable
26         :: frond_lengths, frond_stds, num_fronds,
27             water_speeds, water_angles
26 contains
27     procedure :: init => rope_init

```

```

28     procedure :: deinit => rope_deinit
29 end type rope_state
30
31 type depth_state
32     double precision frond_length, frond_std,
33             num_fronds, water_speeds, water_angles,
34             depth
35     integer depth_layer
36 contains
37     procedure :: set_depth
38     procedure :: length_distribution_cdf
39     procedure :: angle_distribution_pdf
40 end type depth_state
41
42 type optical_properties
43     integer num_vsf
44     type(space_angle_grid) grid
45     double precision, dimension(:), allocatable
46         :: vsf_angles, vsf_vals
47     double precision, dimension(:), allocatable
48         :: abs_water
49     double precision abs_kelp, scat
50     ! On x, y, z grid - including water & kelp.
51     double precision, dimension(:,:,:,:),
52         allocatable :: abs_grid
53     double precision, dimension(:,:,:,:),
54         allocatable :: source_grid
55     ! On theta, phi, theta_prime, phi_prime grid
56     double precision, dimension(:, :, :), allocatable
57         :: vsf, vsf_integral
58 contains
59     procedure :: init => iop_init
60     procedure :: calculate_coef_grids
61     procedure :: zero_source => iop_zero_source
62     procedure :: set_source => iop_set_source
63     procedure :: load_vsf
64     procedure :: eval_vsf
65     procedure :: calc_vsf_on_grid
66     procedure :: deinit => iop_deinit
67     procedure :: vsf_from_function
68 end type optical_properties
69
70 type boundary_condition
71     double precision I0, decay, theta_s, phi_s
72     type(space_angle_grid) grid
73     double precision, dimension(:), allocatable
74         :: bc_grid
75 contains
76     procedure :: bc_gaussian
77     procedure :: init => bc_init
78     procedure :: deinit => bc_deinit
79 end type boundary_condition
80

```

```

73 | contains
74 |
75 |     function bc_gaussian(bc, theta, phi)
76 |         class(boundary_condition) bc
77 |         double precision theta, phi, diff
78 |         double precision bc_gaussian
79 |         diff = angle_diff_3d(theta, phi, bc%theta_s,
80 |                                bc%phi_s)
81 |         bc_gaussian = exp(-bc%decay * diff)
82 |     end function bc_gaussian
83 |
84 |     subroutine bc_init(bc, grid, theta_s, phi_s,
85 |                          decay, I0)
86 |         class(boundary_condition) bc
87 |         type(space_angle_grid) grid
88 |         double precision theta_s, phi_s, decay, I0
89 |         integer p
90 |         double precision theta, phi
91 |         double precision bc_norm
92 |         double precision, allocatable, dimension(:)
93 |                                :: whole_bc_grid
94 |         integer nomega
95 |
96 |         nomega = grid%angles%nomega
97 |
98 |         allocate(bc%bc_grid(nomega/2))
99 |         allocate(whole_bc_grid(nomega))
100 |
101 |         bc%theta_s = theta_s
102 |         bc%phi_s = phi_s
103 |         bc%decay = decay
104 |         bc%I0 = I0
105 |
106 |         ! Only set BC for downwelling light
107 |         do p=1, nomega/2
108 |             theta = grid%angles%theta_p(p)
109 |             phi = grid%angles%phi_p(p)
110 |             bc%bc_grid(p) = bc%bc_gaussian(theta, phi)
111 |         end do
112 |
113 |         ! Normalize
114 |         ! Use 'whole_bc_grid' because angular
115 |             integration
116 |         ! subroutine requires all angles to have
117 |             values,
118 |         ! but 'bc%bc_grid' only has downwelling
119 |             values.
120 |         ! Use zeros for upwelling.
121 |         whole_bc_grid(1:nomega/2) = bc%bc_grid
122 |         whole_bc_grid(nomega/2+1:nomega) = 0

```

```

117      bc_norm = grid%angles%integrate_points(
118          whole_bc_grid)
119      bc%bc_grid = bc%I0 * bc%bc_grid / bc_norm
120  end subroutine bc_init
121
122  subroutine bc_deinit(bc)
123      class(boundary_condition) bc
124      deallocate(bc%bc_grid)
125  end subroutine
126
127  subroutine point_set_cart(point, x, y, z)
128      class(point3d) :: point
129      double precision x, y, z
130      point%x = x
131      point%y = y
132      point%z = z
133      call point%cartesian_to_polar()
134  end subroutine point_set_cart
135
136  subroutine point_set_cyl(point, theta, r, z)
137      class(point3d) :: point
138      double precision theta, r, z
139      point%theta = theta
140      point%r = r
141      point%z = z
142      call point%polar_to_cartesian()
143  end subroutine point_set_cyl
144
145  subroutine polar_to_cartesian(point)
146      class(point3d) :: point
147      point%x = point%r*cos(point%theta)
148      point%y = point%r*sin(point%theta)
149  end subroutine polar_to_cartesian
150
151  subroutine cartesian_to_polar(point)
152      class(point3d) :: point
153      point%r = sqrt(point%x**2 + point%y**2)
154      point%theta = atan2(point%y, point%x)
155  end subroutine cartesian_to_polar
156
157  subroutine frond_set_shape(frond, fs, fr, ft)
158      class(frond_shape) frond
159      double precision fs, fr, ft
160      frond%fs = fs
161      frond%fr = fr
162      frond%ft = ft
163      call frond%calculate_angles()
164  end subroutine frond_set_shape
165
166  subroutine frond_calculate_angles(frond)

```

```

166   class(frond_shape) frond
167   frond%tan_alpha = 2.d0*frond%fs*frond%fr /
168     (1.d0 + frond%fs)
169   frond%alpha = atan(frond%tan_alpha)
170 end subroutine
171
172 subroutine iop_init(iops, grid)
173   class(optical_properties) iops
174   type(space_angle_grid) grid
175
176   iops%grid = grid
177
178   ! Assume that these are preallocated and
179   ! passed to function
180   ! Nevermind, don't assume this.
181   allocate(iops%abs_water(grid%z%num))
182
183   ! Assume that these must be allocated here
184   ! NOTE: vsf_angles are defined on [0, pi].
185   ! (not on [-1, 1])
186   allocate(iops%vsf_angles(iops%num_vsf))
187   allocate(iops%vsf_vals(iops%num_vsf))
188   allocate(iops%vsf(grid%angles%nomega, grid%
189     angles%nomega))
190   allocate(iops%vsf_integral(grid%angles%
191     nomega, grid%angles%nomega))
192   allocate(iops%abs_grid(grid%x%num, grid%y%
193     num, grid%z%num))
194   allocate(iops%source_grid(grid%x%num, grid%y%
195     %num, grid%z%num, grid%angles%nomega))
196
197   call iops%zero_source()
198 end subroutine iop_init
199
200 subroutine iop_zero_source(iops)
201   class(optical_properties) iops
202
203   iops%source_grid = 0
204 end subroutine iop_zero_source
205
206 subroutine iop_set_source(iops, source)
207   class(optical_properties) iops
208   double precision, dimension(:,:,:,:,:) ::
```

```

209     double precision, dimension(:,:,:) :: p_kelp
210
211     integer k
212
213     ! Allow water IOPs to vary over depth
214     do k=1, iops%grid%z%num
215         iops%abs_grid(:,:,:,k) = (iops%abs_kelp -
216             iops%abs_water(k)) * p_kelp(:,:,:,k) +
217             iops%abs_water(k)
218     end do
219
220 end subroutine calculate_coef_grids
221
222 subroutine load_vsf(iops, filename, fmtstr)
223     class(optical_properties) :: iops
224     character(len=*) :: filename, fmtstr
225     double precision, dimension(:,:),
226         allocatable :: tmp_2d_arr
227     integer num_rows, num_cols, skiplines_in
228
229     ! First column is the angle at which the
230     ! measurement is taken
231     ! Second column is the value of the VSF at
232     ! that angle
233     num_rows = iops%num_vsf
234     num_cols = 2
235     skiplines_in = 1 ! Ignore comment on first
236     ! line
237
238     allocate(tmp_2d_arr(num_rows, num_cols))
239
240     tmp_2d_arr = read_array(filename, fmtstr,
241         num_rows, num_cols, skiplines_in)
242     iops%vsf_angles = tmp_2d_arr(:,1)
243     iops%vsf_vals = tmp_2d_arr(:,2)
244
245     ! write(*,*) 'vsf_angles = ', iops%
246     ! vsf_angles
247     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
248
249     ! Pre-evaluate for all pair of angles
250     call iops%calc_vsf_on_grid()
251 end subroutine load_vsf
252
253 function eval_vsf(iops, theta)
254     class(optical_properties) iops
255     double precision theta
256     double precision eval_vsf
257     ! No need to set vsf(0) = 0.

```

```

251      ! It's the area under the curve that matters
252      , not the value.
253      eval_vsf = interp(theta, iops%vsf_angles,
254                          iops%vsf_vals, iops%num_vsf)
255  end function eval_vsf
256
257 subroutine rope_init(rope, grid)
258   class(rope_state) :: rope
259   type(space_angle_grid) :: grid
260
261   rope%nz = grid%z%num
262   allocate(rope%frond_lengths(rope%nz))
263   allocate(rope%frond_stds(rope%nz))
264   allocate(rope%water_speeds(rope%nz))
265   allocate(rope%water_angles(rope%nz))
266   allocate(rope%num_fronds(rope%nz))
267 end subroutine rope_init
268
269 subroutine rope_deinit(rope)
270   class(rope_state) rope
271   deallocate(rope%frond_lengths)
272   deallocate(rope%frond_stds)
273   deallocate(rope%water_speeds)
274   deallocate(rope%water_angles)
275   deallocate(rope%num_fronds)
276 end subroutine rope_deinit
277
278 subroutine set_depth(depth, rope, grid,
279                      depth_layer)
280   class(depth_state) depth
281   type(rope_state) rope
282   type(space_angle_grid) grid
283   integer depth_layer
284
285   depth%frond_length = rope%frond_lengths(
286                                         depth_layer)
287   depth%frond_std = rope%frond_stds(
288                                         depth_layer)
289   depth%water_speeds = rope%water_speeds(
290                                         depth_layer)
291   depth%water_angles = rope%water_angles(
292                                         depth_layer)
293   depth%num_fronds = rope%num_fronds(
294                                         depth_layer)
295   depth%depth_layer = depth_layer
296   depth%depth = grid%z%vals(depth_layer)
297 end subroutine set_depth

```

```

292 |     function length_distribution_cdf(depth, L)
293 |         result(output)
294 |         ! C_L(L)
295 |         class(depth_state) depth
296 |         double precision L, L_mean, L_std
297 |         double precision output
298 |         L_mean = depth%frond_length
299 |         L_std = depth%frond_std
300 |
301 |         call normal_cdf(L, L_mean, L_std, output)
302 |     end function length_distribution_cdf
303 |
304 |     function angle_distribution_pdf(depth, theta_f)
305 |         result(output)
306 |         ! P_{\theta_f}(\theta_f)
307 |         class(depth_state) depth
308 |         double precision theta_f, v_w, theta_w
309 |         double precision output
310 |         double precision diff
311 |
312 |         v_w = depth%water_speeds
313 |         theta_w = depth%water_angles
314 |
315 |         ! von_mises_pdf is only defined on [-pi, pi]
316 |         ! So take difference of angles and input
317 |             into
318 |             ! von_mises dist. centered & x=0.
319 |
320 |             diff = angle_diff_2d(theta_f, theta_w)
321 |
322 |             call von_mises_pdf(diff, 0.d0, v_w, output)
323 |     end function angle_distribution_pdf
324 |
325 |     function angle_mod(theta) result(mod_theta)
326 |         ! Shift theta to the interval [-pi, pi]
327 |         ! which is where von_mises_pdf is defined.
328 |
329 |         double precision theta, mod_theta
330 |
331 |         mod_theta = mod(theta + pi, 2.d0*pi) - pi
332 |     end function angle_mod
333 |
334 |     function angle_diff_2d(theta1, theta2) result(
335 |         diff)
336 |         ! Shortest difference between two angles
337 |             which may be
338 |             ! in different periods.
339 |             double precision theta1, theta2, diff
340 |             double precision modt1, modt2

```

```

338 ! Shift to [0, 2*pi]
339 modt1 = mod(theta1, 2*pi)
340 modt2 = mod(theta2, 2*pi)
341
342 ! https://gamedev.stackexchange.com/questions/4467/comparing-angles-and-working-out-the-difference
343
344 diff = pi - abs(abs(modt1-modt2) - pi)
345 end function angle_diff_2d
346
347 function angle_diff_3d(theta, phi, theta_prime,
348 , phi_prime) result(diff)
349 ! Angle between two vectors in spherical
350 ! coordinates
351 double precision theta, phi, theta_prime,
352 phi_prime
353 double precision alpha, diff
354
355 ! Faster, but produces lots of NaNs (at
356 ! least in Python)
357 !alpha = sin(theta)*sin(theta_prime)*cos(
358 !    theta-theta_prime) + cos(phi)*cos(
359 !    phi_prime)
360
361 ! Slower, but more accurate
362 alpha = (sin(phi)*sin(phi_prime) &
363 * (cos(theta)*cos(theta_prime) + sin(theta)
364 !)*sin(theta_prime)) &
365 + cos(phi)*cos(phi_prime))
366
367 ! Avoid out-of-bounds errors due to rounding
368 alpha = min(1.d0, alpha)
369 alpha = max(-1.d0, alpha)
370
371 diff = acos(alpha)
372 end function angle_diff_3d
373
374 subroutine vsf_from_function(iops, func)
375 class(optical_properties) iops
376 double precision, external :: func
377 integer i
378 type(angle_dim) :: angle1d
379
380 call angle1d%set_bounds(-1.d0, 1.d0)
381 call angle1d%set_num(iops%num_vsf)
382 call angle1d%assign_legendre()
383
384 iops%vsf_angles(:) = acos(angle1d%vals(:))
385 do i=1, iops%num_vsf

```

```

380     iops%vsf_vals(i) = func(iops%vsf_angles(i
381         ))
382     end do
383     call iops%calc_vsf_on_grid()
384     call angle1d%deinit()
385   end subroutine vsf_from_function
386
387   subroutine calc_vsf_on_grid(iops)
388     class(optical_properties) iops
389     double precision th, ph, thp, php
390     integer p, pp
391     integer nomega
392     double precision norm
393
394     nomega = iops%grid%angles%nomega
395
396     do p=1, nomega
397       th = iops%grid%angles%theta_p(p)
398       ph = iops%grid%angles%phi_p(p)
399       do pp=1, nomega
400         thp = iops%grid%angles%theta_p(pp)
401         php = iops%grid%angles%phi_p(pp)
402         iops%vsf(p, pp) = iops%eval_vsf(
403             angle_diff_3d(th,ph,thp,php))
404       end do
405
406       ! Normalize each row of VSF (midpoint
407       ! rule)
408       norm = sum(iops%vsf(p,:) * iops%grid%
409           angles%area_p(:))
410       iops%vsf(p,:) = iops%vsf(p,:) / norm
411
412       ! % / meter light scattered
413       ! from cell pp (2nd ind.) into direction
414       ! p (1st ind.).
415       iops%vsf_integral(p, :) = iops%vsf(p, :)
416       &
417       * iops%grid%angles%area_p(:)
418       !write(*,*) 'vsf_integral (beta_pp)', p,
419       !      ' = ', iops%vsf_integral(p, :)
420     end do
421   end subroutine calc_vsf_on_grid
422
423   subroutine iop_deinit(iops)
424     class(optical_properties) iops
425     deallocate(iops%vsf_angles)
426     deallocate(iops%vsf_vals)
427     deallocate(iops%vsf)

```

```

423 |     deallocate(iops%vsf_integral)
424 |     deallocate(iops%abs_water)
425 |     deallocate(iops%abs_grid)
426 |     deallocate(iops%source_grid)
427 |
428 | end subroutine iop_deinit
429 |
430 | end module kelp_context

light_context.f90
1 | module light_context
2 | #include "lispf.h"
3 | use sag
4 | use rte_sparse_matrices
5 | !use hdf5
6 | implicit none
7 |
8 | type light_state
9 |     double precision, dimension(:,:,:,:),
10 |         allocatable :: irradiance
11 |     double precision, dimension(:,:,:,:,:),
12 |         allocatable :: radiance
13 |     type(space_angle_grid) :: grid
14 |     type(rte_mat) :: mat
15 | contains
16 |     procedure :: init => light_init
17 |     procedure :: init_grid => light_init_grid
18 |     procedure :: calculate_radiance
19 |     procedure :: calculate_irradiance =>
20 |         light_calculate_irradiance
21 |     procedure :: deinit => light_deinit
22 |     !procedure :: to_hdf => light_to_hdf
23 | end type light_state
24 |
25 | contains
26 |
27 |     ! Init for use with mat
28 |     subroutine light_init(light, mat)
29 |         class(light_state) light
30 |         type(rte_mat) mat
31 |         integer nx, ny, nz, nomega
32 |
33 |         light%mat = mat
34 |         light%grid = mat%grid
35 |
36 |         nx = light%grid%x%num
37 |         ny = light%grid%y%num
38 |         nz = light%grid%z%num
39 |         nomega = light%grid%angles%nomega
40 |
41 |         allocate(light%irradiance(nx, ny, nz))

```

```

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

```

```

86          light%radiance(i,j,k,p),
87          light%mat%x, light%mat%
88          ierr)
89      if(light%mat%ierr .ne. 0) then
90          write(*,*) 'IG ERROR:', light
91          %mat%ierr
92      end if
93
94          index = index + 1
95      end do
96      end do
97      end do
98
99      ! call light%mat%initial_guess()
100
101      ! Solve (LIS)
102      write(*,*) 'Solve matrix'
103      call light%mat%solve()
104
105      index = 1
106
107      write(*,*) 'Extract solution'
108      ! Extract solution
109      do k=1, nz
110          do i=1, nx
111              do j=1, ny
112                  do p=1, nomega
113                      call lis_vector_get_value(light%
114                          mat%x, index, &
115                          light%radiance(i,j,k,p),
116                          light%mat%ierr)
117                  if(light%mat%ierr .ne. 0) then
118                      write(*,*) 'EXTRACT ERROR:', light%mat%ierr
119                  end if
120                  index = index + 1
121              end do
122          end do
123      end do
124
125      end subroutine calculate_radiance
126
127      subroutine light_calculate_irradiance(light)
128          class(light_state) light
129          integer i, j, k
130          integer nx, ny, nz
131          double precision, dimension(light%grid%
132              angles%nomega) :: tmp_rad
133
134          nx = light%grid%x%num

```

```

130 |     ny = light%grid%y%num
131 |     nz = light%grid%z%num
132 |
133 |     do i=1, nx
134 |         do j=1, ny
135 |             do k=1, nz
136 |                 ! Use temporary array to avoid
137 |                 ! creating one
138 |                 ! implicitly at every spatial grid
139 |                 ! point
140 |                 tmp_rad = light%radiance(i,j,k,:)
141 |                 light%irradiance(i,j,k) = &
142 |                     light%grid%angles%
143 |                         integrate_points(tmp_rad)
144 |
145 |             end do
146 |         end do
147 |     end do
148 |
149 | end subroutine light_calculate_irradiance
150 |
151 ! subroutine light_to_hdf(light, radfile,
152 !     irradfile)
153 !     class(light_state) light
154 !     character(len=*) radfile
155 !     character(len=*) irradfile
156 !
157 !     call hdf_write_radiance(radfile, light%
158 !         radiance, light%grid)
159 !     call hdf_write_irradiance(irradfile, light%
160 !         irradiance, light%grid)
161 ! end subroutine light_to_hdf
162 |
163 subroutine light_deinit(light)
164     class(light_state) light
165
166     deallocate(light%irradiance)
167     deallocate(light%radiance)
168 end subroutine light_deinit
169
170 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_asymptotic_light_field(
9     grid, bc, iops, source, radiance,
10    num_scatters, num_threads)
11   type(space_angle_grid) grid
12   type(boundary_condition) bc
13   type(optical_properties) iops
14   double precision, dimension(:,:,:,:,:) :::
15     radiance
16   double precision, dimension(:,:,:,:,:) ,
17     allocatable :: rad_scatter
18   double precision, dimension(:,:,:,:,:) :::
19     source
20   integer num_scatters
21   integer nx, ny, nz, nomega
22   integer max_cells
23   integer n
24   logical bc_flag
25   integer num_threads
26
27   double precision bb
28
29   double precision, dimension(:), allocatable
30     :: path_length, path_spacing, a_tilde, gn
31
32   nx = grid%x%num
33   ny = grid%y%num
34   nz = grid%z%num
35   nomega = grid%angles%nomega
36
37   max_cells = calculate_max_cells(grid)
38
39   allocate(path_length(max_cells+1))
40   allocate(path_spacing(max_cells))
41   allocate(a_tilde(max_cells))
42   allocate(gn(max_cells))
43   allocate(rad_scatter(grid%x%num, grid%y%num,
44     grid%z%num, grid%angles%nomega))
45
46   write(*,*) 'before'
47   write(*,*) 'min radiance =', minval(radiance)
48   write(*,*) 'max radiance =', maxval(radiance)
49   write(*,*) 'mean radiance =', sum(radiance)/
50     size(radiance)
51
52   write(*,*) 'advect source + bc'
53   bc_flag = .true.
54   call advect_light( &
55     grid, iops, source, radiance, &
56     path_length, path_spacing, &
57

```

```

49      a_tilde, gn, bc_flag, num_threads, bc)
50
51      write(*,*) 'after'
52      write(*,*) 'min radiance =', minval(radiance
53          )
54      write(*,*) 'max radiance =', maxval(radiance
55          )
56      write(*,*) 'mean radiance =', sum(radiance)/
57          size(radiance)
58
59      rad_scatter = radiance
60      bb = iops%scat
61
62      do n=1, num_scatters
63          write(*,*) 'scatter #', n
64          call scatter(grid, iops, source,
65              rad_scatter, path_length, path_spacing
66              , a_tilde, gn, num_threads)
67          radiance = radiance + bb**n * rad_scatter
68      end do
69
70      write(*,*) 'asymptotics complete'
71
72      deallocate(path_length)
73      deallocate(path_spacing)
74      deallocate(a_tilde)
75      deallocate(gn)
76      deallocate(rad_scatter)
77  end subroutine
78      calculate_asymptotic_light_field
79
80  subroutine
81      calculate_asymptotic_light_field_expanded_source
82          (&
83              grid, bc, iops, source, &
84              source_expansion, radiance, &
85              num_scatters, num_threads)
86      type(space_angle_grid) grid
87      type(boundary_condition) bc
88      type(optical_properties) iops
89      double precision, dimension(:,:,:,:,:) ::
```

radiance

```

90      double precision, dimension(:,:,:,:,:) ::
```

source_expansion

```

91      integer num_scatters
92      integer nx, ny, nz, nomega
93      integer max_cells
94      integer n
95      logical bc_flag
96      integer num_threads
97
98  end subroutine
99
```

```

90 |     double precision bb
91 |
92 |     double precision, dimension(:), allocatable
93 |         :: path_length, path_spacing, a_tilde, gn
94 |     double precision, dimension(:,:,:,:),
95 |         allocatable :: source
96 |     double precision, dimension(:,:,:,:),
97 |         allocatable :: rad_scatter
98 |     double precision, dimension(:,:,:,:),
99 |         allocatable :: scatter_integral
100|
101|     nx = grid%x%num
102|     ny = grid%y%num
103|     nz = grid%z%num
104|     nomega = grid%angles%nomega
105|
106|     max_cells = calculate_max_cells(grid)
107|
108|     allocate(path_length(max_cells+1))
109|     allocate(path_spacing(max_cells))
110|     allocate(a_tilde(max_cells))
111|     allocate(gn(max_cells))
112|     allocate(rad_scatter(grid%x%num, grid%y%num,
113|                           grid%z%num, grid%angles%nomega))
114|     allocate(scatter_integral(nx, ny, nz, nomega
115|                               ))
116|
117|     write(*,*) 'advect source + bc'
118|     bc_flag = .true.
119|     call advect_light( &
120|         grid, iops, source_expansion(:,:,:,:,1)
121|             , radiance, &
122|             path_length, path_spacing, &
123|             a_tilde, gn, bc_flag, num_threads, bc)
124| ! Disable BC for scattering advection
125|     bc_flag = .false.
126|
127|     rad_scatter = radiance
128|     bb = iops%scat
129|
130|     do n=1, num_scatters
131|         write(*,*) 'scatter #', n
132|         call calculate_scatter_source(grid, iops,
133|             rad_scatter, source, scatter_integral
134|             , num_threads)
135|         source = source + source_expansion
136|             (:,:,:,:,n+1)
137|         call advect_light(grid, iops, source,
138|             rad_scatter, path_length, path_spacing
139|             , a_tilde, gn, bc_flag, num_threads)

```

```

129      radiance = radiance + bb**n * rad_scatter
130
131    end do
132
133    write(*,*) 'asymptotics complete'
134
135    deallocate(path_length)
136    deallocate(path_spacing)
137    deallocate(a_tilde)
138    deallocate(gn)
139    deallocate(rad_scatter)
140    deallocate(scatter_integral)
141  end subroutine
142    calculate_asymptotic_light_field_expanded_source
143
144 ! Add attenuated surface light to existing
145 ! radiance
146 subroutine advect_surface_bc(&
147   i, j, k, p, radiance, &
148   path_spacing, num_cells, a_tilde, bc)
149   type(boundary_condition) bc
150   double precision, dimension(:,:,:,:) :: radiance
151   double precision, dimension(:) :: path_spacing, a_tilde
152   integer i, j, k, p
153   integer num_cells
154   double precision atten
155
156   atten = sum(path_spacing(1:num_cells) *
157     a_tilde(1:num_cells))
158   ! Avoid underflow
159   if(atten .lt. 100.d0) then
160     radiance(i,j,k,p) = radiance(i,j,k,p) +
161       bc%bc_grid(p) * exp(-atten)
162   end if
163  end subroutine advect_surface_bc
164
165 ! Perform one scattering event
166 subroutine scatter(grid, iops, source,
167   rad_scatter, path_length, path_spacing,
168   a_tilde, gn, num_threads)
169   type(space_angle_grid) grid
170   type(optical_properties) iops
171   double precision, dimension(:,:,:,:) :: rad_scatter, source
172   double precision, dimension(:,:,:,:),
173     allocatable :: scatter_integral
174   double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn

```

```

168     integer nx, ny, nz, nomega
169     logical bc_flag
170     integer num_threads
171
172     nx = grid%x%num
173     ny = grid%y%num
174     nz = grid%z%num
175     nomega = grid%angles%nomega
176     bc_flag = .false.
177
178     allocate(scatter_integral(nx, ny, nz, nomega
179     ))
180
181     call calculate_scatter_source(grid, iops,
182         rad_scatter, source, scatter_integral,
183         num_threads)
184     call advect_light(grid, iops, source,
185         rad_scatter, path_length, path_spacing,
186         a_tilde, gn, bc_flag, num_threads)
187
188     deallocate(scatter_integral)
189   end subroutine scatter
190
191 ! Calculate source from no-scatter or previous
192 ! scattering layer
193 subroutine calculate_scatter_source(grid, iops
194   , rad_scatter, source, scatter_integral,
195   , num_threads)
196
197   type(space_angle_grid) grid
198   type(optical_properties) iops
199   double precision, dimension(:,:,:,:,:) :::
200       rad_scatter
201   double precision, dimension(:,:,:,:,:) :::
202       source
203   double precision, dimension(:,:,:,:,:) :::
204       scatter_integral
205   type(index_list) indices
206   integer nx, ny, nz, nomega
207   integer i, j, k, p
208   integer num_threads
209
210   nx = grid%x%num
211   ny = grid%y%num
212   nz = grid%z%num
213   nomega = grid%angles%nomega
214
215   ! Use collapse to combine outer two loops
216   ! to avoid manually deciding nesting details
217   .
218
219
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
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
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
759

```

```

206      !$omp parallel do default(none) private(
207          indices) &
208      !$omp private(i,j,k,p) shared(nx,ny,nz,
209          nomega) &
210      !$omp shared(iops, rad_scatter,
211          scatter_integral) &
212      !$omp num_threads(num_threads) collapse(2)
213
214      do k=1, nz
215          do i=1, nx
216              indices%k = k
217              indices%i = i
218              do j=1, ny
219                  indices%j = j
220                  do p=1, nomega
221                      indices%p = p
222                      call calculate_scatter_integral
223                          (&
224                              iops, rad_scatter,&
225                              scatter_integral,&
226                              indices)
227                  end do
228              end do
229          end do
230      end do
231      !$omp end parallel do
232
233      source(:,:,:,:) = -rad_scatter(:,:,:,:) +
234          scatter_integral(:,:,:,:)
235
236      write(*,*) 'source min: ', minval(source)
237      write(*,*) 'source max: ', maxval(source)
238      write(*,*) 'source mean: ', sum(source)/size
239          (source)
240
241      end subroutine calculate_scatter_source
242
243      subroutine calculate_scatter_integral(iops,
244          rad_scatter, scatter_integral, indices)
245          type(optical_properties) iops
246          double precision, dimension(:,:,:,:,:) :::
247              rad_scatter, scatter_integral
248          type(index_list) indices
249
250          scatter_integral(indices%i,indices%j,indices
251              %k,indices%p) &
252              = sum(iops%vsf_integral(indices%p, :) &
253                  * rad_scatter(&
254                      indices%i,&
255                      indices%j,&
256                      indices%p)

```

```

247         indices%k,:))
248
249     end subroutine calculate_scatter_integral
250
251     subroutine advect_light(grid, iops, source,
252         rad_scatter, path_length, path_spacing,
253         a_tilde, gn, bc_flag, num_threads, bc)
254         type(space_angle_grid) grid
255         type(optical_properties) iops
256         double precision, dimension(:,:,:,:,:) :::
257             rad_scatter, source
258         double precision, dimension(:) :::
259             path_length, path_spacing, a_tilde, gn
260         logical bc_flag
261         type(boundary_condition), intent(in),
262             optional :: bc
263         integer i, j, k, p
264         integer num_threads
265
266         !$omp parallel do default(none) &
267         !$omp private(i,j,k,p) &
268         !$omp shared(rad_scatter,source,grid,iops,
269             bc_flag,bc) &
270         !$omp private(path_length,path_spacing,
271             a_tilde,gn) &
272         !$omp num_threads(num_threads) collapse(2)
273         do k=1, grid%z%num
274             do i=1, grid%x%num
275                 do j=1, grid%y%num
276                     do p=1, grid%angles%omega
277                         call integrate_ray(grid, iops,
278                             source,&
279                                 rad_scatter, path_length,
280                                     path_spacing,&
281                                         a_tilde, gn, i, j, k, p,
282                                         bc_flag, bc)
283                     end do
284                 end do
285             end do
286         end do
287         !$omp end parallel do
288     end subroutine advect_light
289
290     ! New algorithm, double integral over
291     ! piecewise-constant 1d funcs
292     subroutine integrate_ray(grid, iops, source,
293         rad_scatter, path_length, path_spacing,
294         a_tilde, gn, i, j, k, p, bc_flag, bc)
295         type(space_angle_grid) :: grid
296         type(optical_properties) :: iops

```

```

284 |     double precision, dimension(:,:,:,:,:) :: source
285 |     double precision, dimension(:,:,:,:,:) :: rad_scatter
286 |     integer :: i, j, k, p
287 |     ! The following are only passed to avoid unnecessary allocation
288 |     double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn
289 |     logical bc_flag
290 |     type(boundary_condition), intent(in),
291 |         optional :: bc
292 |     integer num_cells
293 |
294 |     call traverse_ray(grid, iops, source, i, j,
295 |         k, p, path_length, path_spacing, a_tilde,
296 |         gn, num_cells)
297 |     rad_scatter(i,j,k,p) =
298 |         calculate_ray_integral(num_cells,
299 |             path_length, path_spacing, a_tilde, gn)
300 |
301 |     if(bc_flag .and. p .le. grid%angles%nomega/2) then
302 |         call advect_surface_bc(&
303 |             i, j, k, p, rad_scatter, &
304 |             path_spacing, num_cells, &
305 |             a_tilde, bc)
306 |     end if
307 |
308 |     if((i .eq. 1) &
309 |         .and. (j .eq. 1) &
310 |         !.and. (k .eq. grid%z%num/2) &
311 |         .and. ( &
312 |             (p .eq. 1) .or. (p .eq. grid%angles%nomega) &
313 |             )) then
314 |
315 |         write(*,*) 'ray (', i, ', ', j, ', ', k
316 |         , ', ', p, ')'
317 |         write(*,*) 'num_cells = ', num_cells
318 |         write(*,*) 'path_spacing:'
319 |         write(*,*) path_spacing(1:num_cells)
320 |         write(*,*) 'path_length:'
321 |         write(*,*) path_length(1:num_cells+1)
322 |         write(*,*) 'a_tilde:'
323 |         write(*,*) a_tilde(1:num_cells)
324 |         write(*,*) 'gn:'
325 |         write(*,*) gn(1:num_cells)
326 |         write(*,*)
```

```

322      ! end if
323
324  end subroutine integrate_ray
325
326  function calculate_ray_integral(num_cells, s,
327      ds, a_tilde, gn) result(integral)
328      ! Double integral which accumulates all
329      ! scattered light along the path
330      ! via an angular integral and attenuates it
331      ! by integrating along the path
332      integer :: num_cells
333      double precision, dimension(num_cells) :: ds
334          , a_tilde, gn
335      double precision, dimension(num_cells+1) :: s
336          double precision :: integral
337          double precision bi, di_exp_bi
338          double precision cutoff
339          integer i, j
340
341          ! Maximum absorption coefficient suitable
342          ! for numerical computation
343          cutoff = 10.d0
344
345          integral = 0
346          do i=1, num_cells
347              bi = -a_tilde(i)*s(i+1)
348              do j=i+1, num_cells
349                  bi = bi - a_tilde(j)*ds(j)
350              end do
351
352              ! In this case, so much absorption has
353              ! occurred
354              ! previously on the path that we don't
355              ! need
356              ! to continue, and we might get underflow
357              ! if we do.
358              if(bi .lt. -100.d0) then
359                  di_exp_bi = 0.d0
360              else
361
362                  ! Without this conditional, overflow
363                  ! occurs.
364                  ! Which is unnecessary, because large
365                  ! absorption
366                  ! means very small light added to the
367                  ! ray
368                  ! at this grid cell.
369                  if(a_tilde(i) .lt. cutoff) then
370                      if(a_tilde(i) .eq. 0) then
371                          di_exp_bi = ds(i) * exp(bi)
372                      else

```

```

362          ! In an attempt to avoid
363          ! overflow
364          ! and reduce compute time,
365          ! I'm combining exponentials.
366          ! di*exp(bi) -> di_exp.bi
367          di_exp.bi = (exp(a_tilde(i))*s(
368              +1) + bi) - exp(a_tilde(i)*s(
369                  i) + bi))/a_tilde(i)
370      end if
371      integral = integral + gn(i)*
372          di_exp.bi
373      end if
374  end do
375
376 end function calculate_ray_integral
377
378 ! Calculate maximum number of cells a path
379 ! through the grid could take
380 ! This is a loose upper bound
381 function calculate_max_cells(grid) result(
382     max_cells)
383 type(space_angle_grid) :: grid
384 integer :: max_cells
385 double precision dx, dy, zrange, phi_middle
386
387 ! Angle that will have the longest ray
388 phi_middle = grid%angles%phi(grid%angles%
389     nphi/2)
390 dx = grid%x%spacing(1)
391 dy = grid%y%spacing(1)
392 zrange = grid%z%maxval - grid%z%minval
393
394 max_cells = grid%z%num + ceiling((1/dx+1/dy)
395     *zrange*tan(phi_middle))
396 end function calculate_max_cells
397
398 ! Traverse from surface or bottom to point (xi
399     , yj, zk)
400 ! in the direction omega_p, extracting path
401 ! lengths (ds) and
402 ! function values (f) along the way,
403 ! as well as number of cells traversed (n).
404 subroutine traverse_ray(grid, iops, source, i,
405     j, k, p, s_array, ds, a_tilde, gn,
406     num_cells)
407 type(space_angle_grid) :: grid
408 type(optical_properties) :: iops
409 double precision, dimension(:,:,:,:,:) ::
410     source
411 integer :: i, j, k, p

```

```

400   double precision, dimension(:) :: s_array,
401   ds, a_tilde, gn
402   integer :: num_cells
403
404   integer t
405   double precision p0x, p0y, p0z
406   double precision p1x, p1y, p1z
407   double precision z0
408   double precision s_tilde, s
409   integer dir_x, dir_y, dir_z
410   integer shift_x, shift_y
411   integer cell_x, cell_y, cell_z
412   integer edge_x, edge_y
413   integer first_x, last_x, first_y, last_y,
414   last_z
415   double precision s_next_x, s_next_y,
416   s_next_z, s_next
417   double precision x_factor, y_factor,
418   z_factor
419   double precision ds_x, ds_y
420   double precision, dimension(grid%z%num) :: ds_z
421   double precision smx, smy
422
423   ! Divide by these numbers to get path
424   ! separation
425   ! from separation in individual dimensions
426   x_factor = grid%angles%sin_phi_p(p) * grid%
427   angles%cos_theta_p(p)
428   y_factor = grid%angles%sin_phi_p(p) * grid%
429   angles%sin_theta_p(p)
430   z_factor = grid%angles%cos_phi_p(p)
431
432   ! Destination point
433   p1x = grid%x%vals(i)
434   p1y = grid%y%vals(j)
435   p1z = grid%z%vals(k)
436
437   ! write(*,*) 'START PATH.'
438   ! write(*,*) 'ijk = ', i, j, k
439
440   ! Direction
441   if(p .le. grid%angles%nomega/2) then
442       ! Downwelling light originates from
443       ! surface
444       z0 = grid%z%minval
445       dir_z = 1
446   else
447       ! Upwelling light originates from bottom
448       z0 = grid%z%maxval
449       dir_z = -1
450   end if

```

```

443
444      ! Total path length from origin to
445      ! destination
446      ! (sign is correct for upwelling and
447      ! downwelling)
448      s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p)
449
450      ! Path spacings between edge intersections
451      ! in each dimension
452      ! Set to 2*s_tilde if infinite in this
453      ! dimension so that it's unreachable
454      ! (e.g., if ray is parallel to x axis, then
455      ! no x intersection will occur.)
456      ! Assume x & y spacings are uniform,
457      ! so it's okay to just use the first value.
458      if(x_factor .eq. 0) then
459          ds_x = 2*s_tilde
460      else
461          ds_x = abs(grid%x%spacing(1)/x_factor)
462      end if
463      if(y_factor .eq. 0) then
464          ds_y = 2*s_tilde
465      else
466          ds_y = abs(grid%y%spacing(1)/y_factor)
467      end if
468
469      ! This one is an array because z spacing can
470      ! vary
471      ! z_factor should never be 0,
472      ! because the ray is then horizontal
473      ! and infinite in length.
474      ! z_factor != 0 is ensured when nphi is even
475
476      ds_z(1:grid%z%num) = dir_z * grid%z%spacing
477          (1:grid%z%num)/z_factor
478
479      ! Origin point
480      p0x = p1x - s_tilde * x_factor
481      p0y = p1y - s_tilde * y_factor
482      p0z = p1z - s_tilde * z_factor
483
484      ! Direction of ray in each dimension. 1 =>
485      ! increasing. -1 => decreasing.
486      dir_x = int(sgn(p1x-p0x))
487      dir_y = int(sgn(p1y-p0y))
488
489      ! Shifts
490      ! Conversion from cell_inds to edge_inds
491      ! merge is fortran's ternary operator
492      shift_x = merge(1,0,dir_x>0)
493      shift_y = merge(1,0,dir_y>0)

```

```

485      ! Indices for cell containing origin point
486      cell_x = floor((p0x-grid%x%minval)/grid%x%
487                      spacing(1)) + 1
488      cell_y = floor((p0y-grid%y%minval)/grid%y%
489                      spacing(1)) + 1
490      ! x and y may be in periodic image, so shift
491      ! back.
492      cell_x = mod1(cell_x, grid%x%num)
493      cell_y = mod1(cell_y, grid%y%num)
494
495      ! z starts at top or bottom depending on
496      ! direction.
497      if(dir_z > 0) then
498          cell_z = 1
499      else
500          cell_z = grid%z%num
501      end if
502
503      ! Edge indices preceeding starting cells
504      edge_x = mod1(cell_x + shift_x, grid%x%num)
505      edge_y = mod1(cell_y + shift_y, grid%y%num)
506
507      ! First and last cells in each
508      if(dir_x .gt. 0) then
509          first_x = 1
510          last_x = grid%x%num
511      else
512          first_x = grid%x%num
513          last_x = 1
514      end if
515      if(dir_y .gt. 0) then
516          first_y = 1
517          last_y = grid%y%num
518      else
519          first_y = grid%y%num
520          last_y = 1
521      end if
522      if(dir_z .gt. 0) then
523          last_z = grid%z%num
524      else
525          last_z = 1
526      end if
527
528      ! Calculate periodic images
529      smx = shift_mod(p0x, grid%x%minval, grid%x%
530                      maxval)
531      smy = shift_mod(p0y, grid%y%minval, grid%y%
532                      maxval)

```

```

529      ! Path length to next edge plane in each
530      ! dimension
531      if(abs(x_factor) .lt. 1.d-10) then
532          ! Will never cross, so set above total
533          ! path length
534          s_next_x = 2*s_tilde
535      else if(cell_x .eq. last_x) then
536          ! If starts out at last cell, then
537          ! compare to periodic image
538          s_next_x = (grid%x%edges(first_x) + dir_x
539                      * (grid%x%maxval - grid%x%minval)&
540                      - smx) / x_factor
541      else
542          ! Otherwise, just compare to next cell
543          s_next_x = (grid%x%edges(edge_x) - smx) /
544                      x_factor
545      end if
546
547      ! Path length to next edge plane in each
548      ! dimension
549      if(abs(y_factor) .lt. 1.d-10) then
550          ! Will never cross, so set above total
551          ! path length
552          s_next_y = 2*s_tilde
553      else if(cell_y .eq. last_y) then
554          ! If starts out at last cell, then
555          ! compare to periodic image
556          s_next_y = (grid%y%edges(first_y) + dir_y
557                      * (grid%y%maxval - grid%y%minval)&
558                      - smy) / y_factor
559      else
560          ! Otherwise, just compare to next cell
561          s_next_y = (grid%y%edges(edge_y) - smy) /
562                      y_factor
563      end if
564
565      s_next_z = ds_z(cell_z)
566
567      ! Initialize path
568      s = 0.d0
569      s_array(1) = 0.d0
570
571      ! Start with t=0 so that we can increment
572      ! before storing,
573      ! so that t will be the number of grid cells
574      ! at the end of the loop.
575      t = 0
576
577      ! s is the beginning of the current cell,
578      ! s_next is the end of the current cell.
579      do while (s .lt. s_tilde)
580          ! Move cell counter

```

```

569     t = t + 1
570
571     ! Extract function values
572     a_tilde(t) = iops%abs_grid(cell_x, cell_y
573                               , cell_z)
573     gn(t) = source(cell_x, cell_y, cell_z, p)
574
575     !write(*,*) ''
576     !write(*,*) 's_next_x = ', s_next_x
577     !write(*,*) 's_next_y = ', s_next_y
578     !write(*,*) 's_next_z = ', s_next_z
579     !write(*,*) 'theta, phi =', grid%angles%
579      theta_p(p)*180.d0/pi, grid%angles%
579      phi_p(p)*180.d0/pi
580     !write(*,*) 's = ', s, '/', s_tilde
581     !write(*,*) 'cell_z =', cell_z, '/', grid
581      %z%num
582     !write(*,*) 's_next_z =', s_next_z
583     !write(*,*) 'last_z =', last_z
584     !write(*,*) 'new'
585
586     ! Move to next cell in path
587     if(s_next_x .le. min(s_next_y, s_next_z))
587       then
588         ! x edge is closest
589         s_next = s_next_x
590
591         ! Increment indices (periodic)
592         cell_x = mod1(cell_x + dir_x, grid%x%
592           num)
593         edge_x = mod1(edge_x + dir_x, grid%x%
593           num)
594
595         ! x intersection after the one at s=
595         s_next
596         s_next_x = s_next + ds_x
597
598     else if (s_next_y .le. min(s_next_x,
598       s_next_z)) then
599       ! y edge is closest
600       s_next = s_next_y
601
602       ! Increment indices (periodic)
603       cell_y = mod1(cell_y + dir_y, grid%y%
603         num)
604       edge_y = mod1(edge_y + dir_y, grid%y%
604         num)
605
606       ! y intersection after the one at s=
606       s_next

```

```

607         s_next_y = s_next + ds_y
608
609     else if(s_next_z .le. min(s_next_x,
610             s_next_y)) then
611         ! z edge is closest
612         s_next = s_next_z
613
614         ! Increment indices
615         cell_z = cell_z + dir_z
616
617         !write(*,*) 'z edge, s_next =', s_next
618
619         ! z intersection after the one at s=
620             s_next
621         if(dir_z * (last_z - cell_z) .gt. 0)
622             then
623                 ! Only look ahead if we aren't at
624                     the end
625                 s_next_z = s_next + ds_z(cell_z)
626             else
627                 ! Otherwise, no need to continue.
628                 ! this is our final destination.
629                 ! exit
630                 s_next_z = 2*s_tilde
631                 !write(*,*) 'end. s_next_z =',
632                     s_next_z
633             end if
634
635         end if
636
637         ! Cut off early if this is the end
638         ! This will be the last cell traversed if
639             s_next >= s_tilde
640         s_next = min(s_tilde, s_next)
641
642         ! Store path length
643         s_array(t+1) = s_next
644         ! Extract path length from same cell as
645             function vals
646             ds(t) = s_next - s
647
648             ! Update path length
649             s = s_next
650         end do
651
652         ! Return number of cells traversed
653         num_cells = t
654
655     end subroutine traverse_ray
656 end module asymptotics

```

light_interface.f90

```

1 module light_interface
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

```

```

50 ! Volume scattering function
51 integer, intent(in) :: num_vsf
52 character(len=*) :: vsf_file
53 !double precision, dimension(num_vsf),
54 ! intent(int) :: vsf_angles
55 !double precision, dimension(num_vsf),
56 ! intent(int) :: vsf_vals
57 ! SUNLIGHT
58 double precision, intent(in) :: solar_zenith
59 double precision, intent(in) :: solar_azimuthal
60 double precision, intent(in) :: surface_irrad
61 ! KELP
62 ! Number of Superindividuals in each depth
63 ! level
64 integer, intent(in) :: num_si
65 ! si_area(i,j) = area of superindividual j
66 ! at depth i
67 double precision, dimension(nz, num_si),
68 ! intent(in) :: si_area
69 ! si_ind(i,j) = number of individuals
70 ! represented by superindividual j at depth
71 ! i
72 double precision, dimension(nz, num_si),
73 ! intent(in) :: si_ind
74 ! Thickness of each frond
75 double precision, intent(in) :: frond_thickness
76 ! Ratio of length to width (0,infty)
77 double precision, intent(in) :: frond_aspect_ratio
78 ! Rescaled position of greatest width (0=
79 ! base, 1=tip)
80 double precision, intent(in) :: frond_shape_ratio
81 ! WATER CURRENT
82 double precision, dimension(nz), intent(in)
83 :: current_speeds
84 double precision, dimension(nz), intent(in)
85 :: current_angles
86 ! SPACING
87 double precision, intent(in) :: rope_spacing
88 double precision, dimension(nz), intent(in)
89 :: depth_spacing
90 ! SOLVER PARAMETERS
91 integer, intent(in) :: num_scatters

```

```

84      ! FINAL RESULT
85      real, dimension(nz), intent(out) :: 
86          avg_irrad, perceived_irrad
87
88      ! -----
89
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        allocatable :: pop_length_means, pop_length_stds
108    ! Number of fronds in each depth layer
109    double precision, dimension(:, :, :, :),
110        allocatable :: 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)

```

```

127 | call grid%init()
128 | !call grid%set_uniform_spacing_from_num()
129 | call grid%z%set_spacing_array(depth_spacing)
130 |
131 | call rope%init(grid)
132 |
133 | write(*,*) 'Rope'
134 | ! Calculate kelp distribution
135 | call calculate_length_dist_from_superinds( &
136 | nz, &
137 | num_si, &
138 | si_area, &
139 | si_ind, &
140 | frond_aspect_ratio, &
141 | num_fronds, &
142 | pop_length_means, &
143 | pop_length_stds)
144 |
145 | rope%frond_lengths = pop_length_means
146 | rope%frond_stds = pop_length_stds
147 | rope%num_fronds = num_fronds
148 | rope%water_speeds = current_speeds
149 | rope%water_angles = current_angles
150 |
151 | write(*,*) 'frond_lengths = ', rope%
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 |
177 | !write(*,*) 'iop init'

```

```

172 ! iops%vsf_angles = vsf_angles
173 ! iops%vsf_vals = vsf_vals
174 call iops%load_vsf(vsf_file, fmtstr)
175
176 ! load_vsf already calls calc_vsf_on_grid
177 !call iops%calc_vsf_on_grid()
178 call iops%calculate_coef_grids(p_kelp)
179
180 !write(*,*) 'BC'
181 max_rad = 1.d0 ! Doesn't matter because we'
182     ll rescale
183 decay = 1.d0 ! Does matter, but maybe not
184     much. Determines drop-off from angle
185 call bc%init(grid, solar_zenith,
186     solar_azimuthal, decay, max_rad)
187 ! Rescale surface radiance to match surface
188     irradiance
189 bc%bc_grid = bc%bc_grid * surface_irrad /
190     grid%angles%integrate_points(bc%bc_grid)
191
192 write(*,*) 'bc'
193 write(*,*) bc%bc_grid
194
195 ! write(*,*) 'bc'
196 ! do i=1, grid%y%num
197 !     write(*,'(10F15.3)') bc%bc_grid(i,:)
198 ! end do
199
200 call light%init_grid(grid)
201
202 write(*,*) 'Scatter'
203 call calculate_light_with_scattering(grid,
204     bc, iops, light%radiance, num_scatters)
205
206 write(*,*) 'Irrad'
207 call light%calculate_irradiance()
208
209 ! Calculate output variables
210 call calculate_average_irradiance(grid,
211     light, avg_irrad)
212 call calculate_perceived_irradiance(grid,
213     p_kelp, &
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

```

```

213 ! write(*,*) 'scat_water = ', scat_water
214 write(*,*) 'kelp '
215 write(*,*) p_kelp(:,:, :)
216 write(*,*) 'ft =', frond%ft
217
218 write(*,*) 'irrad'
219 write(*,*) light%irradiance
220
221 write(*,*) 'avg_irrad = ', avg_irrad
222 write(*,*) 'perceived_irrad = ',
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

```

```

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

```

```

296
297     pop_length_stds(k) = std_num / (
298         num_fronds(k) - 1)
299
300     end do
301
302     end subroutine
303     calculate_length_dist_from_superinds
304
305     subroutine calculate_average_irradiance(grid,
306         light, avg_irrad)
307     type(space_angle_grid) grid
308     type(light_state) light
309     real, dimension(:) :: avg_irrad
310     integer k, nx, ny, nz
311
312     nx = grid%x%num
313     ny = grid%y%num
314     nz = grid%z%num
315
316     do k=1, nz
317         avg_irrad(k) = real(sum(light%irradiance
318             (:,:,k)) / nx / ny)
319     end do
320
321     end subroutine calculate_average_irradiance
322
323     subroutine calculate_perceived_irradiance(grid
324         , p_kelp, &
325             perceived_irrad, irradiance)
326     type(space_angle_grid) grid
327     double precision, dimension(:,:,:) :: p_kelp
328     real, dimension(:) :: perceived_irrad
329     double precision, dimension(:,:,:,:) ::
330             irradiance
331     double precision total_kelp
332     integer center_i1, center_i2, center_j1,
333             center_j2
334
335     integer k
336
337     ! Calculate the average irradiance
338     ! experienced over the frond.
339     ! Has same units as irradiance.
340     ! If no kelp, then just take the irradiance
341     ! at the center
342     ! of the grid.
343     do k=1, grid%z%num
344         total_kelp = sum(p_kelp(:,:,:,k))
345         if(total_kelp .eq. 0) then
346             center_i1 = int(ceiling(grid%x%num /
347                 2.d0))

```

```

337     center_j1 = int(ceiling(grid%y%num /
338                           2.d0))
339     ! For even grid, use average of center
340     ! two cells
341     if(mod(grid%x%num, 2) .eq. 0) then
342         center_i2 = center_i1 + 1
343     else
344         center_i2 = center_i1
345     end if
346     if(mod(grid%y%num, 2) .eq. 0) then
347         center_j2 = center_j1 + 1
348     else
349         center_j2 = center_j1
350     end if
351
352     ! Irradiance at the center of the grid
353     ! (at the rope)
354     perceived_irrad(k) = real(sum(
355         irradiance( &
356             center_i1:center_i2, &
357             center_j1:center_j2, k)) &
358             / ((center_i2-center_i1+1) * (
359                 center_j2-center_j1+1)))
360     else
361         ! Average irradiance weighted by kelp
362         ! distribution
363         perceived_irrad(k) = real( &
364             sum(p_kelp(:,:,k)*irradiance(:,:,k
365                 )) &
366                 / total_kelp)
367     end if
368 end do
369
370 end subroutine calculate_perceived_irradiance
371
372 end module light_interface

```