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

December, 2018

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

Approved:

Advisor
Dr. Kevin Kreider

Co-Advisor
Dr. Curtis Clemons

Faculty Reader
Dr. Gerald Young

Department Chair
Dr. Kevin Kreider

Accepted:

Dean of the College
Dr. Linda Subich

Dean of the Graduate School
Dr. Chand Midha

Date

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. The 3D model of this thesis is found to predict significantly lower light levels than the simpler 1D models, especially in regions of high kelp density where a precise description of self-shading is most important.

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, and his colleague, Dima Mishin, for providing me access to and user support for Nautilus, a distributed, heterogeneity's 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 US National Science Foundation under Grant No. EEC-1359256, and by the Norwegian National Research Council, Project number 254883/E40. The Nautilus computational cluster on which the simulations in this thesis were run is supported by NSF awards CNS 0821155, CNS-1338192, CNS-1456638, CNS-1730158, ACI-1540112, and ACI-1541349.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 [20]. 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) [9, 12, 16, 17].

Furthermore, seaweed cultivation has been proposed as a nutrient remediation technique for natural waters [21]. 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 operations. As an alternative to conventional nutrient remediation techniques, the cultivation of the *Saccharina latissima* within the nutrient plume of WWTP ocean outfalls has been proposed [46]. 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. The project's aim is to achieve "successful and predictable production of high quality biomass thereby making significant steps towards industrial macroalgae cultivation in Norway". Figure

Figure 1.1: *Saccharina latissima* being harvested

1.1 shows seaweed being 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 for each site would help determine the best one. Similarly, if a new

cultivation technique is under consideration, simulation can estimate its viability without deploying it on a large scale and risking failure or avoidable inefficiency.

Recently, a growth model [8] for *S. latissima* has been produced and integrated into the SINMOD [45] 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 on seaweed.

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 [3]. 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, 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* [19], 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 simple.

Both Anderson's and Jackson's models were carried out by numerically solving a system of differential equations by integrating over small time steps. In 1990,

M.A. Burgman and V.A. Gerard developed a stochastic population model [10]. 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. [29] 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 [14, 47]. These were some of the first models of kelp growth that did not specifically relate to *Macrocystis pyrifera* ("giant kelp").

The model developed by Broch et. al. at SINTEF [7, 8, 17] uses a super-individual approach, whereby a small number of individual kelp fronds are explicitly simulated at several discrete depth layers. Each super-individual is assumed to represent a certain number of actual individuals in the population. The number of individuals represented by each super-individual may change over the course of the simulation due to population loss. The super-individual approach has the advantage of capturing some of the dynamics at the individual scale, while compromising full detail for the sake of reduced computational cost.

1.3 Background on Radiative Transfer

In terms of optical quantities, of primary interest is the radiance, which describes the rate of energy flow through each point in space in *each* direction. Irradiance, on the other hand, describes the total energy flow through a point in space over *every* direction, and is calculated by integrating radiance over all angles. Irradiance, in turn, determines 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 used extensively in stellar astrophysics [11, 30]; its application to marine biology is fairly recent [26]. 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 the total radiation in the photosynthetic spectrum, known as photosynthetically active radiation (PAR), 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. However, the emission term will be retained in the calculations of this thesis, as it is mathematically useful to verify the correctness of the solution algorithm.

1.4 Overview of Thesis

The remainder of this document is organized as follows. In Chapter II, a probabilistic model is developed to describe the spatial distribution of kelp by assuming simple distributions for the lengths and orientations of fronds. Chapter III begins with a survey of fundamental radiometric quantities and optical properties of matter. The spatial kelp distribution from Chapter II is used to determine optical properties of the combined water-kelp medium, and the radiative transfer equation, an integro-partial differential equation which describes the the light field as a function of position and angle, is discussed. An asymptotic expansion is explored for the case of low scattering, allowing for analytical, ordered approximations to the true light field. In Chapter IV, details are given for the numerical solution of the equations from Chapters II and III. Both the full finite difference solution and the asymptotic approximation are thoroughly developed.

Chapter V is an in-depth discussion of sources of error in both solution procedures. The concepts of verifying codes in general as well as specific calculations are discussed. Exact discretization errors are calculated via the method of manufactured solutions in order to demonstrate that the methods exhibit their theoretical convergence properties, which builds confidence in the correctness of their implementations. A method for estimating errors for realistic cases is also developed.

Next, Chapter VI surveys practical considerations to keep in mind when applying the algorithms in real situations. Relevant model parameter values from the

literature are collected, and estimates are given for those not readily available. Following that is a set of guidelines for choosing which algorithm and which algorithm parameters to use based on the optical scenario to be simulated. Advantages and disadvantages of both approaches are presented. While the finite difference technique can be applied to any situation, it often requires prohibitively large CPU and memory resources, whereas the numerical asymptotic solution is generally faster and its memory footprint never exceeds the capacity of a standard laptop for reasonable grid sizes. The chapter concludes with a comparison to two simpler light models, and specific qualitative differences are noted. As expected, the presence of self-shading in this model results in the prediction of lower light levels regions of high kelp density. However, the presence of scattering in the model increases light levels elsewhere, especially near the surface of the water.

Finally, Chapter VII concludes the thesis by briefly summarizing the model, discussing its achievements and limitations, and suggesting improvements and avenues for future work. Several appendices follow with further details about the algorithm, as well as the full source code of the Fortran model developed.

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. Probability distributions are given for the size and orientation of the individual kelp fronds, which are inverted to determine the probability of a point in space being occupied by kelp. Ultimately, the kelp density at any point in space is calculated, which informs the absorption coefficient of the effective kelp-water medium.

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* inoculated 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 the fronds are modeled as 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.

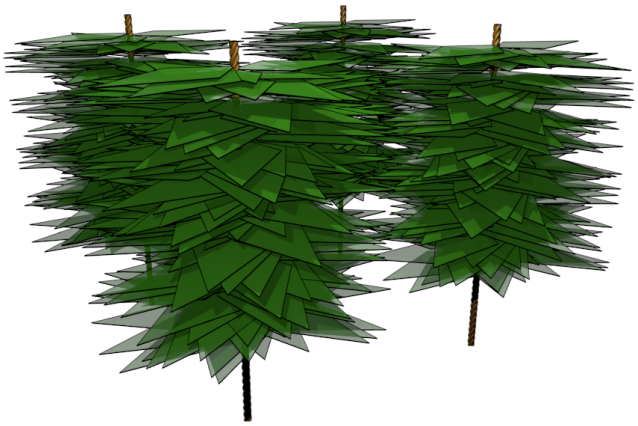

Figure 2.2: Rendering of four nearby vertical kelp ropes as represented in the spatial distribution model. Note the kite-shaped fronds and horizontal orientation.

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 depicted in Figure 2.3. The vectors $\boldsymbol{x} = (x, y, z)$ and $\boldsymbol{\omega} = (\sin\phi\cos\theta, \sin\phi\sin\theta, \cos\phi)$ are also used throughout. Also, the notation $\hat{x}$, $\hat{y}$, $\hat{z}$ is used for the Cartesian unit vectors.

In the following sections, it is necessary to convert between Cartesian and spherical coordinates, which we do using the relations

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

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

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

Then, calculating derivatives from (2.1) yields

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

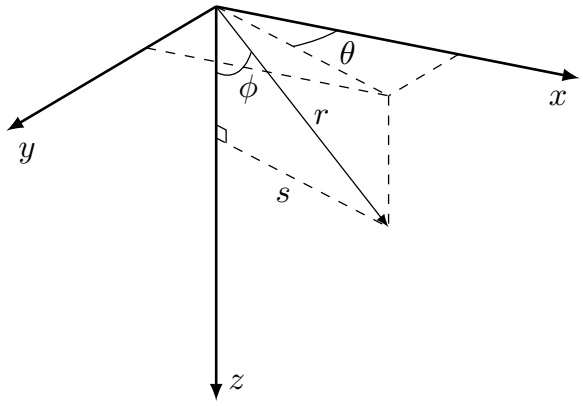

Figure 2.3: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ.

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

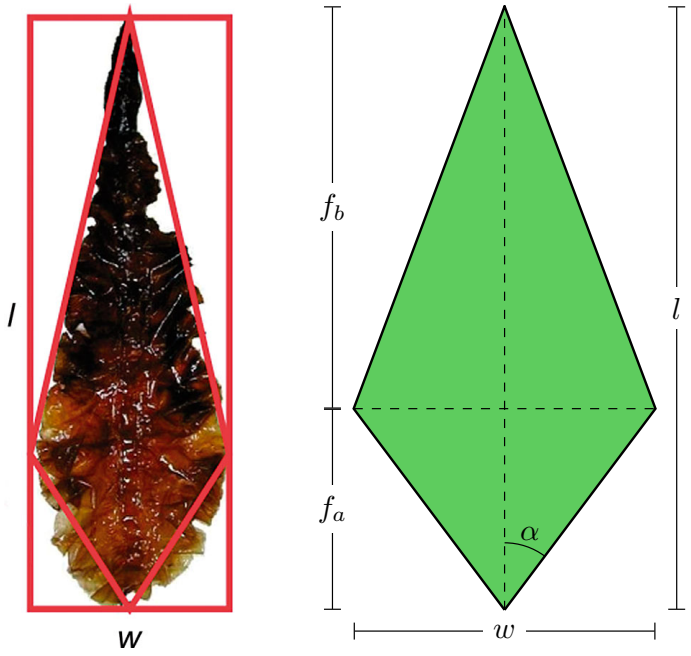

Figure 2.4: Simplified kite-shaped frond. Reproduced with permission from [7].

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, notated f_b, is the shortest distance from that diagonal to the tip. 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

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

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

$$w = \frac{l}{f_r}, \tag{2.3}$$
$$f_a = \frac{l f_s}{1 + f_s}, \tag{2.4}$$
$$f_b = \frac{l}{1 + f_s}. \tag{2.5}$$

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

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

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

$$A = \frac{lw}{2} = \frac{l^2}{2 f_r}, \tag{2.6}$$
$$l = \sqrt{2 A f_r}. \tag{2.7}$$

2.3.2 Length and Angle Distributions

In any given depth layer, 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)

$$P_l(l) = \frac{1}{\sqrt{2\pi\sigma_l^2}} \exp\left(-\frac{(l-\mu_l)^2}{2\sigma_l^2}\right). \tag{2.8}$$

It is further assumed that frond orientation 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 parallel to the current, depending on the stiffness of the frond and stipe. 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 ($\mathrm{s\,m^{-1}}$). 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

$$P_{\theta_f}(\theta_f) = \frac{\exp\left(\eta v_w \cos(\theta_f - \theta_w)\right)}{2\pi I_0(\eta v_w)}, \tag{2.9}$$

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

uniform distribution as κ approaches 0, so

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

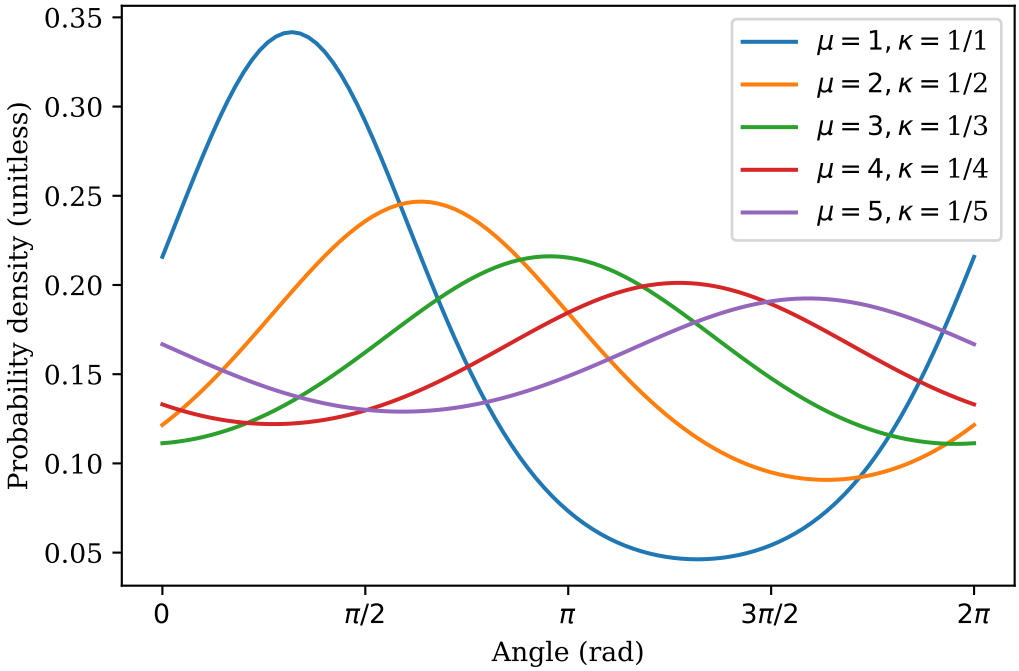

Figure 2.5: The von Mises distribution for a variety of parameters.

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,

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

Therefore, the probability of frond length l and frond angle θ_f coinciding is

$$P_{2D}(\theta_f, l) = P_{\theta_f}(\theta_f) \cdot P_l(l). \tag{2.10}$$

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 Bayes' Theorem,

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

which is currently not taken into consideration in this model.

2.4 Spatial Distribution

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.

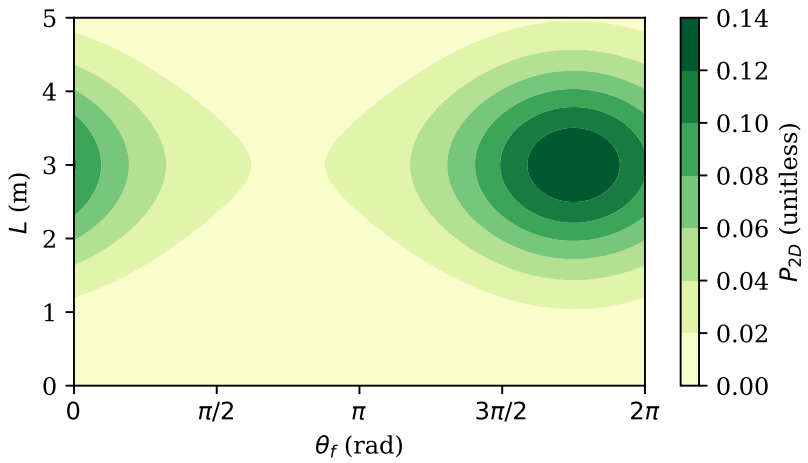

Figure 2.6: 2D length-angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$.

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 polar 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

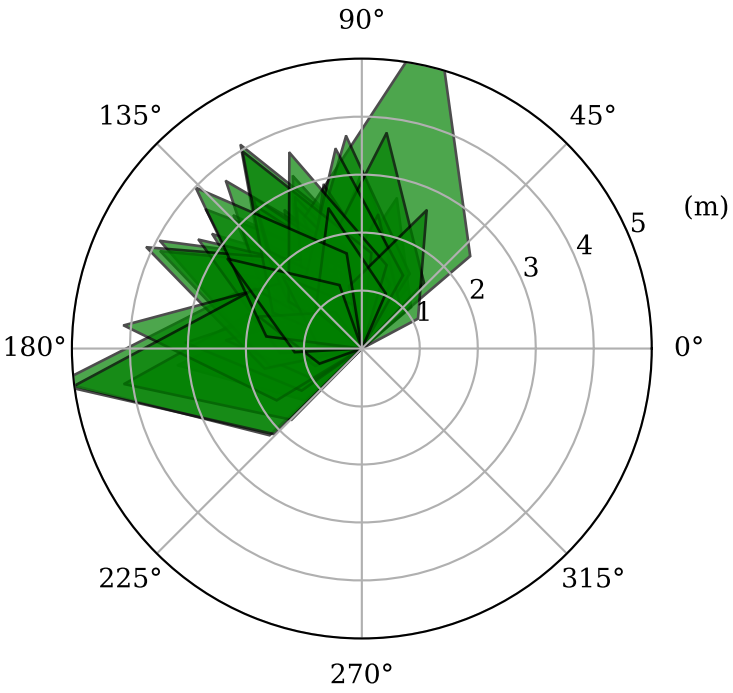

Figure 2.7: A sample of 50 kelp fronds with shape parameters $f_s = 0.5$ and $f_r = 2$ whose lengths are picked from a normal distribution and whose angles are picked from a von Mises distribution.

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

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

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

$$s'_f(\theta'; l) = \frac{l}{\sin\theta' + S(\theta')\frac{2f_r}{1+f_s}\cos\theta'}. \tag{2.11}$$

For convenience, denote the denominator of (2.11) as $d'_f(\theta')$. 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

$$s_f(\theta; l) = s'_f\left(\theta - \theta_f + \frac{\pi}{2}; l\right). \tag{2.12}$$

Similarly, define

$$d_f(\theta) = d'_f\left(\theta - \theta_f + \frac{\pi}{2}\right). \tag{2.13}$$

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) lies within the frond if

$$|\theta_f - \theta| < \alpha \text{ and } s < s_f(\theta).$$

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

$$\theta - \alpha < \theta_f < \theta + \alpha, \tag{2.14}$$

$$l > l_{\min}(\theta, s), \tag{2.15}$$

where

$$l_{\min}(\theta, s) = s \cdot \frac{l}{s_f(\theta; l)} = s \cdot d_f(\theta). \tag{2.16}$$

Then, considering the point to be fixed, (2.14) and (2.15) 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.

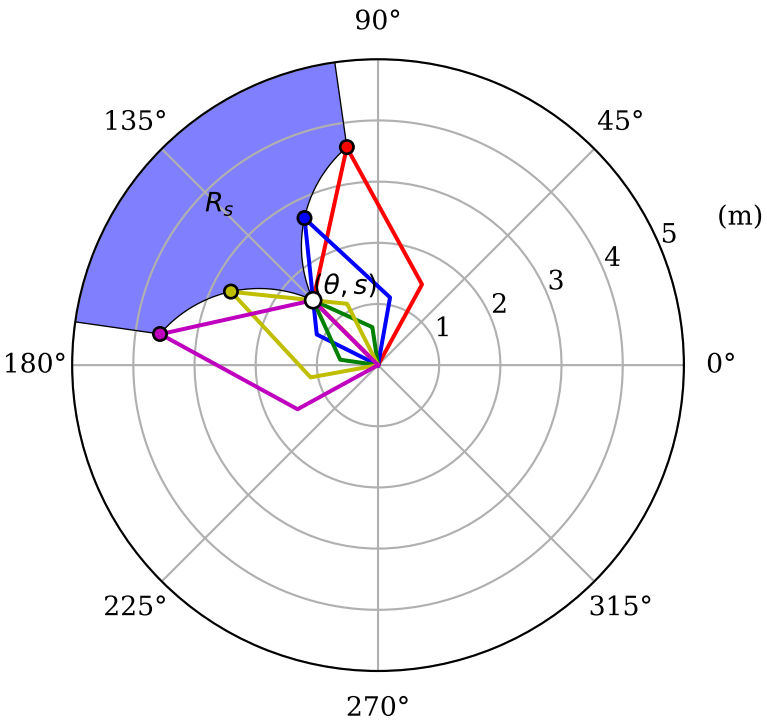

Figure 2.8: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$.

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 $P_{2D}(\theta_f, l)$ from (2.10) over $R_s(\theta, s)$, the occupancy region for the point of interest.

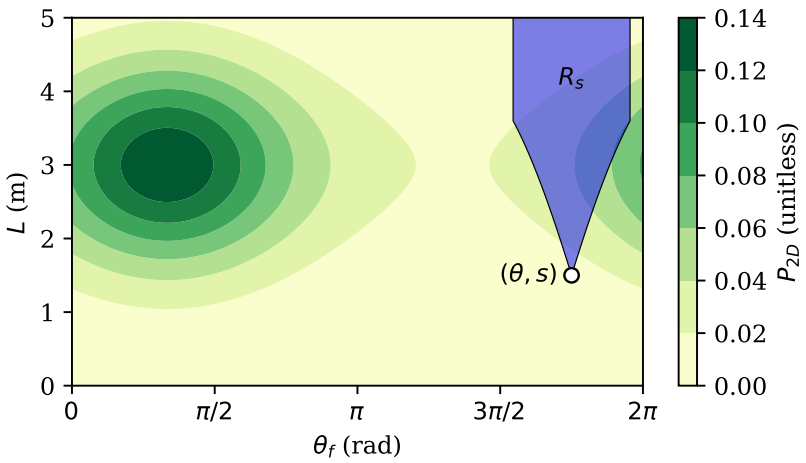

Figure 2.9: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the θ_f-l plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$.

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),

$$\begin{aligned}\tilde{P}_k(\theta, s, z) &= \iint_{R_s(\theta,s)} P_{2D}(\theta_f, l)\, dl\, d\theta_f \\ &= \int_{\theta-\alpha}^{\theta+\alpha} \int_{l_{\min}(\theta_f)}^{\infty} P_{2D}(\theta_f, l)\, dl\, d\theta_f. \end{aligned} \tag{2.17}$$

Assuming that the depth layer has thickness dz and contains n_f fronds of thickness f_t, the proportion of the vertical length of the discrete depth layer occupied by kelp at any position (x, y, z) is given by

$$P_k(x, y) = \frac{n_f f_t}{dz} \tilde{P}_k(x, y). \tag{2.18}$$

In the continuum limit as the number of discrete depth depth layers approaches infinity, $P_k(x, y, z)$ can be interpreted as the probability of the point (x, y, z) being

occupied by kelp. In a three dimensional context, the number of fronds is more likely to be specified by a number density $\rho_f = n_f/dz$ over the vertical length of the rope with units m^{-1}, in which case

$$P_k(x, y, z) = f_t \rho_f(z) \tilde{P}_k(x, y, z). \tag{2.19}$$

Then, since the point $\boldsymbol{x}$ has a probability $P_k(\boldsymbol{x})$ of being occupied by kelp and a probability $(1 - P_k(\boldsymbol{x}))$ of being occupied by the surrounding aquatic medium, the effective absorption coefficient can be calculated as

$$a(\boldsymbol{x}) = P_k(\boldsymbol{x}) a_k + (1 - P_k(\boldsymbol{x})) a_w, \tag{2.20}$$

where a_k is the absorption coefficient of the kelp alone, and a_w is the absorption coefficient of the water and its dissolved and suspended contents.

2.5 Discontinuity at the Rope

While the above model of the kelp distribution is straightforward to evaluate, it does have a noteworthy numerical difficulty in its application. Since the length and orientations are both continuous in the polar coordinates $s > 0$ and θ, the resulting kelp density and effective absorption coefficient are as well. However, they are not necessarily continuous in Cartesian coordinates. In the case of no water current, $v_w = 0$, the flat von Mises distribution produces a continuous solution at the origin. In the more general case, though, there is a high kelp density on one side of the origin in the direction of the current, and a low kelp density immediately on the

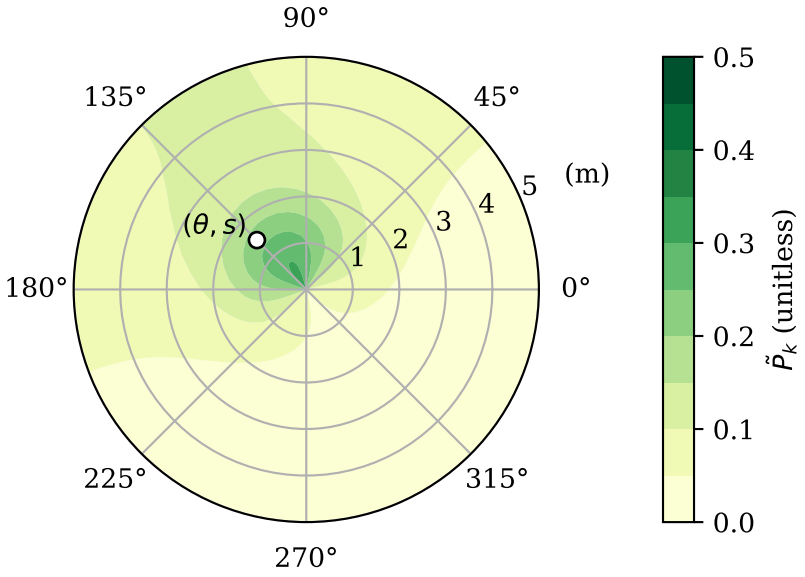

Figure 2.10: Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3, \eta v_w = 1$.

other side. Since the rope is assumed to be infinitely thin and have a fixed position, the sharp corners of the kelp fronds emanate from exactly the same point. Hence, there is in general a discontinuity in the kelp distribution at the origin, and therefore its derivatives are unbounded on the domain.

This is not appealing numerically since the algorithms used in this thesis to solve the differential equation describing the light field are based on interpolation on a discrete Cartesian grid. According to Taylor's theorem, the error incurred by performing such interpolation is bounded by a constant multiple of the appropriate

maximum derivative of the interpolated function over the domain. If the derivatives of the absorption coefficient are unbounded, then so too is the discretization error in the final solution. That is, convergence is not guaranteed in the fine-grid limit.

Luckily, there is a straightforward solution, which is to post-process the spatial kelp distribution with a Gaussian blur in the x and y dimensions. This is achieved via convolution of the solution with a 2D Gaussian kernel centered at the origin. Any blur whatsoever is sufficient to bound the derivatives, and the larger the blur radius, the smaller they become. Obviously, as the blur radius is increased, the kelp distribution tends toward a constant and no longer captures any information about the x-y distribution of the kelp. Therefore, a small blur radius should be used.

2.5.1 One dimensional Gaussian Blur

As a one dimensional analogy, consider the Heaviside step function,

$$H(x) = \begin{cases} 0, & x < 0, \\ \\ 1, & x > 0. \end{cases}$$

Clearly, the function is infinitely steep at the origin. Consider the normalized Gaussian kernel centered at the origin with radius σ_b, given by

$$k(x;\sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{x^2}{2\sigma_b^2}\right). \tag{2.21}$$

A blur is applied by convolving the function with the kernel according to the formula

$$\begin{aligned} \tilde{H}(x) = (k * H)(x) &= \int_{-\infty}^{\infty} H(\tau) k(x-\tau;\sigma_b)\, d\tau \\ &= \int_{0}^{\infty} k(x-\tau;\sigma_b)\, d\tau. \end{aligned}$$

The substitution $u = \tau - x$, $du = d\tau$ yields

$$\tilde{H}(x) = \int_{-x}^{\infty} k(u; \sigma_b)\, du.$$

Note that since the kernel is normalized, the integral from 0 to infinity is $1/2$. Further, since the kernel is even, the integral over $[-x, 0]$ is equal to the integral over $[0, x]$. Hence,

$$\tilde{H}(x) = \frac{1}{2} + \int_{0}^{x} k(u; \sigma_b)\, du.$$

Then, by the fundamental theorem of calculus, the derivative of the convolved function is simply $k(x; \sigma_b)$, whose maximum value is $k(0; \sigma_b) = 1/\sqrt{2\pi\sigma_b^2}$. Then, since the derivative is a linear operator, this result can be generalized to the following statement: Applying a Gaussian blur of radius σ_b to a function with a maximum discontinuity of size J produces a function whose first derivative is bounded by

$$\frac{1}{\sqrt{2\pi}} \frac{J}{\sigma_b}.$$

The same logic applies to directional derivatives of multidimensional scalar functions.

2.5.2 Multidimensional Gaussian Blur

In light of the above arguments, the derivatives of the absorption coefficient are bounded by applying a Gaussian blur to P_k before the calculation of $a(\boldsymbol{x})$. This is done by convolving slices of P_k in the x-y plane with the two-dimensional normalized Gaussian kernel

$$K(x, y; \sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(\frac{x^2 + y^2}{2\sigma_b^2}\right). \tag{2.22}$$

Hence, the blurred solution is

$$P_k^b(x,y,z) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} K(x',y',z;\sigma_b)P_k(x-x',y-y',z)\,dx'\,dy'. \tag{2.23}$$

A physical interpretation of this Gaussian blurring is that P_k^b is the time-averaged kelp distribution, assuming that the rope is allowed to move horizontally, and $K(x,y;\sigma)$ is the PDF of the rope's location distribution. This interpretation is a bit incongruous with the rest of the model since there is no other explicit mention of time-averaging; while the frond length and position distributions can be thought of as continuous approximations to the population distribution at a single point in time, this idea does not apply to the rope's position, since there is only one rope.

2.5.3 Absorption Coefficient Plots

A variety of numerically calculated absorption coefficient fields are shown here in order to demonstrate some key features of the kelp distribution. In each figure, the absorption coefficient is plotted on the color axis, while the norm of its gradient is shown with contours, with brighter contour lines indicating regions of large derivatives. As mentioned above, large derivatives generate large discretization error, and are therefore undesirable. Of course, the distributions must be sufficiently well-defined to capture the characteristics of the kelp. Note that all of the kelp distributions are periodic in order to easily represent multiple vertical lines grown in close proximity.

Figure 2.11(a) shows a sharp kelp distribution with an unrealistically high current velocity, with all fronds identically equal in length, and with no blurring

applied. Note the kite-shaped distribution, as expected. Also note the regions of high derivatives near the origin and inner edges. This sharp distribution requires excessively large numerical grids to approximate well, and is therefore undesirable.

In Figure 2.11(b), the current velocity has been reduced, and a nonzero standard deviation has been given to the frond distribution. With a variety of frond lengths, the edges are no longer as clearly defined, although the general character of the distribution is sensible—the kelp is most dense near the rope in the direction of the current. However, note that there remains a small neighborhood of sharp derivatives encompassing the origin. Such a distribution may still cause numerical difficulties for the algorithm.

Figure 2.11(c) is similar to Figure 2.11(b), except it has been post-processed with a moderate Gaussian blur of radius 40 cm, not so large that it significantly distorts the shape of the distribution, but sufficient to remove the discontinuity in the origin. Such a kelp distribution is ideal, as it balances accuracy with ease of numerical approximation.

In Figure 2.11(d), too large of a blur has been applied; the character of the Gaussian kernel has overwhelmed that of the kelp distribution. While this may still provide a better spatial description of the kelp than a fully horizontally uniform distribution, an unnecessary amount of information has been sacrificed for marginal gains in smoothness over Figure 2.11(c). Such a large blur is unlikely to reduce the discretization error since other factors discussed later contribute to discretization error as well.

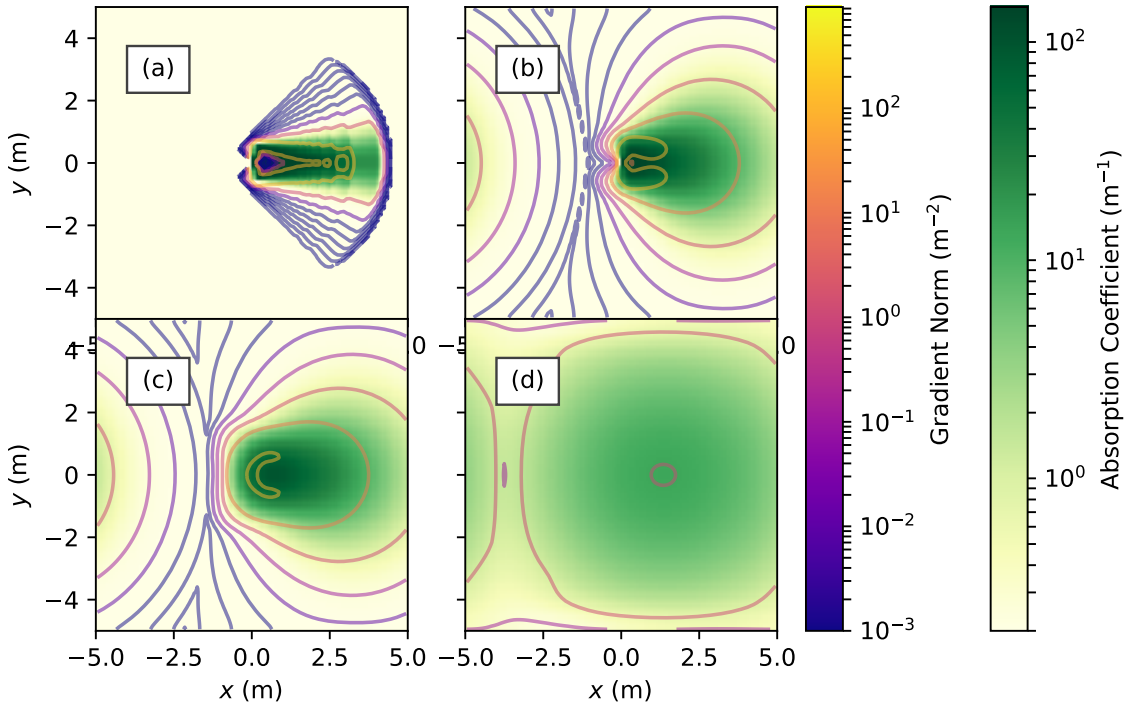

Figure 2.11: z slices of several absorption coefficient distributions from kelp distributions with varying parameters. The norm of the gradient is depicted with contours. (a) $\eta v_w = 90$, $\sigma_l = \sigma_b = 0$. Unrealistically sharp distribution shows kite-shaped character. (b) $\eta v_w = 10$, $\sigma_b = 0$, $\sigma_l = 1\,\text{m}$. More realistic kelp distribution, but still has large derivatives near the origin. (c) $\eta v_w = 10$, $\sigma_b = 0.4\,\text{m}$, $\sigma_l = 1\,\text{m}$. Moderate Gaussian blur bounds derivatives near the origin. (d) $\eta v_w = 10$, $\sigma_b = 2\,\text{m}$, $\sigma_l = 1\,\text{m}$. Over-blurred distribution; should be avoided.

CHAPTER III

LIGHT MODEL

Now that the spatial kelp distribution has been modeled, the radiative transfer equation is introduced, which is used to calculate the light field. An asymptotic series centered at the case of no scattering is developed, which forms the basis for the faster and less memory-intensive of the two solution algorithms presented in Chapter IV.

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. It is necessary to begin by saying that the study of light is riddled with different sets of units which vary between disciplines.

From a physics point of view, light is a form of electromagnetic radiation which carries energy determined exclusively by its wavelength. *Radiometric units* thus describe the transfer of energy, measured in Watts, with quantities such as radiance, irradiance, and radiant flux to describe various types of energy transfer. These quantities come in frequency-dependent and frequency-integrated varieties, depending on the context. From the standpoint of human perception, the importance

of light is not in the raw energy that it transfers, but the degree to which it facilitates vision from a physiological perspective.

Therefore, *Photometric units* take all of the quantities from radiometry, rename them, and weight them by a *luminosity function* which describes frequency dependence of the human eye's sensitivity to light. Various luminosity functions exist which describe the eye's response to light in different circumstances. In photometric units, radiance becomes luminance, irradiance illuminance, and so on.

Meanwhile, plant biologists prefer to measure light by counting photons in the photosynthetic frequency band. The most common quantity in plant biology is *photosynthetically active radiation* (PAR), which is generally measured in moles of photons per square meter per second. All photons in the photosynthetic band (400 nm–700 nm) may be counted equally or weighted according to a plant's photosynthetic response. Radiometric quantities are used throughout this thesis.

3.1.1 Radiometric Quantities

One of the most fundamental quantities in optics is radiant flux Φ, which has units of energy per time (Watts). Considering an element of surface area A, the energy *density* moving through the surface is called irradiance, denoted I, and has units energy per time. Note that the angle ϑ of the surface relative to the light source should also be considered for full detail. Assuming the surface to be flat and the light source to be distant (parallel rays), the flux through the surface is $\Phi = IA\cos\vartheta$. Further, the angular dependence of the light field must be considered. The radiance,

L, expresses this dependence, and is defined as the radiant flux per steradian per projected surface area perpendicular to the direction of propagation of the beam [11]. That is,

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

where $\boldsymbol{\omega}$ is an element of solid angle. Once the radiance L is calculated everywhere, the irradiance is

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

This quantity is also called *scalar irradiance* since it does not consider light through particular surface, but rather weights radiance from all directions equally [22]. For brevity, this quantity is simply called irradiance here.

Irradiance can be approximately converted to PAR units of moles of photons (a mole of photons is also called an Einstein) per second, with the conversion being

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

This is not an exact conversion, but has been found to be accurate to roughly $\pm 10\%$ across a variety of waters [25].

3.1.2 Perceived Irradiance

Assuming that the irradiance $I(\boldsymbol{x})$ is known, the average irradiance at a depth z is calculated as

$$\bar{I}(z) = \frac{\iint I(x, y, z)\, dx\, dy}{\iint 1\, dx\, dy}. \tag{3.2}$$

More relevant, however, is the average irradiance perceived by the kelp. To calculate this value, the irradiance is weighted by the normalized spatial kelp distribution

before taking the mean. Then, the average perceived irradiance at each depth is

$$I_{\text{perc}}(z) = \frac{\iint P_k(x,y,z) I(x,y,z)\, dx\, dy}{\iint P_k(x,y,z)\, dx\, dy}. \tag{3.3}$$

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 $a(\boldsymbol{x})$ (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) $\beta(\Delta) : [-1, 1] \to \mathbb{R}^+$ (units sr^{-1}) defines the probability of light scattering at any given angle from its source, where $\Delta = \cos\vartheta$ is the cosine of the angle between the initial and final directions. Formally, given two directions $\boldsymbol{\omega}$ and $\boldsymbol{\omega}'$, $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$ is the probability density of light scattering from $\boldsymbol{\omega}$ into $\boldsymbol{\omega}'$ (or vice-versa). Now, since a single direction subtends no solid angle, the probability of scattering occurring exactly from $\boldsymbol{\omega}$ to $\boldsymbol{\omega}'$ is 0. Rather, we say

that the probability of radiance being scattered from a direction ω into an element of solid angle Ω is $\int_\Omega \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')\, d\boldsymbol{\omega}'$.

The VSF is normalized such that

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

so that for any ω,

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

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

3.2 The Radiative Transfer Equation

We 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 $\boldsymbol{x}$ and angle $\boldsymbol{\omega}$.

3.2.1 Ray Notation

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

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

Hence, the ray path is parameterized as

$$\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s) = \frac{1}{\tilde{s}}(s\boldsymbol{x} + (\tilde{s} - s)\boldsymbol{x_0}(\boldsymbol{x}, \boldsymbol{\omega})), \tag{3.4}$$

where

$$\boldsymbol{x_0}(\boldsymbol{x}, \boldsymbol{\omega}) = \boldsymbol{x} - \tilde{s}\boldsymbol{\omega} \tag{3.5}$$

is the origin of the ray, and

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

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

3.2.2 Colloquial Description

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

$$\begin{aligned} \left.\frac{dL\left(\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega}\right)}{ds}\right|_{s=\tilde{s}} &= -\left(a(\boldsymbol{x}) + b\right)L(\boldsymbol{x}, \boldsymbol{\omega}) \\ &\quad + b\int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')L(\boldsymbol{x})\, d\boldsymbol{\omega}' + \sigma(\boldsymbol{x}, \boldsymbol{\omega}), \end{aligned} \tag{3.6}$$

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

$$\begin{aligned}\left.\frac{dL\left(\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega},s),\boldsymbol{\omega}\right)}{ds}\right|_{s=\tilde{s}} &= \left[\frac{d\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega},s)}{ds}\cdot\nabla L(\boldsymbol{l}(\boldsymbol{x},\boldsymbol{\omega},s),\boldsymbol{\omega}',\boldsymbol{\omega})\right]\Bigg|_{s=\tilde{s}} \\ &= \boldsymbol{\omega}\cdot\nabla L(\boldsymbol{x},\boldsymbol{\omega}),\end{aligned}$$

which reveals the vector form of the radiative transfer equation,

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

or equivalently,

$$\boldsymbol{\omega}\cdot\nabla L(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L(\boldsymbol{x},\boldsymbol{\omega}) = b\left(\int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L(\boldsymbol{x},\boldsymbol{\omega})\right) + \sigma(\boldsymbol{x},\boldsymbol{\omega}). \tag{3.7}$$

3.2.4 Boundary Conditions

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

$$\begin{aligned}L\left((x_{\min},y,z),\boldsymbol{\omega}\right) &= L\left((x_{\max},y,z),\boldsymbol{\omega}\right), \\ L\left((x,y_{\min},z),\boldsymbol{\omega}\right) &= L\left((x,y_{\max},z),\boldsymbol{\omega}\right).\end{aligned}$$

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

we have

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

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

3.3 Low-Scattering Approximation

In 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 $b \to 0$, it is not necessarily true 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 of the algorithm are discussed in detail in Chapter V.

3.3.1 Asymptotic Expansion

Taking b to be small, we introduce the asymptotic series

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

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

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

Substituting the above into (3.7) yields

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

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

$$\begin{aligned}
&\boldsymbol{\omega}\cdot\nabla L_0(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_0(\boldsymbol{x}) = \sigma_0(\boldsymbol{x},\boldsymbol{\omega}), \qquad (3.8)\\
&\boldsymbol{\omega}\cdot\nabla L_1(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_1(\boldsymbol{x}) = \sigma_1(\boldsymbol{x},\boldsymbol{\omega}) + \int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L_0(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L_0(\boldsymbol{x},\boldsymbol{\omega}),\\
&\boldsymbol{\omega}\cdot\nabla L_2(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_2(\boldsymbol{x}) = \sigma_2(\boldsymbol{x},\boldsymbol{\omega}) + \int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L_1(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L_1(\boldsymbol{x},\boldsymbol{\omega}).\\
&\qquad\qquad\vdots
\end{aligned}$$

In general, for $n \geq 1$,

$$\boldsymbol{\omega}\cdot\nabla L_n(\boldsymbol{x},\boldsymbol{\omega}) + a(\boldsymbol{x})L_n(\boldsymbol{x}) = \sigma_n(\boldsymbol{x},\boldsymbol{\omega}) + \int_{4\pi}\beta(\boldsymbol{\omega}\cdot\boldsymbol{\omega}')L_{n-1}(\boldsymbol{x},\boldsymbol{\omega}')\,d\boldsymbol{\omega}' - L_{n-1}(\boldsymbol{x},\boldsymbol{\omega}). \qquad (3.9)$$

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

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

Grouping by powers of b, we have

$$\begin{cases} L_0(\boldsymbol{x_s}, \boldsymbol{\omega}) = f(\boldsymbol{\omega}) & \text{if } \hat{z} \cdot \boldsymbol{\omega} > 0, \\ L_0(\boldsymbol{x_b}, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} < 0, \end{cases} \tag{3.10}$$

for the first term, and

$$\begin{cases} L_n(\boldsymbol{x_s}, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} > 0, \\ L_n(\boldsymbol{x_b}, \boldsymbol{\omega}) = 0 & \text{if } \hat{z} \cdot \boldsymbol{\omega} < 0, \end{cases} \tag{3.11}$$

for $n > 0$.

3.3.2 Analytical Solution

Given $\boldsymbol{x}, \boldsymbol{\omega}$, consider the path $\boldsymbol{l}(\boldsymbol{x}, \boldsymbol{\omega}, s)$ from (3.4) for $s \in [0, \tilde{s}]$. Denote the leading order radiance, absorption coefficient, and source term along the path by

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

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

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

Then, the first equation from the asymptotic expansion, (3.8) and its associated boundary condition, (3.10), can be rewritten as the first order linear ordinary differ-

ential equation and inhomogeneous boundary condition,

$$\begin{cases} \tilde{\sigma}_0(s) = \dfrac{du_0}{ds}(s) + \tilde{a}(s)u_0(s), \\ u_0(0) = f(\boldsymbol{\omega})H(\boldsymbol{\omega}\cdot\hat{z}), \end{cases} \tag{3.12}$$

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

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

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,

$$\begin{aligned} u_0(s) &= \left[f(\boldsymbol{\omega})H(\boldsymbol{\omega}\cdot\hat{z}) + \int_0^s \exp\left(\int_0^{s'} \tilde{a}(s'')\,ds''\right)\tilde{\sigma}_0(s')\,ds'\right]\exp\left(-\int_0^s \tilde{a}(s')\,ds'\right) \\ &= \exp\left(-\int_0^s \tilde{a}(s')\,ds'\right)f(\boldsymbol{\omega})H(\boldsymbol{\omega}\cdot\hat{z}) \\ &\quad + \int_0^s \exp\left(-\int_{s'}^s \tilde{a}(s'')\,ds''\right)\tilde{\sigma}_0(s')\,ds'. \end{aligned} \tag{3.13}$$

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,

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

This can be similarly extracted along a ray path as

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

Then, since $\tilde{g}_n$ depends only on L_{n-1} and is therefore independent of u_n, (3.9) and its boundary condition (3.11) can be written as the first order linear ordinary differential equation along the ray path,

$$\begin{cases} \tilde{g}_n(s) = \dfrac{du_n}{ds}(s) + \tilde{a}(s) u_n(s) \\ u_n(0) = 0 \end{cases} \tag{3.14}$$

This is exactly (3.12) with $\tilde{\sigma}_0 \to \tilde{g}_n$ and $f(\boldsymbol{\omega}) \to 0$. Hence,

$$u_n(s) = \int_0^s \tilde{g}_n(s') \exp\left(- \int_{s'}^s \tilde{a}(s'')\, ds'' \right) ds'. \tag{3.15}$$

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

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

CHAPTER IV

NUMERICAL SOLUTION

In this chapter, the mathematical details involved in the numerical solution of the equations described in Chapters 2 and 3 are presented. Two solution algorithms are used: finite difference and numerical asymptotics. Both algorithms require a discrete spatial-angular grid, described in Section 4.1. In the finite difference algorithm, the continuum differential equation is approximated at every point in the grid. This forms a large sparse system of linear equations which must be solved simultaneously with an iterative method. The computational cost for this approach is high in terms of CPU usage, and especially in its memory requirement. Meanwhile, the numerical asymptotics algorithm is not as robust, but is significantly faster and does not require the allocation of large arrays.

4.1 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 exist; 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 numbers of grid cells in each dimension are denoted 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. Note that n_ϕ must be even in order to avoid perfectly horizontal rays, as discussed in Section 3.2.1 and Appendix A.

The following indices are assigned to each dimension: $x \to i$, $y \to j$, $z \to k$, $\theta \to l$, $\phi \to m$. It is convenient, however, to use a single index p to refer to directions $\boldsymbol{\omega}$ rather than referring to θ and ϕ separately. Then, the center of a generic grid cell will be denoted as $(x_i, y_j, z_k, \boldsymbol{\omega}_p)$, and the boundaries between adjacent grid cells will be referred to as *edges*. One-indexing is employed throughout this document (i.e., array elements are counted starting at 1, not 0).

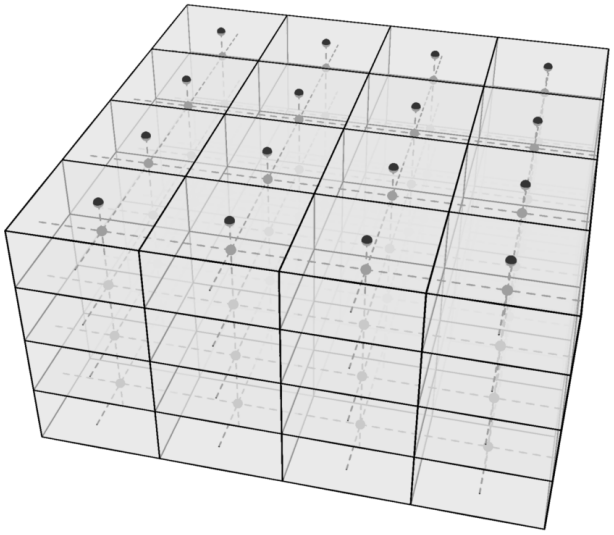

Figure 4.1: Spatial grid. Discrete quantities are calculated at grid cell centers.

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.

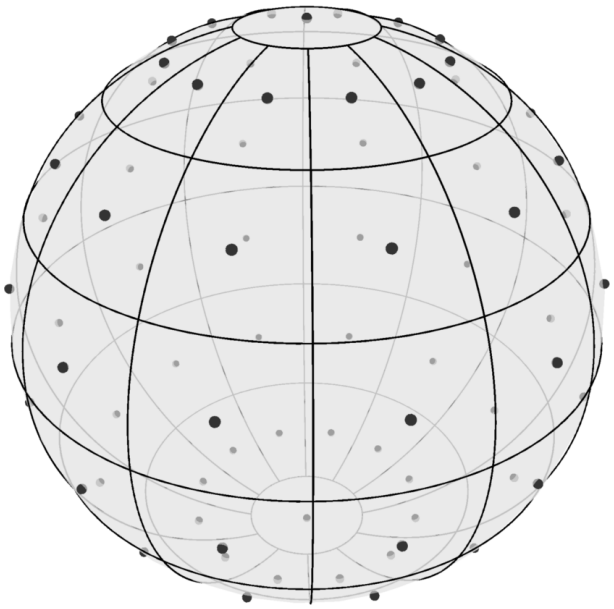

Figure 4.2: Angular grid at each point in space with poles treated separately.

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

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

4.2 Kelp Distribution

In order to determine the appropriate spatial dependence for the absorption coefficient to use in the radiative transfer equation, the spatial kelp distribution, described by (2.18) and its tributaries, is calculated at every point in space. Several computational details related to the kelp distribution are worth mentioning, from the determination of input parameters to algorithmic instructions for the required numerical integration.

4.2.1 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 [41]. Specifically, at each depth k, there are S_k super-individuals, indexed by q. Super-individual q has a frond area A_{kq} and represents S_{kq} individual fronds.

From (2.7), the frond length of the super-individual is $l_{kq} = \sqrt{2A_{kq}f_r}$. Given the super-individual data, we calculate the mean μ and standard deviation σ frond lengths using the formulas

$$\mu_k = \frac{\sum_{q=1}^{S_k} l_{kq}}{\sum_{q=1}^{S_k} S_{kq}}, \tag{4.1}$$

$$\sigma_k = \frac{\sum_{q=1}^{S_k} (l_{kq} - \mu_k)^2}{\sum_{q=1}^{S_k} S_{kq}}. \tag{4.2}$$

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

4.2.2 Integration of Population Distributions

The computational crux of calculating the kelp distribution is to evaluate (2.17) by integrating $P_{2D}(\theta_f, l)$ from (2.10) over the occupancy region $R_s(\theta, s)$, defined by (2.14) and (2.15), for every point (θ, s) in the xy-domain. To recap, the population distribution is

$$\begin{aligned} P_{2D}(\theta_f, l) &= P_{\theta_f}(\theta_f) \cdot P_l(l), \\ P_l(l) &= \frac{1}{\sqrt{2\pi\sigma_l^2}} \exp\left(\frac{-(l-\mu_l)^2}{2\sigma_l^2}\right), \\ P_{\theta_f}(\theta_f) &= \frac{\exp\left(\eta v_w \cos(\theta_f - \theta_w)\right)}{2\pi I_0(\eta v_w)}, \end{aligned}$$

and the occupancy region for (θ, s) is $R_s(\theta, s) \ni (\theta_f, l)$ such that

$$\theta - \alpha < \theta_f < \theta + \alpha,$$

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

To compute the integral, we symbolically convert the two-dimensional integral to a one-dimensional integral in terms of the error function

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt, \tag{4.3}$$

which is a built-in function in most programming languages, including Fortran. So, we evaluate

$$\begin{aligned}
\tilde{P}_k(\theta, s) &= \int_{R_s(\theta,s)} P_{2D}(\theta_f, l)\, d\theta_f\, dl \\
&= \int_{\theta-\alpha}^{\theta+\alpha} \int_{l_{min}(\theta,s)}^{\infty} P_{\theta_f}(\theta_f) \cdot P_l(l)\, dl\, d\theta_f, \\
&= \frac{1}{\sqrt{2\pi\sigma_l^2}} \int_{\theta-\alpha}^{\theta+\alpha} P_{\theta_f}(\theta_f) \int_{l_{min}(\theta,s)}^{\infty} \exp\left(\frac{-(l-\mu_l)^2}{2\sigma_l^2}\right)\, dl\, d\theta_f.
\end{aligned}$$

The substitution $u = (l - \mu_l)/\sqrt{2\sigma_l^2};\ du = dl/\sqrt{2\sigma_l^2}$ produces

$$\begin{aligned}
\tilde{P}_k(\theta, s) &= \frac{1}{\sqrt{\pi}} \int_{\theta-\alpha}^{\theta+\alpha} P_{\theta_f}(\theta_f) \int_{\frac{l_{min}(\theta,s)-\mu_l}{\sqrt{2\sigma_l^2}}}^{\infty} e^{-u^2}\, du\, d\theta_f \\
&= \frac{1}{2} \int_{\theta-\alpha}^{\theta+\alpha} P_{\theta_f}(\theta_f) \left[1 - \mathrm{erf}\left(\frac{l_{min}(\theta,s) - \mu_l}{\sqrt{2\sigma_l^2}}\right)\right] d\theta_f
\end{aligned}$$

Note that the above integrand can be explicitly evaluated at any value of θ_f, and is not restricted to values on a discrete grid. The integral is then evaluated

numerically using a quadrature rule of any chosen degree. At present, a Gauss-Legendre quadrature with degree 101 is applied. If the quadrature degree is chosen too low (e.g. 5), the resulting distribution is marked by numerical artifacts including discontinuities. Since the kelp calculation is orders of magnitude faster than the light field calculation, there is no issue with choosing a high quadrature degree.

4.2.3 Gaussian Convolution

In order to blur the kelp distribution as described in Section 2.5.2, $P_k(x, y, z)$ is first calculated without blurring according to Equations (2.17) and (2.18) at every point (x_i, y_j) in each depth layer, which yields a matrix $P_{ij}^k = P_k(x_i, y_j, z)$. Then, the discrete convolution of P_{ij}^k with the Gaussian kernel $K(x, y; \sigma_b)$ from Equation (2.22) is numerically computed by the same process used to post-process pixelated image with a Gaussian blur.

Notice first that the integral from Equation (2.23) has infinite extent. For numerical computation, a discrete grid with finite bounds must be chosen. Specifically, an array of uniformly spaced quadrature points $(x'_{i'}, y'_{j'})$ is chosen for the integration variables x' and y' with spacings dx and dy to match the existing grid. In order to capture up to two standard deviations of the Gaussian distribution, the number of quadrature points is each dimension is chosen to be

$$n_K = 2r_K + 1, \tag{4.4}$$

where

$$r_K = \text{ceil}\left(\max\left(\frac{2\sigma_b}{dx}, \frac{2\sigma_b}{dy}\right)\right). \tag{4.5}$$

Then, the quadrature points are

$$x'_{i'} = (i' - r_K)dx, \quad i' = 1, \ldots, n_k,$$

$$y'_{j'} = (j' - r_K)dy, \quad j' = 1, \ldots, n_k.$$

The kernel is then evaluated at the quadrature points to form the matrix

$$K_{i'j'} = K\left(x'_{i'}, y'_{j'}; \sigma_b\right).$$

Note that the continuous $K(x, y; \sigma_b)$ is normalized only on the infinite plane; the discrete K_{ij} must therefore be re-normalized on the finite grid as

$$K_{i'j'} = \frac{K_{i'j'}}{\sum_{ij} K_{ij}}.$$

Then, letting

$$\delta_x(i, i') = i - r_K + i' \bmod 1\ n_x,$$

$$\delta_y(j, j') = j - r_K + j' \bmod 1\ n_y,$$

the convolved kelp distribution P^{kb}_{ij} is evaluated as

$$\begin{aligned}
P^{kb}_{ij} &= P^b_k(x_i, y_j, z; \sigma_b) \\
&= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} K(x', y', z; \sigma_b) P_k(x_i - x', y_j - y', z)\, dx'\, dy' \\
&= dx\, dy \sum_{i'=1}^{n_k}\sum_{j'=1}^{n_k} K_{i'j'} P_k\left(x_i - x'_{i'}, y_j - y'_{j'}, z\right) \\
&= dx\, dy \sum_{i'=1}^{n_k}\sum_{j'=1}^{n_k} K_{i'j'} P_k\left(x_{\delta_x(i,i')}, y_{\delta_y(j,j')}, z\right) \\
&= dx\, dy \sum_{i'=1}^{n_k}\sum_{j'=1}^{n_k} K_{i'j'} P^k_{\delta_x(i,i'),\delta_y(j,j')}.
\end{aligned}$$

That is, at each grid point (x_i, y_j), a weighted sum of periodic-adjacent values is taken according to the Gaussian kernel in order to perform the blur.

4.3 Quadrature Rules

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

4.3.1 Spatial Quadrature

Define the *spatial characteristic function* as

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

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

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

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

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

where ds_{ijk} is the total path distance of $\boldsymbol{l}(s)$ through X_{ijk}. Full details of the path integral algorithm for 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 $f(\boldsymbol{\omega})$ over all angles is approximated as

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

Similarly, the amount of light scattered between angular grid cells is found by integrating β over specific regions. Consider two angular grid cells, Ω_p and $\Omega_{p'}$. 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 $\boldsymbol{\omega} \in \Omega_p$ to $\boldsymbol{\omega}' \in \Omega_{p'}$ (or vice versa) is

$$\beta_{pp'} = \frac{1}{\left|\Omega_p\right| \left|\Omega_{p'}\right|} \int_{\Omega_p} \int_{\Omega_{p'}} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')\, d\boldsymbol{\omega}'\, d\boldsymbol{\omega} \approx \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}'),$$

assuming that β is approximately constant over Ω_p and $\Omega_{p'}$. 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

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

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

$$\int_{4\pi} \beta(\boldsymbol{\omega_p} \cdot \boldsymbol{\omega_{p'}}) L(\boldsymbol{x}, \boldsymbol{\omega}')\, d\boldsymbol{\omega} \approx \sum_{p=1}^{n_{\boldsymbol{\omega}}} \beta_{pp'} \left|\Omega'\right| L_{ijkp}. \tag{4.6}$$

4.4 Numerical Asymptotics

The asymptotic approximations (3.13) and (3.15) to the radiative transfer equation (3.7) are evaluated numerically as follows. Given a position $\boldsymbol{x}$ and direction $\boldsymbol{\omega}$, a path through the discrete grid can be constructed using the ray tracing algorithm described in Appendix B. Let $\nu = 1, \ldots, N-1$ index the spatial grid cells traversed (wholly or partially) by the ray, and define the *path-length characteristic function*

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

where $s \in [s_\nu, s_{\nu+1}]$ parameterizes the path segment traversing cell ν and $ds_\nu = s_{\nu+1} - s_\nu$. is the length of the segment. 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

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

Given s, the index of the next edge crossing is

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

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

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

Then, evaluating (3.15) at $s = \tilde{s}$ is calculated as

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

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

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

Define the intermediate variable

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

which permits the simple formula

$$u_n(\tilde{s}) = \sum_{\nu=1}^{N-1} \tilde{g}_{n\nu} d_\nu e^{b_\nu}. \tag{4.7}$$

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 (3.13) is given by

$$u_0(\tilde{s}) = f(\boldsymbol{\omega}) H(\boldsymbol{\omega} \cdot \hat{z}) \exp\left(- \sum_{j=1}^{N-1} \tilde{a}_j ds_j \right) + \sum_{\nu=1}^{N-1} \tilde{\sigma}_{0\nu} d_\nu e^{b_\nu}. \tag{4.8}$$

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 to construct a linear system of equations whose solution approximates that of the analytical equation. The price of a general solution, 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

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

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

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

and the backward difference by

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

For the upper and lower boundaries, an asymmetric finite difference formula is required since the distance between grid centers in the z direction is dz, whereas the distance to the surface is $dz/2$. In general, the Taylor Series of a function f about x is written as

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

Truncating after the first few terms, we have

$$f(x + \varepsilon) = f(x) + f'(x)\varepsilon + \frac{f''(x)}{2}\varepsilon^2 + \mathcal{O}(\varepsilon^3). \tag{4.9}$$

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

$$f(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \tag{4.10}$$

Rearranging (4.9) produces

$$f''(x)\varepsilon^2 = 2f(x + \varepsilon) - 2f(x) - 2f'(x)\varepsilon + \mathcal{O}(\varepsilon^3). \tag{4.11}$$

Combining (4.10) with (4.11) gives

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

Hence,

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

Then, dividing by $3\varepsilon/2$ gives

$$f'(x) = \frac{-4f(x - \frac{\varepsilon}{2}) + 3f(x) + f(x+\varepsilon)}{3\varepsilon} + \mathcal{O}(\varepsilon^2). \tag{4.12}$$

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

$$f'(x) = \frac{-f(x-\varepsilon) - 3f(x) + 4f(x + \frac{\varepsilon}{2})}{3\varepsilon} + \mathcal{O}(\varepsilon^2). \tag{4.13}$$

4.5.2 Difference Equations

For every spatial grid cell, the scattering integral is discretized as in Section 4.4 by

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

or equivalently,

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

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

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

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 mod1 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 (4.10) using the surface radiance value, which gives

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

Combining L_{ijkp} terms and moving the boundary condition to the other side gives

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

Likewise, for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

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, $\mathbb{AX} = \mathbb{B}$. In the coefficient matrix $\mathbb{A}$, 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 practice, some procedure is necessary for constructing an ordered list of the multidimensional grid cells. One option, employed here, is to use a block structure where dimensions are nested within one another. An ordering for the dimensions is chosen, from outermost to innermost. Adjacent rows and columns in the matrix are associated with adjacent grid cells in the innermost

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

In the numerical implementation of this model, we choose the order of dimensions to be $\boldsymbol{\omega}, z, y, x$, with $\boldsymbol{\omega}$ being the outermost and x being the innermost. Recall that θ and ϕ are already combined, both indexed by p, as discussed in Section 4.1 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.

With this storage scheme in mind, the coefficients of the discretized equation applied to $(x_i, y_j, z_k, \omega_p)$ is stored in row

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

of the matrix $\mathbb{A}$. Since the same ordering is used for rows and columns of the coefficient matrix $\mathbb{A}$, L_{ijkp} is located at position r_{ijkp} of the solution vector $\mathbb{X}$, 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 $\mathbb{B}$.

Also relevant is the total size of the system and of the sparse matrices necessary to store. The sizes of $\mathbb{A}$, $\mathbb{X}$, and $\mathbb{B}$ are the number of grid cells, which is just

$n_x n_y n_z n_{\boldsymbol{\omega}}$. Most of these elements, though, are zero, since spatial 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.

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

$$\begin{aligned} N_{\mathbb{A}} &= (n_{\boldsymbol{\omega}} + 6) \cdot n_x n_y (n_z - 2) n_{\boldsymbol{\omega}} \\ &\quad + (n_{\boldsymbol{\omega}} + 5) \cdot n_x n_y n_{\boldsymbol{\omega}} + (n_{\boldsymbol{\omega}} + 6) \cdot n_x n_y n_{\boldsymbol{\omega}} \\ &= n_x n_y n_{\boldsymbol{\omega}} \left[n\omega n_z - n_{\boldsymbol{\omega}} + 6 n_z - 6 + n_{\boldsymbol{\omega}} + 5 \right] \\ &= n_x n_y n_{\boldsymbol{\omega}} \left[n_z (n_{\boldsymbol{\omega}} + 6) - 1 \right]. \end{aligned} \tag{4.14}$$

For example, a $10 \times 10 \times 10 \times 10 \times 10$ grid involves 7,207,800 matrix elements, while a $20 \times 20 \times 20 \times 20 \times 20$ grid requires 1,065,583,200 elements to be stored! Clearly, the grid must be kept reasonably small in order to be solved by modern computers. Also, note that in the absence of an explicit source term $\sigma(\boldsymbol{x}, \boldsymbol{\omega})$, $\mathbb{B}$ only has nonzero entries for the downwelling surface grid cells, of which there are $n_x n_y n_{\boldsymbol{\omega}}/2$.

4.5.4 Iterative Solution

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.

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

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

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 [39], LGMRES [4], IDR [43], and BI-CGSTAB [44].

For the code in this thesis, a parallel implementation of GMRES provided by a Fortran package called LIS (Library of Iterative Solvers) [27] is employed. LIS provides several execution environments: multithreading, multiprocessing, and MPI (Message Passing Interface) [23]. At present, the multithreading environment is used. While this is the simplest to set up since threads within a process share memory and all communication between workers is hidden from the user, it requires that a single process on a single machine be able to allocate enough space for the entire coefficient matrix, which can easily reach tens or hundreds of gigabytes for large grids.

CHAPTER V

CODE VERIFICATION

The purpose of a numerical simulation is generally to accurately predict the behavior of a real system. All mathematical modeling and numerical calculation, however, deals only with approximations. It is therefore crucial to understand the degree to which these approximations reflect the actual scenario of interest, and to what extent their predictions can be trusted. In this chapter, the validity of the numerical approximations to the continuum model is discussed. Concepts relating to determining the correctness of a whole code and the accuracy of specific results are surveyed. The Fortran implementations of the finite difference and numerical asymptotics solutions of the radiative transfer equation are verified with reasonable confidence to be free of detectable coding mistakes.

5.1 Sources of Error in Numerical Simulations

While simulations attempt to realistically reproduce real-life observations, many factors conspire to distort the numerical results they produce. The following is an overview of the errors incurred throughout the process of creating and evaluating numerical models. Some of these errors are *ordered*, meaning that they can be decreased predictably by performing additional computations, while others are not.

First, any mathematical model is based on simplifying assumptions compared to the real system. Any assumptions which do not hold precisely for the real system contribute to differences between solutions to the model equations and the behavior of the real system. Such differences can be termed *modeling errors*, and are a present in the exact solutions to the mathematical equations comprising the model. Once a model is formulated, it contains free parameters which can be varied to emulate different physical scenarios. To evaluate the model, specific parameters must be used which reflect the scenario of interest. The values of these parameters often are obtained experimentally, and may not be known with exact precision. Error resulting from inaccurate physical parameter values can be called *parameter uncertainty error*.

Mathematical models accurate enough to be useful are usually difficult or impossible to solve analytically. Therefore, either mathematical simplifications or numerical approximations, or both, may be employed in order to obtain a solution. For example, in the low-scattering solution presented in Section 3.3, the exact solution is expanded in a Taylor Series, from which only the first few terms are used, with the rest discarded. The error resulting from using a finite number of terms from an infinite series is referred to as *truncation error*. Truncation error is ordered for a convergent series, as an arbitrary number of terms can be used to better approximate the true solution. When a continuous equation is solved numerically, it is necessary to compute the solution on a finite number of discrete points rather than on the complete domain. The error incurred by doing so is called *discretization error*.

Discretization error is also ordered, as arbitrarily fine grids can be chosen in order to better approximate the continuum solution.

Finally, once the model, solution algorithm, and discretization scheme are chosen, a computer is used to perform the actual calculations. Computers do not operate on the full set of real numbers, but rather on a finite subset of the real numbers with a predetermined floating point precision, depending on the hardware and software environment. The loss of accuracy in computations due to the use of floating point numbers is termed *round-off error* [35].

5.2 Verification and Validation

There are two aspects to building confidence in numerical codes, specifically those which solve PDEs: verification and validation. Verification deals with *solving the equations right*, while validation deals with *solving the right equations* [35]. Validation involves comparison with experimental evidence in order to determine that governing equations accurately describe physical phenomenon, and that their solutions match observation. In other words, it is the process of checking that modeling errors are sufficiently small to model the system as accurately as necessary. Validation is an ongoing process; as new experimental data become available, the equations can be solved under appropriate conditions in an attempt to replicate the new observations [37]. Verification, however, is a purely mathematical exercise, and has nothing to do with the physical system being modeled [36]. It deals only with the agreement between an equation and its numerical solution produced by a partic-

ular implementation of an algorithm. It involves checking that ordered sources of numerical error (truncation and discretization errors) decrease as expected when additional computations are performed. 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 present radiative transfer code is left as future work. However, verification of both the finite difference and numerical asymptotics algorithms is demonstrated here. There are two phases of verification. The first is *code verification*, where the overall implementation of an algorithm is tested, and 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 analytical 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. Nevertheless, analyzing a well-chosen unrealistic situation is sufficient to check that the order of the code's discretization error matches the theoretical order of the algorithm.

For realistic conditions, however, a second stage of verification is employed: *verification of calculations*. In this phase, a specific calculation of interest is performed, and the error is *estimated* since it cannot be measured explicitly when the exact solution is not available. This is generally done by repeating the calculation for several grid sizes, as above, then using a technique called Richardson Extrapolation to estimate the limiting solution as the grid spacing approaches zero. This estimated limiting solution is then compared to the actual numerical solutions.

Since the solutions are known on different grids, they cannot be compared pointwise without interpolation. Rather, a single scalar calculated from some integral of the solution is often used as a simple measure of the global order of accuracy [1]. However, the procedure may also be used to estimate discretization for a variable over the entire domain (e.g. for error bars on a plot) by interpolating the variable from each grid at a predetermined set of points, and applying Richardson extrapolation independently at each interpolated point. Of course, the order of the interpolation should be greater than the order of the algorithm under consideration to avoid reducing the observed order of convergence.

5.3 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 (MES). However, such simple cases usually result in such

a loss of generality that they become useless in testing the complicated aspects of the solution algorithm. 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 (MMS), retains arbitrary generality in the equations while making analytical solutions readily available [40]. 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* [36]. Whether a code solves an equation correctly is unrelated to physical realism.

The method of manufactured solutions is performed as follows. Consider a differential equation

$$Du(\boldsymbol{x}) = \sigma(\boldsymbol{x}), \tag{5.1}$$

$$u(\boldsymbol{x}) = f(\boldsymbol{x}) \text{ for } \boldsymbol{x} \in \Sigma, \tag{5.2}$$

where D is a differential operator, u is the solution, 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. Normally, D, σ, and f are known, and solving for u involves determining D^{-1} and calculating $u = D^{-1}\sigma$ subject to (5.2).

The Method of Manufactured Solutions reverses the normal procedure. Here, u is hand-picked at the outset to be easy to calculate, 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 $u(\Sigma) = f$,

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* one 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 acceptable coefficient ranges. 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) such as the Python package Sympy [24] to symbolically compute the source term.

5.3.1 Discretization Error Analysis

Once an exact solution is known, code verification is performed by demonstrating that the discretization error of the code matches the theoretical order of accuracy of the algorithm. Numerical calculations are computed on a sequence of discrete grids from coarse to fine, and errors are calculated by evaluating the exact solution pointwise. Then, the norm of the discretization error is calculated for each grid. To verify an algorithm of order p with grid resolution h, it should be shown that the norm of the pointwise error is approximately proportional to h^p.

This behavior is observed only when the grid spacing is within a neighborhood of zero called the *asymptotic range* [33] (a.k.a. "the sweet spot"). Outside of this neighborhood, order p convergence may not be observed even in a correct implementation since higher order error terms will overpower the order p term. If the observed convergence order does not match the theoretical order, it is either because a coding mistake is present or the asymptotic range has not been achieved [34].

5.3.2 Synthetic Data

In order to perform code verification for the radiative transfer equation, a manufactured solution $L(\boldsymbol{x}, \boldsymbol{\omega})$ for radiance, as well as coefficient functions for the absorption coefficient $a(\boldsymbol{x})$ and volume scattering function $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega'})$, must be chosen *a priori*. Together, the chosen solution and coefficient functions are referred to as *synthetic data*. The code developed for this thesis imposes the following conditions on 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 actual expressions chosen for the synthetic data are quite unwieldy, and are listed in full in Appendix C.

5.3.3 Finite Difference Verification

Since second order finite difference formulas are used, once the asymptotic range for grid spacing has been achieved, decreasing it further should result in the discretization error approaching zero quadratically. The radiative transfer equation involves five discretized variables: (x, y, x, θ, ϕ). A five dimensional resolution space is nontrivial to characterize, so we define the generic spatial and angular grid sizes $n_s = n_x = n_y = n_z$ and $n_a = n_\theta = n_\phi$ for the sake of reducing dimensionality. Then, we use the geometric mean to describe the spatial and angular resolution, as

$$ds = (dx\, dy\, dz)^{1/3}, \tag{5.3}$$

$$da = (d\theta\, d\phi)^{1/2}. \tag{5.4}$$

This reduces the dimensionality of the resolution space to two, but it would be preferable to deal only with 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 5.1, second order convergence is observed, demonstrating that the code is free of coding mistakes affecting the convergence properties of the algorithm.

5.3.4 Numerical Asymptotics Verification

For the numerical asymptotics algorithm, both discretization error and truncation error are present. Therefore, the solutions are not expected to converge to the true solution by increasing the grid size. Neither is it guaranteed that increasing the number of terms will lead to convergence to the true solution. However, since the n term asymptotic solution is a numerical approximation to the n term Taylor series,

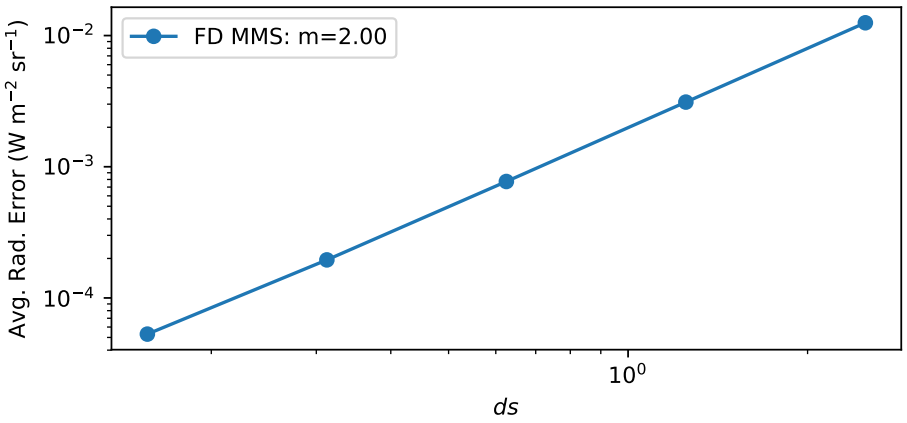

Figure 5.1: Code verification for the finite difference solution. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. A slope of $m = 2$ on a log-log scale demonstrates second order convergence, as expected, demonstrating the correctness of the code.

the norm of the pointwise error should decrease with order $n + 1$ as $b \to 0$. As with discretization error, this can only be observed within some asymptotic range of small b values since higher order error terms dominate the truncation error for large values of b. However, once b is sufficiently small, the truncation error is smaller than the discretization error, and no further improvement results from decreasing b further.

Figure 5.2 demonstrates the convergence of the asymptotic solution as $b \to 0$. The first three approximations are reasonably close, to demonstrating order $n + 1$ convergence, but the $n = 3$ approximation converges slower than expected. It is unclear whether this is due to a coding mistake, the effect of discretization error, or if there is another cause for the sub-optimal convergence.

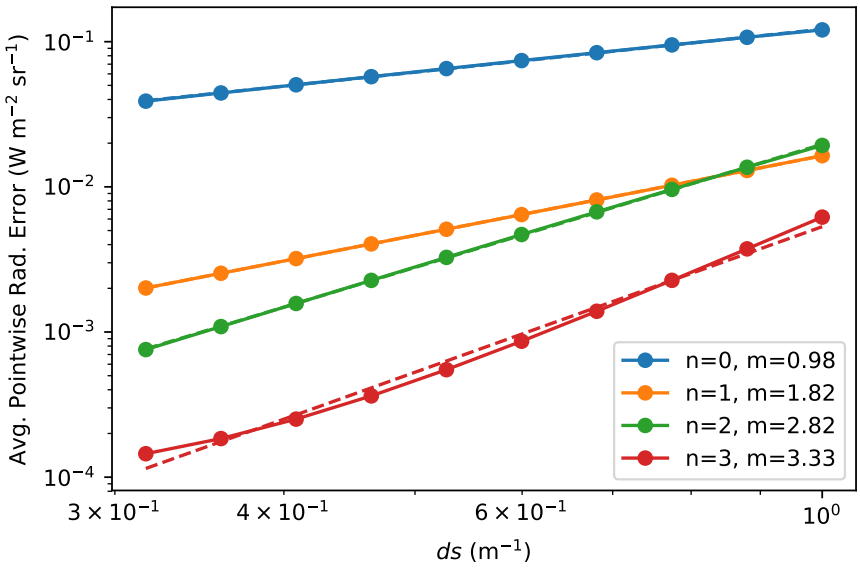

Figure 5.2: Code verification for the numerical asymptotics solution via the Method of Manufactured Solutions. A range of b values are run, using 0–3 terms in the asymptotic series. The legend shows the number of terms (n) and the observed convergence order (m) for each solution.

5.4 Verification of Calculations

As mentioned in Section 5.2, numerical error can be estimated for realistic simulations for cases when the exact solution is not known by analyzing the convergence of a scalar functional of the numerical solution over several grid sizes. This type of error reporting is crucial for accurately communicating the trustworthiness of simulation results, and has long been a common requirement for journals in some computational fields such as computational fluid dynamics (CFD) [38].

5.4.1 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 [32] 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. The basic concept is as follows.

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

$$\phi_e = \lim_{h \to 0} \phi(h),$$

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,

$$\phi - \phi_e = g_0 + g_1 h + g_2 h^2 + g_3 h^3 + \cdots . \tag{5.5}$$

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 technique 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. 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 from an order h^p numerical method to be combined to produce an approximation of order h^{p+1} to the continuum value of ϕ.

5.4.2 Generalized 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 approximately linear in h^2. Therefore, a simple linear fit through multiple points in (h^2, ϕ) space yields ϕ_e as the ϕ-intercept. 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 [48] or Ridge [18] regression can help reduce the influence of outliers.

Additionally, this take on Richardson extrapolation is trivially applied to multiple dimensions. For a grid which has several resolution parameters $h_1, h_2, \ldots, h_n$ (e.g. multiple spatial dimensions, or spatial and angular grid resolutions), if the algorithm is second-order in each resolution parameter, then

$$\phi \approx \phi_e + g_{21} h_1^2 + g_{22} h_2^2 + \ldots + g_{2n} h_n^2.$$

Hence, fitting a hyper-plane through several points in $(h_1^2, h_2^2, \ldots, h_n^2, \phi)$ space similarly produces ϕ_e as the ϕ intercept.

CHAPTER VI

PRACTICAL APPLICATION

This chapter deals with practical considerations when applying the previously discussed numerical algorithms to simulate the light field for realistic scenarios of kelp growing in ocean waters. In this chapter, the spatial kelp distribution model of Chapter II and the light model of Chapter III are implemented as described in Chapter IV and verified in Chapter V and applied to a realistic scenario.

The computational code must be supplied with appropriate physical parameters to emulate the particular seaweed under consideration and the optical properties of the surrounding aquatic medium. Further, the choice of algorithm between numerical asymptotics and finite difference must be made, and algorithm-specific parameters must be chosen, based on the physical situation under consideration, the computational resources available, and the desired level of accuracy in the computed light field.

Guidelines are presented in this chapter to aid the user in making these decisions, and the performance of the model is discussed. Finally, comparison is made to simpler light models, and specific differences are noted. With all of this information, a potential user may decide if and how to use the three-dimensional light model presented in this thesis.

6.1 Physical Parameters

In this section, physical model parameters are discussed. The primary use-case for the present model is that it is run in conjunction with a kelp growth model and ocean model which call it periodically to update the light field. In that case, those models will provide some of the necessary parameters such as the size of the kelp fronds, optical properties of the aquatic medium, and current speed as functions of depth. If the light model is run without kelp growth and ocean models, as is the case for the results in this chapter, then these parameters must be hand-picked to represent a realistic situation of interest. Other parameters external to the kelp growth and ocean models can be found in the literature, as summarized in Table 6.1 and Table 6.2. Still, some parameters remain which are not well described in the literature. In such cases, rough estimates are given or their experimental determination is discussed.

6.1.1 Parameters from Literature

Given here is a table of parameter values found in the literature which are used in the following sections to test this light model under realistic conditions. 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 [8] was given in terms of photons per second, and was converted to $\mathrm{W\,m^{-2}}$ according to the approximate relationship in Equation (3.1).

In [31], detailed measurements of optical properties in various ocean waters are presented. A few of those measurements are reproduced here, using the same

Table 6.1: Physical parameter values.

Parameter Name	Symbol	Value(s)	Citation
Kelp absorptance	A_k	0.8	[13]
Water absorption coefficient	a_w	See Table 6.2	[31]
Scattering coefficient	b	See Table 6.2	[31]
Volume scattering function	β	tabulated	[31, 42]
Frond thickness	f_t	0.4 mm	[15]
Frond aspect ratio	f_r	5.0	[15]
Frond shape parameter	f_s	0.5	estimated
Surface solar irradiance	I_0	$50\,\mathrm{W\,m^{-2}}$	[8]

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 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 turbid water, likely more so than one would expect to find in a seaweed farm. These values give a sense for the range of absorption and scattering coefficients that the model should be tested against. Aside from absorption and scattering coefficients, detailed tabulations of volume scattering functions are also given in [31]. While a simpler linear model is

used for the simulations in this thesis, it is recommended that future simulations interpolate the tabulated VSF from [31].

Table 6.2: Field measurement data of optical properties in the ocean [31]. The site names used in the original paper are used. AUTEC: Bahamas; HAOCE: Coastal southern California; NUC: San Diego Harbor. Absorption, scattering, and total attenuation coefficients (a, b, and $c = a + b$) and their ratios are given.

Site	$a(\mathrm{m}^{-1})$	$b(\mathrm{m}^{-1})$	$c(\mathrm{m}^{-1})$	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

6.1.2 Frond Alignment Coefficient

The *frond alignment coefficient*, η, describes the dependence of frond alignment on current speed, as mentioned in Section 2.3.2. 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 [28] 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 the each frond to make their orientations more easily discernible 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.

6.1.3 Simulation Context

In the case that the model is called by time-dependent kelp growth and ocean models, certain parameters can be passed as arguments to the light calculation subroutines to inform them of the encompassing context. Specifically, the ocean model can provide 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 can also be 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 growth model is expected to provide super-individual data describing the population in each depth layer. Then, Equations (4.1) and (4.2) are used to calculate length and orientation distributions, as described in Section 4.2.1.

Presumably, the ocean model uses a spatial grid much coarser than the grid required for the light model. For example, SINMOD [45] uses a minimum horizontal spatial resolution of 32 m, whereas a grid resolution on the order of centimeters is more appropriate for the light model. While the vertical resolution of the encompassing simulation is probably finer than the horizontal resolution, it may also not be sufficiently fine to use for light field calculations. Assuming that this is the case,

the depth-dependent quantities provided by the encompassing simulations are to be interpolated at the appropriate depths for the light model grid.

While it is reasonable for the ocean model to inform the light model of the surface irradiance and position of the sun, it is unlikely that a full angular radiance distribution is given. Therefore, a simplistic model is used which assigns the highest radiance value to the direction of the sun and lower values to other directions according to the difference in angle, while preserving the total irradiance from the surface. Specifically, the surface boundary condition used is

$$f(\boldsymbol{\omega}) = I_0 \frac{\exp\left(-D_s \cos^{-1}\left(\boldsymbol{\omega}_s \cdot \boldsymbol{\omega}\right)\right)}{\int_{2\pi} \exp\left(-D_s \cos^{-1}\left(\boldsymbol{\omega}_s \cdot \boldsymbol{\omega}\right)\right)\, d\boldsymbol{\omega}}, \tag{6.1}$$

where $\boldsymbol{\omega}_s$ is the propagation direction of the sun's rays and $D_s \in [0, \infty)$ (units rad^{-1}) varies the sharpness of the angular distribution of surface radiance. When $D_s = 0$, the distribution is totally flat, while as $D_s \to \infty$, the distribution approaches a delta function. This can be considered a coarse description of the amount of scattering in the atmosphere, since light arrives at Earth nearly all from a single direction. A value of $D_s = 1\,\text{rad}^{-1}$ is used for the simulations in this chapter.

6.1.4 Standalone Context

The simulation results shown in the later sections of this chapter probe the light model in various ways independent of any encompassing kelp growth or ocean models, and therefore the physical parameters that they would supply must be hand-picked to represent a realistic scenario to which the light model may be applied. Therefore, the following parameter choices are made.

The depth-dependent mean kelp frond length is chosen to be

$$\mu_l(z) = l_{\max} \frac{3z^2 \exp(-z) + 1/2}{12 \exp(-2) + 1/2}, \tag{6.2}$$

which has a local maximum of $\mu_l(2) = l_{\max}$. The value $l_{\max} = 6\,\mathrm{m}$ is used for the sake of agreement with [28], and the vertical number density of fronds is taken to be $\rho_n = 120$ individuals per meter, as in [8]. The kelp frond length standard deviations are chosen to be the constant function $\sigma_l(z) = 1\,\mathrm{m}$. The periodic horizontal domain is $10\,\mathrm{m} \times 10\,\mathrm{m}$, yielding a horizontal rope density of $D = 0.01\,\mathrm{ropes/m^2}$ Also, the current angles are held constant over depth at $\theta_w(z) = 0$, as is the current speed $v_w = 1\,\mathrm{m\,s^{-1}}$ with $\eta = 1\,\mathrm{s\,m^{-1}}$. For the light field, $\boldsymbol{\omega}_s = \hat{z}$ (sunlight from directly above) and $D_s = 1\,\mathrm{rad}^{-1}$ are used.

6.2 Computational Expense

The choice of finite difference or numerical asymptotics and the specific parameters of each has a significant impact on both computation time and solution accuracy. The impact of the choice of algorithm and parameters on both factors is explored in this section in order to aid the reader in making these decisions.

6.2.1 CPU Time

Computation time is a significant consideration, especially when using the light model in the context of a time-dependent kelp growth simulation where it will be called repeatedly. The computational demand can be lessened by updating the light model only a few times per day rather than every time step. Still, accuracy must be

balanced with reasonable resource consumption. If the light model is taking as much time as all of the other aspects combined, any accuracy gained from the light field computation is likely being lost elsewhere in the model.

Also, both algorithms are parallelized using OpenMP [23], so increasing the parallelism decreases the computation time, though it does not reduce the overall resource consumption. Further, some sections of the code are necessarily serial, which reduces the effective parallelism to some degree. When using 8 threads, an average CPU usage of about 7 cores is observed over the course of the entire computation, while for 32 threads, it is around 22. The parallel efficiency decreases as thread count increases because the serial sections bias the average parallelism downward. This is not intended to discourage parallelism, only to represent it accurately; in reality, the speedup is quite noticeable when the thread count is increased.

The choice of algorithm has a significant impact on computation time. The finite difference method relies on an iterative method for solving the linear system. It runs until an error criterion is satisfied, and therefore the exact number of computations is not known ahead of time. On the other hand, the numerical asymptotics algorithm is a direct method that performs a predetermined number of calculations. The computation time of the latter is therefore more predictable.

When using the numerical asymptotics algorithm, the number of terms used in the asymptotic series has a definite impact on computation time. The leading order term involves fewer calculations than the rest since no scattering integration is performed, though the difference is minor. The following terms, however, perform an

identical number of calculations. Therefore, the n term approximation takes about n times as long as the leading order (no-scattering) approximation. Figure 6.1 shows CPU time for 32 cores for both asymptotics and finite difference for five spatial grids.

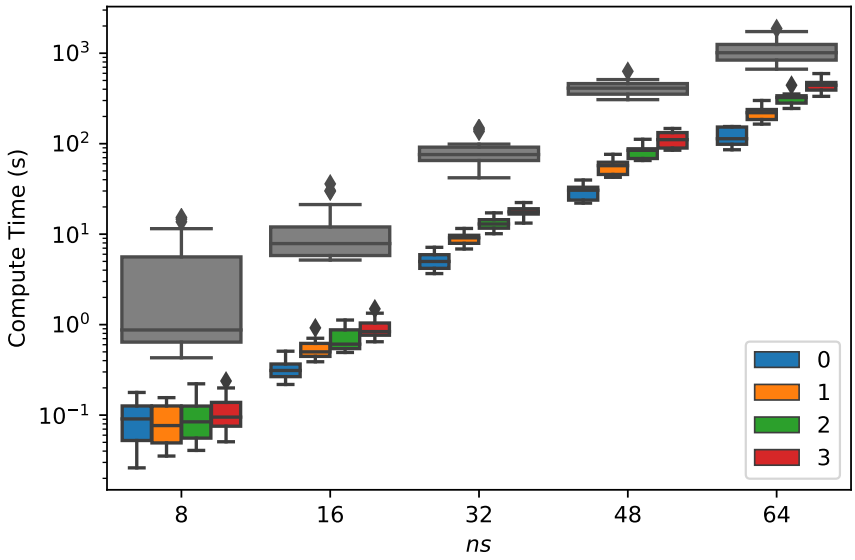

Figure 6.1: Computation time required for numerical asymptotics and finite difference algorithms over a range of spatial grid sizes using $n_a = 10$ and 32 CPUs. Only five grid sizes are shown, with the finite difference shown in gray and numerical asymptotic algorithm for $n = 0, \ldots, 3$ terms in color for each grid. The horizontal offset within each grid size is only for visual clarity. Note the large range in compute times—small simulations take fractions of a second while large grids take upwards of a half-hour.

6.2.2 Memory Usage

Memory usage is perhaps the most important consideration when choosing the algorithm and grid size. While the numerical asymptotics algorithm requires only a few multiples of the memory required to store the radiance itself, the finite difference algorithm requires the generation and storage of a coefficient matrix several orders of magnitude larger than the solution vector. These memory requirements are given for a combination of spatial and angular grid sizes in Table D.1. It is worth noting that using several terms from the asymptotic series does not increase the memory usage, as the same arrays are reused between iterations.

Furthermore, the actual iterative solution of the matrix equation requires the allocation of several multiples of that amount of memory. A good approximation of the memory required to solve the linear system with GMRES (restarted every 100 iterations) is five times the memory required to store the coefficient matrix. Even for large grids, the numerical asymptotics approach has not been observed to use more than five gigabytes of memory, which is well within the memory capacity of a common modern laptop or workstation. On the other hand, the finite difference algorithm uses enormous amounts of memory, from tens of gigabytes for small to medium grids to hundreds of gigabytes for large grids. These estimates are plotted in Figure 6.2, and listed numerically in Table D.2.

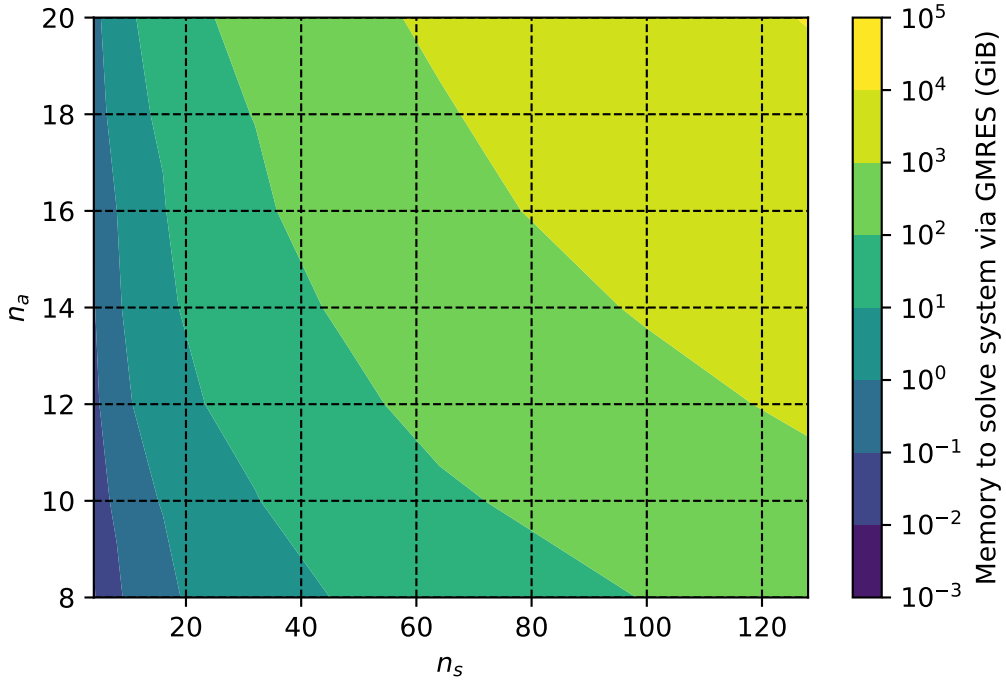

Figure 6.2: Estimated memory required to solve the linear system of equations for the finite difference algorithm using GMRES, restarted every 100 iterations. Table D.2 contains the same data in text form.

6.3 Grid Size and Discretization Error

The size of the spatial-angular grid is an important choice that must be made in order to balance numerical accuracy with computational cost. Since both the numerical asymptotics and finite difference algorithms are second order methods, the discretization error is proportional to the square of the grid spacing, in both the spatial and angular domains. In this section, a realistic simulation (as described in

Section 6.1.4) is performed on many grids, and the discretization error of each is estimated via Richardson Extrapolation, as described in Section 5.4.2. These error estimates allow the spatial and angular components of discretization to be isolated; rules of thumb for both grid spacings are enabled by this analysis.

6.3.1 Error Estimation

A set of simulations is run for a range of spatial and angular grid sizes with optical properties from Petzold's coastal California waters (HAOCE11), listed in Table 6.2. Grid sizes of $n_s = 32, 48, 64, 72$ and $n_a = 4, 8, 10, 12$ are considered; one simulation is run for every combination of these values. The average irradiance over the whole domain is calculated for each simulation, which is used as the scalar metric for comparing solutions on disparate grids. The average irradiances for each simulation are plotted in Figures 6.3 and 6.4 as functions of the squared grid spacings ds^2 and da^2, given by Equations (5.3) and (5.4) with $x_{\max} - x_{\min} = y_{\max} - y_{\min} = 10\,\mathrm{m}$. All parameters aside from grid resolution are held constant, so discretization error is the only source of variation within each plot.

Notice that in Figure 6.4, the average irradiance is linear in squared grid spacing for each slice in the resolution space. This pattern holds for both the finite difference and numerical asymptotics algorithms, indicating that all of the grid sizes considered are within the asymptotic region of both methods, since the observed convergence order matches the theoretical order. This demonstrates that the aver-

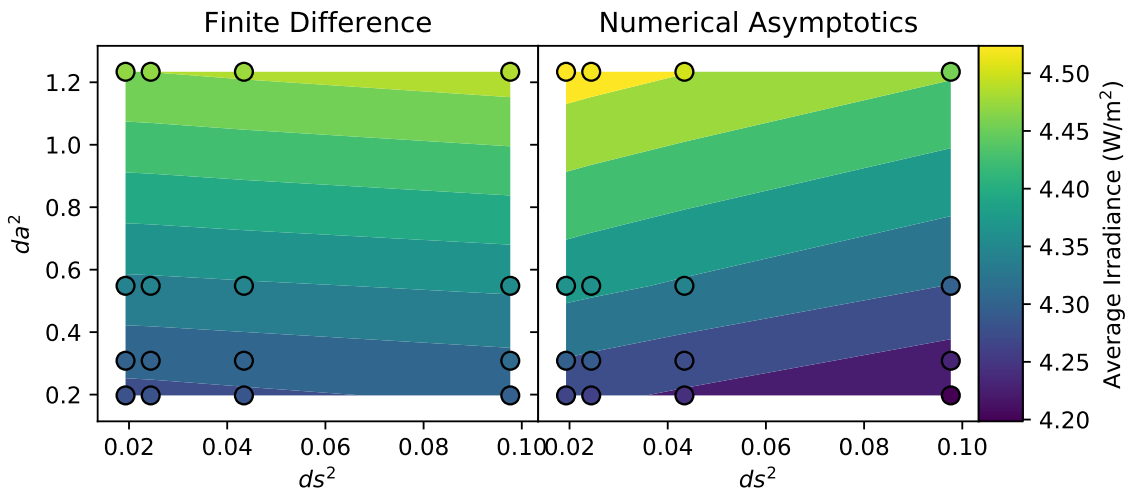

Figure 6.3: Average irradiance plotted against squared resolution for a variety of grid sizes for finite difference and numerical asymptotics with $n = 0$. The linear contours demonstrate that both methods are second order in both resolution parameters.

age irradiance is planar over the whole two-dimensional squared-resolution space, as shown by the parallel linear contours in Figure 6.3.

6.3.2 Error Prediction

The continuum value for the average irradiance is estimated via Richardson extrapolation by fitting a plane through the scalar irradiances in squared-resolution space, as described in Section 5.4.2. Values from this plane are shown in Figure 6.4 as "x"s plotted under the observed errors, shown with circles. The excellent agreement between the prediction and observation is a testament to the validity of the generalized Richardson extrapolation method in this circumstance.

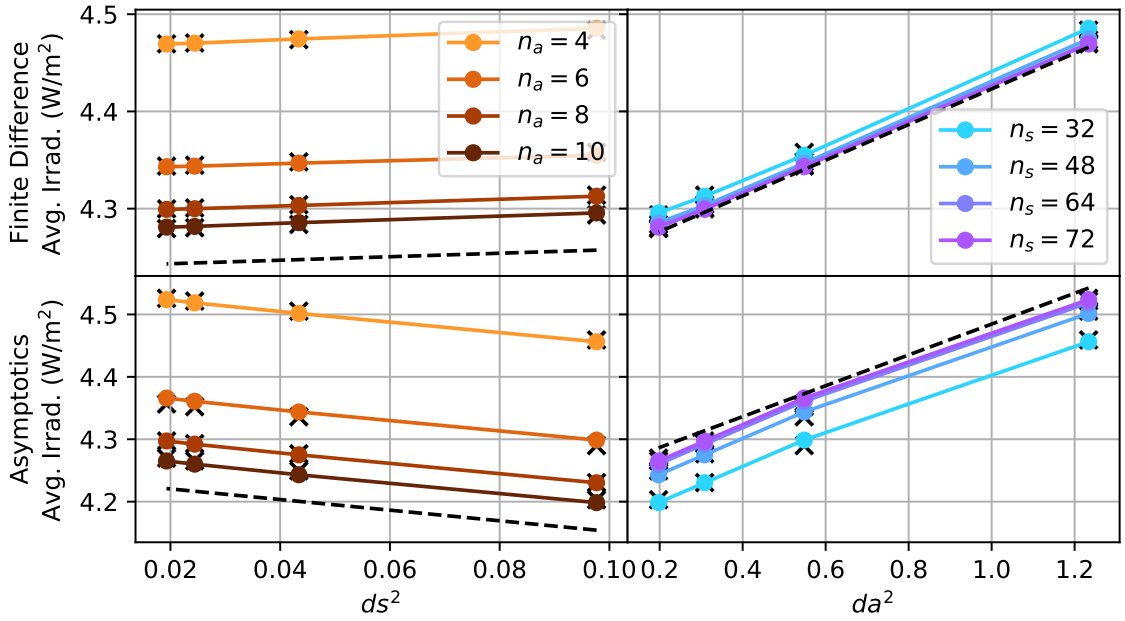

Figure 6.4: Average irradiance versus squared resolution. Each line is a 1D projection from Figure 6.4. Predicted error values are marked with "x"s, and observed error values with circles. Isolated spatial and angular components of discretization error are plotted with a dashed line on the left and right columns respectively.

Specifically, the discretization error for each algorithm is found to be

$$\begin{aligned} \varepsilon_{\text{FD}}(ds, da) &= c_{11} ds^2 + c_{12} da^2, \\ \varepsilon_{\text{asym}}(ds, da) &= c_{21} ds^2 + c_{22} da^2, \end{aligned} \tag{6.3}$$

where

$$\begin{aligned} c_{11} &= 0.180, \quad & C_{12} &= 0.183, \\ c_{21} &= -0.851, \quad & C_{22} &= 0.247. \end{aligned} \tag{6.4}$$

By evaluating this plane at $ds = 0$, $da = 0$, the extrapolated continuum value of average irradiance is found. The generalized Richardson extrapolation plane

is shifted down by the continuum value to construct a model to predict discretization error as a function of grid resolution. The intersections of this model plane with the $ds = 0$ and $da = 0$ planes are the isolated angular and spatial components of discretization error, respectively. These isolated components of discretization error are shown in Figure 6.5 for both algorithms. This figure allows for quick estimation of the grid size necessary to achieve a given error threshold. The total discretization error for a potential grid can be found by taking the sum of the component errors for the algorithm in question.

For example, take the asymptotics algorithm with the grid $n_s = 60$, $n_a = 20$. The asymptotics curves are orange, with the dashed line showing the angular component error and the solid line showing the spatial component. The angular discretization error for this particular grid looks to be about $2 \times 10^{-1}\,\mathrm{W\,m^{-2}}$, while the spatial discretization error is about $3 \times 10^{-2}\,\mathrm{W\,m^{-2}}$. In this case, the spatial grid is unnecessarily fine, since the angular discretization error dominates by almost an order of magnitude. In general, the components of grid size should be chosen to produce roughly similar discretization errors so as to avoid unnecessary computation.

From the other perspective, consider an overall discretization error criterion of $\bar{\varepsilon} = 2 \times 10^{-2}\,\mathrm{W\,m^{-2}}$. This can be achieved by any grid whose spatial and angular error components sum to less than $\bar{\varepsilon}$, but for simplicity, set both component thresholds to $\bar{\varepsilon}_s = \bar{\varepsilon}_a = 1 \times 10^{-2}\,\mathrm{W\,m^{-2}}$. By visual inspection of Figure 6.5, this seems to suggest $n_s = 92$, $n_a = 22$ for the asymptotics algorithm, or $n_s = 42$, $n_a = 20$ for the finite difference algorithm.

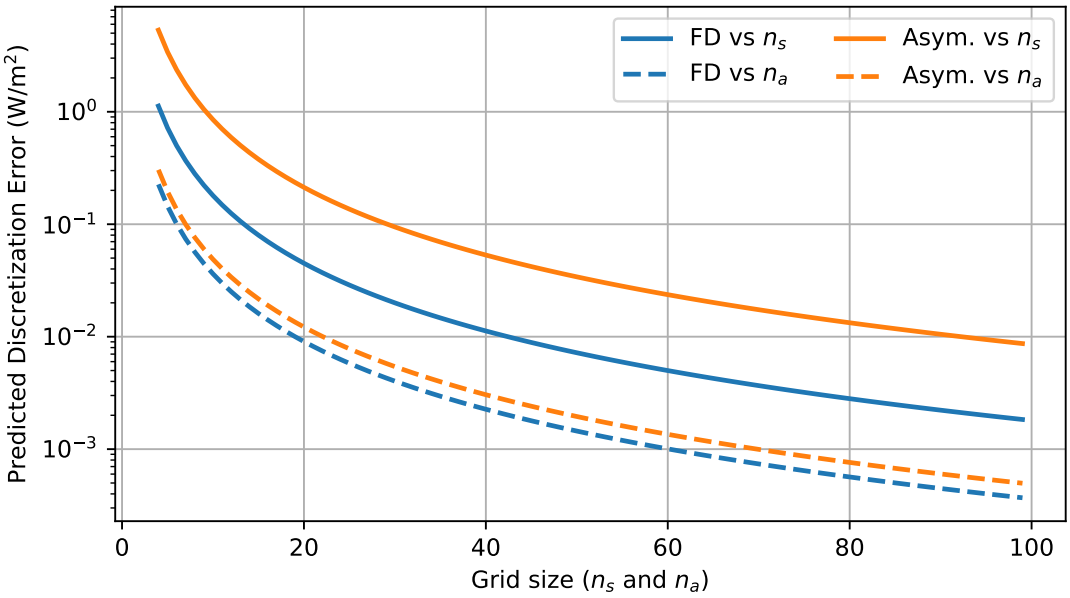

Figure 6.5: Predictions for isolated spatial and angular components of discretization error as a function of grid size for both algorithms. This figure can be used to estimate the grid sizes that each algorithm would require to meet a given error criterion. The total discretization error is the sum of the spatial and angular parts.

It is important to note that the error analysis in this section has been performed for only a single set of optical properties (HAOCE11). Optical properties are sure to have some effect on grid convergence, as they affect the derivatives of the absorption field the solution, thereby impacting the accuracy of quadrature on a particular grid. Furthermore, only $n = 0$ was tested for the asymptotics algorithm. Discretization may compound for higher-order approximations. Also, keep in mind that n_a (n_ϕ in particular) must be an even number, as mentioned in Section 4.1.

6.4 Optical Conditions for Asymptotics

Since the asymptotic approximation is based on a Taylor series expansion around the case of no scattering, it is only valid for relatively small scattering coefficients. Extremely high-scattering scenarios are out of reach for the asymptotic approximation, and the finite difference approach must be used in those cases. Whereas in low-scattering waters, adding terms to the asymptotic series tends to decrease the error in the solution, in high-scattering situations, adding terms causes the error to diverge. Therefore, if the finite difference solution is out of the question for cases of high scattering, the leading order approximation is the best option. An in-depth analysis of the error incurred by the numerical asymptotics algorithm is presented in this section.

6.4.1 Raw Simulation Results

In all of the following results, the properties of the kelp are held constant while the optical properties of the water are varied. Ten values of a_w are taken at equal intervals on a linear scale from $0.1\,\mathrm{m}^{-1}$ to $0.5\,\mathrm{m}^{-1}$ (inclusive), and ten values of b are taken at even intervals on a log scale from $0.01\,\mathrm{m}^{-1}$ to $1.5\,\mathrm{m}^{-1}$ (inclusive). For each set of optical properties, numerical asymptotics is employed with 0, 1, 2, and 3 scattering events, and a finite difference calculation is performed. A spatial-angular grid size of 72×10 is used for all calculations. Each asymptotics calculation is compared to the finite difference calculation with the same optical properties, and a pointwise error is calculated. Figure 6.6 shows this type of comparison for a single value of a_w,

varying b. As discussed in Section 5.3.4, the order n approximation converges with roughly order $n+1$. However, the figure shows that this trend does not continue indefinitely as b decreases. This is because although truncation error decreases as $b \to 0$, discretization error remains constant. Because of this, no results with this grid size show errors below about $0.07\,\mathrm{W\,m^{-2}}$. Simulations with errors below this threshold are therefore discarded in order to isolate trends in the truncation error.

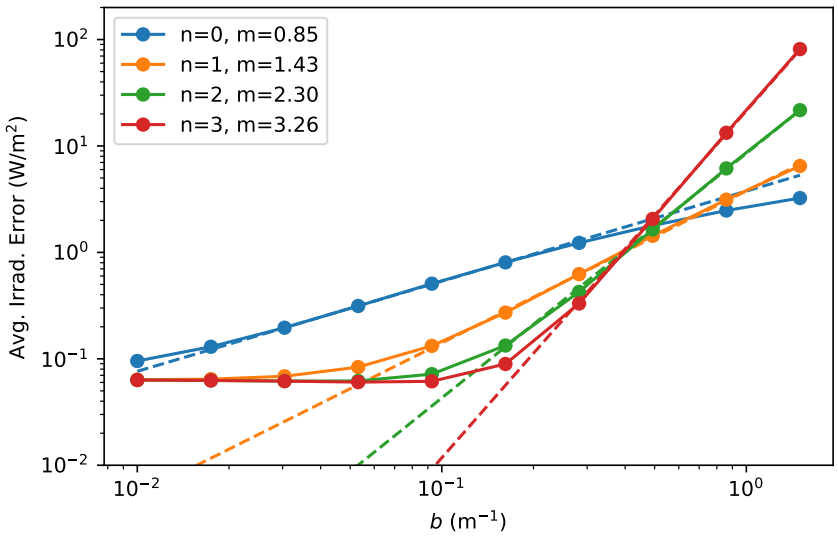

Figure 6.6: Average pointwise difference in irradiance between finite difference and asymptotics solutions for several values of b and n with constant $a = 0.1$ in a realistic kelp scenario using a 72×10 grid. Proper convergence of truncation error is observed between $b = 0.1$ and $b = 0.6$. Below $b = 0.1$, discretization error dominates. Above $b = 0.6$, the asymptotic series diverges.

Figure 6.7 shows a different slice of the simulation results. Here, $n = 0$ is held constant while a_w and b are plotted on the x and (log) y axes, with average

error plotted on the (log) color scale. Note that mean error does not depend only on b. Rather, errors are largest for waters with low absorption and high scattering, and lowest for low scattering, high absorption. The clear pattern in the variation of error over the a_w-b domain indicates that a third parameter can be calculated from a_w and b, which is sufficient to determine the accuracy of the approximation.

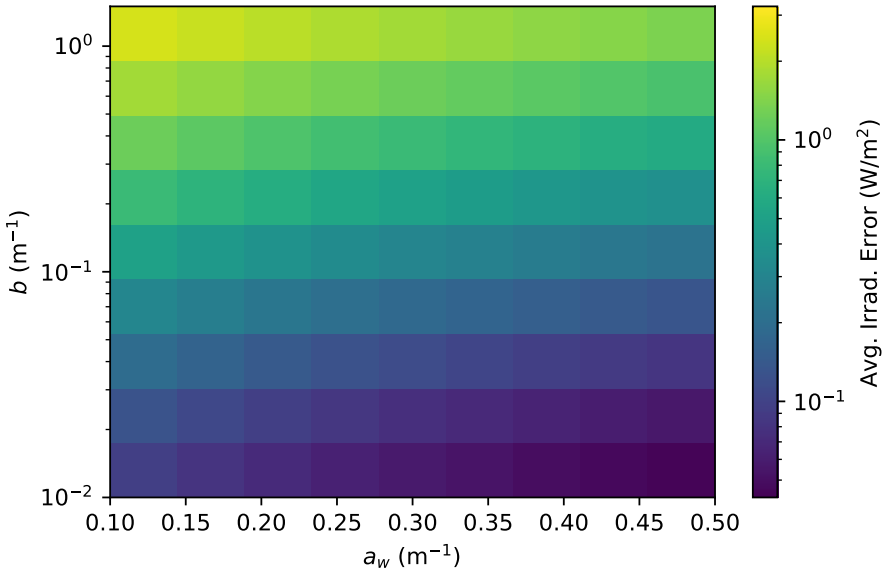

Figure 6.7: Average pointwise difference in irradiance between finite difference and asymptotics solutions for several values of b and a with constant $n = 0$ in a realistic kelp scenario using a 72×10 grid. Note that truncation error is smallest for low-scattering, high-absorption cases, and largest for high-scattering, low absorption.

Figure 6.8 shows a similar view of the simulation results as Figure 6.7, except that now, the different approximation orders are considered for each set of optical properties, and the order with the lowest error is found. The order of the best

approximation is shown for each case, and the error incurred by that approximation is shown on the color axis. In the upper left corner, the most difficult cases are shown. In this region, the leading order approximation is most accurate, showing that additional terms cause the solution to diverge. After a brief transition region, the three-term approximation is most accurate, in agreement with Figure 6.6.

Note that for low scattering, high absorption waters where the error is lowest, the $n = 2$ approximation seems to perform better than $n = 3$. As mentioned previously, this is because a lower bound on the total error is imposed by the discretization error, as suggested by Figure 6.6. In fact, the the two approximations are nearly identical; the former is only slightly better, and no conclusions about truncation error should be drawn from this part of the figure.

In actual usage, one is concerned not with finding the best approximation, but rather the cheapest one which meets some error threshold. Figure 6.9 shows the smallest value of n observed to meet discretization error targets of at most $\bar{\varepsilon} = 1.0\,\mathrm{W\,m^{-2}}$ and $\bar{\varepsilon} = 0.1\,\mathrm{W\,m^{-2}}$ in the left and right columns respectively, with the actual average error incurred shown on the color axis. In both cases, there are some optical properties for which the error threshold cannot be met for any n. This is represented by an "X" in the figure, and the error of the leading order approximation is shown in color. Note that as $\bar{\varepsilon}$ is decreased, the error threshold cannot be satisfied for a larger set of optical properties, and where it is achievable, more scattering terms are required.

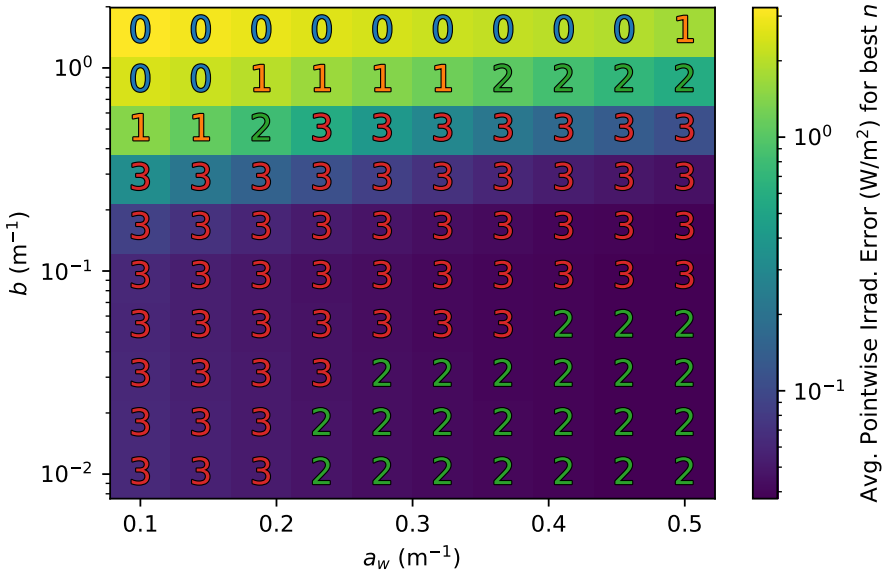

Figure 6.8: The best asymptotics solution for each (a_w, b). The value of n used is written in each cell. For high-scattering cases, the $n > 0$ terms diverge, so $n = 0$ is the best approximation. For most cases, $n = 3$ is the best. For very low-scattering cases, discretization error masks the truncation error trend.

6.4.2 Truncation Error Model

Figure 6.10 concisely represents the simulation results for all optical properties and approximation orders, with error ε plotted on the vertical axis and b plotted on the color axis. In the left column, a_w is plotted on the horizontal axis. On a log-log-log scale, the simplicity of the relationship among these three quantities is striking. Holding b constant, a log-log linear relationship is apparent between the error ε and the absorption coefficient a_w. Meanwhile, holding a_w constant, log-uniform increases

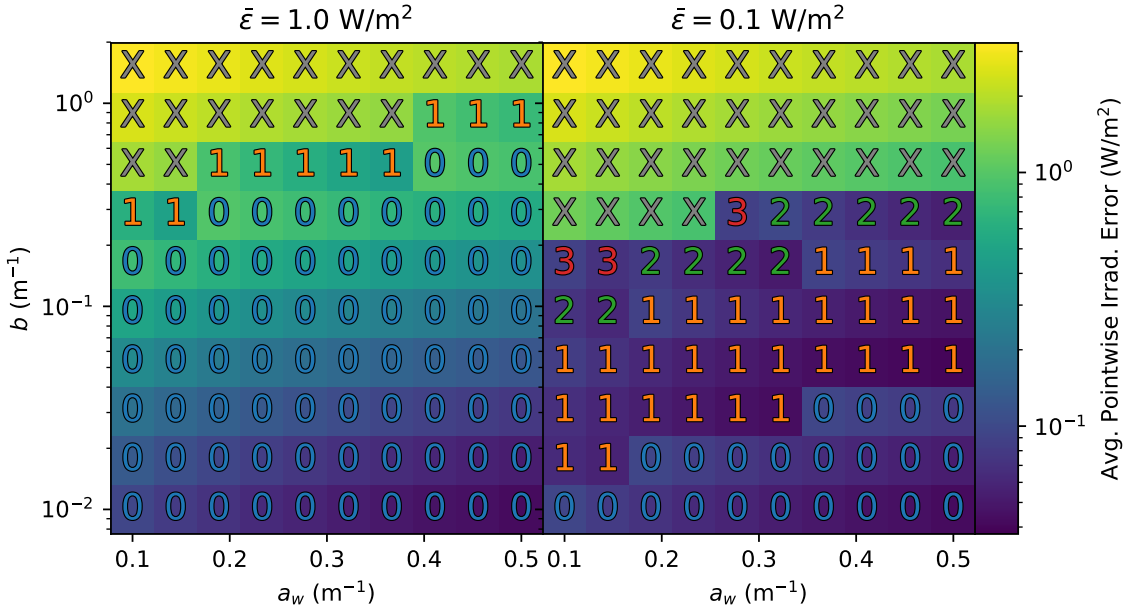

Figure 6.9: For each (a_w, b), the smallest value of n which satisfies the error criterion shown above is printed in the cell. "X" denotes an optical situation in which the criterion cannot be met since adding further terms makes the solution less accurate, not more. As seen on the left, the $n = 0$ solution usually suffices for moderate error criteria.

in b cause log-uniform increases in ε. That is, $\ln \varepsilon$ seems to increase linearly with $\ln b$ and decrease linearly with $\ln a$. Restated, it seems that $\varepsilon \propto b^{d_1}/a_w^{d_2}$.

Then, by fitting a plane to the observed values in $(\ln a_w, \ln b, \ln \varepsilon)$ space, a continuous model can be derived for $\varepsilon(a_w, b)$. Performing this procedure separately for each n value yields values of d_1 and d_2 which appear to be independent of n. Remarkably, the fit for each order of approximation yields $d_1 = 3/4$, $d_2 = 1/2$ to within a few percent variation. The black "x"s in Figure 6.10 are the predictions of

this model, setting $d_1 = 3/4$ and $d_2 = 1/2$ explicitly. The accuracy of the prediction suggests the definition of a third parameter, as mentioned in Section 6.4.1,

$$\xi = \frac{b^{3/4}}{a_w^{1/2}}, \tag{6.5}$$

with the unusual units $\mathrm{m}^{-1/4}$. In the right column of Figure 6.10, ε is plotted as a function of ξ. Notice that all of the results collapse onto a single line. Thus, for the sake of understanding the usefulness of the asymptotic series in approximating the true light field, ξ can be used as a single variable to characterize all waters.

In Figure 6.11, ε is plotted as a function of ξ for all combinations of a_w, b, and n, effectively combining the right column of Figure 6.10 on a single plot. For each n, the log-log plot displays a clear linear relationship, similar to the pattern seen previously in Figures 6.6 and 5.2. However, this figure abstracts that pattern over both a_w and b, whereas previously it was seem only for b. Note that the convergence curves for all lines appear to roughly intersect at a characteristic point. That point, denoted (ξ^*, ε^*), represents a bifurcation in the convergence behavior of the numerical asymptotics algorithm. For waters with $\xi < \xi^*$, accuracy improves with additional terms, whereas for with $\xi > \xi^*$, additional terms decrease accuracy.

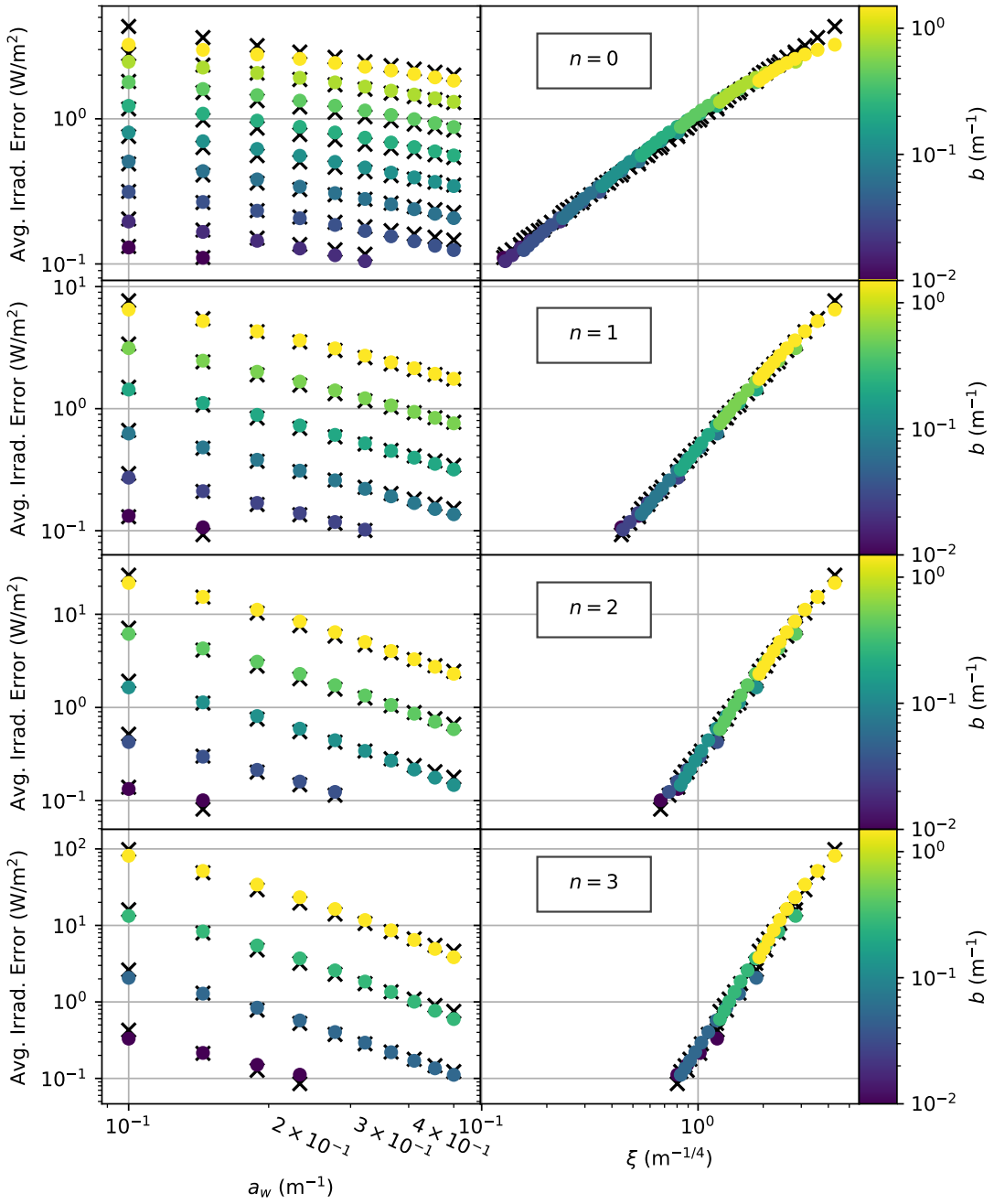

Figure 6.10: Truncation error versus a_w and ξ for each n. Simulations dominated by discretization error have been discarded. The right column shows that ξ sufficiently characterizes an optical situation for the sake of predicting truncation error of the asymptotic approximation. Errors predicted by Equation (6.7) are marked with "x"s.

In order to systematically determine (ξ^*, ε^*), the log-log convergence curve of order $n+1$ is assumed to have slope $n+1$, which is approximately true, at least for $\xi < \xi^*$. Further, it is assumed that there is a single point where all of these curves intersect. This point fully specifies the convergence curves, all other free parameters having been eliminated by the previous assumptions. Thus, the error for the order n approximation is assumed to satisfy

$$\ln\left(\varepsilon_n(\xi)\right) - \ln\varepsilon^* = (n+1)\left(\ln\xi - \ln\xi^*\right), \tag{6.6}$$

or equivalently,

$$\varepsilon_n(\xi) = \varepsilon^* \left(\frac{\xi}{\xi^*}\right)^{n+1}. \tag{6.7}$$

Then, a residual function is constructed which accumulates squared differences between the fit functions and their corresponding data points, weighting all squared errors by $1/\varepsilon$ in order to deem the simulations with lower errors more important since errors diverge for large ξ values and do not fit the linear function quite as well. A numerical optimization algorithm is then used to search the two dimensional parameter space for the minimizers of the residual, which produces the point (ξ^*, ε^*). This procedure yields the result

$$\xi^* = 1.58\,\mathrm{m}^{-1/4}, \tag{6.8}$$

$$\varepsilon^* = 1.30\,\mathrm{W\,m}^{-2}, \tag{6.9}$$

as shown in Figure 6.11. Thus, waters characterized below this threshold are worth considering for solution via numerical asymptotics, whereas others are better suited for a finite difference solution.

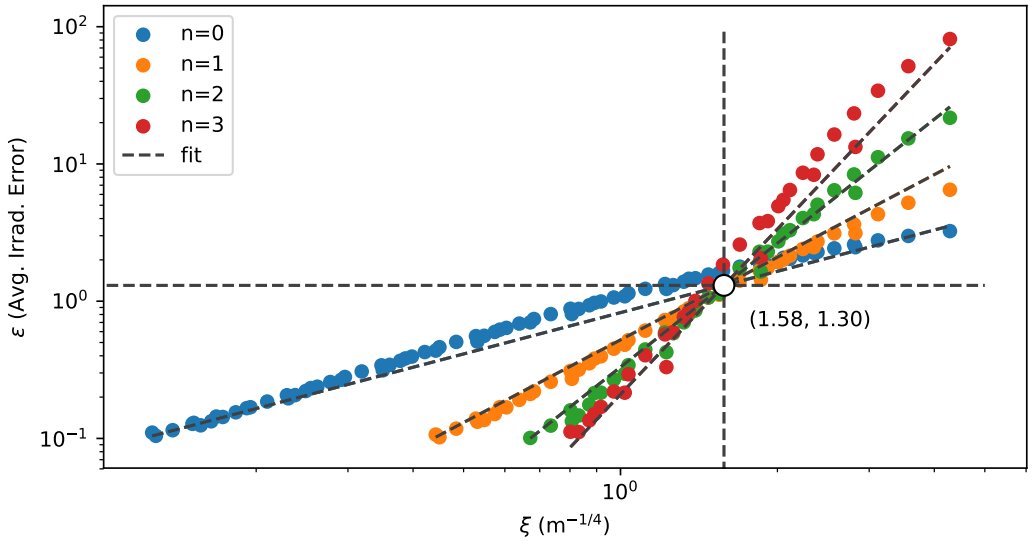

Figure 6.11: All data from the right column of 6.10 on a single plot demonstrates a characteristic $\xi = \xi^*$ above which the asymptotic series approach is inappropriate. This observation permits a simple model for predicting truncation error.

The determination of ξ^* and ε^* marks the construction of a simple analytical model given by Equation (6.7) which predicts the errors for any set of aquatic optical properties. Figure 6.12 shows the predictions of this model for several values of a_w, b, and n, with ε plotted on the color axis. Note that ξ^* is a contour in (a_w, b) space, and is marked on the figure with a dashed white line. $\xi < \xi^*$ is the region to the lower right of the threshold, and $\xi > \xi^*$ is the region to the upper left. Also note that this model and its predictions deal only with truncation error, and that discretization error distorts the actual solution accuracy, as seen in Figures 6.6 and 6.8.

This model can now be used to produce general guidelines for the choice of n given a desired error. Assuming that a maximum error $\varepsilon = \bar{\varepsilon}$ is permissible, the minimum order of n which produces ε is desired. This order, denoted $\bar{n}$, can determined by solving Equation (6.6) for n when $\varepsilon = \bar{\varepsilon}$ and rounding up to the nearest integer. That is,

$$\bar{n} = \text{ceil}\left(\frac{\ln\bar{\varepsilon} - \ln\varepsilon^*}{\ln\xi - \ln\xi^*} - 1\right),$$

which is equivalent almost everywhere to

$$\bar{n} = \text{floor}\left(\frac{\ln\bar{\varepsilon} - \ln\varepsilon^*}{\ln\xi - \ln\xi^*}\right). \tag{6.10}$$

Figure 6.13 shows $\bar{n}$ up to $\bar{n} = 3$ for $\bar{\varepsilon} = 0.1$, $\bar{\varepsilon} = 0.05$, and $\bar{\varepsilon} = 0.01$. The ξ^* contour is plotted. Assuming that $\bar{\varepsilon} < \varepsilon^*$, if higher contours of $\bar{n}$ were plotted, they would approach the ξ^* contour as $\bar{n} \to \infty$ since no accuracy better than ε^* is achievable by any order approximation when $\xi > \xi^*$.

To summarize this analysis, once an error threshold $\bar{\varepsilon}$ is chosen, the optical properties a_w and b determine the optimal numerical approach according to Figure 6.13 and Equation (6.10). If $\xi^*(a_w, b) > \xi^*$, then the finite difference algorithm should be used if possible. If memory or computation time requires the numerical asymptotics algorithm to be used, then the $n = 0$ approximation should be used, effectively ignoring scattering in order to avoid a diverging solution.

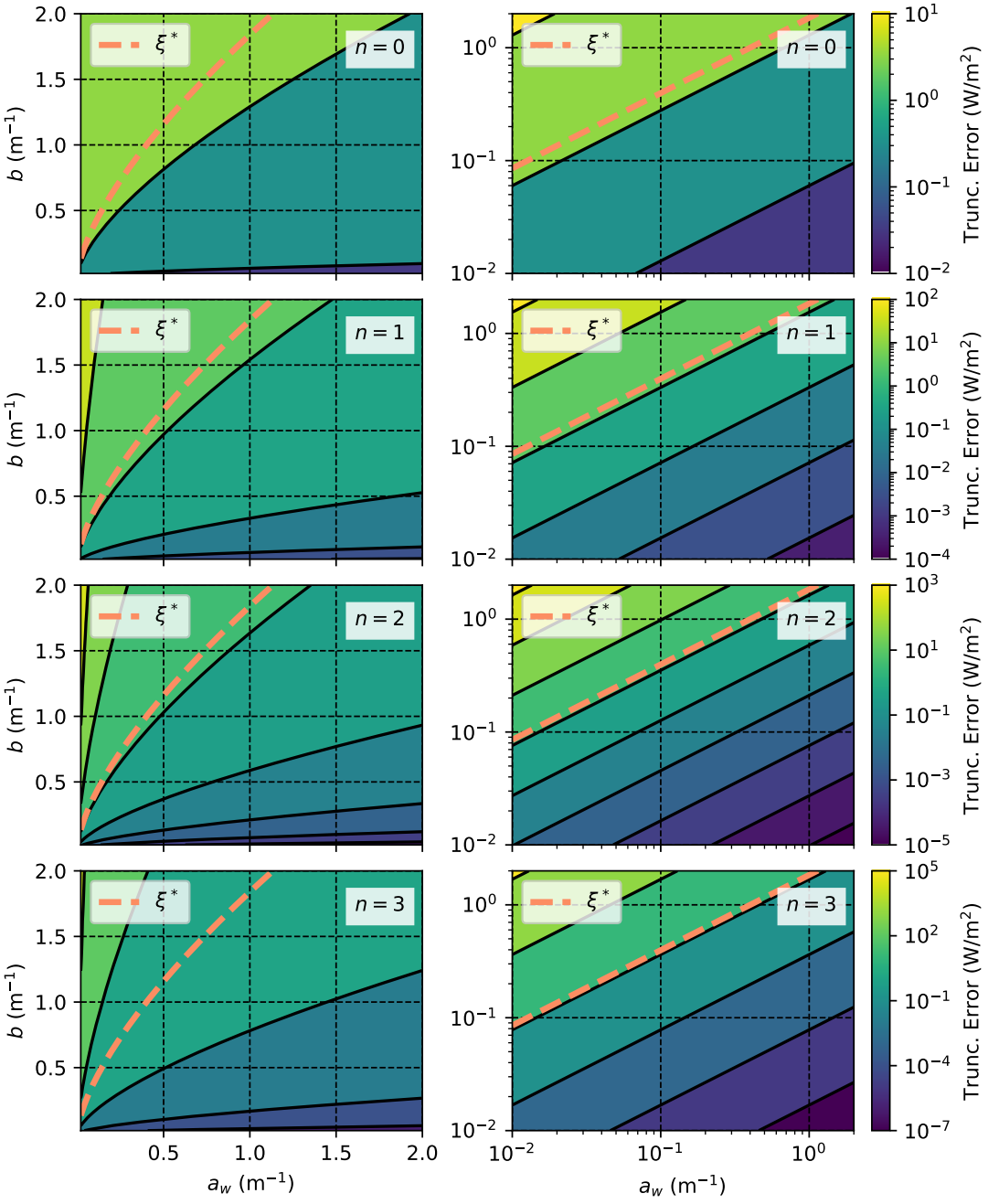

Figure 6.12: Predicted truncation error for a range of a_w and b values. n varies over plot rows. Linear-linear scale on the left, log-log scale on the right. The asymptotic series is appropriate only for (a_w, b) values to the right of the ξ^* contour.

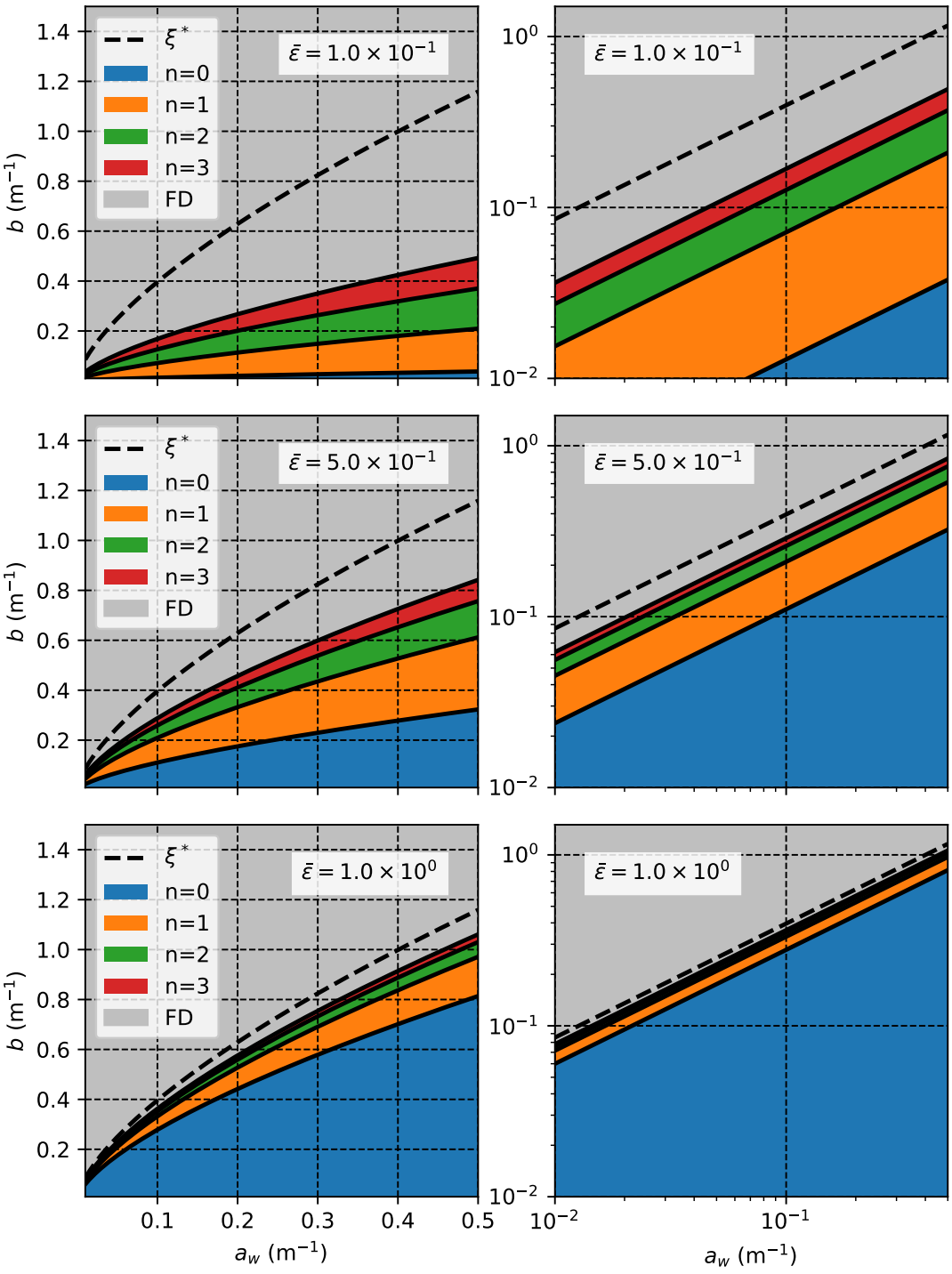

Figure 6.13: Recommended $n = \bar{n}$ value as a function of (a_w, b) to achieve the truncation error criteria $\bar{\varepsilon} = 0.1$, 0.5, and 1.0 W/m^2 for the three plot rows.

If $\bar{n} > 3$ (i.e., the gray region between ξ^* and $\bar{n} = 3$), it is theoretically possible to achieve the desired accuracy in this region by continuing to add terms, however, this is not recommended since the closer ξ is to ξ^*, the more likely it is that higher order terms will cause the solution to diverge due to some unexpected deviation in the actual performance of the algorithm from the theoretical error model presented here. As a compromise, $n = 2$ or $n = 3$ could be used in this region to balance the severity of divergence in the case of failure with the improved accuracy of the solution in the case of successful convergence. Finally, in any of the easier regions defined by $\bar{n} \leq 3$, the optimal trade-off between accuracy and computation time is achieved by using $n = \bar{n}(a_w, b)$.

6.5 Comparison to Other Light Models

Now that recommendations have been given for the choice of solution method, with the computational cost and accuracy of each having been discussed, all that remains is to compare the solutions obtained from the model presented in this thesis to other available models for the light field. Two simple models are used; in both cases, the number of calculations can be counted on two hands, whereas the model of this thesis involves many millions of calculations. Whether the insight gained from the more complex model is worth the computational expense depends on the purpose of the calculation, and is a decision for the reader to make according to their best judgment for the situation.

6.5.1 Simpler Models

The first model is the simplest conceivable light model—exponential attenuation with a constant absorption coefficient $a = a_w$. That is, the simplest solution is to ignore the kelp entirely. Of course, a depth-dependent absorption coefficient is likely to be used in a real simulation, but for the sake of the comparison, a constant a_w is maintained. The irradiance in this case is simply

$$I(z) = \exp\left(-a_w z\right). \tag{6.11}$$

The second light model, presented in [8], accounts for the kelp by adding a depth-dependent term to the absorption coefficient related to the area and spatial density of the seaweed. The model is described by

$$\frac{dI}{dz} = -\left(a_w + k_{\text{kelp}}(z)\right), \tag{6.12}$$

$$k_{\text{kelp}}(z) = -\ln(1 - A_k(1 - (1 - AD)^{\rho_f(z)})), \tag{6.13}$$

where $A_k = 0.7$ is the kelp absorptance, $D = 0.01$ ropes/m^2 is the number of vertical kelp ropes per ocean surface area, $A(z)$ is the average area of the kelp fronds per meter vertical rope, and $\rho_f = 120$ fronds/m is the number of kelp fronds per vertical meter of rope. Then,

$$D = \frac{1}{(x_{\max} - x_{\min})\,(y_{\max} - y_{\min})}, \tag{6.14}$$

and the mean frond A is calculated from the mean frond length according to Equation (2.6) as

$$A = \frac{{\mu_l}^2}{2f_r}. \tag{6.15}$$

Once $k_{\text{kelp}}(z)$ is calculated according to Equation (6.13), then the solution of Equation (6.12) is a simplified version of Equation (3.12) with $\tilde{a}(z) = a_w + k_{\text{kelp}}(z)$ and $\tilde{\sigma}_0 = 0$. The solution is therefore the analogous simplification of Equation (3.13), namely

$$I(z) = I_0 \exp\left(-a_w z - \int_0^z k_{\text{kelp}}(z')\, dz'\right). \tag{6.16}$$

6.5.2 Description of Comparison

In Figure 6.15, the light model of this thesis is compared to simpler models under the conditions described in Section 6.1.4 with the optical properties of coastal California (HAOCE11 from Table 6.2). In the figure, the first model described in this chapter, which ignores the kelp entirely, is labeled "No Kelp." The second model, which uses an additional term in the absorption coefficient to describe the kelp, is labeled "Simple Kelp." From the 3D model of this thesis, the average irradiance and perceived irradiance calculated from a finite difference solution on a 72×10 grid are plotted. As described in Section 3.1.2, the perceived irradiance is a weighted average of the irradiance over the kelp distribution, whereas the average irradiance is weighted evenly over the entire horizontal domain.

Both quantities are plotted with error bands calculated via Richardson extrapolation. The calculation was performed for all combinations of $n_s = 32, 48, 64, 72$ and $n_a = 6, 8, 10$, and the average and perceived irradiances calculated from those solutions were cubically interpolated at many z values. Then, the standard 2D Richardson extrapolation procedure was applied independently for each z, using the

interpolated irradiance value at that z as the scalar to extrapolate, as described in Section 5.4.2. The plotted curves are the irradiances from the 72×10 grid. The error bands are three times the estimated error in order to make the estimate conservative, as recommended by [33]. For a detailed procedure for error reporting in numerical simulations, see [1].

6.5.3 Comparison: Irradiance Curves

First, note that the no-kelp model predicts the highest light levels throughout the domain, as should be expected since it ignores the dominant source of attenuation. The simple kelp model agrees with the no-kelp model at the surface, but then reduces quickly once the kelp, which has a maximum frond length at $z = 2$, becomes sufficiently dense. At the bottom of the domain where there is little kelp remaining, the two curves are seen to be parallel in the log-linear scale, showing that the absorption coefficients are roughly equal, as at the surface. The average irradiance 3D model appears to agree quite well with the simple kelp model, especially near the surface and bottom, where both models are approximately reduced to the first model. In the $z = 2$ region where the most kelp is present, though, the 3D model predicts higher absorption than the simpler 1D model. However, between $z = 3$ and $z = 4$, the light in the 3D kelp model attenuates more slowly than the 1D model, perhaps because light is able to penetrate the water in regions where the kelp is not present, and the multitude of angles in which the light travels allow some of the rays to reach even the lower depths without being absorbed by the kelp.

The perceived irradiance is clearly much lower than the average irradiance and the other models. This makes sense, as the light is dimmest in the regions of highest kelp density, and the perceived irradiance weights these regions the highest, practically ignoring the edges of the domain where light is absorbed only by the water; these regions skew the average irradiance upward, and are not actually representative of the light field where photosynthesis is occurring. From this point of view, it seems that the other two light models significantly overestimate the amount of light available where it actually of interest.

Note that the 3D model does not predict the lowest light fields everywhere; the average irradiance is observed to be larger than the no-kelp irradiance in the second plot in Figure 6.15. This is due to the effect of scattering shifting light from lower in the domain closer to the surface. This tendency for scattering to insulate the upper region of the water may prove to be important in understanding how photosynthesis near the ocean's surface differs from photosynthesis at depth.

6.5.4 Comparison: Total Radiant Flux

Figure 6.14 shows the total radiant flux in Watts predicted by each model, calculated by integrating each depth-dependent light field over the vertical domain, *weighted by the frond area density* (total frond area per vertical meter). While not exactly analogous, total radiant flux can be regarded as a rough proxy for the amount of photosynthesis occurring in the entire kelp population at all depths. Reducing the entire

light field to an interpretable scalar is quite useful, as it summarizes the magnitude of the difference between the four light models.

The radiant flux predicted by the 3D average irradiance is just 13% of the flux disregarding kelp, and 57% of the simple kelp flux. Meanwhile, the perceived irradiance predicts only 3% the flux of the no-kelp model and 11% the flux of the simple kelp model. Even among the 3D models, using the perceived irradiance predicts 19% the flux of the average irradiance. Calculating the perceived irradiance is possible only because the 3D spatial kelp distribution is known, which is a unique feature of this model. The large degree of these differences in model predictions highlight the importance of considering this spatial distribution.

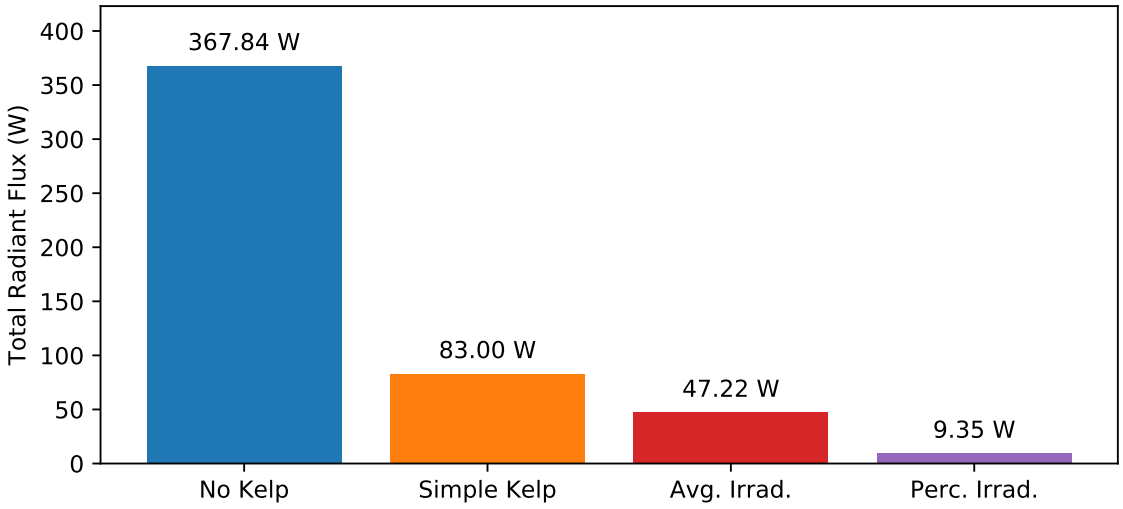

Figure 6.14: Total radiant flux in Watts through kelp predicted by each model. The magnitude of the differences highlight the importance of considering the three-dimensional spatial kelp distribution.

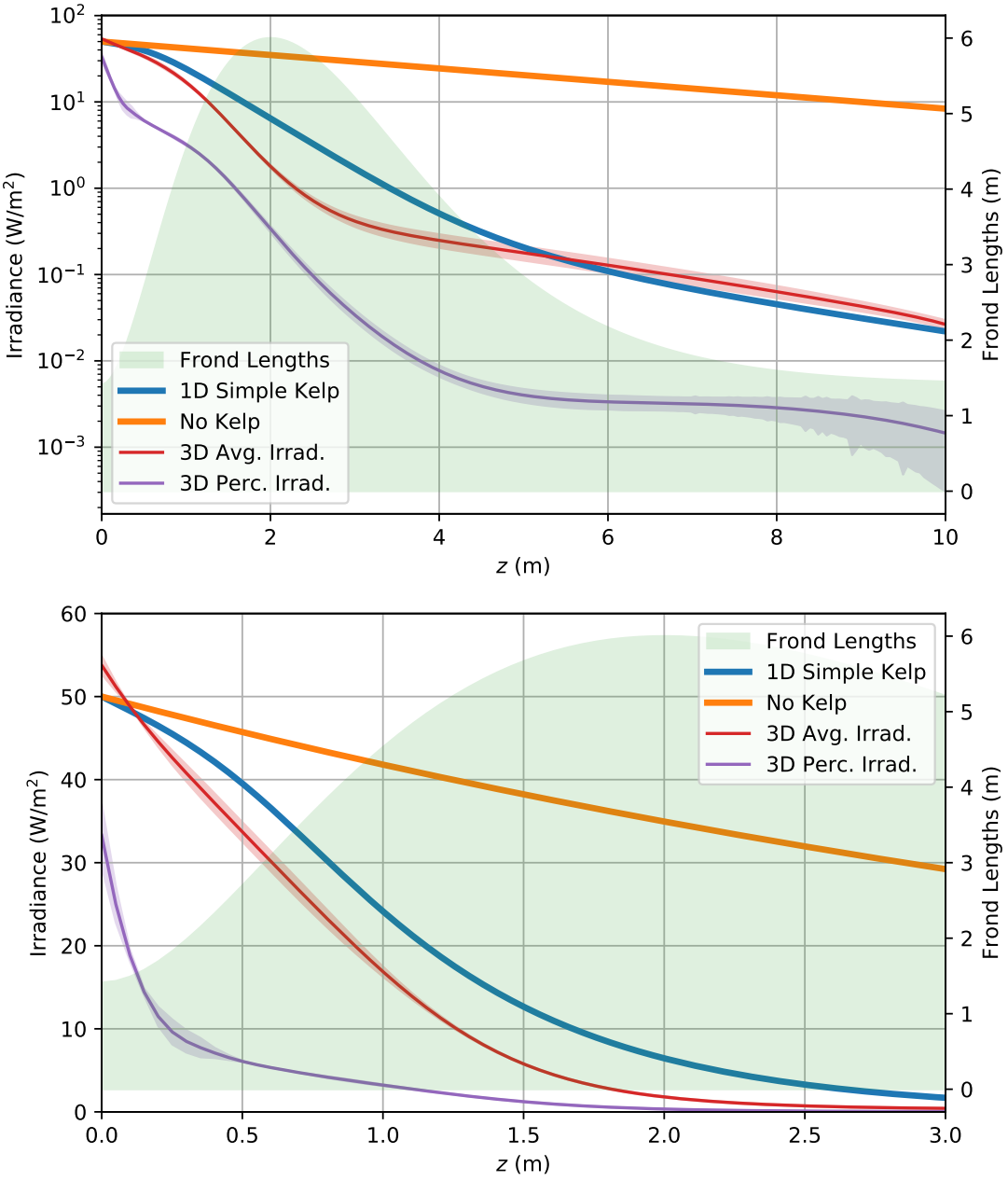

Figure 6.15: Average irradiance and perceived irradiance from finite difference compared to simpler light models for the case of coastal California water (HAOCE11) with realistic kelp growth. Frond lengths over depth are shown on the right-hand axis. Estimated discretization error is shown by error bands for perceived irradiance and average irradiance. Note the different scales in the two plots.

CHAPTER VII

CONCLUSION

This thesis presents a model for the light field in a vertical-rope kelp farming operation, emphasizing the effect of the seaweed itself on the overall light field. A three-dimensional model for the spatial distribution of the seaweed is developed, which then informs the absorption field of the combined seaweed-water medium. This absorption field is then used as a coefficient in the monochromatic radiative transfer equation, which calculates the full light field over all positions and angles in the domain, accounting for both attenuation and scattering in the medium.

Two numerical solution techniques are presented: finite difference and numerical asymptotics. The finite difference approach is applicable to any type of water, although it can be prohibitively expensive in terms of CPU time, and even more so in terms of memory usage. On the other hand, the numerical asymptotics algorithm is much cheaper computationally by both measures, though it is only accurate for low-scattering scenarios. Within the appropriate range of optical properties of the aquatic medium, the accuracy and computation time of the solution can be tuned by choosing the number of terms to include in the asymptotic series.

When compared to simpler one-dimensional models for the light field, the model presented in this thesis is found to predict lower light levels. This is expected,

as the full three-dimensional model considers self-shading due to the kelp in greater detail than the others. Further, the average irradiance as a function of depth predicted by the three-dimensional model is found to agree fairly well with a simpler model for kelp shading. However, the average irradiance considers areas of low shading far from the kelp which are irrelevant to photosynthesis. When the light field is examined only in the regions where kelp is actually growing, much less light is predicted. This indicates that simpler models may be overestimating the amount of light available for photosynthesis, which would, in light-limited situations involving high kelp density or low nutrient concentration, predict unrealistically large overall biomass yields in a time-dependent kelp growth simulation.

7.1 Model Summary

The following is a summary of the primary assumptions used in the formulation of the model. The assumptions are vast simplifications from the real system, and leave plenty of room for future improvement.

- All fronds in the population are congruent kites of equal, uniform thickness.
- Fronds are perfectly flat and horizontal.
- Fronds emanate from an infinitely thin, perfectly vertical rope, with no stipe.
- Population frond lengths are normally distributed in each depth layer.
- Fronds are oriented according to a von Mises distribution whose sharpness is proportional to current velocity and independent of frond length.

- The absorption coefficient of the aquatic medium depends only on depth.
- All fronds in the population have the same absorptance.
- The scattering coefficient is constant and equal for both kelp and water.
- Only volumetric optical effects, not surface effects, are considered.
- Frequency dependence is neglected.

7.2 Future Work

As with any scientific or mathematical investigation, many new opportunities for exploration have arisen from the development of this model, including improvements to the mathematical model itself, improvements to its numerical solution, and its application to answer real-world questions about kelp cultivation. Certainly, the exploration of any of the ideas presented here will lead to many more opportunities for future inquiry and so on *ad infinitum*, as is the nature of knowledge creation.

7.2.1 Model Improvements

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. The implementation of long-lines may prove to be closely related to allowing non-horizontal frond orientation. 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 [28].

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, the assumption of independent distributions would have to be abandoned, as mentioned at the end of Section 2.3.3. Considering non-kite frond shapes, as well as out-of-plane bending may also improve the spatial description of the seaweed, though would pose major implementation challenges.

Improved parameter estimation is also important, especially the frond alignment coefficient η. Better values for the frond absorptance A_k and thickness f_t would also be advantageous. Further, it may be worthwhile to consider how the A_k and f_t vary within a single frond, over depth, and over the life cycle of the kelp plant.

Additionally, the light model itself has a few opportunities for improvement. At present, the scattering coefficient is taken to be constant over the whole domain. In reality, kelp and water have different scattering properties, which should be considered in the same way that the absorption field is presently determined by Equation (2.20). This poses an additional challenge for the numerical asymptotics solution, as the asymptotic expansion must be taken in terms of a scalar parameter. Perhaps the maximum value $\bar{b}$ of the scattering field $b(\boldsymbol{x}, \bar{b})$ could be used, which of course

would require that b be expanded in terms of $\bar{b}$ as is currently done with L and σ in Equations (3.3.1) and (3.3.1) respectively.

Also, it is important to note that in reality, photosynthesis is a frequency-dependent process, and a frequency-dependent light model may therefore significantly improve seasonal growth predictions. This suggests that inelastic (a.k.a. Raman) scattering also be considered, whereby light changes not only direction but also frequency during a scattering event. Of course, frequency-dependence adds another dimension to the already five-dimensional simulation, which may push a numerical solution beyond the capability of modern computers.

Finally, a major outstanding task is the experimental validation of the model presented in this thesis. Chapter V deals with ensuring that the numerical algorithms presented here *solve the equations right*, but it remains to be shown whether they *solve the right equations*. Comparison with experimental data is an essential step in developing and refining computation models, and is sure to provide insight into potential improvements.

7.2.2 Numerical Improvements

Aside from improvements of the mathematical model, plenty of improvements to the numerical solution are possible. Perhaps the most important and achievable such improvement the following specific modification to the numerical asymptotics algorithm which is likely to speed up the solution by several orders of magnitude. At present, the numerical asymptotics algorithm considers each spatial-angular grid

point, determines the ray path back to its intersection with the domain boundary, and solves Equation (3.14) via Equation (4.7) over the full ray path in order to determine the effects of absorption and scattering since the previous term in the asymptotic series. Rather than integrate the full ray for each grid point, it may be possible to implement a much cheaper local algorithm which solves Equation (3.14) only between adjacent grid cells. At present, computation time is a far greater bottleneck than memory usage for the asymptotic approximation. If this algorithmic improvement indeed speeds up the computation considerably, it would make solutions feasible on much larger grids than are currently within reach.

A somewhat similar modification is available for the finite difference solution which would make larger grids achievable. However, this does not actually involve an algorithmic improvement, nor does it involve reducing the total CPU time or memory allocation required for computation. Rather, it involves distributing the computational load between computers in a way that is not currently implemented. As mentioned in Section 4.5.4, the finite difference linear system is presently solved by the multithreaded GMRES algorithm provided by the LIS package. While multithreading can only achieve parallelism within a single process on a single machine, LIS also makes available MPI implementations of its parallel algorithms. MPI is a protocol which facilitates work-sharing between processes either on the same machine or over a network. Therefore, using the MPI environment permits the coefficient matrix to be divided and stored in a distributed fashion on several nodes within a cluster, which, depending on the computational resources available, is likely to drastically

increase the grid sizes available for computation. However, network communication is often significantly slower than inter-thread communication. Therefore, unless a high-speed fiber-optic network is used, the MPI environment is likely to require a sacrifice in terms of computation time for the sake of the increased memory capacity. None of this has yet been explored in the context of this project, and is ripe for exploration in future work.

Another potential improvement, though not likely to have as significant of an impact as the two above, is to restructure the angular grid to use a more accurate quadrature, which would reduce the necessary number of angular grid points to achieve a particular level of accuracy, and therefore reduce the required computation time and memory allocation. Among the possible options are the Lebedev grid and "spherical t-design"[2, 5, 6]. In fact, many other numerical algorithms altogether are available for solving the radiative transfer equation, including finite element methods and Monte Carlo methods among others.

7.2.3 Application to Seaweed Cultivation

The final and perhaps most interesting and useful avenue for future work is the application of the 3D light model developed in this thesis to answer real questions about seaweed cultivation. While there are many ways in which the model and its numerical solution can be improved, a working implementation is now available, and should be taken advantage of! In particular, this light model is well-suited for use in a time-dependent growth model such as [8], as discussed in Section 6.1.3. For

example, one line of questioning which naturally arises related to light modeling is "How does the placement of kelp cultivation ropes affect the potential biomass cultivation potential of a given area? What is the optimal rope spacing to maximize said production?" These type of questions are particularly natural to investigate with this light model, as the use of periodic boundary equates domain width and rope spacing.

Also, as mentioned in Chapter I, the growth of seaweed near WWTP ocean outfalls has been proposed for nutrient remediation of the aquatic ecosystem. One concern which arises from this proposal is whether the optical conditions near the WWTP are suitable for kelp cultivation. Therefore, modeling the optical properties of such water as a function of nutrient concentrations in order to run appropriate kelp growth simulations may prove valuable for addressing such concerns. Similar inquiry may be worthwhile when considering kelp cultivation in close proximity to other potential sources of turbidity such as salmon farms [7] or other forms of aquaculture.

BIBLIOGRAPHY

[1] Procedure for Estimation and Reporting of Uncertainty Due to Discretization in CFD Applications. *Journal of Fluids Engineering*, 130(7):078001–078001–4, July 2008.

[2] C. An and S. Chen. Numerical Integration over the Unit Sphere by using spherical t-design. *arXiv:1611.02785 [math]*, Nov. 2016. arXiv: 1611.02785.

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

[4] 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.

[5] J. Beckmann, H. N. Mhaskar, and J. Prestin. Local numerical integration on the sphere. *GEM-International Journal on Geomathematics*, 5(2):143–162, 2014.

[6] C. H. L. Beentjes. Quadrature on a Spherical Surface. Technical report, Mathematical Institute, University of Oxford, Oxford, UK.

[7] O. Broch, I. Ellingsen, S. Forbord, X. Wang, Z. Volent, M. Alver, A. Handå, K. Andresen, D. Slagstad, K. Reitan, Y. Olsen, and J. Skjermo. Modelling

the cultivation and bioremediation potential of the kelp Saccharina latissima in close proximity to an exposed salmon farm in Norway. *Aquaculture Environment Interactions*, 4(2):187–206, Aug. 2013.

[8] O. J. Broch and D. Slagstad. Modelling seasonal growth and composition of the kelp Saccharina latissima. *Journal of Applied Phycology*, 24(4):759–776, Aug. 2012.

[9] V. Brzeski and G. Newkirk. Integrated coastal food production systems – a review of current literature. page 17, July 1996.

[10] 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.

[11] S. Chandrasekhar. *Radiative Transfer*. Dover, 1960.

[12] 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.

[13] 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 Depths. *Journal of Phycology*, 42(6):1225–1234, Dec. 2006.

[14] 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.

[15] S. Foldal. *Morphological relations of cultivated Saccharina latissima at three stations along the Norwegian coast*. Master's thesis in Marine Coastal Development, Norwegian University of Science and Technology, Trondheim, Norway, 2018.

[16] 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.

[17] 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.

[18] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, Feb. 1970.

[19] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp Macrocystis pyrifera. *Marine Biology*, 95(4):611–624, 1987.

[20] D. G. Jones. Corn-Based Ethanol Production in the United States and the Propensity for Pesticide Use. Master's thesis, Aug. 2015.

[21] 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.

[22] J. T. O. Kirk. *Light and Photosynthesis in Aquatic Ecosystems*. Cambridge University Press, Apr. 1994. Google-Books-ID: It5GePwa2EIC.

[23] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Stuttgart, High-Performance Computing Center Stuttgart, 1993.

[24] A. Meurer, C. P. Smith, M. Paprocki, S. B. Kirpichev, and M. Rocklin. SymPy: symbolic computing in Python. page 27, 2017.

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

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

[27] A. Nishida. Experience in Developing an Open Source Scalable Software Infrastructure in Japan. In D. Taniar, O. Gervasi, B. Murgante, E. Pardede, and B. O. Apduhan, editors, *Computational Science and Its Applications – ICCSA 2010*, pages 448–462. Springer Berlin Heidelberg, 2010.

[28] C. Norvik. *Design of Artificial Seaweeds for Assessment of Hydrodynamic Properties of Seaweed Farms*. Master's thesis in Marine Coastal Development, Norwegian University of Science and Technology, Trondheim, Norway, 2017.

[29] 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.

[30] 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.

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

[32] L. F. Richardson. On the approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam. *Proc. R. Soc. Lond. A*, 83(563):335–336, Mar. 1910.

[33] P. J. Roache. Perspective: A Method for Uniform Reporting of Grid Refinement Studies. *Journal of Fluids Engineering*, 116(3):405–413, Sept. 1994.

[34] P. J. Roache. *Verification and validation in computational science and engineering*. Hermosa, Albuquerque, NM, 1998.

[35] P. J. Roache. Verification of Codes and Calculations. *AIAA Journal*, 36(5):696–702, May 1998.

[36] P. J. Roache. Code Verification by the Method of Manufactured Solutions. *Journal of Fluids Engineering*, 124(1):4–10, Mar. 2002.

[37] P. J. Roache. Building PDE codes to be verifiable and validatable. *Computing in Science Engineering*, 6(5):30–38, Sept. 2004.

[38] P. J. Roache, K. N. Ghia, and F. M. White. Editorial Policy Statement on the Control of Numerical Accuracy. *Journal of Fluids Engineering*, 108(1):2–2, Mar. 1986.

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

[40] K. Salari and P. Knupp. Code Verification by the Method of Manufactured Solutions. Technical Report SAND2000-1444, Sandia National Labs., Albuquerque, NM (US); Sandia National Labs., Livermore, CA (US), June 2000.

[41] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Ecological Modelling*, 80:161–170, Mar. 1994.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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

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

Denote the edges as

$$x^e_i = (i - 1)dx \text{ for } i = 1, \ldots, n_x,$$
$$y^e_j = (j - 1)dy \text{ for } j = 1, \ldots, n_y,$$
$$z^e_k = (k - 1)dz \text{ for } k = 1, \ldots, n_z.$$

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

Now, we define the azimuthal angle such that

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

For the sake of periodicity, we need

$$\theta_1 = 0,$$

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

which requires

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

For the polar angle, we similarly let

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

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

$$\phi_1 = 0,$$

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

This gives us

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

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

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

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

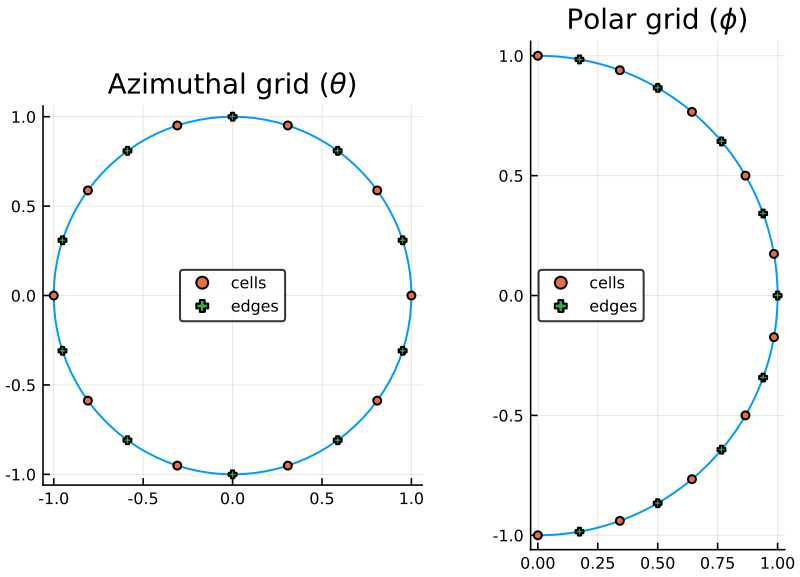

Figure A.1: Angular grid cell centers and edges.

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

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

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

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

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 $\boldsymbol{\omega}_p$ as Ω_p, and the solid angle subtended by Ω_p is denoted $|\Omega_p|$. The areas of the grid cells are calculated as follows. Note that there is a temporary abuse of notation in that the same symbols ($d\theta$ and $d\phi$) are being used for infinitesimal differential and for finite grid spacing. For the poles, we have

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

For all other angular grid cells,

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

APPENDIX B

RAY TRACING ALGORITHM

In order to evaluate a path integral through the 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 $\boldsymbol{p_1} = (p_{1x}, p_{1y}, p_{1z})$ and a corresponding path $\boldsymbol{l}(\boldsymbol{x_1}, \boldsymbol{\omega}, s)$. To find the location of discontinuities in the integrand, we first calculate the distance from its origin, $\boldsymbol{p_0} = \boldsymbol{x_0}(\boldsymbol{p_1}, \boldsymbol{\omega}) = (p_{0x}, p_{0y}, p_{0z})$ (as in (3.5)) to grid edges in each dimension separately. Given

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

simply as a boolean value to track of the ray direction.

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

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

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

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

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

Then,

$$i_0^e = i_0 + \sigma_x, \tag{B.16}$$

$$j_0^e = j_0 + \sigma_y, \tag{B.17}$$

$$k_0^e = k_0 + \sigma_z. \tag{B.18}$$

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

$$s_i^x = \hat{s}\frac{x_i^e - p_{0x}}{p_{1x} - p_{0x}}, \tag{B.19}$$

$$s_j^y = \hat{s}\frac{y_j^e - p_{0y}}{p_{1y} - p_{0y}}, \tag{B.20}$$

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

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 E.

APPENDIX C

SYNTHETIC DATA

In order to perform code verification via the Method of Manufactured solutions, analytical functions for radiance, absorption coefficient, and volume scattering function must be chosen which are simple to evaluate, are differentiable, and satisfy the constraints imposed by the algorithm implementation that are listed in Section 5.3.2. The functions chosen to meet the above conditions are

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

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

$$\beta(\Delta) = \frac{\Delta+1}{4\pi}, \tag{C.3}$$

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 [24], the boundary conditions

and source function are calculated to be

$$f(\theta,\phi) = \alpha\left(-\gamma+1\right)\left(\sin\left(\phi+\theta\right)+1\right), \tag{C.4}$$

$$\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)\left(\sin\left(\phi+\theta\right)+1\right)\\
&\cdot\left(-\gamma+z+\frac{\tanh\left(\left(b+1\right)\left(\gamma-z\right)\right)}{\tanh\left(\gamma\left(b+1\right)\right)}\right)\left(b+\sin\left(\frac{2\pi x}{\alpha}\right)+\sin\left(\frac{2\pi y}{\alpha}\right)+\tanh\left(-\gamma+z\right)+5\right)\\
&- b\Bigg[\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\left(\phi\right)\sin\left(\theta\right)}{3}+\frac{\cos\left(\phi\right)}{3}\right)\left(-\gamma+z+\frac{\tanh\left((b+1)(\gamma-z)\right)}{\tanh\left(\gamma(b+1)\right)}\right)}{4\pi}\\
&-\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\left((b+1)(\gamma-z)\right)}{\tanh\left(\gamma(b+1)\right)}\right)}{4\pi}\\
&\cdot\left(-\frac{\pi\sin\left(\phi\right)\sin\left(\theta\right)}{2}-\frac{\sin\left(\phi\right)\sin\left(\theta\right)}{3}-\frac{\cos\left(\phi\right)}{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\left((b+1)(\gamma-z)\right)}{\tanh\left(\gamma(b+1)\right)}\right)}{4\pi}\\
&\cdot\left(\frac{\sin\left(\phi\right)\sin\left(\theta\right)}{3}-\frac{2\pi\sin\left(\phi\right)\cos\left(\theta\right)}{3}+\frac{\cos\left(\phi\right)}{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\left((b+1)(\gamma-z)\right)}{\tanh\left(\gamma(b+1)\right)}\right)}{4\pi}\\
&\cdot\left(-\frac{\pi\sin\left(\phi\right)\sin\left(\theta\right)}{2}-\frac{\sin\left(\phi\right)\sin\left(\theta\right)}{3}+\frac{2\pi\sin\left(\phi\right)\cos\left(\theta\right)}{3}-\frac{\cos\left(\phi\right)}{3}+2\pi\right)\Bigg]\\
&+2\pi z\left(\sin\left(\phi+\theta\right)+1\right)\left(-\gamma+z+\frac{\tanh\left(\left(b+1\right)\left(\gamma-z\right)\right)}{\tanh\left(\gamma\left(b+1\right)\right)}\right)\sin\left(\phi\right)\sin\left(\theta\right)\cos\left(\frac{2\pi y}{\alpha}\right)\\
&+2\pi z\left(\sin\left(\phi+\theta\right)+1\right)\left(-\gamma+z+\frac{\tanh\left(\left(b+1\right)\left(\gamma-z\right)\right)}{\tanh\left(\gamma\left(b+1\right)\right)}\right)\sin\left(\phi\right)\cos\left(\theta\right)\cos\left(\frac{2\pi x}{\alpha}\right)\\
&+\Bigg[\alpha\left(z\left(\sin\left(\frac{2\pi x}{\alpha}\right)+\sin\left(\frac{2\pi y}{\alpha}\right)\right)+1\right)\\
&\cdot\left(\frac{\left(-b-1\right)\left(-\tanh^2\left(\left(b+1\right)\left(\gamma-z\right)\right)+1\right)}{\tanh\left(\gamma\left(b+1\right)\right)}+1\right)\left(\sin\left(\phi+\theta\right)+1\right)\\
&+\alpha\left(\sin\left(\frac{2\pi x}{\alpha}\right)+\sin\left(\frac{2\pi y}{\alpha}\right)\right)\left(\sin\left(\phi+\theta\right)+1\right)\\
&\cdot\left(-\gamma+z+\frac{\tanh\left(\left(b+1\right)\left(\gamma-z\right)\right)}{\tanh\left(\gamma\left(b+1\right)\right)}\right)\Bigg]\cos\left(\phi\right).
\end{aligned} \tag{C.5}$$

APPENDIX D

MEMORY USAGE

The number of elements in the finite difference coefficient matrix is given by Equation (4.14). Assuming that double precision floating point numbers are used, the number of bytes required to store the matrix values is found by multiplying the result of (4.14) by 8 bytes per number. These values are listed in Table D.1.

Since the matrix is sparse, the indices of the elements must also be stored. Due to the large number of elements involved, long integers must be used as indices, which also require 8 bytes to store. The exact amount of memory required depends on the storage scheme, which in this case is the Compressed Sparse Row (CSR) format provided by LIS [27]. Furthermore, the iterative method required to calculate a solution to the matrix equation allocates additional working arrays. From personal experience, the memory requirement for LIS solution via GMRES, restarted every 100 iterations, is well-estimated by multiplying the values in Table D.1 by 5. These solution memory estimates are listed in Table D.2.

Table D.1: Memory to store one copy of the finite difference coefficient matrix. n_s varies over rows and n_a over columns.

n_s \ n_a	8	10	12	14	16	18	20
4	1.36 MiB	3.51 MiB	7.61 MiB	14.59 MiB	25.57 MiB	41.88 MiB	65 MiB
8	10.91 MiB	28.15 MiB	60.94 MiB	116.79 MiB	204.7 MiB	335.17 MiB	520.2 MiB
16	87.4 MiB	225.34 MiB	487.76 MiB	934.67 MiB	1.6 GiB	2.62 GiB	4.06 GiB
32	699.61 MiB	1.76 GiB	3.81 GiB	7.3 GiB	12.8 GiB	20.95 GiB	32.52 GiB
48	2.31 GiB	5.94 GiB	12.87 GiB	24.65 GiB	43.2 GiB	70.73 GiB	109.76 GiB
64	5.47 GiB	14.09 GiB	30.5 GiB	58.43 GiB	102.4 GiB	167.65 GiB	260.18 GiB
72	7.78 GiB	20.06 GiB	43.42 GiB	83.2 GiB	145.8 GiB	238.7 GiB	370.45 GiB
100	20.86 GiB	53.76 GiB	116.34 GiB	222.91 GiB	390.63 GiB	639.54 GiB	992.51 GiB
128	43.74 GiB	112.74 GiB	243.99 GiB	467.48 GiB	819.22 GiB	1.31 TiB	2.03 TiB

Table D.2: Memory to solve the linear system of equations with GMRES restarted every 100 iterations. This seems to require about five times the memory required to store the matrix. In the table, n_s varies over rows, and n_a over columns.

n_s \ n_a	8	10	12	14	16	18	20
4	6.81 MiB	17.57 MiB	38.05 MiB	72.94 MiB	127.87 MiB	209.39 MiB	325.01 MiB
8	54.57 MiB	140.74 MiB	304.7 MiB	583.96 MiB	1023.51 MiB	1.64 GiB	2.54 GiB
16	437.01 MiB	1.1 GiB	2.38 GiB	4.56 GiB	8 GiB	13.1 GiB	20.32 GiB
32	3.42 GiB	8.81 GiB	19.06 GiB	36.52 GiB	64 GiB	104.77 GiB	162.6 GiB
48	11.53 GiB	29.72 GiB	64.33 GiB	123.25 GiB	215.99 GiB	353.63 GiB	548.8 GiB
64	27.34 GiB	70.46 GiB	152.48 GiB	292.16 GiB	512 GiB	838.24 GiB	1.27 TiB
72	38.92 GiB	100.32 GiB	217.11 GiB	415.99 GiB	729 GiB	1.17 TiB	1.81 TiB
100	104.29 GiB	268.79 GiB	581.7 GiB	1.09 TiB	1.91 TiB	3.12 TiB	4.85 TiB
128	218.72 GiB	563.7 GiB	1.19 TiB	2.28 TiB	4 TiB	6.55 TiB	10.16 TiB

APPENDIX E

FORTRAN CODE

The full Fortran implementation of the model described in this thesis! The main subroutine is listed first: `light_interface.f90/full_light_calculations`. The electronic version of this code, as well as supplemental analysis can be found online at the following URLs:

`https://github.com/OliverEvans96/swdlyt`

`https://gitlab.com/OliverEvans96/swdlyt`

`https://gitlab.nautilus.optiputer.net/OliverEvans96/swdlyt`

Please don't hesitate to get in touch to discuss the code or any part of the model! You can reach me at `oliverevans96@gmail.com`. I commend you for making it this far, and good luck in all of your endeavors!

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
         ABSORPTION COEFFICIENT
11     abs_water, &
12     scat, &
```

```
13      num_vsf, &
14      vsf_file, &
15      ! SUNLIGHT
16      solar_zenith, &
17      solar_azimuthal, &
18      surface_irrad, &
19      ! KELP &
20      num_si, &
21      si_area, &
22      si_ind, &
23      frond_thickness, &
24      frond_aspect_ratio, &
25      frond_shape_ratio, &
26      ! WATER CURRENT
27      current_speeds, &
28      current_angles, &
29      ! SPACING
30      rope_spacing, &
31      depth_spacing, &
32      ! SOLVER PARAMETERS
33      nx, &
34      ny, &
35      nz, &
36      ntheta, &
37      nphi, &
38      fd_flag, &
39      num_scatters, &
40      num_threads, &
41      ! FINAL RESULTS
42      perceived_irrad, &
43      avg_irrad)
44
45      implicit none
46
47      ! OPTICAL PROPERTIES
48      integer, intent(in) :: nx, ny, nz, ntheta,
           nphi
49      ! Absorption and scattering coefficients
50      double precision, intent(in) ::
           absorptance_kelp, scat
51      double precision, dimension(nz), intent(in)
           :: abs_water
52      ! Volume scattering function
53      integer, intent(in) :: num_vsf
54      character(len=*) :: vsf_file
55      !double precision, dimension(num_vsf),
           intent(int) :: vsf_angles
56      !double precision, dimension(num_vsf),
           intent(int) :: vsf_vals
57
58      ! SUNLIGHT
59      double precision, intent(in) :: solar_zenith
```

```
60      double precision, intent(in) ::
           solar_azimuthal
61      double precision, intent(in) ::
           surface_irrad
62
63      ! KELP
64      ! Number of Superindividuals in each depth
           level
65      integer, intent(in) :: num_si
66      ! si_area(i,j) = area of superindividual j
           at depth i
67      double precision, dimension(nz, num_si),
           intent(in) :: si_area
68      ! si_ind(i,j) = number of inidividuals
           represented by superindividual j at depth
            i
69      double precision, dimension(nz, num_si),
           intent(in) :: si_ind
70      ! Thickness of each frond
71      double precision, intent(in) ::
           frond_thickness
72      ! Ratio of length to width (0,infty)
73      double precision, intent(in) ::
           frond_aspect_ratio
74      ! Rescaled position of greatest width (0=
           base, 1=tip)
75      double precision, intent(in) ::
           frond_shape_ratio
76
77      ! WATER CURRENT
78      double precision, dimension(nz), intent(in)
           :: current_speeds
79      double precision, dimension(nz), intent(in)
           :: current_angles
80
81      ! SPACING
82      ! Horizontal distance between vertical kelp
           ropes = domain width
83      double precision, intent(in) :: rope_spacing
84      ! dz, varies over depth
85      double precision, dimension(nz), intent(in)
           :: depth_spacing
86      ! SOLVER PARAMETERS
87      ! Whether to use the finite difference
           algorithm (otherwise, just asymptotics.)
88      logical, intent(in) :: fd_flag
89      ! Number of scattering events for
           asymptotics. Use 0 if fd_flag = true.
90      integer, intent(in) :: num_scatters
91      ! Number of OMP threads to use for parallel
           subroutines
```

```
92  |     integer, intent(in) :: num_threads
93  |     character*(256) :: lis_opts
94  |     integer :: lis_iter
95  |     double precision :: lis_time, lis_resid
96  |
97  |     ! FINAL RESULT
98  |     real, dimension(nz), intent(out) ::
    |        avg_irrad, perceived_irrad
99  |
100 |     !-------------!
101 |
102 |     double precision xmin, xmax, ymin, ymax,
    |        zmin, zmax
103 |     character(len=5), parameter :: fmtstr = 'E13
    |        .4'
104 |     !double precision, dimension(num_vsf) ::
    |        vsf_angles, vsf_vals
105 |     double precision decay
106 |     integer quadrature_degree
107 |     ! Number of periodic images of the domain to
    |         consider for kelp distribution
108 |     integer :: n_images
109 |
110 |     type(space_angle_grid) grid
111 |     type(rte_mat) mat
112 |     type(optical_properties) iops
113 |     type(light_state) light
114 |     type(rope_state) rope
115 |     type(frond_shape) frond
116 |     type(boundary_condition) bc
117 |
118 |     double precision, dimension(:), allocatable
    |        :: pop_length_means, pop_length_stds
119 |     ! Number of fronds in each depth layer
120 |     double precision, dimension(:), allocatable
    |        :: num_fronds
121 |     double precision, dimension(:,:,:),
    |        allocatable :: p_kelp
122 |     double precision, dimension(nx, ny, nz,
    |        ntheta*(nphi-2)+2) :: radiance
123 |     double precision, dimension(:,:,:,:),
    |        allocatable :: source
124 |
125 |     write(*,*) 'Light calculation'
126 |
127 |     allocate(pop_length_means(nz))
128 |     allocate(pop_length_stds(nz))
129 |     allocate(num_fronds(nz))
130 |     allocate(p_kelp(nx, ny, nz))
131 |
132 |     xmin = -rope_spacing/2
```

```
133 |     xmax = rope_spacing/2
134 |
135 |     ymin = -rope_spacing/2
136 |     ymax = rope_spacing/2
137 |
138 |     zmin = 0.d0
139 |     zmax = sum(depth_spacing)
140 |
141 |     write(*,*) 'Grid'
142 |     call grid%set_bounds(xmin, xmax, ymin, ymax,
    |         zmin, zmax)
143 |     call grid%set_num(nx, ny, nz, ntheta, nphi)
144 |     call grid%init()
145 |     !call grid%set_uniform_spacing_from_num()
146 |     call grid%z%set_spacing_array(depth_spacing)
147 |
148 |     allocate(source( &
149 |          grid%x%num, &
150 |          grid%y%num, &
151 |          grid%z%num, &
152 |          grid%angles%nomega))
153 |
154 |     ! Initialize source to zero
155 |     source(:,:,:,:) = 0.d0
156 |
157 |     call rope%init(grid)
158 |
159 |     write(*,*) 'Rope'
160 |     ! Calculate kelp distribution
161 |     call calculate_length_dist_from_superinds( &
162 |     nz, &
163 |     num_si, &
164 |     si_area, &
165 |     si_ind, &
166 |     frond_aspect_ratio, &
167 |     num_fronds, &
168 |     pop_length_means, &
169 |     pop_length_stds)
170 |
171 |     rope%frond_lengths = pop_length_means
172 |     rope%frond_stds = pop_length_stds
173 |     rope%num_fronds = num_fronds
174 |     rope%water_speeds = current_speeds
175 |     rope%water_angles = current_angles
176 |
177 |     write(*,*) 'frond_lengths  =', rope%
    |        frond_lengths
178 |     write(*,*) 'frond_stds  =', rope%frond_stds
179 |     write(*,*) 'num_fronds  =', rope%num_fronds
```

```
180     write(*,*) 'water_speeds  =', rope%
           water_speeds
181     write(*,*) 'water_angles  =', rope%
           water_angles
182     write(*,*) 'Frond'
183
184     ! INIT FROND
185     call frond%set_shape(frond_shape_ratio,
           frond_aspect_ratio, frond_thickness)
186     write(*,*) 'ft =', frond%ft
187     ! CALCULATE KELP
188     quadrature_degree = 100
189     n_images = 2
190     call calculate_kelp_on_grid(grid, p_kelp,
           frond, rope, quadrature_degree, n_images,
            num_threads)
191     ! INIT IOPS
192     iops%num_vsf = num_vsf
193     call iops%init(grid)
194     write(*,*) 'IOPs'
195     iops%abs_kelp = absorptance_kelp /
           frond_thickness
196     iops%abs_water = abs_water
197     iops%scat = scat
198
199     !write(*,*) 'iop init'
200     !iops%vsf_angles = vsf_angles
201     !iops%vsf_vals = vsf_vals
202     write(*,*) 'Load VSF'
203     call iops%load_vsf(vsf_file, fmtstr)
204
205     ! load_vsf already calls calc_vsf_on_grid
206     !call iops%calc_vsf_on_grid()
207     write(*,*) 'Calculate kelp coef. grid.'
208     call iops%calculate_coef_grids(p_kelp)
209
210     !write(*,*) 'BC'
211     decay = 1.d0 ! Determines drop-off in
           surface BC as a func of angle diff from (
           theta_s, phi_s)
212     write(*,*) 'Init BC'
213     write(*,*) 'I0 = ', surface_irrad
214     write(*,*) 'phi_s = ', solar_zenith
215     write(*,*) 'theta_s = ', solar_azimuthal
216     write(*,*) 'decay = ', decay
217     call bc%init(grid, solar_azimuthal,
           solar_zenith, decay, surface_irrad)
218     write(*,*) 'bc%I0 = ', bc%I0
219     write(*,*) 'bc%phi_s = ', bc%phi_s
220     write(*,*) 'bc%theta_s = ', bc%theta_s
```

```
221 |     write(*,*) 'bc%decay = ', bc%decay
222 |     !write(*,*) 'bc'
223 |     !write(*,*) bc%bc_grid
224 |     call write_vec(bc%bc_grid, grid%angles%
    |        nomega/2, "bc.txt")
225 |     call write_array(iops%abs_grid(:,ny/2,:), nx
    |        , nz, "abs.txt")
226 |
227 |     if(num_scatters .ge. 0) then
228 |         write(*,*) 'Calculate asymptotic light
    |            field'
229 |         call calculate_asymptotic_light_field(&
230 |              grid, bc, iops, source, &
231 |              radiance, num_scatters, num_threads
    |                 )
232 |     else
233 |         radiance = 0
234 |     end if
235 |
236 |
237 |     if(fd_flag) then
238 |
239 |       ! INIT MAT
240 |       write(*,*) 'FD Sparse Matrix'
241 |       ! Set boundary condition
242 |       call mat%init(grid, iops)
243 |       call mat%set_bc(bc)
244 |       call gen_matrix(mat, num_threads)
245 |
246 |       ! Set solver options
247 |       if(num_scatters .ge. 0) then
248 |           mat%initx_zeros = .false.
249 |       else
250 |           mat%initx_zeros = .true.
251 |       end if
252 |       lis_opts = '-i gmres -restart 100'
253 |       call mat%set_solver_opts(trim(lis_opts))
254 |
255 |       ! Initialize & set initial guess
256 |       write(*,*) 'Light init'
257 |       call light%init(mat)
258 |       light%radiance = radiance
259 |
260 |       ! Solve system
261 |       write(*,*) 'Calculate Radiance'
262 |       call light%calculate_radiance()
263 |
264 |       call mat%get_solver_stats(lis_iter,
    |          lis_time, lis_resid)
265 |       call mat%deinit()
266 |     else
```

```
267 |         call light%init_grid(grid)
268 |         light%radiance = radiance
269 |       endif
270 |
271 |       write(*,*) 'Irrad'
272 |       call light%calculate_irradiance()
273 |       ! Calculate output variables
274 |       call calculate_average_irradiance(grid,
    |          light, avg_irrad)
275 |       call calculate_perceived_irradiance(grid,
    |          p_kelp, &
276 |            perceived_irrad, light%irradiance)
277 |
278 |       !write(*,*) 'vsf_angles = ', iops%vsf_angles
279 |       !write(*,*) 'vsf_vals = ', iops%vsf_vals
280 |       !write(*,*) 'vsf norm  = ', grid%
    |          integrate_angle_2d(iops%vsf(1,1,:,:))
281 |
282 |       !0w7rite(*,*) 'abs_water = ', abs_water
283 |       ! write(*,*) 'scat_water = ', scat_water
284 |       !write(*,*) 'kelp '
285 |       !write(*,*) p_kelp(:,:,:)
286 |
287 |       !write(*,*) 'irrad'
288 |       !write(*,*) light%irradiance
289 |
290 |       call write_array(p_kelp(:,ny/2,:), nx, nz, "
    |          kelp.txt")
291 |       call write_array(light%irradiance(:,ny/2,:),
    |           nx, nz, "irrad.txt")
292 |
293 |       write(*,*) 'avg_irrad = ', avg_irrad
294 |       write(*,*) 'perceived_irrad = ',
    |          perceived_irrad
295 |
296 |       write(*,*) 'deinit'
297 |       call bc%deinit()
298 |       !write(*,*) 'a'
299 |       call iops%deinit()
300 |       !write(*,*) 'b'
301 |       call light%deinit()
302 |       !write(*,*) 'c'
303 |       call rope%deinit()
304 |       !write(*,*) 'd'
305 |       call grid%deinit()
306 |       !write(*,*) 'e'
307 |
308 |       deallocate(pop_length_means)
309 |       deallocate(pop_length_stds)
310 |       deallocate(num_fronds)
```

```
311     deallocate(p_kelp)
312
313     !write(*,*) 'done'
314   end subroutine full_light_calculations
315
316   subroutine
        calculate_length_dist_from_superinds( &
317     nz, &
318     num_si, &
319     si_area, &
320     si_ind, &
321     frond_aspect_ratio, &
322     num_fronds, &
323     pop_length_means, &
324     pop_length_stds)
325
326     implicit none
327
328     ! Number of depth levels
329     integer, intent(in) :: nz
330     ! Number of Superindividuals in each depth
          level
331     integer, intent(in) :: num_si
332     ! si_area(i,j) = area of superindividual j
          at depth i
333     double precision, dimension(nz, num_si),
          intent(in) :: si_area
334     ! si_area(i,j) = number of inidividuals
          represented by superindividual j at depth
           i
335     double precision, dimension(nz, num_si),
          intent(in) :: si_ind
336     double precision, intent(in) ::
          frond_aspect_ratio
337
338     double precision, dimension(nz), intent(out)
          :: num_fronds
339     ! Population mean area at each depth level
340     double precision, dimension(nz), intent(out)
          :: pop_length_means
341     ! Population area standard deviation at each
          depth level
342     double precision, dimension(nz), intent(out)
          :: pop_length_stds
343
344     !---------------!
345
346     integer i, k
347     ! Numerators for mean and std
348     double precision mean_num, std_num
349     ! Convert area to length
```

```
350     double precision, dimension(num_si) ::
           si_length
351
352     do k=1, nz
353        mean_num = 0.d0
354        std_num = 0.d0
355        num_fronds(k) = 0
356
357        do i=1, num_si
358           si_length(i) = sqrt(2.d0*
                 frond_aspect_ratio*si_area(k,i))
359           mean_num = mean_num + si_length(i)
360           num_fronds(k) = num_fronds(k) + si_ind
                 (k,i)
361        end do
362
363        pop_length_means(k) = mean_num /
              num_fronds(k)
364
365        do i=1, num_si
366           std_num = std_num + (si_length(i) -
                 pop_length_means(k))**2
367        end do
368
369        pop_length_stds(k) = std_num / (
              num_fronds(k) - 1)
370
371     end do
372
373   end subroutine
        calculate_length_dist_from_superinds
374
375   subroutine calculate_average_irradiance(grid,
        light, avg_irrad)
376     type(space_angle_grid) grid
377     type(light_state) light
378     real, dimension(:) :: avg_irrad
379     integer k, nx, ny, nz
380
381     nx = grid%x%num
382     ny = grid%y%num
383     nz = grid%z%num
384
385     do k=1, nz
386        avg_irrad(k) = real(sum(light%irradiance
              (:,:,k)) / nx / ny)
387     end do
388   end subroutine calculate_average_irradiance
389
390   subroutine calculate_perceived_irradiance(grid
        , p_kelp, &
```

```
391         perceived_irrad, irradiance)
392       type(space_angle_grid) grid
393       double precision, dimension(:,:,:) :: p_kelp
394       real, dimension(:) :: perceived_irrad
395       double precision, dimension(:,:,:) ::
            irradiance
396       double precision total_kelp
397       integer center_i1, center_i2, center_j1,
            center_j2
398
399       integer k
400
401       ! Calculate the average irradiance
            experienced over the frond.
402       ! Has same units as irradiance.
403       ! If no kelp, then just take the irradiance
            at the center
404       ! of the grid.
405       do k=1, grid%z%num
406          total_kelp = sum(p_kelp(:,:,k))
407          if(total_kelp .eq. 0) then
408             center_i1 = int(ceiling(grid%x%num /
                  2.d0))
409             center_j1 = int(ceiling(grid%y%num /
                  2.d0))
410             ! For even grid, use average of center
                  two cells
411             ! For odd grid, just use center cell
412             if(mod(grid%x%num, 2) .eq. 0) then
413                center_i2 = center_i1 + 1
414             else
415                center_i2 = center_i1
416             end if
417             if(mod(grid%y%num, 2) .eq. 0) then
418                center_j2 = center_j1 + 1
419             else
420                center_j2 = center_j1
421             end if
422
423
424             ! Irradiance at the center of the grid
                  (at the rope)
425             perceived_irrad(k) = real(sum(
                  irradiance( &
426                   center_i1:center_i2, &
427                   center_j1:center_j2, k)) &
428                   / ((center_i2-center_i1+1) * (
                        center_j2-center_j1+1)))
429          else
430             ! Average irradiance weighted by kelp
                  distribution
```

```
431 |           perceived_irrad(k) = real( &
432 |               sum(p_kelp(:,:,k)*irradiance(:,:,k
    |                  )) &
433 |               / total_kelp)
434 |       end if
435 |     end do
436 |
437 |   end subroutine calculate_perceived_irradiance
438 |
439 | end module light_interface
```

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(
    |      grid, bc, iops, source, radiance,
    |      num_scatters, num_threads)
  9 |     type(space_angle_grid) grid
 10 |     type(boundary_condition) bc
 11 |     type(optical_properties) iops
 12 |     double precision, dimension(:,:,:,:) ::
    |        radiance
 13 |     double precision, dimension(:,:,:,:),
    |        allocatable :: rad_scatter
 14 |     double precision, dimension(:,:,:,:) ::
    |        source
 15 |     integer num_scatters
 16 |     integer nx, ny, nz, nomega
 17 |     integer max_cells
 18 |     integer n
 19 |     logical bc_flag
 20 |     integer num_threads
 21 |
 22 |     double precision bb
 23 |
 24 |     double precision, dimension(:), allocatable
    |        :: path_length, path_spacing, a_tilde, gn
 25 |
 26 |     nx = grid%x%num
 27 |     ny = grid%y%num
 28 |     nz = grid%z%num
 29 |     nomega = grid%angles%nomega
 30 |
 31 |     max_cells = calculate_max_cells(grid)
 32 |
```

```
33 |     allocate(path_length(max_cells+1))
34 |     allocate(path_spacing(max_cells))
35 |     allocate(a_tilde(max_cells))
36 |     allocate(gn(max_cells))
37 |     allocate(rad_scatter(grid%x%num, grid%y%num,
   |         grid%z%num, grid%angles%nomega))
38 |
39 |     write(*,*) 'before'
40 |     write(*,*) 'min radiance =', minval(radiance
   |        )
41 |     write(*,*) 'max radiance =', maxval(radiance
   |        )
42 |     write(*,*) 'mean radiance =', sum(radiance)/
   |        size(radiance)
43 |
44 |     write(*,*) 'advect source + bc'
45 |     bc_flag = .true.
46 |     call advect_light( &
47 |          grid, iops, source, radiance, &
48 |          path_length, path_spacing, &
49 |          a_tilde, gn, bc_flag, num_threads, bc)
50 |
51 |     write(*,*) 'after'
52 |     write(*,*) 'min radiance =', minval(radiance
   |        )
53 |     write(*,*) 'max radiance =', maxval(radiance
   |        )
54 |     write(*,*) 'mean radiance =', sum(radiance)/
   |        size(radiance)
55 |
56 |     rad_scatter = radiance
57 |     bb = iops%scat
58 |
59 |     do n=1, num_scatters
60 |        write(*,*) 'scatter #', n
61 |        call scatter(grid, iops, source,
   |           rad_scatter, path_length, path_spacing
   |           , a_tilde, gn, num_threads)
62 |        radiance = radiance + bb**n * rad_scatter
63 |     end do
64 |
65 |     write(*,*) 'asymptotics complete'
66 |
67 |     deallocate(path_length)
68 |     deallocate(path_spacing)
69 |     deallocate(a_tilde)
70 |     deallocate(gn)
71 |     deallocate(rad_scatter)
72 |   end subroutine
   |      calculate_asymptotic_light_field
```

```
73 |
74 |   subroutine
   |      calculate_asymptotic_light_field_expanded_source
   |      (&
75 |         grid, bc, iops, source, &
76 |         source_expansion, radiance, &
77 |         num_scatters, num_threads)
78 |     type(space_angle_grid) grid
79 |     type(boundary_condition) bc
80 |     type(optical_properties) iops
81 |     double precision, dimension(:,:,:,:) ::
   |        radiance
82 |     double precision, dimension(:,:,:,:,:) ::
   |        source_expansion
83 |     integer num_scatters
84 |     integer nx, ny, nz, nomega
85 |     integer max_cells
86 |     integer n
87 |     logical bc_flag
88 |     integer num_threads
89 |
90 |     double precision bb
91 |
92 |     double precision, dimension(:), allocatable
   |        :: path_length, path_spacing, a_tilde, gn
93 |     double precision, dimension(:,:,:,:),
   |        allocatable :: source
94 |     double precision, dimension(:,:,:,:),
   |        allocatable :: rad_scatter
95 |     double precision, dimension(:,:,:,:),
   |        allocatable :: scatter_integral
96 |
97 |     nx = grid%x%num
98 |     ny = grid%y%num
99 |     nz = grid%z%num
100|     nomega = grid%angles%nomega
101|
102|     max_cells = calculate_max_cells(grid)
103|
104|     allocate(path_length(max_cells+1))
105|     allocate(path_spacing(max_cells))
106|     allocate(a_tilde(max_cells))
107|     allocate(gn(max_cells))
108|     allocate(rad_scatter(grid%x%num, grid%y%num,
   |         grid%z%num, grid%angles%nomega))
109|     allocate(scatter_integral(nx, ny, nz, nomega
   |        ))
110|
111|     write(*,*) 'advect source + bc'
112|     bc_flag = .true.
113|     call advect_light( &
```

```
114           grid, iops, source_expansion(:,:,:,:,1)
                 , radiance, &
115           path_length, path_spacing, &
116           a_tilde, gn, bc_flag, num_threads, bc)
117     ! Disable BC for scattering advection
118     bc_flag = .false.
119
120     rad_scatter = radiance
121     bb = iops%scat
122
123     do n=1, num_scatters
124        write(*,*) 'scatter #', n
125        call calculate_scatter_source(grid, iops,
                rad_scatter, source, scatter_integral
               , num_threads)
126        source = source + source_expansion
               (:,:,:,:,n+1)
127        call advect_light(grid, iops, source,
               rad_scatter, path_length, path_spacing
               , a_tilde, gn, bc_flag, num_threads)
128
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
        calculate_asymptotic_light_field_expanded_source
142
143   ! Add attenuated surface light to existing
        radiance
144   subroutine advect_surface_bc(&
145        i, j, k, p, radiance, &
146        path_spacing, num_cells, a_tilde, bc)
147     type(boundary_condition) bc
148     double precision, dimension(:,:,:,:) ::
          radiance
149     double precision, dimension(:) ::
          path_spacing, a_tilde
150     integer i, j, k, p
151     integer num_cells
152     double precision atten
153
```

```
154 |     atten = sum(path_spacing(1:num_cells) *
    |        a_tilde(1:num_cells))
155 |     ! Avoid underflow
156 |     if(atten .lt. 100.d0) then
157 |        radiance(i,j,k,p) = radiance(i,j,k,p) +
    |           bc%bc_grid(p) * exp(-atten)
158 |     end if
159 |   end subroutine advect_surface_bc
160 |
161 |   ! Perform one scattering event
162 |   subroutine scatter(grid, iops, source,
    |      rad_scatter, path_length, path_spacing,
    |      a_tilde, gn, num_threads)
163 |     type(space_angle_grid) grid
164 |     type(optical_properties) iops
165 |     double precision, dimension(:,:,:,:) ::
    |        rad_scatter, source
166 |     double precision, dimension(:,:,:,:),
    |        allocatable :: scatter_integral
167 |     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 |     call calculate_scatter_source(grid, iops,
    |        rad_scatter, source, scatter_integral,
    |        num_threads)
181 |     call advect_light(grid, iops, source,
    |        rad_scatter, path_length, path_spacing,
    |        a_tilde, gn, bc_flag, num_threads)
182 |
183 |     deallocate(scatter_integral)
184 |   end subroutine scatter
185 |
186 |   ! Calculate source from no-scatter or previous
    |      scattering layer
187 |   subroutine calculate_scatter_source(grid, iops
    |      , rad_scatter, source, scatter_integral,
    |      num_threads)
188 |     type(space_angle_grid) grid
189 |     type(optical_properties) iops
```

```
190     double precision, dimension(:,:,:,:) ::
           rad_scatter
191     double precision, dimension(:,:,:,:) ::
           source
192     double precision, dimension(:,:,:,:) ::
           scatter_integral
193     type(index_list) indices
194     integer nx, ny, nz, nomega
195     integer i, j, k, p
196     integer num_threads
197
198     nx = grid%x%num
199     ny = grid%y%num
200     nz = grid%z%num
201     nomega = grid%angles%nomega
202
203     ! Use collapse to combine outer two loops
204     ! to avoid manually deciding nesting details
           .
205
206     !$omp parallel do default(none) private(
           indices) &
207     !$omp private(i,j,k,p) shared(nx,ny,nz,
           nomega) &
208     !$omp shared(iops, rad_scatter,
           scatter_integral) &
209     !$omp num_threads(num_threads) collapse(2)
210
211     do k=1, nz
212        do i=1, nx
213           indices%k = k
214           indices%i = i
215           do j=1, ny
216              indices%j = j
217              do p=1, nomega
218                 indices%p = p
219                 call calculate_scatter_integral
                       (&
220                     iops, rad_scatter,&
221                     scatter_integral,&
222                     indices)
223              end do
224           end do
225        end do
226     end do
227     !$omp end parallel do
228
229     source(:,:,:,:) = -rad_scatter(:,:,:,:) +
           scatter_integral(:,:,:,:)
230
231     write(*,*) 'source min: ', minval(source)
```

```
232 |     write(*,*) 'source max: ', maxval(source)
233 |     write(*,*) 'source mean: ', sum(source)/size
    |        (source)
234 |
235 |   end subroutine calculate_scatter_source
236 |
237 |   subroutine calculate_scatter_integral(iops,
    |      rad_scatter, scatter_integral, indices)
238 |     type(optical_properties) iops
239 |     double precision, dimension(:,:,:,:) ::
    |        rad_scatter, scatter_integral
240 |     type(index_list) indices
241 |
242 |     scatter_integral(indices%i,indices%j,indices
    |        %k,indices%p) &
243 |           = sum(iops%vsf_integral(indices%p, :) &
244 |           * rad_scatter(&
245 |             indices%i,&
246 |             indices%j,&
247 |             indices%k,:))
248 |
249 |   end subroutine calculate_scatter_integral
250 |
251 |   subroutine advect_light(grid, iops, source,
    |      rad_scatter, path_length, path_spacing,
    |      a_tilde, gn, bc_flag, num_threads, bc)
252 |     type(space_angle_grid) grid
253 |     type(optical_properties) iops
254 |     double precision, dimension(:,:,:,:) ::
    |        rad_scatter, source
255 |     double precision, dimension(:) ::
    |        path_length, path_spacing, a_tilde, gn
256 |     logical bc_flag
257 |     type(boundary_condition), intent(in),
    |        optional :: bc
258 |     integer i, j, k, p
259 |     integer num_threads
260 |
261 |     !$omp parallel do default(none) &
262 |     !$omp private(i,j,k,p) &
263 |     !$omp shared(rad_scatter,source,grid,iops,
    |        bc_flag,bc) &
264 |     !$omp private(path_length,path_spacing,
    |        a_tilde,gn) &
265 |     !$omp num_threads(num_threads) collapse(2)
266 |     do k=1, grid%z%num
267 |        do i=1, grid%x%num
268 |           do j=1, grid%y%num
269 |              do p=1, grid%angles%nomega
```

```
270                call integrate_ray(grid, iops,
                      source,&
271                     rad_scatter, path_length,
                           path_spacing,&
272                     a_tilde, gn, i, j, k, p,
                           bc_flag, bc)
273             end do
274           end do
275         end do
276       end do
277       !$omp end parallel do
278     end subroutine advect_light
279
280     ! New algorithm, double integral over
          piecewise-constant 1d funcs
281     subroutine integrate_ray(grid, iops, source,
          rad_scatter, path_length, path_spacing,
          a_tilde, gn, i, j, k, p, bc_flag, bc)
282       type(space_angle_grid) :: grid
283       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),
            optional :: bc
291
292       integer num_cells
293
294       call traverse_ray(grid, iops, source, i, j,
            k, p, path_length, path_spacing, a_tilde,
            gn, num_cells)
295       rad_scatter(i,j,k,p) =
            calculate_ray_integral(num_cells,
            path_length, path_spacing, a_tilde, gn)
296
297       if(bc_flag .and. p .le. grid%angles%nomega
            /2) then
298         call advect_surface_bc(&
299              i, j, k, p, rad_scatter, &
300              path_spacing, num_cells, &
301              a_tilde, bc)
302       end if
303
304       ! if((i .eq. 1) &
```

```
305 |     !       .and. (j .eq. 1) &
306 |     !       !.and. (k .eq. grid%z%num/2) &
307 |     !       .and. ( &
308 |     !       (p .eq. 1) .or. (p .eq. grid%angles%
    |        nomega) &
309 |     !       )) then
310 |
311 |     !     write(*,*) 'ray (', i, ', ', j, ', ', k
    |        , ', ', p, ')'
312 |     !     write(*,*) 'num_cells = ', num_cells
313 |     !     write(*,*) 'path_spacing:'
314 |     !     write(*,*) path_spacing(1:num_cells)
315 |     !     write(*,*) 'path_length:'
316 |     !     write(*,*) path_length(1:num_cells+1)
317 |     !     write(*,*) 'a_tilde:'
318 |     !     write(*,*) a_tilde(1:num_cells)
319 |     !     write(*,*) 'gn:'
320 |     !     write(*,*) gn(1:num_cells)
321 |     !     write(*,*)
322 |     ! end if
323 |
324 |   end subroutine integrate_ray
325 |
326 |   function calculate_ray_integral(num_cells, s,
    |      ds, a_tilde, gn) result(integral)
327 |     ! Double integral which accumulates all
    |        scattered light along the path
328 |     ! via an angular integral and attenuates it
    |        by integrating along the path
329 |     integer :: num_cells
330 |     double precision, dimension(num_cells) :: ds
    |        , a_tilde, gn
331 |     double precision, dimension(num_cells+1) ::
    |        s
332 |     double precision :: integral
333 |     double precision bi, di_exp_bi
334 |     double precision cutoff
335 |     integer i, j
336 |
337 |     ! Maximum absorption coefficient suitable
    |        for numerical computation
338 |     cutoff = 10.d0
339 |
340 |     integral = 0
341 |     do i=1, num_cells
342 |        bi = -a_tilde(i)*s(i+1)
343 |        do j=i+1, num_cells
344 |           bi = bi - a_tilde(j)*ds(j)
345 |        end do
346 |
```

```
347         ! In this case, so much absorption has
               occurred
348         ! previously on the path that we don't
               need
349         ! to continue, and we might get underflow
                if we do.
350         if(bi .lt. -100.d0) then
351            di_exp_bi = 0.d0
352         else
353
354           ! Without this conditional, overflow
                occurs.
355           ! Which is unnecessary, because large
                absorption
356           ! means very small light added to the
                ray
357           ! at this grid cell.
358           if(a_tilde(i) .lt. cutoff) then
359               if(a_tilde(i) .eq. 0) then
360                   di_exp_bi = ds(i) * exp(bi)
361               else
362                  ! In an attempt to avoid
                       overflow
363                  ! and reduce compute time,
364                  ! I'm combining exponentials.
365                  ! di*exp(bi) -> di_exp_bi
366                  di_exp_bi = (exp(a_tilde(i)*s(i
                       +1) + bi) - exp(a_tilde(i)*s(
                       i) + bi))/a_tilde(i)
367               end if
368               integral = integral + gn(i)*
                     di_exp_bi
369           end if
370         end if
371     end do
372
373   end function calculate_ray_integral
374
375   ! Calculate maximum number of cells a path
        through the grid could take
376   ! This is a loose upper bound
377   function calculate_max_cells(grid) result(
        max_cells)
378     type(space_angle_grid) :: grid
379     integer :: max_cells
380     double precision dx, dy, zrange, phi_middle
381
382     ! Angle that will have the longest ray
383     phi_middle = grid%angles%phi(grid%angles%
          nphi/2)
384     dx = grid%x%spacing(1)
385     dy = grid%y%spacing(1)
```

```
386       zrange = grid%z%maxval - grid%z%minval
387
388       max_cells = grid%z%num + ceiling((1/dx+1/dy)
              *zrange*tan(phi_middle))
389     end function calculate_max_cells
390
391     ! Traverse from surface or bottom to point (xi
          , yj, zk)
392     ! in the direction omega_p, extracting path
          lengths (ds) and
393     ! function values (f) along the way,
394     ! as well as number of cells traversed (n).
395     subroutine traverse_ray(grid, iops, source, i,
           j, k, p, s_array, ds, a_tilde, gn,
          num_cells)
396       type(space_angle_grid) :: grid
397       type(optical_properties) :: iops
398       double precision, dimension(:,:,:,:) ::
              source
399       integer :: i, j, k, p
400       double precision, dimension(:) :: s_array,
              ds, a_tilde, gn
401       integer :: num_cells
402
403       integer t
404       double precision p0x, p0y, p0z
405       double precision p1x, p1y, p1z
406       double precision z0
407       double precision s_tilde, s
408       integer dir_x, dir_y, dir_z
409       integer shift_x, shift_y
410       integer cell_x, cell_y, cell_z
411       integer edge_x, edge_y
412       integer first_x, last_x, first_y, last_y,
              last_z
413       double precision s_next_x, s_next_y,
              s_next_z, s_next
414       double precision x_factor, y_factor,
              z_factor
415       double precision ds_x, ds_y
416       double precision, dimension(grid%z%num) ::
              ds_z
417       double precision smx, smy
418
419       ! Divide by these numbers to get path
              separation
420       ! from separation in individual dimensions
421       x_factor = grid%angles%sin_phi_p(p) * grid%
              angles%cos_theta_p(p)
422       y_factor = grid%angles%sin_phi_p(p) * grid%
              angles%sin_theta_p(p)
423       z_factor = grid%angles%cos_phi_p(p)
```

```
424 |
425 |     ! Destination point
426 |     p1x = grid%x%vals(i)
427 |     p1y = grid%y%vals(j)
428 |     p1z = grid%z%vals(k)
429 |
430 |     !write(*,*) 'START PATH.'
431 |     !write(*,*) 'ijk = ', i, j, k
432 |
433 |     ! Direction
434 |     if(p .le. grid%angles%nomega/2) then
435 |        ! Downwelling light originates from
    |           surface
436 |        z0 = grid%z%minval
437 |        dir_z = 1
438 |     else
439 |        ! Upwelling light originates from bottom
440 |        z0 = grid%z%maxval
441 |        dir_z = -1
442 |     end if
443 |
444 |     ! Total path length from origin to
    |        destination
445 |     ! (sign is correct for upwelling and
    |        downwelling)
446 |     s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p
    |        )
447 |
448 |     ! Path spacings between edge intersections
    |        in each dimension
449 |     ! Set to 2*s_tilde if infinite in this
    |        dimension so that it's unreachable
450 |     ! (e.g., if ray is parallel to x axis, then
    |        no x intersection will occur.)
451 |     ! Assume x & y spacings are uniform,
452 |     ! so it's okay to just use the first value.
453 |     if(x_factor .eq. 0) then
454 |        ds_x = 2*s_tilde
455 |     else
456 |        ds_x = abs(grid%x%spacing(1)/x_factor)
457 |     end if
458 |     if(y_factor .eq. 0) then
459 |        ds_y = 2*s_tilde
460 |     else
461 |        ds_y = abs(grid%y%spacing(1)/y_factor)
462 |     end if
463 |
464 |     ! This one is an array because z spacing can
    |        vary
465 |     ! z_factor should never be 0,
466 |     ! because the ray is then horizontal
467 |     ! and infinite in length.
```

```
468     ! z_factor != 0 is ensured when nphi is even
          .
469     ds_z(1:grid%z%num) = dir_z * grid%z%spacing
          (1:grid%z%num)/z_factor
470
471     ! Origin point
472     p0x = p1x - s_tilde * x_factor
473     p0y = p1y - s_tilde * y_factor
474     p0z = p1z - s_tilde * z_factor
475
476     ! Direction of ray in each dimension. 1 =>
          increasing. -1 => decreasing.
477     dir_x = int(sgn(p1x-p0x))
478     dir_y = int(sgn(p1y-p0y))
479
480     ! Shifts
481     ! Conversion from cell_inds to edge_inds
482     ! merge is fortran's ternary operator
483     shift_x = merge(1,0,dir_x>0)
484     shift_y = merge(1,0,dir_y>0)
485
486     ! Indices for cell containing origin point
487     cell_x = floor((p0x-grid%x%minval)/grid%x%
          spacing(1)) + 1
488     cell_y = floor((p0y-grid%y%minval)/grid%y%
          spacing(1)) + 1
489     ! x and y may be in periodic image, so shift
           back.
490     cell_x = mod1(cell_x, grid%x%num)
491     cell_y = mod1(cell_y, grid%y%num)
492
493     ! z starts at top or bottom depending on
          direction.
494     if(dir_z > 0) then
495        cell_z = 1
496     else
497        cell_z = grid%z%num
498     end if
499
500     ! Edge indices preceeding starting cells
501     edge_x = mod1(cell_x + shift_x, grid%x%num)
502     edge_y = mod1(cell_y + shift_y, grid%y%num)
503
504     ! First and last cells in each
505     if(dir_x .gt. 0) then
506        first_x = 1
507        last_x = grid%x%num
508     else
509        first_x = grid%x%num
510        last_x = 1
511     end if
```

```
512       if(dir_y .gt. 0) then
513          first_y = 1
514          last_y = grid%y%num
515       else
516          first_y = grid%y%num
517          last_y = 1
518       end if
519       if(dir_z .gt. 0) then
520          last_z = grid%z%num
521       else
522          last_z = 1
523       end if
524
525       ! Calculate periodic images
526       smx = shift_mod(p0x, grid%x%minval, grid%x%
             maxval)
527       smy = shift_mod(p0y, grid%y%minval, grid%y%
             maxval)
528
529       ! Path length to next edge plane in each
             dimension
530       if(abs(x_factor) .lt. 1.d-10) then
531          ! Will never cross, so set above total
               path length
532          s_next_x = 2*s_tilde
533       else if(cell_x .eq. last_x) then
534          ! If starts out at last cell, then
               compare to periodic image
535          s_next_x = (grid%x%edges(first_x) + dir_x
                 * (grid%x%maxval - grid%x%minval)&
536                   - smx) / x_factor
537       else
538          ! Otherwise, just compare to next cell
539          s_next_x = (grid%x%edges(edge_x) - smx) /
                 x_factor
540       end if
541
542       ! Path length to next edge plane in each
             dimension
543       if(abs(y_factor) .lt. 1.d-10) then
544          ! Will never cross, so set above total
               path length
545          s_next_y = 2*s_tilde
546       else if(cell_y .eq. last_y) then
547          ! If starts out at last cell, then
               compare to periodic image
548          s_next_y = (grid%y%edges(first_y) + dir_y
                 * (grid%y%maxval - grid%y%minval)&
549                   - smy) / y_factor
550       else
551          ! Otherwise, just compare to next cell
```

```
552         s_next_y = (grid%y%edges(edge_y) - smy) /
                y_factor
553       end if
554
555       s_next_z = ds_z(cell_z)
556
557       ! Initialize path
558       s = 0.d0
559       s_array(1) = 0.d0
560
561       ! Start with t=0 so that we can increment
              before storing,
562       ! so that t will be the number of grid cells
              at the end of the loop.
563       t = 0
564
565       ! s is the beginning of the current cell,
566       ! s_next is the end of the current cell.
567       do while (s .lt. s_tilde)
568          ! Move cell counter
569          t = t + 1
570
571          ! Extract function values
572          a_tilde(t) = iops%abs_grid(cell_x, cell_y
                , 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%
                theta_p(p)*180.d0/pi, grid%angles%
                phi_p(p)*180.d0/pi
580          !write(*,*) 's = ', s, '/', s_tilde
581          !write(*,*) 'cell_z =', cell_z, '/', grid
                %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))
                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%
                   num)
```

```
593             edge_x = mod1(edge_x + dir_x, grid%x%
                   num)
594
595             ! x intersection after the one at s=
                   s_next
596             s_next_x = s_next + ds_x
597
598          else if (s_next_y .le. min(s_next_x,
                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%
                   num)
604             edge_y = mod1(edge_y + dir_y, grid%y%
                   num)
605
606             ! y intersection after the one at s=
                   s_next
607             s_next_y = s_next + ds_y
608
609          else if(s_next_z .le. min(s_next_x,
                s_next_y)) then
610             ! z edge is closest
611             s_next = s_next_z
612
613             ! Increment indices
614             cell_z = cell_z + dir_z
615
616             !write(*,*) 'z edge, s_next =', s_next
617
618             ! z intersection after the one at s=
                   s_next
619             if(dir_z * (last_z - cell_z) .gt. 0)
                   then
620                ! Only look ahead if we aren't at
                      the end
621                s_next_z = s_next + ds_z(cell_z)
622             else
623                ! Otherwise, no need to continue.
624                ! this is our final destination.
625             !   exit
626                s_next_z = 2*s_tilde
627                !write(*,*) 'end. s_next_z =',
                      s_next_z
628             end if
629
630          end if
631
632          ! Cut off early if this is the end
```

```
633         ! This will be the last cell traversed if
                s_next >= s_tilde
634         s_next = min(s_tilde, s_next)
635
636         ! Store path length
637         s_array(t+1) = s_next
638         ! Extract path length from same cell as
               function vals
639         ds(t) = s_next - s
640
641         ! Update path length
642         s = s_next
643       end do
644
645       ! Return number of cells traversed
646       num_cells = t
647
648     end subroutine traverse_ray
649   end module asymptotics
```

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   ! Use 64-bit integers for LIS
8   ! Necessary for FD solution w/ large matrices
9   #define LONG__LONG
10  #include "lisf.h"
11  implicit none
12
13  type solver_opts
14     integer maxiter_inner, maxiter_outer
15     double precision tol_abs, tol_rel
16  end type solver_opts
17
18  type rte_mat
19     type(space_angle_grid) grid
20     type(optical_properties) iops
21     type(solver_opts) params
22     integer nx, ny, nz, nomega
23     integer i, j, k, p
24     integer(index_kind) nonzero, n_total
25     integer x_block_size, y_block_size,
          z_block_size, omega_block_size
26
27     double precision, dimension(:), allocatable
          :: surface_vals
28
```

```
29    ! CSR format
30    ! http://www.scipy-lectures.org/advanced/
        scipy_sparse/csr_matrix.html
31    ! with LIS method 2 (LIS manual, p.19)
32    integer(index_kind), dimension(:),
        allocatable :: ptr, col
33    double precision, dimension(:), allocatable
        :: data
34
35    ! Lis Matrix and vectors
36    LIS_MATRIX A
37    LIS_VECTOR b, x
38    LIS_SOLVER solver
39    LIS_INTEGER ierr
40    character(len=1024) solver_opts
41    logical initx_zeros
42
43    ! Pointer to solver subroutine
44    ! Set to mgmres by default
45    !procedure(solver_interface), pointer, nopass
        :: solver => mgmres_st
46
47  contains
48    procedure :: init => mat_init
49    procedure :: deinit => mat_deinit
50    procedure :: calculate_size
51    procedure :: set_solver_opts =>
        mat_set_solver_opts
52    procedure :: set_row => mat_set_row
53    procedure :: assign => mat_assign
54    procedure :: add => mat_add
55    procedure :: assign_rhs => mat_assign_rhs
56    procedure :: add_rhs => mat_add_rhs
57    !procedure :: store_index => mat_store_index
58    !procedure :: find_index => mat_find_index
59    procedure :: set_bc => mat_set_bc
60    procedure :: solve => mat_solve
61    procedure :: get_solver_stats
62    procedure :: ind => mat_ind
63    !procedure :: to_hdf => mat_to_hdf
64    procedure :: attenuate
65    procedure :: angular_integral
66    procedure :: add_source
67
68    ! Derivative subroutines
69    procedure x_cd2
70    procedure x_cd2_first
71    procedure x_cd2_last
72    procedure y_cd2
73    procedure y_cd2_first
74    procedure y_cd2_last
75    procedure z_cd2
76    procedure z_fd2
```

```
77    procedure z_bd2
78    procedure z_surface_bc
79    procedure z_bottom_bc
80
81  end type rte_mat
82
83  interface
84    ! Define interface for external procedure
85    ! https://stackoverflow.com/questions
        /8549415/how-to-declare-the-interface-
        section-for-a-procedure-argument-which-in-
        turn-ref
86    subroutine solver_interface(n_total, nonzero,
          row, col, data, &
87          sol, rhs, maxiter_outer, maxiter_inner,
              &
88          tol_abs, tol_rel)
89      use type_consts
90      integer(index_kind) ::  n_total, nonzero
91      integer, dimension(nonzero) :: row, col
92      double precision, dimension(nonzero) ::
          data
93      double precision, dimension(nonzero) :: sol
94      double precision, dimension(n_total) :: rhs
95      integer :: maxiter_outer, maxiter_inner
96      double precision :: tol_abs, tol_rel
97    end subroutine solver_interface
98  end interface
99
100 contains
101
102   subroutine mat_init(mat, grid, iops)
103     class(rte_mat) mat
104     type(space_angle_grid) grid
105     type(optical_properties) iops
106     integer(index_kind) nnz, n_total
107
108     LIS_INTEGER comm_world
109
110     comm_world = LIS_COMM_WORLD
111
112     mat%grid = grid
113     mat%iops = iops
114
115     call mat%calculate_size()
116
117     mat%solver_opts = ''
118     mat%ierr = 0
119
120     n_total = mat%n_total
121     nnz = mat%nonzero
```

```
122 |
123 |     write(*,*) 'lis_init'
124 |     call lis_initialize(mat%ierr)
125 |
126 |     call lis_solver_create(mat%solver, mat%ierr)
127 |
128 |     call lis_matrix_create(comm_world, mat%A,
    |        mat%ierr)
129 |     call lis_vector_create(comm_world, mat%b,
    |        mat%ierr)
130 |     call lis_vector_create(comm_world, mat%x,
    |        mat%ierr)
131 |
132 |     call lis_matrix_set_size(mat%A, n_total,
    |        n_total, mat%ierr)
133 |     call lis_vector_set_size(mat%b, n_total,
    |        n_total, mat%ierr)
134 |     call lis_vector_set_size(mat%x, n_total,
    |        n_total, mat%ierr)
135 |
136 |     call lis_vector_set_all(0.0d0, mat%x, mat%
    |        ierr)
137 |     call lis_vector_set_all(0.0d0, mat%b, mat%
    |        ierr)
138 |
139 |     if(mat%ierr .ne. 0) then
140 |        write(*,*) 'INIT ERR: ', mat%ierr
141 |        call exit(1)
142 |     end if
143 |
144 |     ! CSR Format
145 |     ! http://www.scipy-lectures.org/advanced/
    |        scipy_sparse/csr_matrix.html
146 |     write(*,*) 'Allocate CSR arrays'
147 |     allocate(mat%ptr(n_total+1))
148 |     allocate(mat%col(nnz))
149 |     allocate(mat%data(nnz))
150 |     allocate(mat%surface_vals(grid%angles%nomega
    |        ))
151 |
152 |     mat%ptr(n_total+1) = nnz
153 |   end subroutine mat_init
154 |
155 |   subroutine mat_deinit(mat)
156 |     class(rte_mat) mat
157 |
158 |     call lis_matrix_destroy(mat%A, mat%ierr)
159 |     call lis_vector_destroy(mat%b, mat%ierr)
160 |     call lis_vector_destroy(mat%x, mat%ierr)
```

```
161 |     call lis_solver_destroy(mat%solver, mat%ierr
    |        )
162 |     call lis_finalize(mat%ierr)
163 |
164 |     if(mat%ierr .ne. 0) then
165 |        write(*,*) 'DEINIT ERR: ', mat%ierr
166 |        call exit(1)
167 |     end if
168 |
169 |     deallocate(mat%ptr)
170 |     deallocate(mat%col)
171 |     deallocate(mat%data)
172 |     deallocate(mat%surface_vals)
173 |   end subroutine mat_deinit
174 |
175 |   subroutine calculate_size(mat)
176 |     class(rte_mat) mat
177 |     integer(index_kind) nx, ny, nz, nomega
178 |
179 |     nx = mat%grid%x%num
180 |     ny = mat%grid%y%num
181 |     nz = mat%grid%z%num
182 |     nomega = mat%grid%angles%nomega
183 |
184 |     !mat%nonzero = nx * ny * ntheta * nphi * ( (
    |        nz-1) * (6 + ntheta * nphi) + 1)
185 |     mat%nonzero = nx * ny * nomega * (nz * (
    |        nomega + 6) - 1)
186 |     mat%n_total = nx * ny * nz * nomega
187 |     write(*,*) 'nnz = ', mat%nonzero
188 |     write(*,*) 'n_total = ', mat%n_total
189 |
190 |     !mat%theta_block_size = 1
191 |     !mat%phi_block_size = mat%theta_block_size *
    |         ntheta
192 |     mat%omega_block_size = 1
193 |     mat%y_block_size = int(mat%omega_block_size
    |        * nomega)
194 |     mat%x_block_size = int(mat%y_block_size * ny
    |        )
195 |     mat%z_block_size = int(mat%x_block_size * nx
    |        )
196 |
197 |   end subroutine calculate_size
198 |
199 | !  subroutine mat_to_hdf(mat,filename)
200 | !    class(rte_mat) mat
201 | !    character(len=*) filename
202 | !    call write_coo(filename, mat%row, mat%col,
    |   mat%data, mat%nonzero)
```

```
203 !  end subroutine mat_to_hdf
204
205   subroutine mat_set_bc(mat, bc)
206     class(rte_mat) mat
207     class(boundary_condition) bc
208     integer p
209
210     do p=1, mat%grid%angles%nomega/2
211        mat%surface_vals(p) = bc%bc_grid(p)
212     end do
213   end subroutine mat_set_bc
214
215   subroutine mat_solve(mat)
216     class(rte_mat) mat
217     character(len=64) init_opt
218
219     ! write(*,*) 'mat%n_total =', mat%n_total
220     ! write(*,*) 'mat%nonzero =', mat%nonzero
221     ! open(unit=1, file='ptr.txt')
222     ! open(unit=2, file='col.txt')
223     ! open(unit=3, file='data.txt')
224     ! write(1,*) mat%ptr
225     ! write(2,*) mat%col
226     ! write(3,*) mat%data
227     ! close(1)
228     ! close(2)
229     ! close(3)
230
231     ! Create matrix
232     write(*,*) 'LIS Set CSR'
233     call lis_matrix_set_csr(mat%nonzero, mat%ptr
        , mat%col, mat%data, mat%A, mat%ierr)
234     write(*,*) 'LIS Assemble'
235     call lis_matrix_assemble(mat%A, mat%ierr)
236
237     ! Set solver options
238     if(mat%initx_zeros) then
239         init_opt = "-initx_zeros true -print out
            "
240     else
241         init_opt = "-initx_zeros false -print
            out"
242     end if
243
244     write(*,*) 'LIS set solver options'
245     write(*,*) 'opt: ', trim(init_opt)
246     call lis_solver_set_option(init_opt, mat%
        solver, mat%ierr)
247     if(len(trim(mat%solver_opts)) .gt. 0) then
```

```
248         write(*,*) 'opt: ', trim(mat%solver_opts
              )
249         call lis_solver_set_option(mat%
              solver_opts, mat%solver, mat%ierr)
250       end if
251
252       ! Solve
253       write(*,*) 'LIS Solve'
254       call lis_solve(mat%A, mat%b, mat%x, mat%
            solver, mat%ierr)
255       write(*,*) 'LIS Solve done'
256
257     end subroutine mat_solve
258
259     subroutine get_solver_stats(mat, lis_iter,
          lis_time, lis_resid)
260       class(rte_mat) mat
261       integer lis_iter
262       double precision lis_time
263       double precision lis_resid
264
265       call lis_solver_get_iter(mat%solver,
            lis_iter, mat%ierr)
266       call lis_solver_get_time(mat%solver,
            lis_time, mat%ierr)
267       call lis_solver_get_residualnorm(mat%solver,
            lis_resid, mat%ierr)
268     end subroutine get_solver_stats
269
270     subroutine mat_set_solver_opts(mat,
          solver_opts)
271       class(rte_mat) mat
272       character(len=*) solver_opts
273       mat%solver_opts = solver_opts
274     end subroutine mat_set_solver_opts
275
276     function mat_ind(mat, i, j, k, p) result(ind)
277       ! Assuming var ordering: z, x, y, omega
278       class(rte_mat) mat
279       integer i, j, k, p
280       integer(index_kind) ind
281
282       ind = (i-1) * mat%x_block_size + (j-1) * mat
            %y_block_size + &
283           (k-1) * mat%z_block_size + p * mat%
                omega_block_size
284     end function mat_ind
285
286     subroutine mat_set_row(mat, ent, row_num)
287       ! Start new row for CSR format
```

```
288     class(rte_mat) mat
289     integer(index_kind) ent, row_num
290     ! 0-indexing for LIS
291     mat%ptr(row_num) = ent - 1
292   end subroutine mat_set_row
293
294   subroutine mat_assign(mat, ent, val, i, j, k,
          p)
295     ! It's assumed that this is the first time
            this entry is defined
296     class(rte_mat) mat
297     double precision val
298     integer i, j, k, p
299     integer(index_kind) ent
300
301     ! LIS method 2 (LIS manual, p. 19) requires
            0-indexing
302     mat%col(ent) = mat%ind(i, j, k, p) - 1
303     mat%data(ent) = val
304
305     ent = ent + 1
306   end subroutine mat_assign
307
308   subroutine mat_add(mat, repeat_ent, val)
309     ! Use this when you know that this entry has
            already been assigned
310     ! and you'd like to add this value to the
            existing value.
311
312     class(rte_mat) mat
313     double precision val
314     integer(index_kind) repeat_ent
315
316     ! Entry number where value is already stored
317     mat%data(repeat_ent) = mat%data(repeat_ent)
          + val
318   end subroutine mat_add
319
320   subroutine mat_assign_rhs(mat, row_num, data)
321     class(rte_mat) mat
322     double precision data
323     integer(index_kind) row_num
324
325     call lis_vector_set_value(LIS_INS_VALUE,
          row_num, data, mat%b, mat%ierr)
326     if(mat%ierr .ne. 0) then
327       write(*,*) 'RHS ERR: ', mat%ierr
328       call exit(1)
329     end if
330   end subroutine mat_assign_rhs
331
```

```
332 |   subroutine mat_add_rhs(mat, row_num, data)
333 |     class(rte_mat) mat
334 |     double precision data
335 |     integer(index_kind) row_num
336 |
337 |     call lis_vector_set_value(LIS_ADD_VALUE,
    |        row_num, data, mat%b, mat%ierr)
338 |     if(mat%ierr .ne. 0) then
339 |        write(*,*) 'RHS ERR: ', mat%ierr
340 |        call exit(1)
341 |     end if
342 |   end subroutine mat_add_rhs
343 |
344 |   ! subroutine mat_store_index(mat, row_num,
    |      col_num)
345 |   !   ! Remember where we stored information for
    |       this matrix element
346 |   !   class(rte_mat) mat
347 |   !   integer row_num, col_num
348 |   !   !mat%index_map(row_num, col_num) = mat%ent
349 |   ! end subroutine
350 |
351 |   ! function mat_find_index(mat, row_num,
    |      col_num) result(index)
352 |   !   ! Find the position in row, col, data
    |      where this entry
353 |   !   ! is defined.
354 |   !   class(rte_mat) mat
355 |   !   integer row_num, col_num, index
356 |
357 |   !   index = mat%index_map(row_num, col_num)
358 |
359 |   !   ! This took up 95% of execution time.
360 |   !   ! Only search up to most recently assigned
    |       index
361 |   !   ! do index=1, mat%ent-1
362 |   !   !    if( (mat%row(index) .eq. row_num) .
    |      and. (mat%col(index) .eq. col_num)) then
363 |   !   !       exit
364 |   !   !    end if
365 |   !   ! end do
366 |   ! end function mat_find_index
367 |
368 |   subroutine attenuate(mat, indices, repeat_ent)
369 |     ! Has to be called after angular_integral
370 |     ! Because they both write to the same matrix
    |         entry
371 |     ! And adding here is more efficient than a
    |       conditional
372 |     ! in the angular loop.
373 |     class(rte_mat) mat
```

```
374 |     double precision attenuation
375 |     type(index_list) indices
376 |     double precision aa, bb
377 |     integer(index_kind) repeat_ent
378 |
379 |     aa = mat%iops%abs_grid(indices%i, indices%j,
    |         indices%k)
380 |     bb = mat%iops%scat
381 |     attenuation = aa + bb
382 |
383 |     call mat%add(repeat_ent, attenuation)
384 |   end subroutine attenuate
385 |
386 |   subroutine add_source(mat, indices, row_num)
387 |     ! Has to be called after angular_integral
388 |     ! Because they both write to the same matrix
    |         entry
389 |     ! And adding here is more efficient than a
    |        conditional
390 |     ! in the angular loop.
391 |     class(rte_mat) mat
392 |     type(index_list) indices
393 |     integer(index_kind) row_num
394 |     double precision source_val
395 |
396 |     source_val = mat%iops%source_grid(indices%i,
    |         indices%j, indices%k, indices%p)
397 |
398 |     call mat%add_rhs(row_num, source_val)
399 |   end subroutine add_source
400 |
401 |   subroutine x_cd2(mat, indices, ent)
402 |     class(rte_mat) mat
403 |     double precision val, dx
404 |     type(index_list) indices
405 |     integer i, j, k, p
406 |     integer(index_kind) ent
407 |
408 |     i = indices%i
409 |     j = indices%j
410 |     k = indices%k
411 |     p = indices%p
412 |
413 |     dx = mat%grid%x%spacing(1)
414 |
415 |     val = mat%grid%angles%sin_phi_p(p) &
416 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
417 |
418 |     call mat%assign(ent,-val,i-1,j,k,p)
419 |     call mat%assign(ent,val,i+1,j,k,p)
```

```
420 |   end subroutine x_cd2
421 |
422 |   subroutine x_cd2_first(mat, indices, ent)
423 |     class(rte_mat) mat
424 |     double precision val, dx
425 |     integer nx
426 |     type(index_list) indices
427 |     integer i, j, k, p
428 |     integer(index_kind) ent
429 |
430 |     i = indices%i
431 |     j = indices%j
432 |     k = indices%k
433 |     p = indices%p
434 |
435 |     dx = mat%grid%x%spacing(1)
436 |     nx = mat%grid%x%num
437 |
438 |     val = mat%grid%angles%sin_phi_p(p) &
439 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
440 |
441 |     call mat%assign(ent,-val,nx,j,k,p)
442 |     call mat%assign(ent,val,i+1,j,k,p)
443 |   end subroutine x_cd2_first
444 |
445 |   subroutine x_cd2_last(mat, indices, ent)
446 |     class(rte_mat) mat
447 |     double precision val, dx
448 |     type(index_list) indices
449 |     integer i, j, k, p
450 |     integer(index_kind) ent
451 |
452 |     i = indices%i
453 |     j = indices%j
454 |     k = indices%k
455 |     p = indices%p
456 |
457 |     dx = mat%grid%x%spacing(1)
458 |
459 |     val = mat%grid%angles%sin_phi_p(p) &
460 |          * mat%grid%angles%cos_theta_p(p) / (2.
    |             d0 * dx)
461 |
462 |     call mat%assign(ent,-val,i-1,j,k,p)
463 |     call mat%assign(ent,val,1,j,k,p)
464 |   end subroutine x_cd2_last
465 |
466 |   subroutine y_cd2(mat, indices, ent)
467 |     class(rte_mat) mat
```

```
468 |     double precision val, dy
469 |     type(index_list) indices
470 |     integer i, j, k, p
471 |     integer(index_kind) ent
472 |
473 |     i = indices%i
474 |     j = indices%j
475 |     k = indices%k
476 |     p = indices%p
477 |
478 |     dy = mat%grid%y%spacing(1)
479 |
480 |     val = mat%grid%angles%sin_phi_p(p) &
481 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
482 |
483 |     call mat%assign(ent,-val,i,j-1,k,p)
484 |     call mat%assign(ent,val,i,j+1,k,p)
485 |   end subroutine y_cd2
486 |
487 |   subroutine y_cd2_first(mat, indices, ent)
488 |     class(rte_mat) mat
489 |     double precision val, dy
490 |     integer ny
491 |     type(index_list) indices
492 |     integer i, j, k, p
493 |     integer(index_kind) ent
494 |
495 |     i = indices%i
496 |     j = indices%j
497 |     k = indices%k
498 |     p = indices%p
499 |
500 |     dy = mat%grid%y%spacing(1)
501 |     ny = mat%grid%y%num
502 |
503 |     val = mat%grid%angles%sin_phi_p(p) &
504 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
505 |
506 |     call mat%assign(ent,-val,i,ny,k,p)
507 |     call mat%assign(ent,val,i,j+1,k,p)
508 |   end subroutine y_cd2_first
509 |
510 |   subroutine y_cd2_last(mat, indices, ent)
511 |     class(rte_mat) mat
512 |     double precision val, dy
513 |     type(index_list) indices
514 |     integer i, j, k, p
515 |     integer(index_kind) ent
```

```
516 |
517 |     i = indices%i
518 |     j = indices%j
519 |     k = indices%k
520 |     p = indices%p
521 |
522 |     dy = mat%grid%y%spacing(1)
523 |
524 |     val = mat%grid%angles%sin_phi_p(p) &
525 |          * mat%grid%angles%sin_theta_p(p) / (2.
    |             d0 * dy)
526 |
527 |     call mat%assign(ent,-val,i,j-1,k,p)
528 |     call mat%assign(ent,val,i,1,k,p)
529 |   end subroutine y_cd2_last
530 |
531 |   subroutine z_cd2(mat, indices, ent)
532 |     class(rte_mat) mat
533 |     double precision val, dz
534 |     type(index_list) indices
535 |     integer i, j, k, p
536 |     integer(index_kind) ent
537 |
538 |     i = indices%i
539 |     j = indices%j
540 |     k = indices%k
541 |     p = indices%p
542 |
543 |     dz = mat%grid%z%spacing(indices%k)
544 |
545 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
    |         dz)
546 |
547 |     call mat%assign(ent,-val,i,j,k-1,p)
548 |     call mat%assign(ent,val,i,j,k+1,p)
549 |   end subroutine z_cd2
550 |
551 |   subroutine z_fd2(mat, indices, ent, repeat_ent
    |     )
552 |     ! Has to be called after angular_integral
553 |     ! Because they both write to the same matrix
    |         entry
554 |     ! And adding here is more efficient than a
    |        conditional
555 |     ! in the angular loop.
556 |     class(rte_mat) mat
557 |     double precision val, val1, val2, val3, dz
558 |     type(index_list) indices
559 |     integer i, j, k, p
560 |     integer(index_kind) ent, repeat_ent
561 |
```

```
562 |     i = indices%i
563 |     j = indices%j
564 |     k = indices%k
565 |     p = indices%p
566 |
567 |     dz = mat%grid%z%spacing(indices%k)
568 |
569 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
    |         dz)
570 |
571 |     val1 = -3.d0 * val
572 |     val2 = 4.d0 * val
573 |     val3 = -val
574 |
575 |     call mat%add(repeat_ent, val1)
576 |     call mat%assign(ent,val2,i,j,k+1,p)
577 |     call mat%assign(ent,val3,i,j,k+2,p)
578 |   end subroutine z_fd2
579 |
580 |   subroutine z_bd2(mat, indices, ent, repeat_ent
    |     )
581 |     ! Has to be called after angular_integral
582 |     ! Because they both write to the same matrix
    |       entry
583 |     ! And adding here is more efficient than a
    |       conditional
584 |     ! in the angular loop.
585 |     class(rte_mat) mat
586 |     double precision val, val1, val2, val3, dz
587 |     type(index_list) indices
588 |     integer i, j, k, p
589 |     integer(index_kind) ent, repeat_ent
590 |
591 |     i = indices%i
592 |     j = indices%j
593 |     k = indices%k
594 |     p = indices%p
595 |
596 |     dz = mat%grid%z%spacing(indices%k)
597 |
598 |     val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
    |         dz)
599 |
600 |     val1 = 3.d0 * val
601 |     val2 = -4.d0 * val
602 |     val3 = val
603 |
604 |     call mat%add(repeat_ent, val1)
605 |     call mat%assign(ent,val2,i,j,k-1,p)
606 |     call mat%assign(ent,val3,i,j,k-2,p)
607 |   end subroutine z_bd2
```

```
608 |
609 |   subroutine angular_integral(mat, indices, ent)
610 |     class(rte_mat) mat
611 |     ! Primed angular integration variables
612 |     integer pp
613 |     double precision val
614 |     type(index_list) indices
615 |     integer(index_kind) ent
616 |
617 |     do pp=1, mat%grid%angles%nomega
618 |        val = -mat%iops%scat * mat%iops%
    |           vsf_integral(indices%p, pp)
619 |        call mat%assign(ent, val, indices%i,
    |           indices%j, indices%k, pp)
620 |     end do
621 |   end subroutine angular_integral
622 |
623 |   subroutine z_surface_bc(mat, indices, row_num,
    |      ent, repeat_ent)
624 |     class(rte_mat) mat
625 |     double precision bc_val
626 |     type(index_list) indices
627 |     double precision val1, val2, dz
628 |     integer(index_kind) row_num, ent, repeat_ent
629 |
630 |     dz = mat%grid%z%spacing(1)
631 |
632 |     val1 = mat%grid%angles%cos_phi_p(indices%p)
    |        / (3.d0 * dz)
633 |     val2 = 3.d0 * val1
634 |     bc_val = 4.d0 * val1 * mat%surface_vals(
    |        indices%p)
635 |
636 |     call mat%assign(ent,val1,indices%i,indices%j
    |        ,2,indices%p)
637 |     call mat%add(repeat_ent, val2)
638 |     call mat%assign_rhs(row_num, bc_val)
639 |   end subroutine z_surface_bc
640 |
641 |     subroutine z_bottom_bc(mat, indices, ent,
    |        repeat_ent)
642 |     class(rte_mat) mat
643 |     type(index_list) indices
644 |     double precision val1, val2, dz
645 |     integer nz
646 |     integer(index_kind) ent, repeat_ent
647 |
648 |     dz = mat%grid%z%spacing(1)
649 |     nz = mat%grid%z%num
650 |
```

```
    val1 = -mat%grid%angles%cos_phi_p(indices%p)
        / (3.d0 * dz)
    val2 = 3.d0 * val1

    call mat%assign(ent,val1,indices%i,indices%j
       ,nz-1,indices%p)
    call mat%add(repeat_ent, val2)
  end subroutine z_bottom_bc

end module rte_sparse_matrices
```

kelp_context.f90

```
module kelp_context
use sag
use prob
implicit none

! Point in cylindrical coordinates
type point3d
   double precision x, y, z, theta, r
 contains
   procedure :: set_cart => point_set_cart
   procedure :: set_cyl => point_set_cyl
   procedure :: cartesian_to_polar
   procedure :: polar_to_cartesian
end type point3d

type frond_shape
  double precision fs, fr, tan_alpha, alpha, ft
contains
  procedure :: set_shape => frond_set_shape
  procedure :: calculate_angles =>
     frond_calculate_angles
end type frond_shape

type rope_state
   integer nz
   double precision, dimension(:), allocatable
      :: frond_lengths, frond_stds, num_fronds,
      water_speeds, water_angles
contains
    procedure :: init => rope_init
    procedure :: deinit => rope_deinit
end type rope_state

type depth_state
   double precision frond_length, frond_std,
      num_fronds, water_speeds, water_angles,
      depth
   integer depth_layer
contains
   procedure :: set_depth
```

```
36 |    procedure :: length_distribution_cdf
37 |    procedure :: angle_distribution_pdf
38 | end type depth_state
39 |
40 | type optical_properties
41 |    integer num_vsf
42 |    type(space_angle_grid) grid
43 |    double precision, dimension(:), allocatable
   |       :: vsf_angles, vsf_vals
44 |    double precision, dimension(:), allocatable
   |       :: abs_water
45 |    double precision abs_kelp, scat
46 |    ! On x, y, z grid - including water & kelp.
47 |    double precision, dimension(:,:,:),
   |       allocatable :: abs_grid
48 |    double precision, dimension(:,:,:,:),
   |       allocatable :: source_grid
49 |    ! On theta, phi, theta_prime, phi_prime grid
50 |    double precision, dimension(:,:), allocatable
   |        :: vsf, vsf_integral
51 |  contains
52 |    procedure :: init => iop_init
53 |    procedure :: calculate_coef_grids
54 |    procedure :: zero_source => iop_zero_source
55 |    procedure :: set_source => iop_set_source
56 |    procedure :: load_vsf
57 |    procedure :: eval_vsf
58 |    procedure :: calc_vsf_on_grid
59 |    procedure :: deinit => iop_deinit
60 |    procedure :: vsf_from_function
61 | end type optical_properties
62 |
63 | type boundary_condition
64 |    double precision I0, decay, theta_s, phi_s
65 |    type(space_angle_grid) grid
66 |    double precision, dimension(:), allocatable
   |       :: bc_grid
67 |  contains
68 |    procedure :: bc_gaussian
69 |    procedure :: init => bc_init
70 |    procedure :: deinit => bc_deinit
71 | end type boundary_condition
72 |
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,
   |         bc%phi_s)
80 |     bc_gaussian = exp(-bc%decay * diff)
```

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

```
125 |
126 |   subroutine point_set_cart(point, x, y, z)
127 |     class(point3d) :: point
128 |     double precision x, y, z
129 |     point%x = x
130 |     point%y = y
131 |     point%z = z
132 |     call point%cartesian_to_polar()
133 |   end subroutine point_set_cart
134 |
135 |   subroutine point_set_cyl(point, theta, r, z)
136 |     class(point3d) :: point
137 |     double precision theta, r, z
138 |     point%theta = theta
139 |     point%r = r
140 |     point%z = z
141 |     call point%polar_to_cartesian()
142 |   end subroutine point_set_cyl
143 |
144 |   subroutine polar_to_cartesian(point)
145 |     class(point3d) :: point
146 |     point%x = point%r*cos(point%theta)
147 |     point%y = point%r*sin(point%theta)
148 |   end subroutine polar_to_cartesian
149 |
150 |   subroutine cartesian_to_polar(point)
151 |     class(point3d) :: point
152 |     point%r = sqrt(point%x**2 + point%y**2)
153 |     point%theta = atan2(point%y, point%x)
154 |   end subroutine cartesian_to_polar
155 |
156 |   subroutine frond_set_shape(frond, fs, fr, ft)
157 |     class(frond_shape) frond
158 |     double precision fs, fr, ft
159 |     frond%fs = fs
160 |     frond%fr = fr
161 |     frond%ft = ft
162 |     call frond%calculate_angles()
163 |   end subroutine frond_set_shape
164 |
165 |   subroutine frond_calculate_angles(frond)
166 |     class(frond_shape) frond
167 |     frond%tan_alpha = 2.d0*frond%fs*frond%fr /
    |        (1.d0 + frond%fs)
168 |     frond%alpha = atan(frond%tan_alpha)
169 |   end subroutine
170 |
171 |   subroutine iop_init(iops, grid)
172 |     class(optical_properties) iops
173 |     type(space_angle_grid) grid
```

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

```
216     end do
217
218   end subroutine calculate_coef_grids
219
220
221   subroutine load_vsf(iops, filename, fmtstr)
222     class(optical_properties) :: iops
223     character(len=*) :: filename, fmtstr
224     double precision, dimension(:,:),
          allocatable :: tmp_2d_arr
225     integer num_rows, num_cols, skiplines_in
226
227     ! First column is the angle at which the
          measurement is taken
228     ! Second column is the value of the VSF at
          that angle
229     num_rows = iops%num_vsf
230     num_cols = 2
231     skiplines_in = 1 ! Ignore comment on first
          line
232
233     allocate(tmp_2d_arr(num_rows, num_cols))
234
235     tmp_2d_arr = read_array(filename, fmtstr,
          num_rows, num_cols, skiplines_in)
236     iops%vsf_angles = tmp_2d_arr(:,1)
237     iops%vsf_vals = tmp_2d_arr(:,2)
238
239     ! write(*,*) 'vsf_angles = ', iops%
          vsf_angles
240     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
241
242     ! Pre-evaluate for all pair of angles
243     call iops%calc_vsf_on_grid()
244   end subroutine load_vsf
245
246   function eval_vsf(iops, theta)
247     class(optical_properties) iops
248     double precision theta
249     double precision eval_vsf
250     ! No need to set vsf(0) = 0.
251     ! It's the area under the curve that matters
          , not the value.
252     eval_vsf = interp(theta, iops%vsf_angles,
          iops%vsf_vals, iops%num_vsf)
253
254   end function eval_vsf
255
256   subroutine rope_init(rope, grid)
257     class(rope_state) :: rope
258     type(space_angle_grid) :: grid
```

```
259 |
260 |     rope%nz = grid%z%num
261 |     allocate(rope%frond_lengths(rope%nz))
262 |     allocate(rope%frond_stds(rope%nz))
263 |     allocate(rope%water_speeds(rope%nz))
264 |     allocate(rope%water_angles(rope%nz))
265 |     allocate(rope%num_fronds(rope%nz))
266 |   end subroutine rope_init
267 |
268 |   subroutine rope_deinit(rope)
269 |     class(rope_state) rope
270 |     deallocate(rope%frond_lengths)
271 |     deallocate(rope%frond_stds)
272 |     deallocate(rope%water_speeds)
273 |     deallocate(rope%water_angles)
274 |     deallocate(rope%num_fronds)
275 |   end subroutine rope_deinit
276 |
277 |   subroutine set_depth(depth, rope, grid,
    |      depth_layer)
278 |     class(depth_state) depth
279 |     type(rope_state) rope
280 |     type(space_angle_grid) grid
281 |     integer depth_layer
282 |
283 |     depth%frond_length = rope%frond_lengths(
    |        depth_layer)
284 |     depth%frond_std = rope%frond_stds(
    |        depth_layer)
285 |     depth%water_speeds = rope%water_speeds(
    |        depth_layer)
286 |     depth%water_angles = rope%water_angles(
    |        depth_layer)
287 |     depth%num_fronds = rope%num_fronds(
    |        depth_layer)
288 |     depth%depth_layer = depth_layer
289 |     depth%depth = grid%z%vals(depth_layer)
290 |   end subroutine set_depth
291 |
292 |   function length_distribution_cdf(depth, L)
    |      result(output)
293 |     ! C_L(L)
294 |     class(depth_state) depth
295 |     double precision L, L_mean, L_std
296 |     double precision output
297 |
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
          ) result(output)
305     ! P_{\theta_f}(\theta_f)
306     class(depth_state) depth
307     double precision theta_f, v_w, theta_w
308     double precision output
309     double precision diff
310
311     v_w = depth%water_speeds
312     theta_w = depth%water_angles
313
314     ! von_mises_pdf is only defined on [-pi, pi]
315     ! So take difference of angles and input
            into
316     ! von_mises dist. centered & x=0.
317
318     diff = angle_diff_2d(theta_f, theta_w)
319
320     call von_mises_pdf(diff, 0.d0, v_w, output)
321   end function angle_distribution_pdf
322
323   function angle_mod(theta) result(mod_theta)
324     ! Shift theta to the interval [-pi, pi]
325     ! which is where von_mises_pdf is defined.
326
327     double precision theta, mod_theta
328
329     mod_theta = mod(theta + pi, 2.d0*pi) - pi
330   end function angle_mod
331
332   function angle_diff_2d(theta1, theta2) result(
          diff)
333     ! Shortest difference between two angles
            which may be
334     ! in different periods.
335     double precision theta1, theta2, diff
336     double precision modt1, modt2
337
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
    |      , phi_prime) result(diff)
348 |     ! Angle between two vectors in spherical
    |        coordinates
349 |     double precision theta, phi, theta_prime,
    |        phi_prime
350 |     double precision alpha, diff
351 |
352 |     ! Faster, but produces lots of NaNs (at
    |        least in Python)
353 |     !alpha = sin(theta)*sin(theta_prime)*cos(
    |        theta-theta_prime) + cos(phi)*cos(
    |        phi_prime)
354 |
355 |
356 |     ! Slower, but more accurate
357 |     alpha = (sin(phi)*sin(phi_prime) &
358 |       * (cos(theta)*cos(theta_prime) + sin(theta
    |          )*sin(theta_prime)) &
359 |       + cos(phi)*cos(phi_prime))
360 |
361 |     ! Avoid out-of-bounds errors due to rounding
362 |     alpha = min(1.d0, alpha)
363 |     alpha = max(-1.d0, alpha)
364 |
365 |     diff = acos(alpha)
366 |   end function angle_diff_3d
367 |
368 |   subroutine vsf_from_function(iops, func)
369 |     class(optical_properties) iops
370 |     double precision, external :: func
371 |     integer i
372 |     type(angle_dim) :: angle1d
373 |
374 |     call angle1d%set_bounds(-1.d0, 1.d0)
375 |     call angle1d%set_num(iops%num_vsf)
376 |     call angle1d%assign_legendre()
377 |
378 |     iops%vsf_angles(:) = acos(angle1d%vals(:))
379 |     do i=1, iops%num_vsf
380 |        iops%vsf_vals(i) = func(iops%vsf_angles(i
    |           ))
381 |     end do
382 |
383 |     call iops%calc_vsf_on_grid()
384 |
385 |     call angle1d%deinit()
386 |   end subroutine vsf_from_function
387 |
388 |   subroutine calc_vsf_on_grid(iops)
389 |     class(optical_properties) iops
```

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

```
module light_context
! Use 64-bit integers for LIS
! Necessary for FD solution w/ large matrices
#define LONG__LONG
#include "lisf.h"
  use sag
  use rte_sparse_matrices
  !use hdf5
  implicit none

  type light_state
     double precision, dimension(:,:,:),
        allocatable :: irradiance
     double precision, dimension(:,:,:,:),
        allocatable :: radiance
     type(space_angle_grid) :: grid
     type(rte_mat) :: mat
   contains
     procedure :: init => light_init
     procedure :: init_grid => light_init_grid
     procedure :: calculate_radiance
     procedure :: calculate_irradiance =>
        light_calculate_irradiance
     procedure :: deinit => light_deinit
     !procedure :: to_hdf => light_to_hdf
  end type light_state

contains

  ! Init for use with mat
  subroutine light_init(light, mat)
    class(light_state) light
    type(rte_mat) mat
    integer nx, ny, nz, nomega

    light%mat = mat
    light%grid = mat%grid

    nx = light%grid%x%num
    ny = light%grid%y%num
    nz = light%grid%z%num
    nomega = light%grid%angles%nomega

    allocate(light%irradiance(nx, ny, nz))
    allocate(light%radiance(nx, ny, nz, nomega))
  end subroutine light_init

  ! Init for use without mat
  subroutine light_init_grid(light, grid)
    class(light_state) light
    type(space_angle_grid) grid
    integer nx, ny, nz, nomega
```

```
50  |
51  |     light%grid = grid
52  |
53  |     nx = light%grid%x%num
54  |     ny = light%grid%y%num
55  |     nz = light%grid%z%num
56  |     nomega = light%grid%angles%nomega
57  |
58  |     allocate(light%irradiance(nx, ny, nz))
59  |     allocate(light%radiance(nx, ny, nz, nomega))
60  |   end subroutine light_init_grid
61  |
62  |   subroutine calculate_radiance(light)
63  |     class(light_state) light
64  |     integer i, j, k, p
65  |     integer nx, ny, nz, nomega
66  |     integer(index_kind) index
67  |     LIS_INTEGER lis_test_int
68  |
69  |     nx = light%grid%x%num
70  |     ny = light%grid%y%num
71  |     nz = light%grid%z%num
72  |     nomega = light%grid%angles%nomega
73  |
74  |     ! call lis_vector_get_size(light%mat%x, ln,
    |        gn)
75  |
76  |     ! write(*,*) 'ln =', ln
77  |     ! write(*,*) 'gn =', gn
78  |
79  |     index = 1
80  |
81  |     write(*,*) 'Set matrix values'
82  |     write(*,*) 'index_kind = ', storage_size(
    |        index)
83  |     write(*,*) 'lis_kind = ', storage_size(
    |        lis_test_int)
84  |     ! Set initial guess from provided radiance
85  |     ! Traverse solution vector in order
86  |     ! so as to avoid calculating index
87  |     do k=1, nz
88  |        do i=1, nx
89  |            do j=1, ny
90  |              do p=1, nomega
91  |                 call lis_vector_set_value(
    |                    LIS_INS_VALUE, index, &
92  |                      light%radiance(i,j,k,p),
    |                         light%mat%x, light%mat%
    |                         ierr)
93  |                 if(light%mat%ierr .ne. 0) then
```

```
94                      write(*,*) 'IG ERROR:', light
                           %mat%ierr
95                   end if
96
97                   index = index + 1
98                end do
99             end do
100         end do
101      end do
102
103      !call light%mat%initial_guess()
104
105      ! Solve (LIS)
106      write(*,*) 'Solve matrix'
107      call light%mat%solve()
108
109      index = 1
110
111      write(*,*) 'Extract solution'
112      ! Extract solution
113      do k=1, nz
114         do i=1, nx
115            do j=1, ny
116               do p=1, nomega
117                  call lis_vector_get_value(light%
                        mat%x, index, &
118                        light%radiance(i,j,k,p),
                           light%mat%ierr)
119                  if(light%mat%ierr .ne. 0) then
120                     write(*,*) 'EXTRACT ERROR:',
                           light%mat%ierr
121                  end if
122                  index = index + 1
123               end do
124            end do
125         end do
126      end do
127   end subroutine calculate_radiance
128
129   subroutine light_calculate_irradiance(light)
130      class(light_state) light
131      integer i, j, k
132      integer nx, ny, nz
133      double precision, dimension(light%grid%
            angles%nomega) :: tmp_rad
134
135      nx = light%grid%x%num
136      ny = light%grid%y%num
137      nz = light%grid%z%num
138
139      do i=1, nx
```

```
140         do j=1, ny
141            do k=1, nz
142               ! Use temporary array to avoid
                     creating one
143               ! implicitly at every spatial grid
                     point
144               tmp_rad = light%radiance(i,j,k,:)
145               light%irradiance(i,j,k) = &
146                    light%grid%angles%
                        integrate_points(tmp_rad)
147            end do
148         end do
149      end do
150
151    end subroutine light_calculate_irradiance
152
153 !  subroutine light_to_hdf(light, radfile,
       irradfile)
154 !    class(light_state) light
155 !    character(len=*) radfile
156 !    character(len=*) irradfile
157 !
158 !    call hdf_write_radiance(radfile, light%
       radiance, light%grid)
159 !    call hdf_write_irradiance(irradfile, light%
       irradiance, light%grid)
160 !  end subroutine light_to_hdf
161
162    subroutine light_deinit(light)
163      class(light_state) light
164
165      deallocate(light%irradiance)
166      deallocate(light%radiance)
167    end subroutine light_deinit
168 end module
```

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,
          ddx, ddy)
17      use rte_sparse_matrices
18      import deriv_interface
19      type(space_angle_grid) grid
20      type(rte_mat) mat
21      type(index_list) indices
22      procedure(deriv_interface) :: ddx, ddy
23    end subroutine angle_loop_interface
24 end interface
25
26 contains
27
28 subroutine whole_space_loop(mat, indices,
       num_threads)
29   type(rte_mat) mat
30   type(index_list) indices
31   integer i, j, k
32   integer num_threads
33
34   procedure(deriv_interface), pointer :: ddx,
       ddy
35   procedure(angle_loop_interface), pointer ::
       angle_loop
36
37   !$omp parallel do default(none) shared(mat) &
38   !$omp private(ddx,ddy,angle_loop, k, i, j)
       private(indices) &
39   !$omp num_threads(num_threads) collapse(3)
40   do k=1, mat%grid%z%num
41      do i=1, mat%grid%x%num
42         do j=1, mat%grid%y%num
43            indices%k = k
44            if(k .eq. 1) then
45               angle_loop => surface_angle_loop
46            else if(k .eq. mat%grid%z%num) then
47               angle_loop => bottom_angle_loop
48            else
49               angle_loop => interior_angle_loop
50            end if
51
52            indices%i = i
53            if(indices%i .eq. 1) then
54               ddx => x_cd2_first
55            else if(indices%i .eq. mat%grid%x%num
                 ) then
56               ddx => x_cd2_last
57            else
58               ddx => x_cd2
```

```
59 |             end if
60 |
61 |             indices%j = j
62 |             if(indices%j .eq. 1) then
63 |                ddy => y_cd2_first
64 |             else if(indices%j .eq. mat%grid%y%num
   |                ) then
65 |                ddy => y_cd2_last
66 |             else
67 |                ddy => y_cd2
68 |             end if
69 |
70 |             call angle_loop(mat, indices, ddx,
   |                ddy)
71 |          end do
72 |       end do
73 |    end do
74 |    !$omp end parallel do
75 | end subroutine whole_space_loop
76 |
77 | function calculate_start_ent(grid, indices)
   |    result(ent)
78 |    type(space_angle_grid) grid
79 |    type(index_list) indices
80 |    integer(index_kind) ent
81 |    integer(index_kind) boundary_nnz, interior_nnz
82 |    integer(index_kind) num_boundary, num_interior
83 |    integer(index_kind) num_this_x, num_this_z
84 |
85 |    ! Nonzero matrix entries for an surface or
   |       bottom spatial grid cell
86 |    ! Definitely an integer since nomega is even
87 |    boundary_nnz = grid%angles%nomega * (2 * grid%
   |       angles%nomega + 11) / 2
88 |    ! Nonzero matrix entries for an interior
   |       spatial grid cell
89 |    interior_nnz = grid%angles%nomega * (grid%
   |       angles%nomega + 6)
90 |
91 |    ! Order: z, x, y, omega
92 |    ! Total number traversed so far in each
   |       spatial category
93 |    ! row
94 |    num_this_x = indices%j - 1
95 |    ! depth layer
96 |    num_this_z = (indices%i - 1) * grid%y%num +
   |       num_this_x
97 |
98 |    ! Calculate number of spatial grid cells of
   |       each type which have
99 |    ! already been traversed up to this point
```

```
100 |   if(indices%k .eq. 1) then
101 |      num_boundary = num_this_z
102 |      num_interior = 0
103 |   else if(indices%k .eq. grid%z%num) then
104 |      num_boundary = (grid%x%num * grid%y%num) +
    |         num_this_z
105 |      num_interior = (grid%z%num-2) * grid%x%num
    |         * grid%y%num
106 |   else
107 |      num_boundary = grid%x%num * grid%y%num
108 |      num_interior = num_this_z + (indices%k-2) *
    |          grid%x%num * grid%y%num
109 |   end if
110 |
111 |   ent = num_boundary * boundary_nnz +
    |      num_interior * interior_nnz + 1
112 | end function calculate_start_ent
113 |
114 | function calculate_repeat_ent(ent, p) result(
    |    repeat_ent)
115 |   integer p
116 |   integer(index_kind) ent, repeat_ent
117 |   ! Entry number for row=mat%ind(i,j,k,p), col=
    |      mat%ind(i,j,k,p),
118 |   ! which will be modified multiple times in
    |      this matrix row
119 |   repeat_ent = ent + p - 1
120 | end function calculate_repeat_ent
121 |
122 | subroutine interior_angle_loop(mat, indices, ddx
    |    , ddy)
123 |   type(rte_mat) mat
124 |   type(index_list) indices
125 |   procedure(deriv_interface) :: ddx, ddy
126 |   integer p
127 |   integer(index_kind) ent, repeat_ent
128 |   integer(index_kind) row_num
129 |
130 |   ! Determine which matrix row to start at
131 |   ent = calculate_start_ent(mat%grid, indices)
132 |   indices%p = 1
133 |   row_num = mat%ind(indices%i, indices%j,
    |      indices%k, indices%p)
134 |
135 |   do p=1, mat%grid%angles%nomega
136 |      indices%p = p
137 |      repeat_ent = calculate_repeat_ent(ent, p)
138 |      call mat%set_row(ent, row_num)
139 |      call mat%angular_integral(indices, ent)
140 |      call ddx(mat, indices, ent)
```

```
141 |       call ddy(mat, indices, ent)
142 |       call mat%z_cd2(indices, ent)
143 |       call mat%attenuate(indices, repeat_ent)
144 |       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,
    |    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,
    |      indices%k, indices%p)
161 |
162 |   ! Downwelling
163 |   do p=1, mat%grid%angles%nomega / 2
164 |       indices%p = p
165 |       repeat_ent = calculate_repeat_ent(ent, p)
166 |       call mat%set_row(ent, row_num)
167 |       call mat%angular_integral(indices, ent)
168 |       call ddx(mat, indices, ent)
169 |       call ddy(mat, indices, ent)
170 |       call mat%z_surface_bc(indices, row_num, ent
    |          , repeat_ent)
171 |       call mat%attenuate(indices, repeat_ent)
172 |       call mat%add_source(indices, row_num)
173 |       row_num = row_num + 1
174 |   end do
175 |   ! Upwelling
176 |   do p=mat%grid%angles%nomega/2+1, mat%grid%
    |      angles%nomega
177 |       indices%p = p
178 |       repeat_ent = calculate_repeat_ent(ent, p)
179 |       call mat%set_row(ent, row_num)
180 |       call mat%angular_integral(indices, ent)
181 |       call ddx(mat, indices, ent)
182 |       call ddy(mat, indices, ent)
183 |       call mat%z_fd2(indices, ent, repeat_ent)
184 |       call mat%attenuate(indices, repeat_ent)
185 |       call mat%add_source(indices, row_num)
186 |       row_num = row_num + 1
```

```
187 |   end do
188 | end subroutine surface_angle_loop
189 |
190 | subroutine bottom_angle_loop(mat, indices, ddx,
    |    ddy)
191 |   type(rte_mat) mat
192 |   type(index_list) indices
193 |   integer p
194 |   integer(index_kind) row_num, ent, repeat_ent
195 |   procedure(deriv_interface) :: ddx, ddy
196 |
197 |   ! Determine which matrix row to start at
198 |   ent = calculate_start_ent(mat%grid, indices)
199 |   indices%p = 1
200 |   row_num = mat%ind(indices%i, indices%j,
    |      indices%k, indices%p)
201 |
202 |   ! Downwelling
203 |   do p=1, mat%grid%angles%nomega/2
204 |      indices%p = p
205 |      repeat_ent = calculate_repeat_ent(ent, p)
206 |      call mat%set_row(ent, row_num)
207 |      call mat%angular_integral(indices, ent)
208 |      call ddx(mat, indices, ent)
209 |      call ddy(mat, indices, ent)
210 |      call mat%z_bd2(indices, ent, repeat_ent)
211 |      call mat%attenuate(indices, repeat_ent)
212 |      call mat%add_source(indices, row_num)
213 |      row_num = row_num + 1
214 |   end do
215 |   ! Upwelling
216 |   do p=mat%grid%angles%nomega/2+1, mat%grid%
    |      angles%nomega
217 |      indices%p = p
218 |      repeat_ent = calculate_repeat_ent(ent, p)
219 |      call mat%set_row(ent, row_num)
220 |      call mat%angular_integral(indices, ent)
221 |      call ddx(mat, indices, ent)
222 |      call ddy(mat, indices, ent)
223 |      call mat%z_bottom_bc(indices, ent,
    |         repeat_ent)
224 |      call mat%attenuate(indices, repeat_ent)
225 |      call mat%add_source(indices, row_num)
226 |      row_num = row_num + 1
227 |   end do
228 | end subroutine bottom_angle_loop
229 |
230 | subroutine gen_matrix(mat, num_threads)
231 |   type(rte_mat) mat
232 |   type(index_list) indices
```

```
233   integer num_threads
234
235   call indices%init()
236
237   call whole_space_loop(mat, indices,
          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
```

kelp3d.f90

```
1  ! Kelp 3D
2  ! Oliver Evans
3  ! 8/31/2017
4
5  ! Given superindividual/water current data at
       each depth, generate kelp distribution at
       each point in 3D space
6
7  module kelp3d
8
9  use kelp_context
10
11 implicit none
12
13 contains
14
15 subroutine generate_grid(xmin, xmax, nx, ymin,
       ymax, ny, zmin, zmax, nz, ntheta, nphi, grid,
        p_kelp)
16   double precision xmin, xmax, ymin, ymax, zmin,
         zmax
17   integer nx, ny, nz, ntheta, nphi
18   type(space_angle_grid) grid
19   double precision, dimension(:,:,:),
       allocatable :: p_kelp
20
```

```
21 |   call grid%set_bounds(xmin, xmax, ymin, ymax,
   |     zmin, zmax)
22 |   call grid%set_num(nx, ny, nz, ntheta, nphi)
23 |
24 |   allocate(p_kelp(nx,ny,nz))
25 |
26 | end subroutine generate_grid
27 |
28 | subroutine kelp3d_deinit(grid, rope, p_kelp)
29 |   type(space_angle_grid) grid
30 |   type(rope_state) rope
31 |   double precision, dimension(:,:,:),
   |     allocatable :: p_kelp
32 |   call rope%deinit()
33 |   call grid%deinit()
34 |   deallocate(p_kelp)
35 | end subroutine kelp3d_deinit
36 |
37 | subroutine calculate_kelp_on_grid(grid, p_kelp,
   |    frond, rope, quadrature_degree, n_images,
   |    num_threads)
38 |   type(space_angle_grid), intent(in) :: grid
39 |   type(frond_shape), intent(in) :: frond
40 |   type(rope_state), intent(in) :: rope
41 |   type(point3d) point
42 |   integer, intent(in) :: quadrature_degree
43 |   integer, optional :: n_images
44 |   double precision, dimension(grid%x%num, grid%y
   |     %num, grid%z%num) :: p_kelp
45 |   type(depth_state) depth
46 |   integer num_threads
47 |
48 |   integer i, j, k, nx, ny, nz
49 |   double precision x, y, z
50 |   ! Number of periodic images
51 |   ! to consider in each horizontal direction
52 |   ! for kelp distribution
53 |   ! n_images=1 => 3x3 meta-grid
54 |   ! n_images=2 => 5x5 meta-grid (only necessary
   |     for very dense kelp ropes)
55 |   integer im_i, im_j
56 |   double precision x_width, y_width
57 |
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 |   !$omp parallel do default(shared) private(x,y,
   |     z) &
72 |   !$omp firstprivate(point,depth) &
73 |   !$omp private(i,j,k,im_i,im_j) shared(nx,ny,nz
   |     ,n_images) &
74 |   !$omp shared(frond,rope,grid,quadrature_degree
   |     ) &
75 |   !$omp shared(p_kelp,x_width,y_width) &
76 |   !$omp num_threads(num_threads) collapse(3) &
77 |   !$omp schedule(dynamic, 10) ! 10 grid points
   |     per thread
78 |   do k=1, nz
79 |     do i=1, nx
80 |       do j=1, ny
81 |         z = grid%z%vals(k)
82 |         call depth%set_depth(rope, grid, k)
83 |         do im_i=-n_images, n_images
84 |           x = im_i*x_width + grid%x%vals(i)
85 |           do im_j=-n_images, n_images
86 |             y = im_j*y_width + grid%y%vals(j)
87 |             call point%set_cart(x, y, z)
88 |             p_kelp(i, j, k) = p_kelp(i,j,k) &
89 |                   + kelp_proportion(point, frond
   |                      , grid, depth,
   |                      quadrature_degree)
90 |          end do
91 |         end do
92 |       end do
93 |     end do
94 |   end do
95 |   !omp end do
96 | end subroutine calculate_kelp_on_grid
97 |
98 | subroutine shading_region_limits(theta_low_lim,
   |    theta_high_lim, point, frond)
99 |   type(point3d), intent(in) :: point
100|   type(frond_shape), intent(in) :: frond
101|   double precision, intent(out) :: theta_low_lim
   |     , theta_high_lim
102|
103|   theta_low_lim = point%theta - frond%alpha
104|   theta_high_lim = point%theta + frond%alpha
105| end subroutine shading_region_limits
106|
```

```
107 |function prob_kelp(point, frond, depth,
    |   quadrature_degree)
108 |! P_s(theta_p, r_p) - This is the proportion of
    |   the population of this depth layer which can
    |   be found in this Cartesian grid cell.
109 |  type(point3d), intent(in) :: point
110 |  type(frond_shape), intent(in) :: frond
111 |  type(depth_state), intent(in) :: depth
112 |  integer, intent(in) :: quadrature_degree
113 |  double precision prob_kelp
114 |  double precision theta_low_lim, theta_high_lim
115 |
116 |  call shading_region_limits(theta_low_lim,
    |     theta_high_lim, point, frond)
117 |  prob_kelp = integrate_ps(theta_low_lim,
    |     theta_high_lim, quadrature_degree, point,
    |     frond, depth)
118 |end function prob_kelp
119 |
120 |function kelp_proportion(point, frond, grid,
    |   depth, quadrature_degree)
121 |  ! This is the proportion of the volume of the
    |     Cartesian grid cell occupied by kelp
122 |  type(point3d), intent(in) :: point
123 |  type(frond_shape), intent(in) :: frond
124 |  type(depth_state), intent(in) :: depth
125 |  type(space_angle_grid), intent(in) :: grid
126 |  integer, intent(in) :: quadrature_degree
127 |  double precision p_k, n, t, dz
128 |  double precision kelp_proportion
129 |
130 |  n = depth%num_fronds
131 |  dz = grid%z%spacing(depth%depth_layer)
132 |  t = frond%ft
133 |  !write(*,*) 'KELP PROPORTION'
134 |  !write(*,*) 'n=', n
135 |  !write(*,*) 'dz=', dz
136 |  !write(*,*) 't=', t
137 |  !write(*,*) 'coef=', n*t/dz
138 |  p_k = prob_kelp(point, frond, depth,
    |     quadrature_degree)
139 |  kelp_proportion = n*t/dz * p_k
140 |end function kelp_proportion
141 |
142 |function integrate_ps(theta_low_lim,
    |   theta_high_lim, quadrature_degree, point,
    |   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,
    |       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,
    |      theta_high_lim)
154 |   call theta_f%set_num(quadrature_degree)
155 |   call theta_f%assign_legendre()
156 |
157 |   allocate(integrand_vals(theta_f%num))
158 |
159 |   do i=1, theta_f%num
160 |     integrand_vals(i) = ps_integrand(theta_f%
    |        vals(i), point, frond, depth)
161 |   end do
162 |
163 |   integral = theta_f%integrate_points(
    |      integrand_vals)
164 |
165 |   deallocate(integrand_vals)
166 |   call theta_f%deinit()
167 |
168 | end function integrate_ps
169 |
170 | function ps_integrand(theta_f, point, frond,
    |    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,
    |      frond)
179 |
180 |   angular_part = depth%angle_distribution_pdf(
    |      theta_f)
181 |   length_part =  1 - depth%
    |      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,
    |   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%
    |      fs)
198 |   l_min = point%r * (sin(tpp) + angular_sign(tpp
    |      ) * 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,
    |   theta_f, L, fs, fr
204 !   double precision, intent(out) :: frond_edge
205 !
206 !   frond_edge = relative_frond_edge(theta -
    |   theta_f + pi/2.d0)
207 !
208 ! end function frond_edge
209 !
210 ! function relative_frond_edge(theta_prime, L,
    |   fs, fr)
211 ! ! r_f'(\theta')
212 !   double precision, intent(in) :: theta_prime,
    |    L, fs, fr
213 !   double precision, intent(out) ::
    |   relative_frond_edge
214 !
215 !   relative_frond_edge = L / (sin(theta_prime)
    |   + angular_sign(theta_prime * alpha(fs, fr) *
    |   cos(theta_prime)))
216 ! end function relative_frond_edge
217 |
218 function angular_sign(theta_prime)
219 ! S(\theta')
220 |   double precision, intent(in) :: theta_prime
221 |   double precision angular_sign
222 |
223 |   ! This seems to be incorrect in summary.pdf as
    |      of 9/9/18
```

```
224 |   ! In the report, it's written as sgn(
    |      theta_print - pi/2.d0)
225 |   ! This results in L_min < 0 - not good!
226 |   angular_sign = sgn(pi/2.d0 - theta_prime)
227 | end function angular_sign
228 |
229 | subroutine gaussian_blur_2d(A, sigma, dx, dy, nk
    |    , num_threads)
230 |   ! 2D Gaussian blur (periodic BC) with std
    |      sigma
231 |   ! with kernel radius of nk (full size (2*nk+1)
    |      x(2*nk+1))
232 |   ! applied to matrix A with element spacings dx
    |       and dy.
233 |   double precision, intent(inout), dimension(:,
    |      :) :: A
234 |   double precision, intent(in) :: sigma, dx, dy
235 |   ! kernel half width
236 |   integer, intent(in) :: nk
237 |   ! kernel full width
238 |   integer kw
239 |   integer num_threads
240 |
241 |   ! A matrix size
242 |   integer nx, ny
243 |
244 |   ! indices
245 |   integer i1, j1
246 |   integer i2, j2
247 |   integer i, j
248 |   ! kernel
249 |   double precision, dimension(:,:), allocatable
    |      :: k
250 |   ! output matrix
251 |   double precision, dimension(:,:), allocatable
    |      :: B
252 |   ! kernel independent variables
253 |   double precision x, y
254 |
255 |   if(sigma > 0) then
256 |     nx = size(A, 1)
257 |     ny = size(A, 2)
258 |
259 |     kw = 2*nk + 1
260 |
261 |     allocate(B(nx, ny))
262 |     allocate(k(kw, kw))
263 |     !write(*,*) 'creating kernel', sigma, nk
264 |     ! Create kernel
265 |     do i1=-nk, nk
266 |       x = i1*dx
267 |       i = i1+nk+1
```

```
      do j1=-nk, nk
          y = j1*dy
          j = j1+nk+1
          k(i,j) = exp(-(x**2+y**2)/(2*sigma**2)
             )
      end do

    end do
    ! normalize kernel
    k = k / sum(k)

    !write(*,*) 'convolving'
    ! convolve
    !$omp parallel do default(private) private(x
       ,y) &
    !$omp private(i,j,i1,j1,i2,j2) shared(nx,ny,
       nk,kw) &
    !$omp shared(A,B,k) &
    !$omp num_threads(num_threads) collapse(2) &
    !$omp schedule(dynamic, 10) ! 10 grid points
        per thread
    do i1=1, nx
      do j1=1, ny
          B(i1, j1) = 0
          do i2=1, kw
            do j2=1, kw
                i = mod1(i1 - nk + i2 - 1, nx)
                j = mod1(j1 - nk + j2 - 1, ny)
                B(i1, j1) = B(i1, j1) + k(i2, j2
                   ) * A(i, j)
            end do
          end do
      end do
    end do
    !omp end parallel do
    !write(*,*) 'done convolving'

    ! Update original matrix
    A(:,:) = B(:,:)
    deallocate(k)
    deallocate(B)
    !write(*,*) 'gb2d done.'
  end if
end subroutine gaussian_blur_2d

end module kelp3d
```

sag.f90

```
module sag
use utils
```

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

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

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

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

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

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

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

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

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

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

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

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

```
!  subroutine set_num_from_spacing(space)
!    class(space_dim) :: space
!    !space%num = num_from_spacing(space%minval
   , space%maxval, space%spacing)

!  end subroutine set_num_from_spacing

 subroutine space_deinit(space)
   class(space_dim) :: space
   deallocate(space%vals)
   deallocate(space%edges)
   deallocate(space%spacing)
 end subroutine space_deinit

 !! SAG !!

 subroutine sag_set_bounds(grid, xmin, xmax,
    ymin, ymax, zmin, zmax)
   class(space_angle_grid) :: grid
   double precision xmin, xmax, ymin, ymax,
      zmin, zmax

   call grid%x%set_bounds(xmin, xmax)
   call grid%y%set_bounds(ymin, ymax)
   call grid%z%set_bounds(zmin, zmax)
 end subroutine sag_set_bounds

 subroutine sag_set_uniform_spacing(grid, dx,
    dy, dz)
   class(space_angle_grid) :: grid
   double precision dx, dy, dz
   call grid%x%set_uniform_spacing(dx)
   call grid%y%set_uniform_spacing(dy)
   call grid%z%set_uniform_spacing(dz)
 end subroutine sag_set_uniform_spacing

 subroutine sag_set_num(grid, nx, ny, nz,
    ntheta, nphi)
   class(space_angle_grid) :: grid
   integer nx, ny, nz, ntheta, nphi
   call grid%x%set_num(nx)
   call grid%y%set_num(ny)
   call grid%z%set_num(nz)
   call grid%angles%set_num(ntheta, nphi)
 end subroutine sag_set_num

 subroutine sag_init(grid)
   class(space_angle_grid) :: grid

   call grid%x%assign_linspace()
```

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

```
598 |   subroutine sag_deinit(grid)
599 |     class(space_angle_grid) :: grid
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
    |      , ymax]
610 |   subroutine affine_transform(x, xmin, xmax,
    |      ymin, ymax)
611 |     double precision x, xmin, xmax, ymin, ymax
612 |     x = ymin + (ymax-ymin)/(xmax-xmin) * (x-xmin
    |       )
613 |   end subroutine affine_transform
614 |
615 |   function num_from_spacing(xmin, xmax, dx)
    |      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)
    |      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
```

utils.f90

```
1  | ! General utilities which might be useful in
   |    other settings
2  | module utils
3  | implicit none
4  |
5  | ! Constants
6  | double precision, parameter :: pi = 4.D0 * datan
   |    (1.D0)
7  |
8  | contains
9  |
10 | ! Determine base directory relative to current
   |    directory
```

```
11 ! by looking for Makefile, which is in the base
       dir
12 ! Assuming that this is executed from within the
       git repo.
13 function getbasedir()
14     implicit none
15
16     ! INPUTS:
17     ! Number of paths to check
18     integer, parameter :: numpaths = 3
19     ! Maximum length of path names
20     integer, parameter :: maxlength = numpaths *
           2 - 1
21     ! Paths to check for Makefile
22     character(len=maxlength), parameter,
          dimension(numpaths) :: check_paths &
23             = (/ '.    ', '..   ', '../..' /)
24     ! Temporary path string
25     character(len=maxlength) tmp_path
26     ! Whether Makefile has been found yet
27     logical found
28     ! Path counter
29     integer ii
30     ! Lengths of paths
31     integer, dimension(numpaths) ::  pathlengths
32
33     ! OUTPUT:
34     ! getbasedir - relative path to base
          directory
35     ! Will either return '.', '..', or '../..'
36     character(len=maxlength) getbasedir
37
38
39     ! Determine length of each path
40     pathlengths(1) = 1
41     do ii = 2, numpaths
42         pathlengths(ii) = 2 + 3 * (ii - 2)
43     end do
44
45     ! Loop through paths
46     do ii = 1, numpaths
47         ! Determine this path
48         tmp_path = check_paths(ii)
49
50         ! Check whether Makefile is in this
              directory
51         !write(*,*) 'Checking "', tmp_path(1:
              pathlengths(ii)), '"'
52         inquire(file=tmp_path(1:pathlengths(ii))
               // '/Makefile', exist=found)
53         ! If so, stop. Otherwise, keep looking.
54         if(found) then
```

```
55 |             getbasedir = tmp_path(1:pathlengths(
   |                ii))
56 |             exit
57 |         end if
58 |     end do
59 |
60 |     ! If it hasn't been found, then this script
   |        was probably called
61 |     ! from outside of the repository.
62 |     if(.not. found) then
63 |         write(*,*) 'BASE DIR NOT FOUND.'
64 |     end if
65 |
66 | end function
67 |
68 | ! Determine array size from min, max and step
69 | ! If alignment is off, array will overstep the
   |    maximum
70 | function bnd2max(xmin,xmax,dx)
71 |     implicit none
72 |
73 |     ! INPUTS:
74 |     ! xmin - minimum x value in array
75 |     ! xmax - maximum x value in array (inclusive
   |        )
76 |     ! dx - step size
77 |     double precision, intent(in) :: xmin, xmax,
   |        dx
78 |
79 |     ! OUTPUT:
80 |     ! step2max - maximum index of array
81 |     integer bnd2max
82 |
83 |     ! Calculate array size
84 |     bnd2max = int(ceiling((xmax-xmin)/dx))
85 | end function
86 |
87 | ! Create array from bounds and number of
   |    elements
88 | ! xmax is not included in array
89 | function bnd2arr(xmin,xmax,imax)
90 |     implicit none
91 |
92 |     ! INPUTS:
93 |     ! xmin - minimum x value in array
94 |     ! xmax - maximum x value in array (exclusive
   |        )
95 |     double precision, intent(in) :: xmin, xmax
96 |     ! imax - number of elements in array
97 |     integer imax
98 |
99 |     ! OUTPUT:
```

```
100 |     ! bnd2arr - array to generate
101 |     double precision, dimension(imax) :: bnd2arr
102 |
103 |     ! BODY:
104 |
105 |     ! Counter
106 |     integer ii
107 |     ! Step size
108 |     double precision dx
109 |
110 |     ! Calculate step size
111 |     dx = (xmax - xmin) / imax
112 |
113 |     ! Generate array
114 |     do ii = 1, imax
115 |         bnd2arr(ii) = xmin + (ii-1) * dx
116 |     end do
117 |
118 | end function
119 |
120 | function mod1(i, n)
121 |   implicit none
122 |   integer i, n, m
123 |   integer mod1
124 |
125 |   m = modulo(i, n)
126 |
127 |   if(m .eq. 0) then
128 |      mod1 = n
129 |   else
130 |      mod1 = m
131 |   end if
132 |
133 | end function mod1
134 |
135 | function sgn_int(x)
136 |   integer x, sgn_int
137 |   ! Standard signum function
138 |   sgn_int = sign(1,x)
139 |   if(x .eq. 0.) sgn_int = 0
140 | end function sgn_int
141 |
142 | function sgn(x)
143 |   double precision x, sgn
144 |   ! Standard signum function
145 |   sgn = sign(1.d0,x)
146 |   if(x .eq. 0.) sgn = 0
147 | end function sgn
148 |
149 | ! Interpolate single point from 1D data
150 | function interp(x0,xx,yy,nn)
151 |     implicit none
```

```
152 |
153 |     ! INPUTS:
154 |     ! x0 - x value at which to interpolate
155 |     double precision, intent(in) :: x0
156 |     ! xx - ordered x values at which y data is
        |        sampled
157 |     ! yy - corresponding y values to interpolate
158 |     double precision, dimension (nn), intent(in)
        |         :: xx,yy
159 |     ! nn - length of data
160 |     integer, intent(in) :: nn
161 |
162 |     ! OUTPUT:
163 |     ! interp - interpolated y value
164 |     double precision interp
165 |
166 |     ! BODY:
167 |
168 |     ! Index of lower-adjacent data (xx(i) < x0 <
        |         xx(i+1))
169 |     integer ii
170 |     ! Slope of liine between (xx(ii),yy(ii)) and
        |         (xx(ii+1),yy(ii+1))
171 |     double precision mm
172 |
173 |     ! If out of bounds, then return endpoint
        |        value
174 |     if (x0 < xx(1)) then
175 |        interp = yy(1)
176 |     else if (x0 > xx(nn)) then
177 |        interp = yy(nn)
178 |     else
179 |
180 |       ! Determine ii
181 |       do ii = 1, nn
182 |           if (xx(ii) > x0) then
183 |               ! We've now gone one index too far
        |                 .
184 |               exit
185 |           end if
186 |       end do
187 |
188 |       ! Determine whether we're on the right
        |          endpoint
189 |       if(ii-1 < nn) then
190 |           ! If this is a legitimate
        |              interpolation, then
191 |           ! subtract since we went one index too
        |              far
192 |           ii = ii - 1
193 |
194 |           ! Calculate slope
```

```
195 |           mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
    |               xx(ii))
196 |
197 |           ! Return interpolated value
198 |           interp = yy(ii) + mm * (x0 - xx(ii))
199 |       else
200 |           ! If we're actually interpolating the
    |              right endpoint,
201 |           ! then just return it.
202 |           interp = yy(nn)
203 |       end if
204 |
205 |    end if
206 |
207 | end function
208 |
209 | ! Calculate unshifted position of periodic image
210 | ! Assuming xmin, xmax are extreme attainable
    |    values of x
211 | function shift_mod(x, xmin, xmax)
212 |   double precision x, xmin, xmax
213 |   double precision mod_part, shift_mod
214 |   mod_part = mod(x-xmin, xmax-xmin)
215 |   if(mod_part .ge. 0) then
216 |      ! In this case, mod_part is distance
    |         between image & lower bound
217 |      shift_mod = xmin + mod_part
218 |   else
219 |      ! In this case, mod_part is distance
    |         between image & upper bound
220 |      shift_mod = xmax + mod_part
221 |   endif
222 | end function shift_mod
223 |
224 | ! Bilinear interpolation on evenly spaced 2D
    |    grid
225 | ! Assume upper endpoint is not included and is
    |    identical
226 | ! to the lower endpoint, which is included.
227 | function bilinear_array_periodic(x, y, nx, ny,
    |    x_vals, y_vals, fun_vals)
228 |   implicit none
229 |   double precision x, y
230 |   integer nx, ny
231 |   double precision, dimension(:) :: x_vals,
    |      y_vals
232 |   double precision, dimension(:,:) :: fun_vals
233 |
234 |   double precision dx, dy, xmin, ymin
235 |   integer i0, j0, i1, j1
236 |   double precision x0, x1, y0, y1
237 |   double precision z00, z10, z01, z11
```

```
238 |
239 |   double precision bilinear_array_periodic
240 |
241 |   xmin = x_vals(1)
242 |   ymin = y_vals(1)
243 |   dx = x_vals(2) - x_vals(1)
244 |   dy = y_vals(2) - y_vals(1)
245 |
246 |   ! Add 1 for one-indexing
247 |   i0 = int(floor((x-xmin)/dx))+1
248 |   j0 = int(floor((y-ymin)/dy))+1
249 |
250 |   x0 = x_vals(i0)
251 |   y0 = y_vals(j0)
252 |
253 |   ! Periodic wrap
254 |   if(i0 .lt. nx) then
255 |      i1 = i0 + 1
256 |      x1 = x_vals(i1)
257 |   else
258 |      i1 = 1
259 |      x1 = x_vals(nx) + dx
260 |   endif
261 |
262 |   if(j0 .lt. ny) then
263 |      j1 = j0 + 1
264 |      y1 = y_vals(j1)
265 |   else
266 |      j1 = 1
267 |      y1 = y_vals(ny) + dy
268 |   endif
269 |
270 |   z00 = fun_vals(i0,j0)
271 |   z10 = fun_vals(i1,j0)
272 |   z01 = fun_vals(i0,j1)
273 |   z11 = fun_vals(i1,j1)
274 |
275 |   bilinear_array_periodic = bilinear(x, y, x0,
    |      y0, x1, y1, z00, z01, z10, z11)
276 |end function bilinear_array_periodic
277 |
278 |! Bilinear interpolation on evenly spaced 2D
    |   grid
279 |! Assume upper and lower endpoints are included
280 |function bilinear_array(x, y, x_vals, y_vals,
    |   fun_vals)
281 |   implicit none
282 |   double precision x, y
283 |   double precision, dimension(:) :: x_vals,
    |      y_vals
284 |   double precision, dimension(:,:) :: fun_vals
```

```
285 |
286 |   double precision dx, dy, xmin, ymin
287 |   integer i0, j0, i1, j1
288 |   double precision x0, x1, y0, y1
289 |   double precision z00, z10, z01, z11
290 |
291 |   double precision bilinear_array
292 |
293 |   xmin = x_vals(1)
294 |   ymin = y_vals(1)
295 |   dx = x_vals(2) - x_vals(1)
296 |   dy = y_vals(2) - y_vals(1)
297 |
298 |   ! Add 1 for one-indexing
299 |   i0 = int(floor((x-xmin)/dx))+1
300 |   j0 = int(floor((y-ymin)/dy))+1
301 |   i1 = i0 + 1
302 |   j1 = j0 + 1
303 |
304 |   ! Bounds checking
305 |   ! if(i0 .lt. 1) then
306 |   !    i0 = 1
307 |   !    i1 = 1
308 |   ! else if(i1 .gt. nx) then
309 |   !    i0 = nx
310 |   !    i1 = nx
311 |   ! endif
312 |   ! if(j0 .lt. 1) then
313 |   !    j0 = 1
314 |   !    j1 = 1
315 |   ! else if(j1 .gt. ny) then
316 |   !    j0 = ny
317 |   !    j1 = ny
318 |   ! endif
319 |
320 |   x0 = x_vals(i0)
321 |   x1 = x_vals(i1)
322 |   y0 = y_vals(j0)
323 |   y1 = y_vals(j1)
324 |
325 |   z00 = fun_vals(i0,j0)
326 |   z10 = fun_vals(i1,j0)
327 |   z01 = fun_vals(i0,j1)
328 |   z11 = fun_vals(i1,j1)
329 |
330 |   bilinear_array = bilinear(x, y, x0, y0, x1, y1
    |      , z00, z01, z10, z11)
331 | end function bilinear_array
332 |
333 | ! ilinear interpolation of a function of two
    |   variables
```

```
! over a rectangle of points.
! Weight each point by the area of the sub-rectangle involving
! the point (x,y) and the point diagonally across the rectangle
function bilinear(x, y, x0, y0, x1, y1, z00, z01, z10, z11)
  implicit none
  double precision x, y
  double precision x0, y0, x1, y1, z00, z01, z10, z11
  double precision a, b, c, d
  double precision bilinear

  a = (x-x0)*(y-y0)
  b = (x1-x)*(y-y0)
  c = (x-x0)*(y1-y)
  d = (x1-x)*(y1-y)

  bilinear = (a*z11 + b*z01 + c*z10 + d*z00) / (a + b + c + d)
end function bilinear

! Integrate using left endpoint rule
! Assuming the right endpoint is not included in arr
function lep_rule(arr,dx,nn)
    implicit none

    ! INPUTS:
    ! arr - array to integrate
    double precision, dimension(nn) :: arr
    ! dx - array spacing (mesh size)
    double precision dx
    ! nn - length of arr
    integer, intent(in) :: nn

    ! OUTPUT:
    ! lep_rule - integral w/ left endpoint rule
    double precision lep_rule

    ! BODY:

    ! Counter
    integer ii

    ! Set output to zero
    lep_rule = 0.0d0

    ! Accumulate integral
    do ii = 1, nn
        lep_rule = lep_rule + arr(ii) * dx
```

```
    end do

end function

! Integrate using trapezoid rule
! Assuming both endpoints are included in arr
function trap_rule_dx(arr, dx, nn)
  implicit none
  double precision, dimension(nn) :: arr
  double precision dx
  integer ii, nn
  double precision trap_rule_dx

  trap_rule_dx = 0.0d0

  do ii=1, nn-1
     trap_rule_dx = trap_rule_dx + 0.5d0 * dx *
        (arr(ii) + arr(ii+1))
  end do

end function trap_rule_dx

! Integrate using trapezoid rule
! Assuming both endpoints are included in arr
function trap_rule_uneven(xx, yy, nn)
  implicit none
  double precision, dimension(nn) :: xx
  double precision, dimension(nn) :: yy
  integer ii, nn
  double precision trap_rule_uneven

  trap_rule_uneven = 0.0d0

  do ii=1, nn-1
     trap_rule_uneven = trap_rule_uneven + 0.5d0
         * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii
        +1))
  end do
end function trap_rule_uneven

function trap_rule_dx_uneven(dx, yy, nn)
  implicit none
  double precision, dimension(nn-1) :: dx
  double precision, dimension(nn) :: yy
  integer ii, nn
  double precision trap_rule_dx_uneven

  trap_rule_dx_uneven = 0.0d0

  do ii=1, nn-1
     trap_rule_dx_uneven = trap_rule_dx_uneven +
         0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
```

```
  end do
end function trap_rule_dx_uneven

! Integrate using midpoint rule
! First and last bins, only use inner half
function midpoint_rule_halfends(dx, yy, nn)
   result(integral)
  implicit none
  integer ii, nn
  double precision, dimension(nn) :: dx, yy
  double precision integral

  if(nn > 1) then
    integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(
       nn))

    do ii=2, nn-1
      integral = integral + dx(ii)*yy(ii)
    end do
  else
    integral = 0.d0
  end if
end function midpoint_rule_halfends

! Normalize 1D array and return integral w/ left
    endpoint rule
function normalize_dx(arr,dx,nn)
    implicit none

    ! INPUTS:
    ! arr - array to normalize
    double precision, dimension(nn) :: arr
    ! dx - array spacing (mesh size)
    double precision dx
    ! nn - length of arr
    integer, intent(in) :: nn

    ! OUTPUT:
    ! normalize - integral before normalization
       (left endpoint rule)
    double precision normalize_dx

    ! BODY:

    ! Calculate integral
    normalize_dx = lep_rule(arr,dx,nn)

    ! Normalize array
    arr = arr / normalize_dx

  end function normalize_dx

```

```
476 |! Normalize 1D unevenly-spaced array and
477 |! return integral w/ trapezoid rule
478 |! Will not be quite accurate if rightmost
    |   endpoint is not included
479 |! (Very small for VSF, so not a big deal there)
480 |! Modifies yy in place
481 |function normalize_uneven(xx, yy, nn) result(
    |   norm)
482 |  implicit none
483 |
484 |  ! INPUTS:
485 |  ! xx, yy - array values of data to normalize
486 |  double precision, dimension(nn) :: xx, yy
487 |  ! nn - length of arr
488 |  integer, intent(in) :: nn
489 |
490 |  ! OUTPUT:
491 |  ! normalize - integral before normalization (
    |     left endpoint rule)
492 |  double precision norm
493 |
494 |  ! BODY:
495 |
496 |  ! Calculate integral
497 |  ! PERHAPS WE SHOULD USE TRAPEZOID RULE
498 |  norm = trap_rule_uneven(xx, yy, nn)
499 |
500 |  ! Normalize array
501 |  yy(:) = yy(:) / norm
502 |
503 |end function normalize_uneven
504 |
505 |! Read 2D array from file
506 |function read_array(filename,fmtstr,nn,mm,
    |   skiplines_in)
507 |    implicit none
508 |
509 |    ! INPUTS:
510 |    ! filename - path to file to be read
511 |    ! fmtstr - input format (no parentheses, don
    |       't specify columns)
512 |    ! e.g. 'E10.2', not '(2E10.2)'
513 |    character(len=*), intent(in) :: filename,
    |       fmtstr
514 |    ! nn - Number of data rows in file
515 |    ! mm - number of data columns in file
516 |    integer, intent(in) :: nn, mm
517 |    ! skiplines - optional - number of lines to
    |       skip from header
518 |    integer, optional :: skiplines_in
519 |    integer skiplines
520 |
```

```
521 |     ! OUTPUT:
522 |     double precision, dimension(nn,mm) ::
        |        read_array
523 |
524 |     ! BODY:
525 |
526 |     ! Row counter
527 |     integer ii
528 |     ! File unit number
529 |     integer, parameter :: un = 10
530 |     ! Final format to use
531 |     character(len=256) finfmt
532 |
533 |     ! Generate final format string
534 |     write(finfmt,'(A,I1,A,A)') '(', mm, fmtstr,
        |        ')'
535 |
536 |     ! Print message
537 |     !write(*,*) 'Reading data from "', trim(
        |        filename), '"'
538 |     !write(*,*) 'using format "', trim(finfmt),
        |        '"'
539 |
540 |     ! Open file
541 |     open(unit=un, file=trim(filename), status='
        |        old', form='formatted')
542 |
543 |     ! Skip lines if desired
544 |     if(present(skiplines_in)) then
545 |         skiplines = skiplines_in
546 |         do ii = 1, skiplines
547 |             ! Read without variable ignores the
        |                line
548 |             read(un,*)
549 |         end do
550 |     else
551 |         skiplines = 0
552 |     end if
553 |
554 |     ! Loop through lines
555 |     do ii = 1, nn
556 |         ! Read one row at a time
557 |         read(unit=un, fmt=trim(finfmt))
        |            read_array(ii,:)
558 |     end do
559 |
560 |     ! Close file
561 |     close(unit=un)
562 |
563 | end function
564 |
565 | ! Print 2D array to stdout
```

```
566 subroutine print_int_array(arr,nn,mm,fmtstr_in)
567   implicit none
568
569   ! INPUTS:
570   ! arr - array to print
571   integer, dimension (nn,mm), intent(in) :: arr
572   ! nn - number of data rows in file
573   ! nn - number of data columns in file
574   integer, intent(in) :: nn, mm
575   ! fmtstr - output format (no parentheses, don'
        t specify columns)
576   ! e.g. 'E10.2', not '(2E10.2)'
577   character(len=*), optional :: fmtstr_in
578   character(len=256) fmtstr
579
580   ! NO OUTPUTS
581
582   ! BODY
583
584   ! Row counter
585   integer ii
586   ! Final format to use
587   character(len=256) finfmt
588
589   ! Determine string format
590   if(present(fmtstr_in)) then
591      fmtstr = fmtstr_in
592   else
593      fmtstr = 'I10'
594   end if
595
596   ! Generate final format string
597   write(finfmt,'(A,I4,A,A)') '(', mm, trim(
        fmtstr), ')'
598
599   ! Loop through rows
600   do ii = 1, nn
601      ! Print one row at a time
602      write(*,finfmt) arr(ii,:)
603   end do
604
605   ! Print blank line after
606   write(*,*) ' '
607
608 end subroutine print_int_array
609
610 subroutine print_array(arr,nn,mm,fmtstr_in)
611     implicit none
612
613     ! INPUTS:
614     ! arr - array to print
```

```
615 |     double precision, dimension (nn,mm), intent(
    |        in) :: arr
616 |     ! nn - number of data rows in file
617 |     ! nn - number of data columns in file
618 |     integer, intent(in) :: nn, mm
619 |     ! fmtstr - output format (no parentheses,
    |        don't specify columns)
620 |     ! e.g. 'E10.2', not '(2E10.2)'
621 |     character(len=*), optional :: fmtstr_in
622 |     character(len=256) fmtstr
623 |
624 |     ! NO OUTPUTS
625 |
626 |     ! BODY
627 |
628 |     ! Row counter
629 |     integer ii
630 |     ! Final format to use
631 |     character(len=256) finfmt
632 |
633 |     ! Determine string format
634 |     if(present(fmtstr_in)) then
635 |         fmtstr = fmtstr_in
636 |     else
637 |         fmtstr = 'ES10.2'
638 |     end if
639 |
640 |     ! Generate final format string
641 |     write(finfmt,'(A,I4,A,A)') '(', mm, trim(
    |        fmtstr), ')'
642 |
643 |     ! Loop through rows
644 |     do ii = 1, nn
645 |         ! Include row number
646 |         !write(*,'(I10)', advance='no') ii
647 |         ! Print one row at a time
648 |         write(*,finfmt) arr(ii,:)
649 |     end do
650 |
651 |     ! Print blank line after
652 |     write(*,*) ' '
653 |
654 | end subroutine
655 |
656 | ! Write 1D array to file
657 | subroutine write_vec(arr,nn,filename,fmtstr_in)
658 |     implicit none
659 |
660 |     ! INPUTS:
661 |     ! arr - array to print
662 |     double precision, dimension (nn), intent(in)
    |         :: arr
```

```
663     ! nn - number of data rows in file
664     ! nn - number of data columns in file
665     integer, intent(in) :: nn
666     ! filename - file to write to
667     character(len=*) filename
668     ! fmtstr - output format (no parentheses,
          don't specify columns)
669     ! e.g. 'E10.2', not '(2E10.2)'
670     character(len=*), optional :: fmtstr_in
671     character(len=256) fmtstr
672
673     ! NO OUTPUTS
674
675     ! BODY
676
677     ! Row counter
678     integer ii
679     ! Final format to use
680     character(len=256) finfmt
681     ! Dummy file unit to use
682     integer, parameter :: un = 20
683
684     ! Open file for writing
685     open(unit=un, file=trim(filename), status='
          replace', form='formatted')
686
687     ! Determine string format
688     if(present(fmtstr_in)) then
689         fmtstr = fmtstr_in
690     else
691         fmtstr = 'E10.2'
692     end if
693
694     ! Generate final format string
695     write(finfmt,'(A,A,A)') '(', trim(fmtstr), '
          )'
696
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,
       fmtstr_in)
710     implicit none
```

```
711 |
712 |     ! INPUTS:
713 |     ! arr - array to print
714 |     double precision, dimension (nn,mm), intent(
    |        in) :: arr
715 |     ! nn - number of data rows in file
716 |     ! nn - number of data columns in file
717 |     integer, intent(in) :: nn, mm
718 |     ! filename - file to write to
719 |     character(len=*) filename
720 |     ! fmtstr - output format (no parentheses,
    |        don't specify columns)
721 |     ! e.g. 'E10.2', not '(2E10.2)'
722 |     character(len=*), optional :: fmtstr_in
723 |     character(len=256) fmtstr
724 |
725 |     ! NO OUTPUTS
726 |
727 |     ! BODY
728 |
729 |     ! Row counter
730 |     integer ii
731 |     ! Final format to use
732 |     character(len=256) finfmt
733 |     ! Dummy file unit to use
734 |     integer, parameter :: un = 20
735 |
736 |     ! Open file for writing
737 |     open(unit=un, file=trim(filename), status='
    |        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(
    |        fmtstr), ')'
748 |
749 |     ! Loop through rows
750 |     do ii = 1, nn
751 |         ! Print one row at a time
752 |         write(un,finfmt) arr(ii,:)
753 |     end do
754 |
755 |     ! Close file
756 |     close(unit=un)
757 |
758 | end subroutine
```

```
759 |
760 |subroutine zeros(x, n)
761 |  implicit none
762 |  integer n, i
763 |  double precision, dimension(n) :: x
764 |
765 |  do i=1, n
766 |     x(i) = 0
767 |  end do
768 |end subroutine zeros
769 |
770 |end module
```