

©2018

OLIVER GRAHAM EVANS

ALL RIGHTS RESERVED

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Oliver Graham Evans

May, 2018

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

Approved:

Advisor
Dr. Kevin Kreider

Accepted:

Dean of the College
Dr. Linda Subich

Co-Advisor
Dr. Curtis Clemons

Dean of the Graduate School
Dr. Chand Midha

Faculty Reader
Dr. Gerald Young

Date

Department Chair
Dr. Kevin Kreider

ABSTRACT

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

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

This project was supported in part by the 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

	Page
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Background on Kelp Models	4
1.3 Background on Radiative Transfer	7
1.4 Overview of Thesis	8
II. KELP MODEL	10
2.1 Physical Setup	10
2.2 Coordinate System	12
2.3 Population Distributions	14
2.4 Spatial Distribution	18
2.5 Discontinuity at the Rope	25
III. LIGHT MODEL	32
3.1 Optical Definitions	32
3.2 The Radiative Transfer Equation	36

3.3	Low-Scattering Approximation	39
IV.	NUMERICAL SOLUTION	44
4.1	Discrete Grid	44
4.2	Kelp Distribution	48
4.3	Quadrature Rules	52
4.4	Numerical Asymptotics	55
4.5	Finite Difference	57
V.	CODE VERIFICATION	67
5.1	Sources of Error in Numerical Simulations	67
5.2	Verification and Validation	69
5.3	Code Verification: Method of Manufactured Solutions	71
5.4	Verification of Calculations	77
VI.	PRACTICAL APPLICATION	81
6.1	Physical Parameters	82
6.2	Computational Expense	88
6.3	Grid Size and Discretization Error	92
6.4	Optical Conditions for Asymptotics	99
6.5	Comparison to Other Light Models	115
VII.	CONCLUSION	122
7.1	Model Summary	123
7.2	Future Work	124

APPENDICES	137
APPENDIX A. GRID DETAILS	138
APPENDIX B. RAY TRACING ALGORITHM	142
APPENDIX C. SYNTHETIC DATA	145
APPENDIX D. MEMORY USAGE	147
APPENDIX E. FORTRAN CODE	150

LIST OF TABLES

Table	Page
4.1 Breakdown of nonzero matrix elements by derivative case.	64
6.1 Physical parameter values.	83
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 attenuation coefficients (a, b, c) are given, and their ratios.	84
D.1 Memory to store one copy of the finite difference coefficient matrix. n_s varies over rows and n_a over columns.	148
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.	149

LIST OF FIGURES

Figure	Page
1.1 <i>Saccharina latissima</i> being harvested	3
2.1 <i>Saccharina latissima</i> innoculated onto a thread wrapped around a rope on which it is to be grown.	11
2.2 Rendering of four nearby vertical kelp ropes as represented in the spatial distribution model. Note the kite-shaped fronds and horizontal orientation.	12
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	13
2.4 Simplified kite-shaped frond. Reproduced with permission from [7]. . .	14
2.5 von Mises distribution for a variety of parameters.	17
2.6 2D length-angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$	19
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.	20
2.8 Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$	23
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 $(\theta, s) = (3\pi/4, 3/2)$. .	24
2.10 Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3$, $\eta v_w = 1$	26

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) $\eta v_w = 10$, $\sigma_b = 0$, $\sigma_l = 1$ m. More realistic kelp distribution, but still has large derivatives near the origin. (c) $\eta v_w = 10$, $\sigma_b = 0.4$ m, $\sigma_l = 1$ m. Moderate Gaussian blur slightly reduces derivatives near the origin. (d) $\eta v_w = 10$, $\sigma_b = 2$ m, $\sigma_l = 1$ m. Overblurred distribution; should be avoided.	31
4.1	Spatial grid. Discrete quantities are calculated at grid cell centers.	46
4.2	Angular grid at each point in space with poles treated separately.	47
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.	76
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.	77
6.1	Computation time required for numerical asymptotics and finite difference algorithms over a range of spatial grid sizes using 32 CPUs. Only five grid sizes are shown, with the finite difference and numerical asymptotic algorithm for $n = 0, \dots, 3$ terms shown for each grid. The horizontal offset within each grid size is only for visual clarity.	90
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.	92
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.	94

6.4	Average irradiance plotted against squared resolution. Each line is a one-dimensional 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.	95
6.5	Predictions for isolated spatial and angular components of discretization error as a function of grid size for both numerical algorithms. Once a target error threshold has been chosen, this figure can be used to estimate the grid sizes that each algorithm would require. The total discretization error is the sum of the spatial and angular parts.	98
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 $b = 0.2$ and $b = 0.5$. Below $b = 0.2$, discretization error dominates. Above $b = 0.5$, the asymptotic series diverges.	101
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.	102
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.	103
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 suffices for large error criteria.	104
6.10	Truncation error plotted against 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. Predicted values, marked with “x”s, are very accurate.	107

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 diverges. This observation permits a simple model for predicting truncation error.	110
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 converges only for (a_w, b) values to the right of the ξ^* contour.	111
6.13	Recommended $n = \bar{n}$ value as a function of (a_w, b) to achieve the truncation error criteria $\bar{\varepsilon} = 0.1, 0.5, 1.0\text{W/m}^2$ for the three plot rows.	114
6.14	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. Discretization error is shown by error bands for perc. irrad. and avg. irrad. Note the different scales in the two plots.	120
6.15	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.	121
A.1	Angular grid	140

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 [42]. 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 of each area would be of great assistance in choosing the best site. Sim-

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

Recently, a growth model [8] for *S. latissima* has been produced and integrated into the SINMOD [41] 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 simplistic.

Both Anderson’s and Jackson’s model were carried out by numerically solving a system of differential equations over small time intervals. In 1990, M.A. Burgman and V.A. Gerard developed a stochastic population model [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, 43]. 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 [8, 7, 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 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 convergence properties which build confidence in their correctness. 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* innoculated onto a thread wrapped around a rope on which it is to be grown.

We consider only the case of a rigid vertical rope which does not sway in the current. The mature *Saccharina latissima* plant consists of a single frond (leaf), a stipe (stem) and a holdfast (root structure). For the sake of this model, only the kelp frond is considered, and its base is attached directly to the rope. The “gentle undulation approximation” is employed, whereby 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 defined in Figure 2.3. The vectors $\mathbf{x} = (x, y, z)$ 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

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

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

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

Then, calculating derivatives from (2.1) yields

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

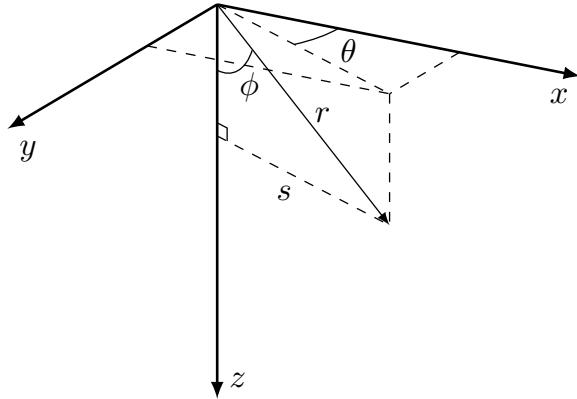


Figure 2.3: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ .

2.3 Population Distributions

In order to construct a spatial distribution of kelp fronds, a simple kite-shaped geometry is introduced, and frond lengths and azimuthal orientations are assumed to be distributed predictably. Since it is assumed that fronds extend perfectly horizontally, no angular elevation distribution is required.

2.3.1 Frond Shape

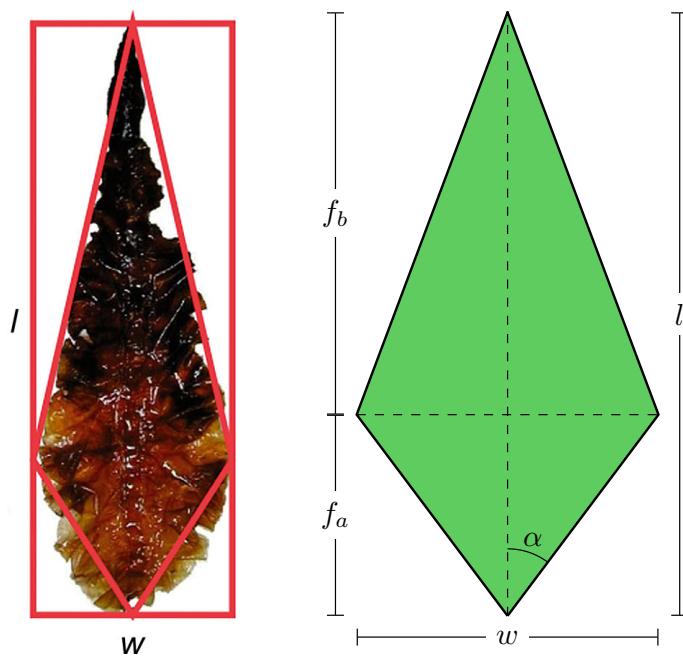


Figure 2.4: Simplified kite-shaped frond. Reproduced with permission from [7].

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

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

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

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

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

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

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

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

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

2.3.2 Length and Angle Distributions

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)

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

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(s m^{-1}). Then, use $\mu = \theta_w$ and $\kappa = \eta v_w$ as the von Mises distribution parameters. Note that θ_w and v_w vary over depth, while η is assumed constant for the population. Then, the PDF for the von Mises frond angle distribution is

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

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

uniform distribution as κ approaches 0, so

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

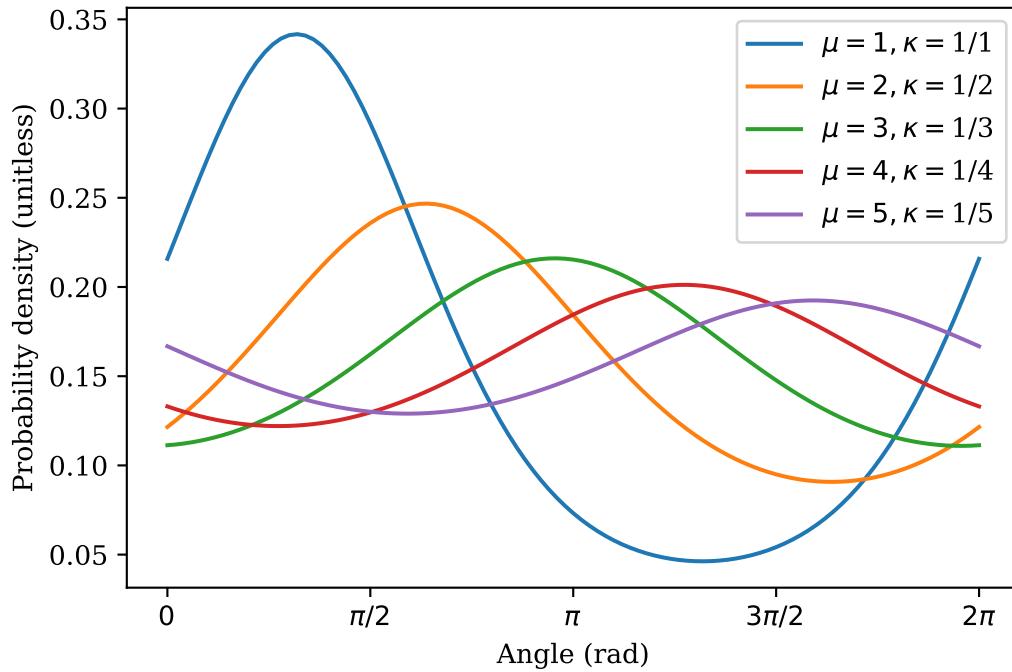


Figure 2.5: von Mises distribution for a variety of parameters.

2.3.3 Joint Length–Orientation Distribution

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

probabilities. That is,

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

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

$$P_{2D}(\theta_f, l) = P_{\theta_f}(\theta_f) \cdot P_l(l). \quad (2.5)$$

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

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

which is currently not taken into consideration in this model.

2.4 Spatial Distribution

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

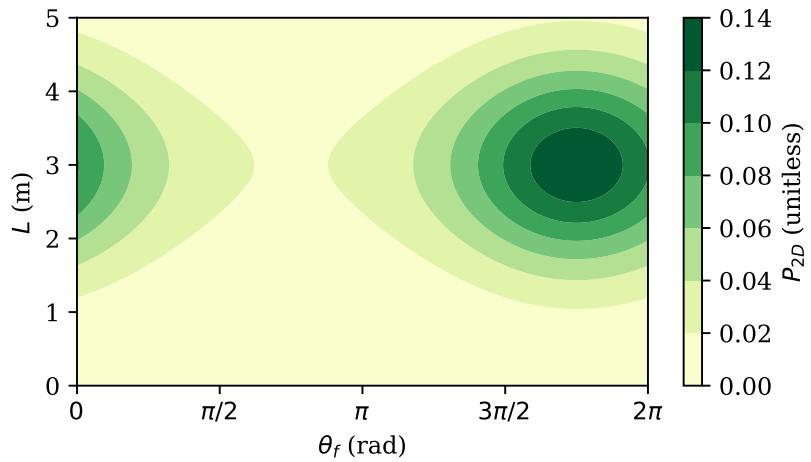


Figure 2.6: 2D length–angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$.

2.4.1 Rotated Coordinate System

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

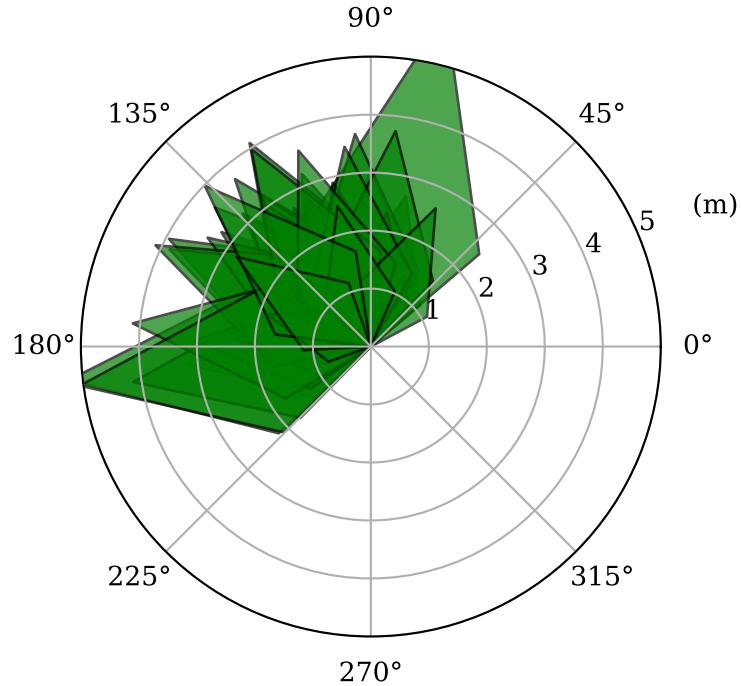


Figure 2.7: A sample of 50 kelp fronds with shape parameters $f_s = 0.5$ and $f_r = 2$ whose lengths are picked from a normal distribution and whose angles are picked from a von Mises distribution.

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

$$s'_f(\theta'; l) = \frac{l}{\sin \theta' + S(\theta') \frac{2f_r}{1+f_s} \cos \theta'}. \quad (2.6)$$

For convenience, denote the denominator of (2.6) 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

$$s_f(\theta; l) = s'_f \left(\theta - \theta_f + \frac{\pi}{2} \right). \quad (2.7)$$

Similarly, define

$$d_f(\theta) = d'_f \left(\theta - \theta_f + \frac{\pi}{2} \right). \quad (2.8)$$

2.4.3 Conditions for Occupancy

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

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

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

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

$$\theta - \alpha < \theta_f < \theta + \alpha, \quad (2.9)$$

$$l > l_{\min}(\theta, s), \quad (2.10)$$

where

$$l_{\min}(\theta, s) = s \cdot \frac{l}{s_f(\theta; l)} = s \cdot d_f(\theta). \quad (2.11)$$

Then, considering the point to be fixed, (2.9) and (2.10) 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.

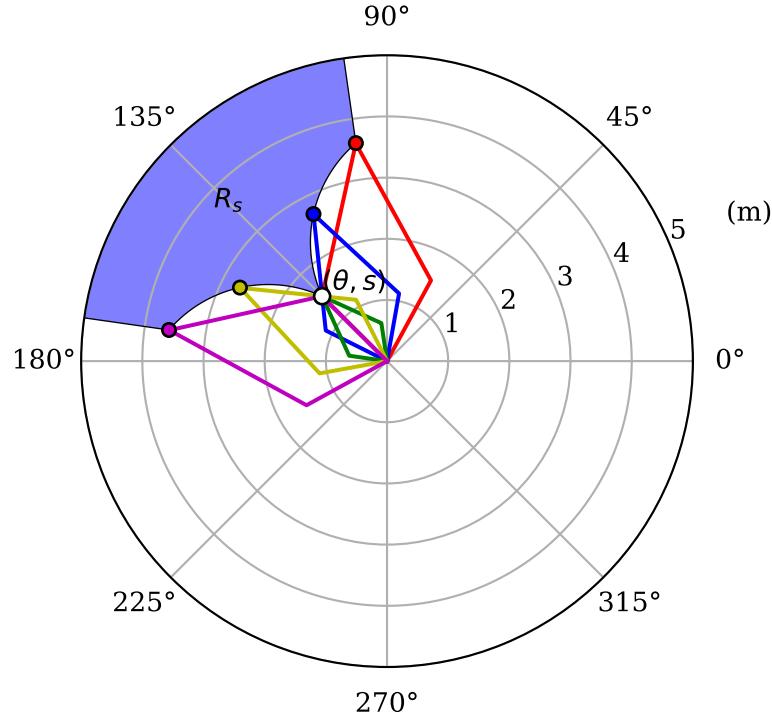


Figure 2.8: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$.

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 (2.5) over $R_s(\theta, s)$, the occupancy region for the point of interest.

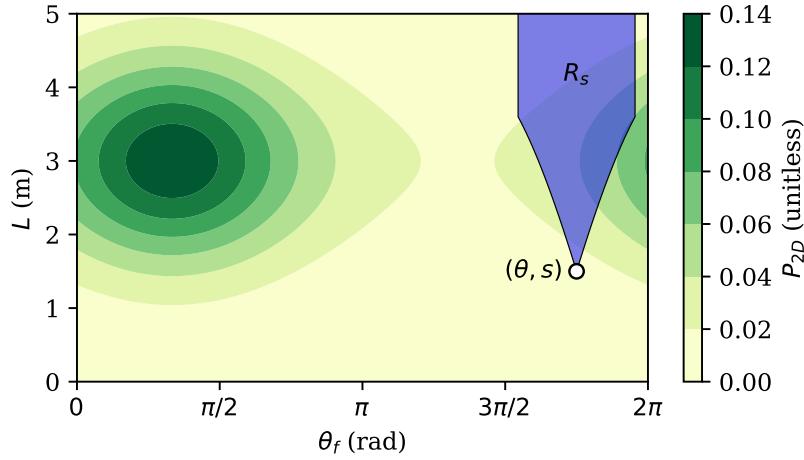


Figure 2.9: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the θ_f-l plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$.

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) ,

$$\begin{aligned}\tilde{P}_k(\theta, s, z) &= \iint_{R_s(\theta, s)} P_{2D}(\theta_f, l) dl d\theta_f \\ &= \int_{\theta-\alpha}^{\theta+\alpha} \int_{l_{\min}(\theta_f)}^{\infty} P_{2D}(\theta_f, l) dl d\theta_f.\end{aligned}\quad (2.12)$$

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

$$P_k(x, y) = \frac{n_f f_t}{dz} \tilde{P}_k(x, y). \quad (2.13)$$

In the continuum limit as the number of discrete 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

$$P_k(x, y, z) = f_t \rho_f(z) \tilde{P}_k(x, y, z). \quad (2.14)$$

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

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

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 significant 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

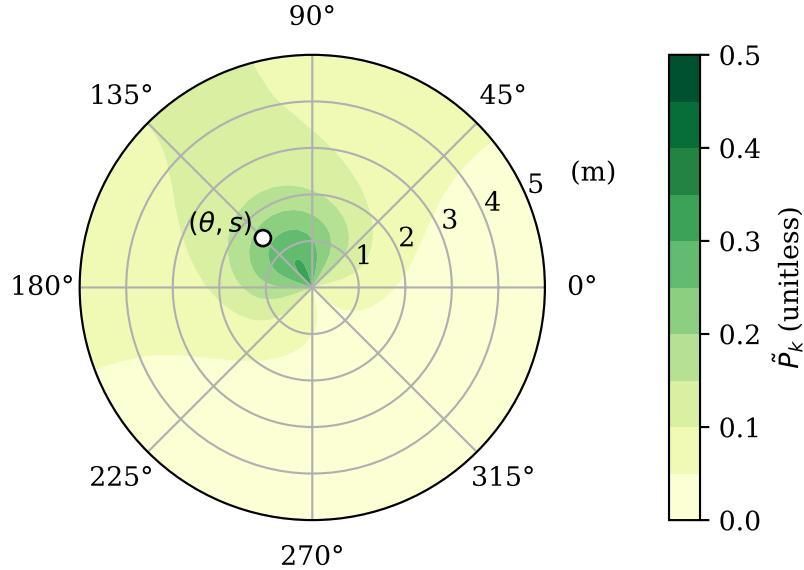


Figure 2.10: Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3$, $\eta v_w = 1$.

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, the solution is not guaranteed to converge.

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 xy 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

$$k(x; \sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{x^2}{2\sigma_b^2}\right). \quad (2.15)$$

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 $a(\mathbf{x})$. This is done by convolving slices of P_k in the xy -plane with the two-dimensional normalized Gaussian kernel

$$K(x, y; \sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(\frac{x^2 + y^2}{2\sigma_b^2}\right). \quad (2.16)$$

Hence, the blurred solution is

$$P_k^b(x, y, z) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K(x', y', z; \sigma_b) P_k(x - x', y - y', z) dx' dy'. \quad (2.17)$$

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 numerical 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, and 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.

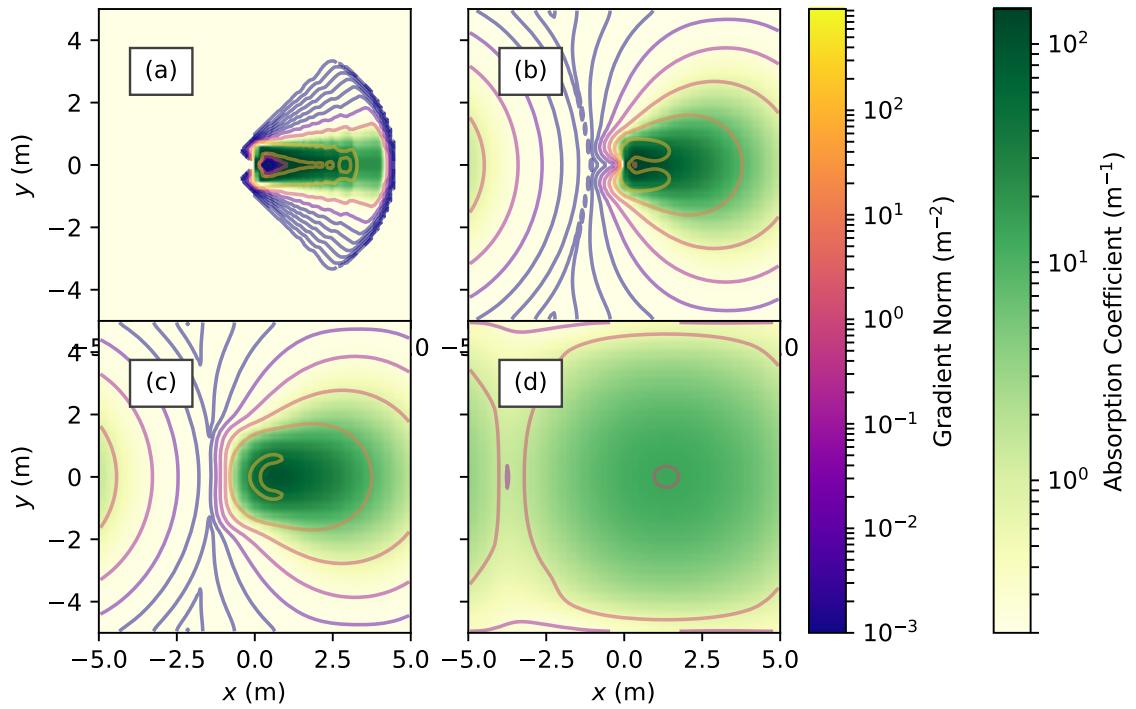


Figure 2.11: z -slices of several absorption coefficient distributions from kelp distributions with varying parameters. The norm of the gradient is depicted with contours.

(a) $\eta v_w = 90$, $\sigma_l = \sigma_b = 0$. Unrealistically sharp distribution shows kite-shaped character. (b) $\eta v_w = 10$, $\sigma_b = 0$, $\sigma_l = 1$ m. More realistic kelp distribution, but still has large derivatives near the origin. (c) $\eta v_w = 10$, $\sigma_b = 0.4$ m, $\sigma_l = 1$ m. Moderate Gaussian blur slightly reduces derivatives near the origin. (d) $\eta v_w = 10$, $\sigma_b = 2$ m, $\sigma_l = 1$ m. Overblurred 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.

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 becomes 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,

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

where ω is an element of solid angle. Once the radiance L is calculated everywhere, the irradiance is

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

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 will be 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 [25] given by

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

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

3.1.2 Perceived Irradiance

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

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

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

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

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

3.1.3 Inherent Optical Properties

We now define a few inherent optical properties (IOPs) which depend only on the medium of propagation. The absorption coefficient $a(\mathbf{x})$ (units m^{-1}) defines the proportional loss of radiance per unit length due to absorption by the medium. For example, this includes radiant energy which is converted to heat. The scattering coefficient b (units m^{-1}), defines the proportional loss of radiance per unit length due to scattering, and is assumed to be constant over space. Scattered light is not lost from the light field, it simply changes direction.

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

the probability of scattering occurring exactly from ω to ω' is 0. Rather, we say that the probability of radiance being scattered from a direction ω into an element of solid angle Ω is $\int_{\Omega} \beta(\omega \cdot \omega') d\omega'$.

The VSF is normalized such that

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

so that for any ω ,

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

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

3.2 The Radiative Transfer Equation

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

3.2.1 Ray Notation

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

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

Hence, the ray path is parameterized as

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

where

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

is the origin of the ray, and

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

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

3.2.2 Colloquial Description

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

1. Radiance is decreased due to absorption;
2. Radiance is decreased due to scattering out of the path to other directions;
3. Radiance is increased due to scattering into the path from other directions;
4. Radiance is increased or decreased due to light sources or sinks.

3.2.3 Equation of Transfer

Combining these phenomena yields the radiative transfer equation along $\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s)$ evaluated at $(\mathbf{x}, \boldsymbol{\omega})$,

$$\begin{aligned} \left. \frac{dL(\mathbf{l}(\mathbf{x}, \boldsymbol{\omega}, s), \boldsymbol{\omega})}{ds} \right|_{s=\tilde{s}} &= -(a(\mathbf{x}) + b)L(\mathbf{x}, \boldsymbol{\omega}) \\ &+ b \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L(\mathbf{x}) d\boldsymbol{\omega}' + \sigma(\mathbf{x}, \boldsymbol{\omega}), \end{aligned} \quad (3.4)$$

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

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

which reveals the vector form of the radiative transfer equation,

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

or equivalently,

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

3.2.4 Boundary Conditions

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

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

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

In the z direction, we specify a spatially uniform downwelling light just under the surface of the water by a function $f(\boldsymbol{\omega})$. 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 \mathbf{x}_s be a point on the surface of the domain and \mathbf{x}_b a point on the bottom,

we have

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

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

3.3 Low-Scattering Approximation

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

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

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

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

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

Substituting the above into (3.5) yields

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

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

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

$$\boldsymbol{\omega} \cdot \nabla L_1(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_1(\mathbf{x}) = \sigma_1(\mathbf{x}, \boldsymbol{\omega}) + \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_0(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_0(\mathbf{x}, \boldsymbol{\omega}),$$

$$\boldsymbol{\omega} \cdot \nabla L_2(\mathbf{x}, \boldsymbol{\omega}) + a(\mathbf{x})L_2(\mathbf{x}) = \sigma_2(\mathbf{x}, \boldsymbol{\omega}) + \int_{4\pi} \beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}') L_1(\mathbf{x}, \boldsymbol{\omega}') d\boldsymbol{\omega}' - L_1(\mathbf{x}, \boldsymbol{\omega}).$$

⋮

In general, for $n \geq 1$,

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

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

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

Grouping by powers of b , we have

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

for the first term, and

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

for $n > 0$.

3.3.2 Analytical Solution

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

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

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

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

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

ential equation

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

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

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

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,

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

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

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

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

these two sources is called the effective source,

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

This can be similarly extracted along a ray path as

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

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

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

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

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

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

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

CHAPTER IV

NUMERICAL SOLUTION

In this chapter, the mathematical details involved in the numerical solution of the 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.

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:

$$x \rightarrow i,$$

$$y \rightarrow j,$$

$$z \rightarrow k,$$

$$\theta \rightarrow l,$$

$$\phi \rightarrow m.$$

It is convenient, however, to use a single index p to refer to directions ω rather than referring to θ and ϕ separately. Then, the center of a generic grid cell will be denoted as $(x_i, y_j, z_k, \omega_p)$, and the boundaries between adjacent grid cells will be referred to as *edges*. One-indexing is employed throughout this document (i.e., array elements are counted starting at 1, not 0).

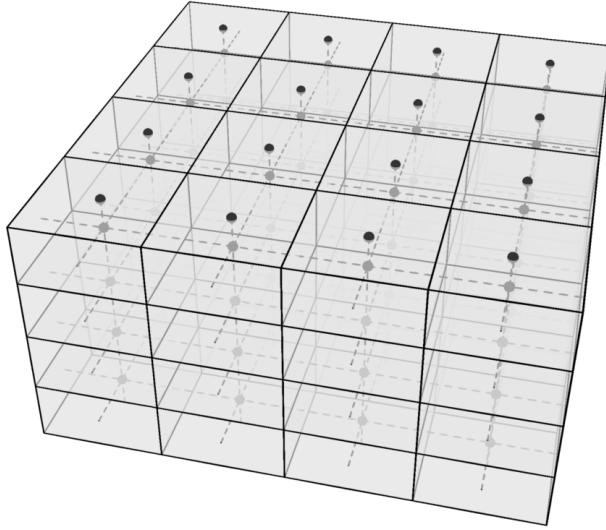


Figure 4.1: Spatial grid. Discrete quantities are calculated at grid cell centers.

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.

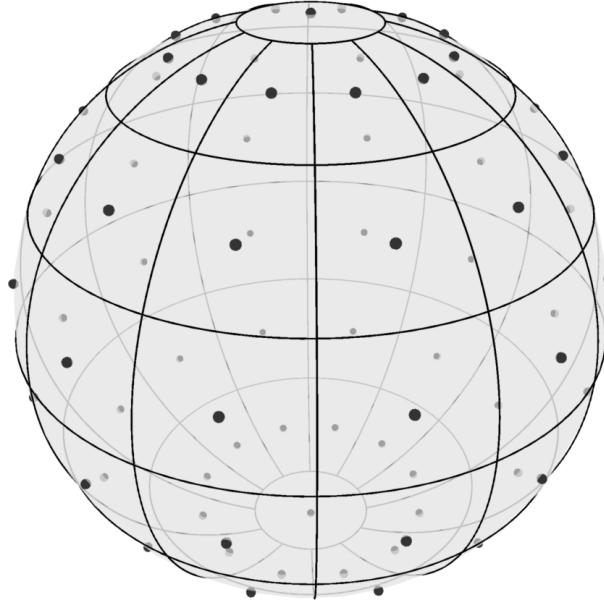


Figure 4.2: Angular grid at each point in space with poles treated separately.

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 ω_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 $n_\omega = n_\theta(n_\phi - 2) + 2$. Then, cells are indexed by $p = 1, \dots, n_\omega$ and are ordered such that $p = 1$ and $p = n_\omega$ refer to the north and south poles respectively, $p \leq n_\omega/2$ refers to the northern hemisphere, and $p > n_\omega/2$ refers to the southern hemisphere. Further, the symbol Ω_p is used to refer to the two dimension angular interval centered

at ω_p . The solid angle subtended by Ω_p is denoted $|\Omega_p|$. 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

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 [37]. 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 (2.4), 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

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

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

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

4.2.2 Integration of Population Distributions

The computational crux of calculating the kelp distribution is to evaluate (2.12) by integrating $P_{2D}(\theta_f, l)$ from (2.5) over the occupancy region $R_s(\theta, s)$, defined by (2.9) and (2.10), for every point (θ, s) in the xy -domain. To recap, the population distribution is

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

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

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

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

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

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

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

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 (2.12) and (2.13) 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 $K(x, y; \sigma_b)$ from Equation (2.16) is numerically computed by the same process used to post-process pixelated image with a Gaussian blur.

Notice first that the integral from Equation (2.17) 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 the large majority of the Gaussian distribution, the number of quadrature points in each dimension is chosen to be

$$n_K = 2r_K + 1, \quad (4.4)$$

where

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

Then, the quadrature points are

$$x'_{i'} = (i' - r_K)dx, \quad i' = 1, \dots, n_k, \quad y'_{j'} = (j' - r_K)dy, \quad j' = 1, \dots, n_k.$$

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

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

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

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

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_{ij}^{kb} is evaluated as

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

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

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

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

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

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

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

where ds_{ijk} is the total path distance of $\mathbf{l}(s)$ 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

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

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

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

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

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

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

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

4.4 Numerical Asymptotics

The asymptotic approximations (3.11) and (3.13) to the radiative transfer equation (3.5) 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 $\nu = 1, \dots, N - 1$ index the spatial grid cells traversed (wholly or partially) by the ray, and define the *path-length characteristic function*

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

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

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

Given s , the index of the next edge crossing is

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

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

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

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

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

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

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

Define the intermediate variable

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

which permits the simple formula

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

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 (3.11) is given by

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

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 in order to construct a linear system of equations whose solution approximates that of the analytical equation. The price of a general solution, however, is greatly increased computational cost, both in terms of memory and CPU usage.

4.5.1 Discretization

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

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

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

difference is given by

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

and the backward difference by

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

For the upper and lower boundaries, we need an asymmetric finite difference method 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

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

Truncating after the first few terms, we have

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

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

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

Rearranging (4.9) produces

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

Combining (4.10) with (4.11) gives

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

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

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

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

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

4.5.2 Difference Equations

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

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

or equivalently,

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

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

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

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

For downwelling light at the surface, we apply the asymmetric second order difference approximation (4.10) using the surface radiance value, which gives

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

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

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

Likewise, for the bottom boundary condition, we have

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

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

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

Grouping L_{ijkp} terms gives

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

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

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

4.5.3 Structure of Linear System

For each spatial-angular grid cell, one of the above equations is applied. The equation applied at each grid cell involves adjacent radiance values due to the discretized derivatives. Thus, a coupled system of linear equations is produced, which can be written as a sparse matrix equation, $\mathbb{A}\mathbb{X} = \mathbb{B}$. 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 ω, z, y, x , with ω being the outermost and x being the innermost. Recall that θ and ϕ are already combined, both indexed by p , as discussed in Section 4.1 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

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

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 $n_x n_y n_z n_\omega$. 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.

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

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

By multiplying the first column of Table 4.1 by the second and summing over the rows, the total number of nonzero matrix elements is calculated to be

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

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 numbers 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 $\sigma(\mathbf{x}, \boldsymbol{\omega})$, \mathbb{B} only has nonzero entries for the downwelling surface grid cells, of which there are $n_x n_y n_{\omega}/2$.

4.5.4 Iterative Solution

Because of the large number of dimensions (three spatial, two angular), the matrix can easily have upwards of millions of nonzero elements, even for modest grid sizes. Direct methods such as Gaussian elimination, QR factorization, and singular value decomposition are therefore infeasible due to memory requirements. We therefore turn to iterative solvers. Many such solvers are available, including GMRES [36], LGMRES [4], IDR [39], and BI-CGSTAB [40].

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 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, 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* [33].

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* [33]. Validation involves comparison with experimental evidence in order to determine that governing equations accurately describe physical phenomenon, and that their solutions match observation. That is, 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 [34]. Verification, however, is a purely mathematical exercise, and has nothing to do with the physical system being modeled. It deals only with the agreement between an equation and its numerical solution produced by a particular implementation of an

algorithm. 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.

5.3 Code Verification: 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. Of course, there is a trade-off: the solutions are not physically realistic. However, this

is not an issue; as stated previously, *verification is a purely mathematical endeavor*.

Determining that a code solves an equation correctly is unrelated to physical realism.

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

$$Du(\mathbf{x}) = \sigma(\mathbf{x}), \quad (5.1)$$

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

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 ?? 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 soution pointwise. Then, some norm (usually l_1 , l_2 , or l_∞) 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 effect occurs only for small grid resolutions, when the first nonzero term in the error series dominates the others. If large grid spacings are used, then order p convergence may not be observed even in a correct implementation since higher order error terms will overpower the order p term. The range of small grid spacings for which the lowest order term dominates is known as the *asymptotic range* (a.k.a.

“the sweet spot”). If the verification procedure shows that the implementation does not demonstrate the theoretical order of accuracy of the algorithm, this is most likely [32] an indication that either a coding mistake is present or the asymptotic range has not been achieved.

5.3.2 Synthetic Data

In order to perform code verification for the radiative transfer equation, it is necessary to first choose a manufactured solution $L(\mathbf{x}, \boldsymbol{\omega})$ for radiance, as well as coefficient functions for the absorption coefficient $a(\mathbf{x})$ and volume scattering function $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$. 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 non-trivial to characterize, so we define generic spatial and angular resolutions for the sake of reducing dimensionality. Let $n_s = n_x = n_y = n_z$ and $n_a = n_\theta = n_\phi$. Then, we use the geometric mean to describe the spatial and angular resolution, as

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

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

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 which would affect 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, while the

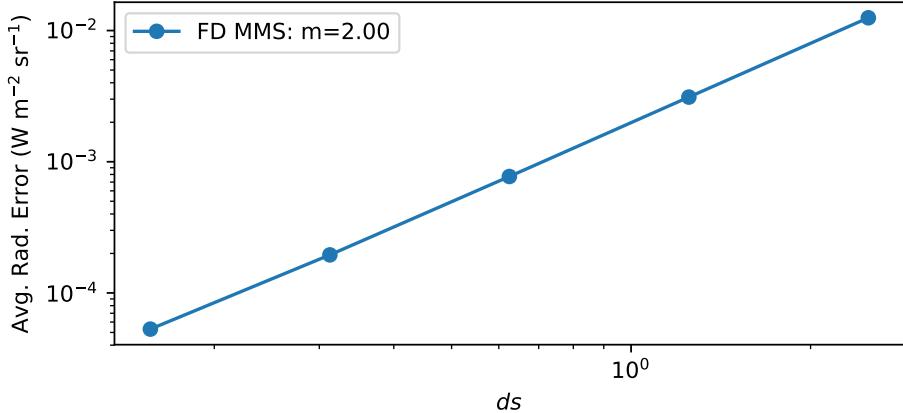


Figure 5.1: Code verification for the finite difference solution. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. A slope of $m = 2$ on a log–log scale demonstrates second order convergence, as expected, demonstrating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

discretization error is not verifiable in that sense, the truncation error *is*. That is, since the n –term asymptotic solution is a numerical approximation to the n –term Taylor series, the norm of the pointwise error should decrease with order $n + 1$ as $b \rightarrow 0$. 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 .

Figure 5.2 demonstrates the convergence of the asymptotic solution as $b \rightarrow 0$. 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.

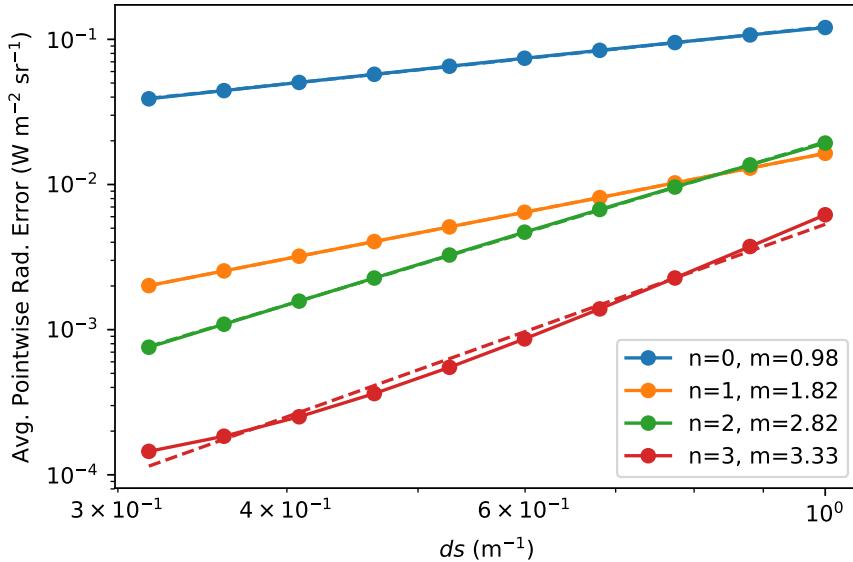


Figure 5.2: Code verification for the numerical asymptotics solution via the Method of Manufactured Solutions. A range of b values are run, using 0 - 3 terms in the asymptotic series. The legend shows the number of terms (n) and the observed convergence order (m) for each solution.

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) [35].

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 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

$$\phi_e = \lim_{h \rightarrow 0} \phi(h), \quad (5.5)$$

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

$$\phi - \phi_e = g_0 + g_1 h + g_2 h^2 + g_3 h^3 + \dots . \quad (5.6)$$

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 [44] 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, \dots, h_n (e.g. multiple spatial dimensions, or spatial and angular grid resolutions), if the algorithm is second-order in each resolution parameter, then

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

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

CHAPTER VI

PRACTICAL APPLICATION

This chapter deals with the practical considerations when applying the previously discussed numerical algorithms to simulate the light field for realistic scenarios of kelp growing in ocean waters. This chapter is the culmination of this thesis: the spatial kelp distribution model of Chapter II and the radiative transfer model of Chapter III are combined using the numerical algorithms 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 based on the physical situation under consideration, the computational resources available, and the desired level of accuracy in the computed light field. Lastly, algorithm-specific parameters must also be tuned according to these considerations.

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 W m^{-2} according to (3.1).

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, 38]
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 W m^{-2}	[8]

In [31], detailed measurements of optical properties in various ocean waters are presented. A few of those measurements are reproduced here, using the same site names as in the original report. There are three categories of water provided: AUTEC is from Tongue of the Ocean, Bahama Islands, and represents 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 attenuation coefficients (a, b, c) are given, and their ratios.

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

6.1.2 Frond Alignment Coefficient

The *frond alignment coefficient*, η , describes the dependence of frond alignment on current speed. To the author’s knowledge, no such parameter is available in the literature. However, similar measurements have been made in the MACROSEA

project by Norvik [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 each frond to make their orientation 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, (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 [41] 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 encom-

passing 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

$$f(\omega) = I_0 \frac{\exp(-D_s \cos^{-1}(\omega_s \cdot \omega))}{\int_{2\pi} \exp(-D_s \cos^{-1}(\omega_s \cdot \omega)) d\omega}, \quad (6.1)$$

where ω_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 $D_s \rightarrow \infty$, 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

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

which has a local maximum of $\mu_l(2) = l_{\max}$. The value $l_{\max} = 6\text{ m}$ is used for the sake of agreement with [28]. The kelp frond length standard deviations are chosen to be the constant function $\sigma_l(z) = 1\text{ m}$. Also, the current angles are held constant over depth at $\theta_w(z) = 0$, as is the current speed $v_w = 1\text{ m s}^{-1}$ with $\eta = 1\text{ s m}^{-1}$. For the light field, $\omega_s = \hat{z}$ (sunlight from directly above) and $D_s = 1\text{ rad}^{-1}$ 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 for the matrix equation, 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.

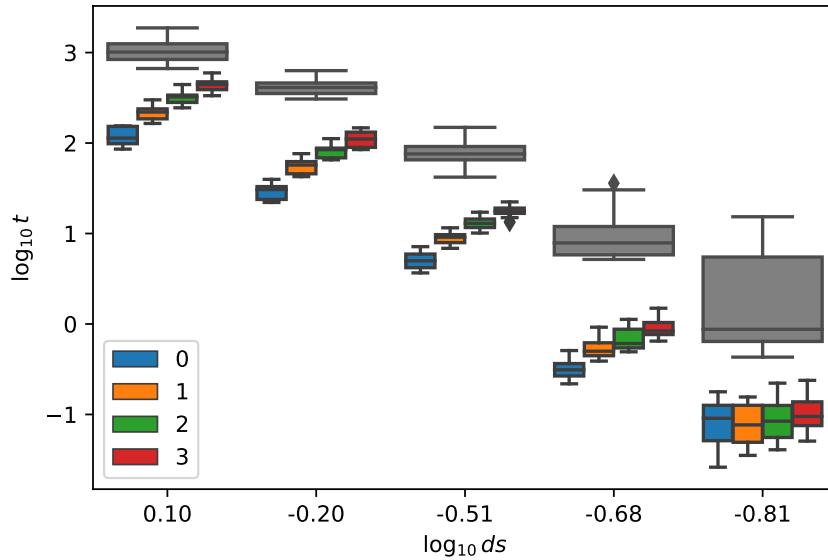


Figure 6.1: Computation time required for numerical asymptotics and finite difference algorithms over a range of spatial grid sizes using 32 CPUs. Only five grid sizes are shown, with the finite difference and numerical asymptotic algorithm for $n = 0, \dots, 3$ terms shown for each grid. The horizontal offset within each grid size is only for visual clarity.

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.

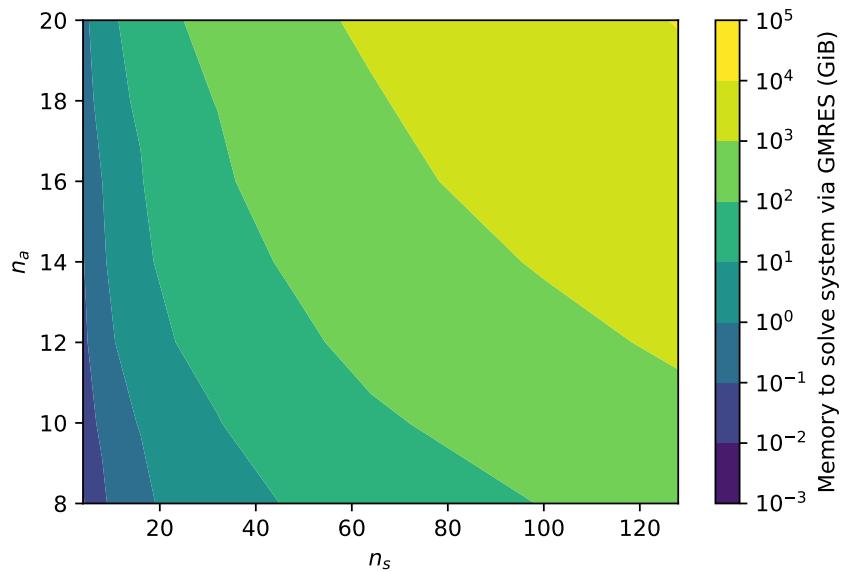


Figure 6.2: Estimated memory required to solve the linear system of equations for the finite difference algorithm using GMRES, restarted every 100 iterations. Table D.2 contains the same data in text form.

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 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 Estimatation

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 and da , given by Equations (5.3) and (5.4) with $x_{\max} - x_{\min} = y_{\max} - y_{\min} = 10$ m. All parameters aside from grid resolution are held constant, so discretization error is the only source of variation within each plot.

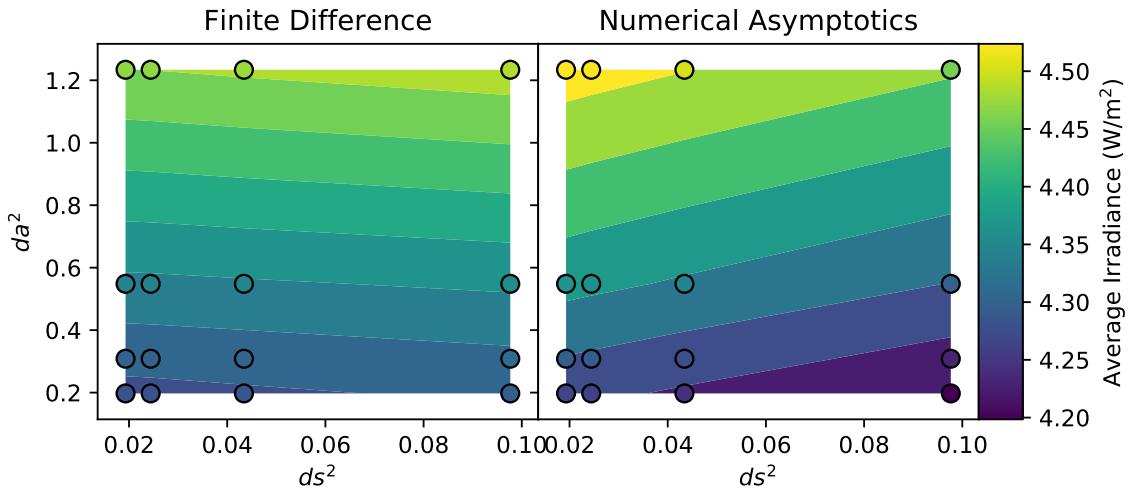


Figure 6.3: Average irradiance plotted against squared resolution for a variety of grid sizes for finite difference and numerical asymptotics with $n = 0$. The linear contours demonstrate that both methods are second order in both resolution parameters.

Notice that in Figure 6.4, the average irradiance is linear in squared grid spacing for each slice in the resolution space, which 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 average irradiance is planar over the whole two-dimensional squared-resolution space, as shown by the parallel linear contours in Figure 6.3.

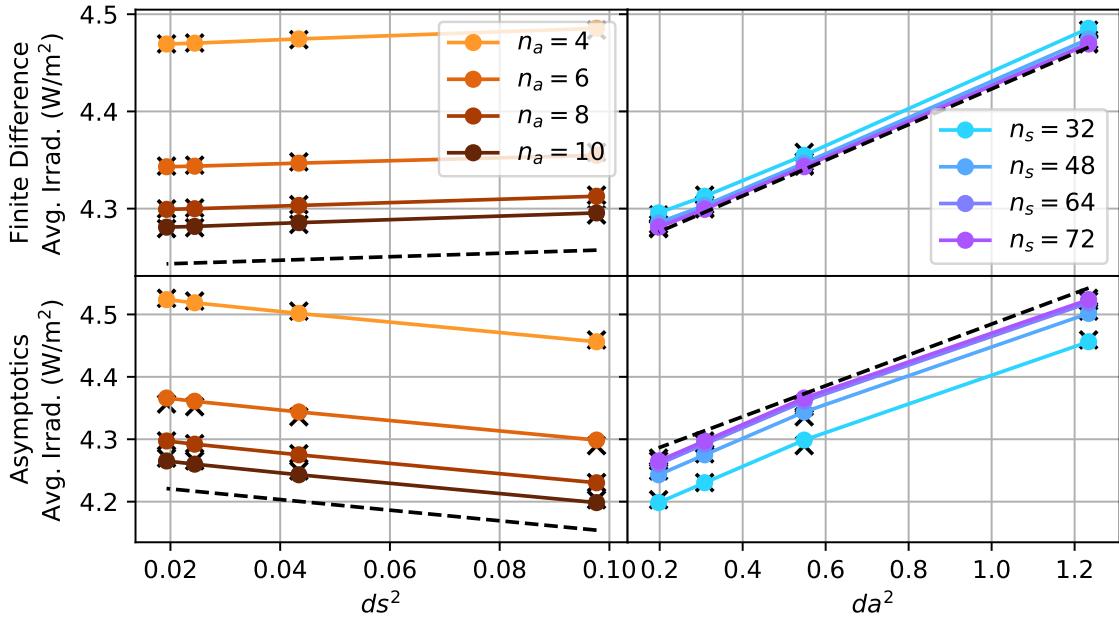


Figure 6.4: Average irradiance plotted against squared resolution. Each line is a one-dimensional projection from Figure 6.4. Predicted error values are marked with “x”s, and observed error values with circles. Isolated spatial and angular components of discretization error are plotted with a dashed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

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 be-

tween the prediction and observation is a testament to the validity of the generalized Richardson extrapolation method in this circumstance.

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

$$\varepsilon_{\text{FD}}(ds, da) = c_{11}ds^2 + c_{12}da^2, \quad (6.3)$$

$$\varepsilon_{\text{asym}}(ds, da) = c_{21}ds^2 + c_{22}da^2, \quad (6.4)$$

where

$$c_{11} = 0.180, \quad C_{12} = 0.183, \quad (6.5)$$

$$c_{21} = -0.851, \quad C_{22} = 0.247. \quad (6.6)$$

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 for 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 $2 \times 10^{-1} \text{ W m}^{-2}$, while the spatial discretization error is about $3 \times 10^{-2} \text{ W m}^{-2}$. 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 target error of $\bar{\varepsilon} = 2 \times 10^{-2} \text{ W m}^{-2}$. 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 $\bar{\varepsilon}_s = \bar{\varepsilon}_a = 1 \times 10^{-2} \text{ W m}^{-2}$. 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.

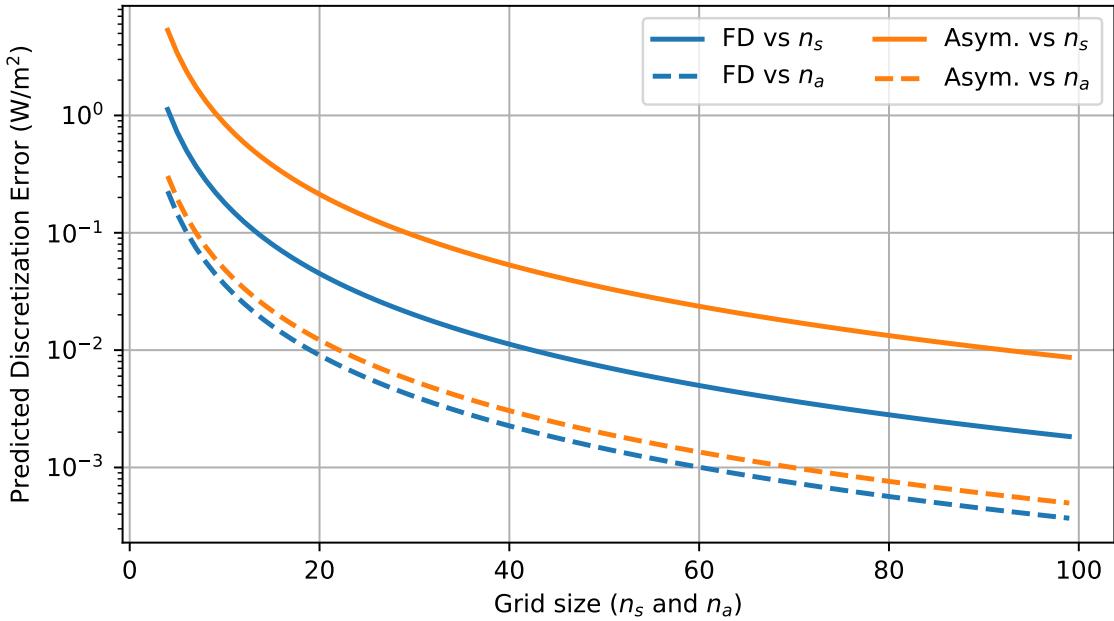


Figure 6.5: Predictions for isolated spatial and angular components of discretization error as a function of grid size for both numerical algorithms. Once a target error threshold has been chosen, this figure can be used to estimate the grid sizes that each algorithm would require. The total discretization 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 only for 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 m^{-1} to 0.5 m^{-1} (inclusive), and ten values of b are taken at even intervals on a log scale from 0.01 m^{-1} to 1.5 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 the truncation error is decreasing, the discretization error remains constant. Therefore, no results with this grid size will show errors below about 0.07 W m^{-2} . As a result, simulations with errors below this threshold are discarded in order to gain insight into trends in truncation error.

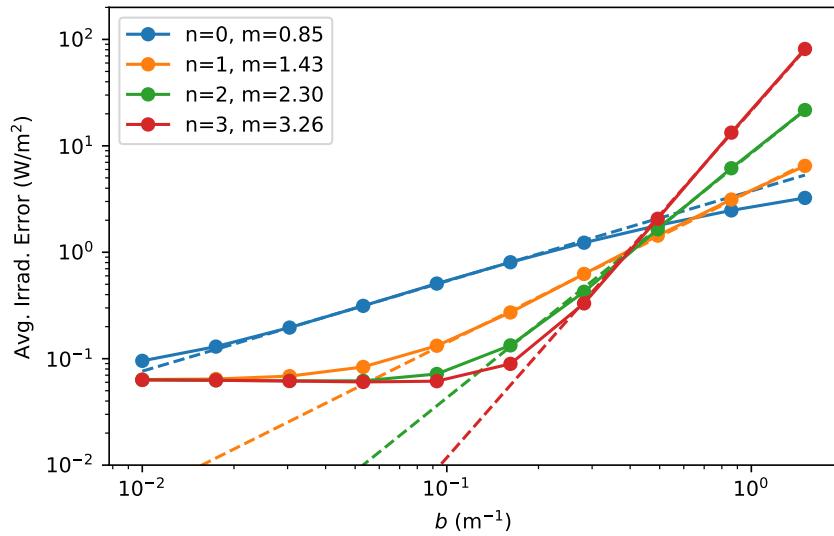


Figure 6.6: Average pointwise difference in irradiance between finite difference and asymptotics solutions for several values of b and n with constant $a = 0.1$ in a realistic kelp scenario using a 72×10 grid. Proper convergence of truncation error is observed between $b = 0.2$ and $b = 0.5$. Below $b = 0.2$, discretization error dominates. Above $b = 0.5$, 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 sufficiently from a_w and b which is sufficient to determine the accuracy of the approximation.

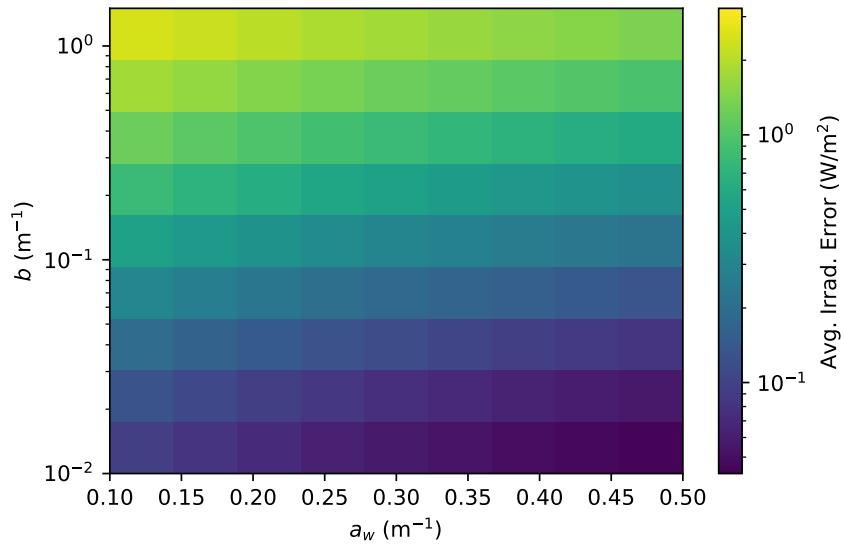


Figure 6.7: Average pointwise difference in irradiance between finite difference and asymptotics solutions for several values of b and a with constant $n = 0$ in a realistic kelp scenario using a 72×10 grid. Note that truncation error is smallest for low-scattering, high-absorption cases, and largest 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 zeroth order approximation is most accurate, showing that additional terms cause the solution to diverge. After a brief transition region, the third order approximation is most accurate, in agreement with Figure 6.6.

Note that for low scattering, high absorption waters where the error is lowest, the second order approximation seems to perform better than the third order. 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 second and third order 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.

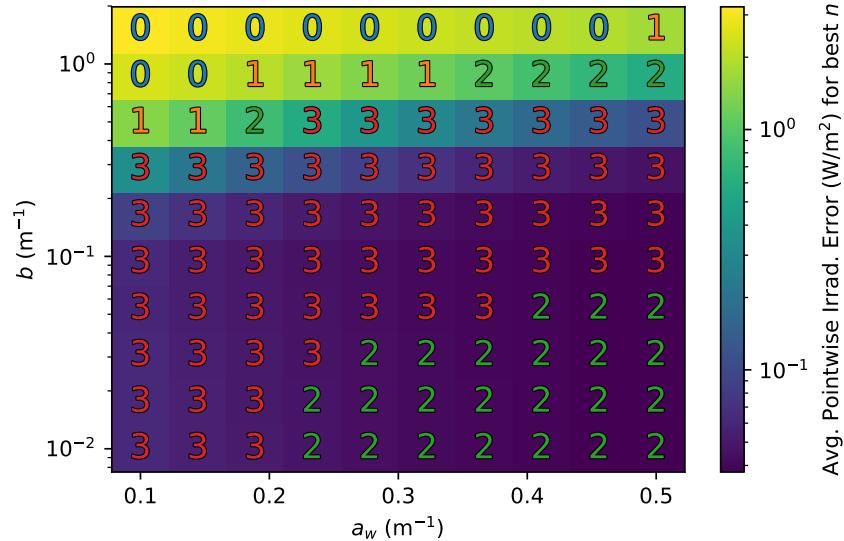


Figure 6.8: The best asymptotics solution for each (a_w, b) . The value of n used is written in each cell. For high-scattering cases, the $n > 0$ terms diverge, so $n = 0$ is the best approximation. For most cases, $n = 3$ is the best. For very low-scattering cases, discretization error masks the truncation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

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 $\bar{\varepsilon} = 1.0 \text{ W m}^{-2}$ and $\bar{\varepsilon} = 0.1 \text{ W m}^{-2}$ in the left and right columns 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.

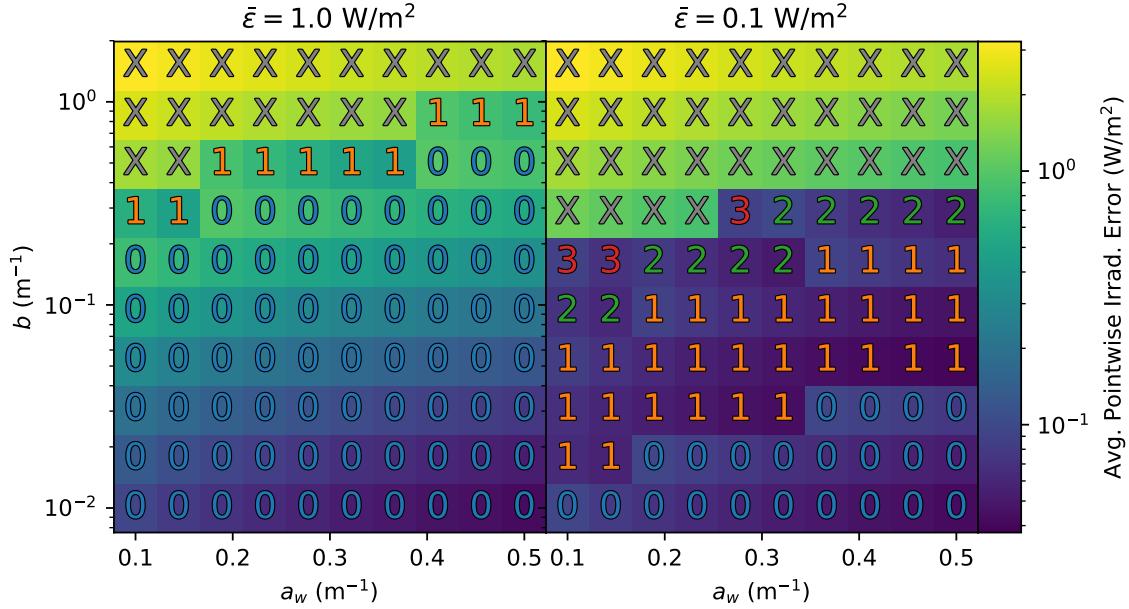


Figure 6.9: For each (a_w, b) , the smallest value of n which satisfies the error criterion shown above is printed in the cell. “X” denotes an optical situation in which the criterion cannot be met since adding further terms makes the solution less accurate, not more. As seen on the left the $n = 0$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

6.4.2 Truncation Error Model

Figure 6.10 concisely represents the simulation results for all optical properties and all 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 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 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

$$\varepsilon \propto \frac{b^{c_1}}{a_w^{c_2}}. \quad (6.7)$$

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 c_1 and c_2 which appear to be independent of n . Remarkably, the fit for each order of approximation yields $c_1 = 3/4$, $c_2 = 1/2$ to within a few percent variation. The black “x”s in Figure 6.10 are the predictions of this model, setting $c_1 = 3/4$ and $c_2 = 1/2$ explicitly. Notice how accurately the observed errors are predicted by the model. This also suggests the definition of a third parameter, as mentioned in Section 6.4.1,

$$\xi = \frac{b^{3/4}}{a_w^{1/2}}, \quad (6.8)$$

with the unusual units $\text{m}^{-1/4}$. 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.

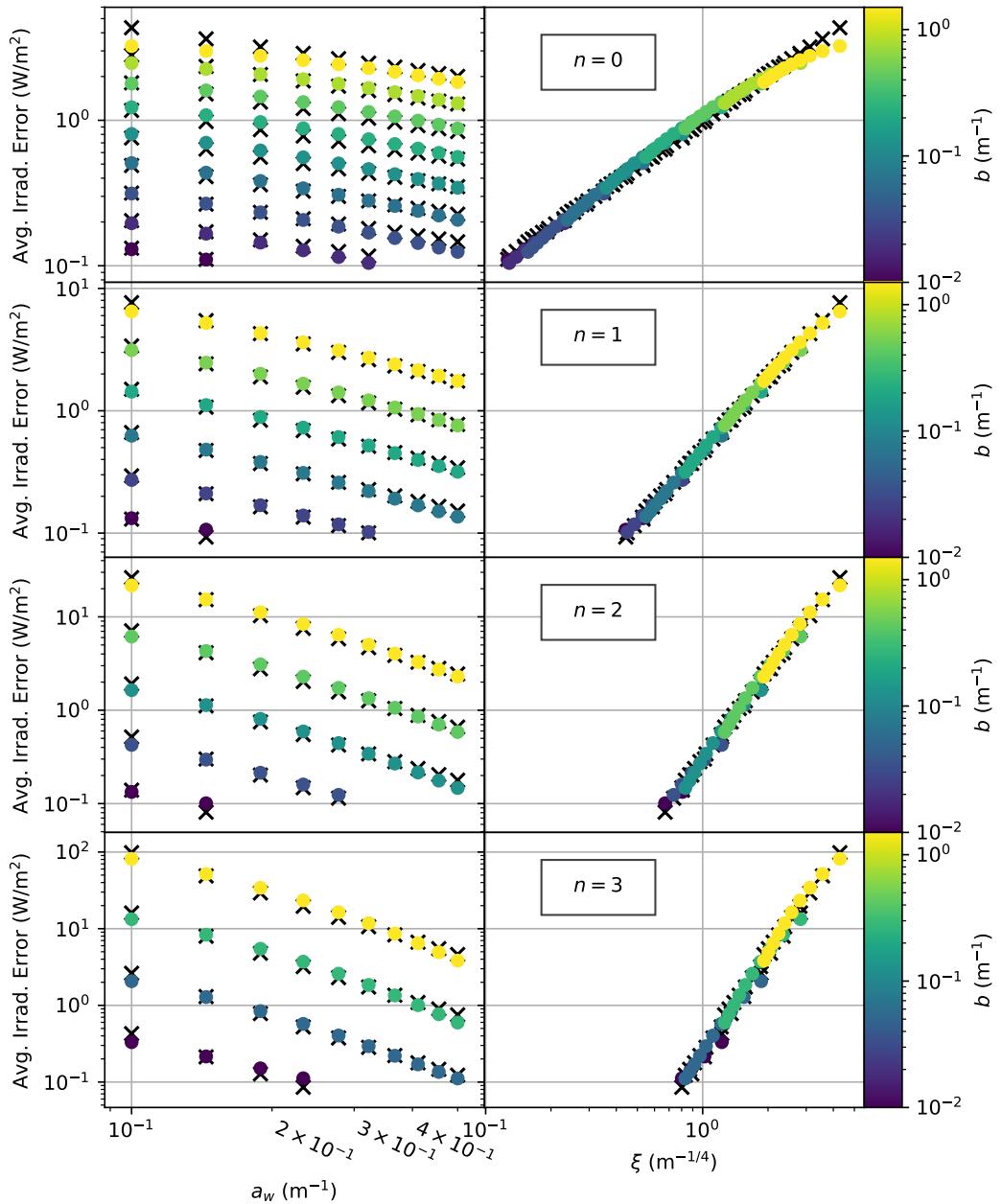


Figure 6.10: Truncation error plotted against a_w and ξ for each n . Simulations dominated by discretization error have been discarded. The right column shows that ξ sufficiently characterizes an optical situation for the sake of predicting truncation error of the asymptotic approximation. Predicted values, marked with “x”s, are very accurate.

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 seen only for b . Note that the convergence curves for all lines appear to roughly intersect at a characteristic point. That point, denoted (ξ^*, ε^*) , is significant because it represents a bifurcation in the convergence behavior of the numerical asymptotics algorithm. For waters with $\xi < \xi^*$, adding terms to the series increases the accuracy of the approximation, whereas for waters with ξ above the threshold value ξ^* , adding terms decreases the accuracy.

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 satisfies

$$\ln(\varepsilon_n(\xi)) - \ln \varepsilon^* = (n + 1)(\ln \xi - \ln \xi^*), \quad (6.9)$$

or equivalently,

$$\varepsilon_n(\xi) = \varepsilon^* \left(\frac{\xi}{\xi^*} \right)^{n+1}. \quad (6.10)$$

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 runs 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

$$\xi^* = 1.58 \text{ m}^{-1/4}, \quad (6.11)$$

$$\varepsilon^* = 1.30 \text{ W m}^{-2}, \quad (6.12)$$

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.

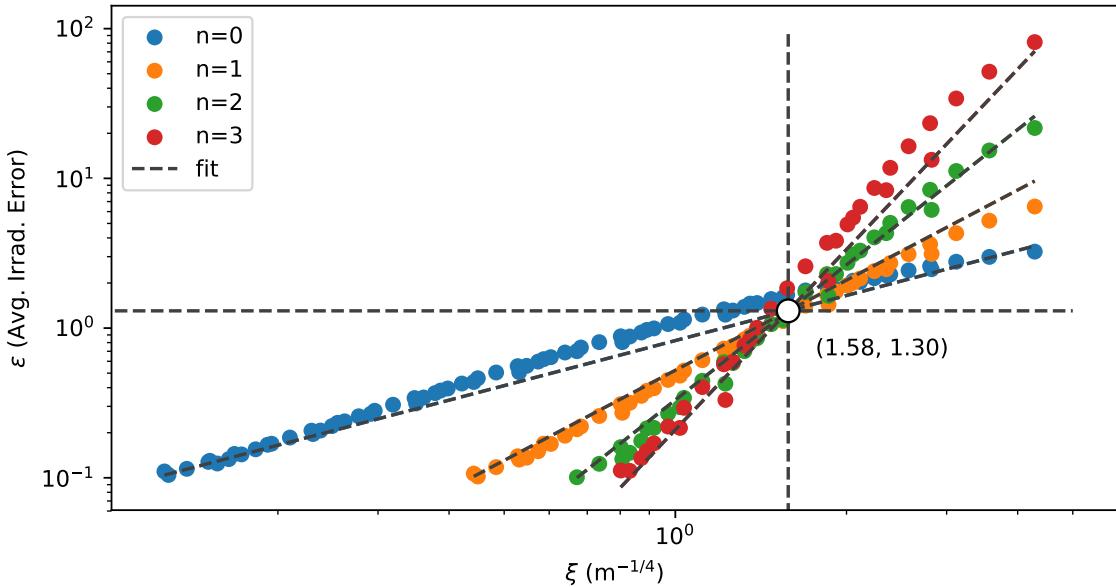


Figure 6.11: All data from the right column of 6.10 on a single plot demonstrates a characteristic $\xi = \xi^*$ above which the asymptotic series diverges. This observation permits a simple model for predicting truncation error.

The determination of ξ^* and ε^* marks the construction of a simple analytical model given by Equation (6.10) 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 keep in mind 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.

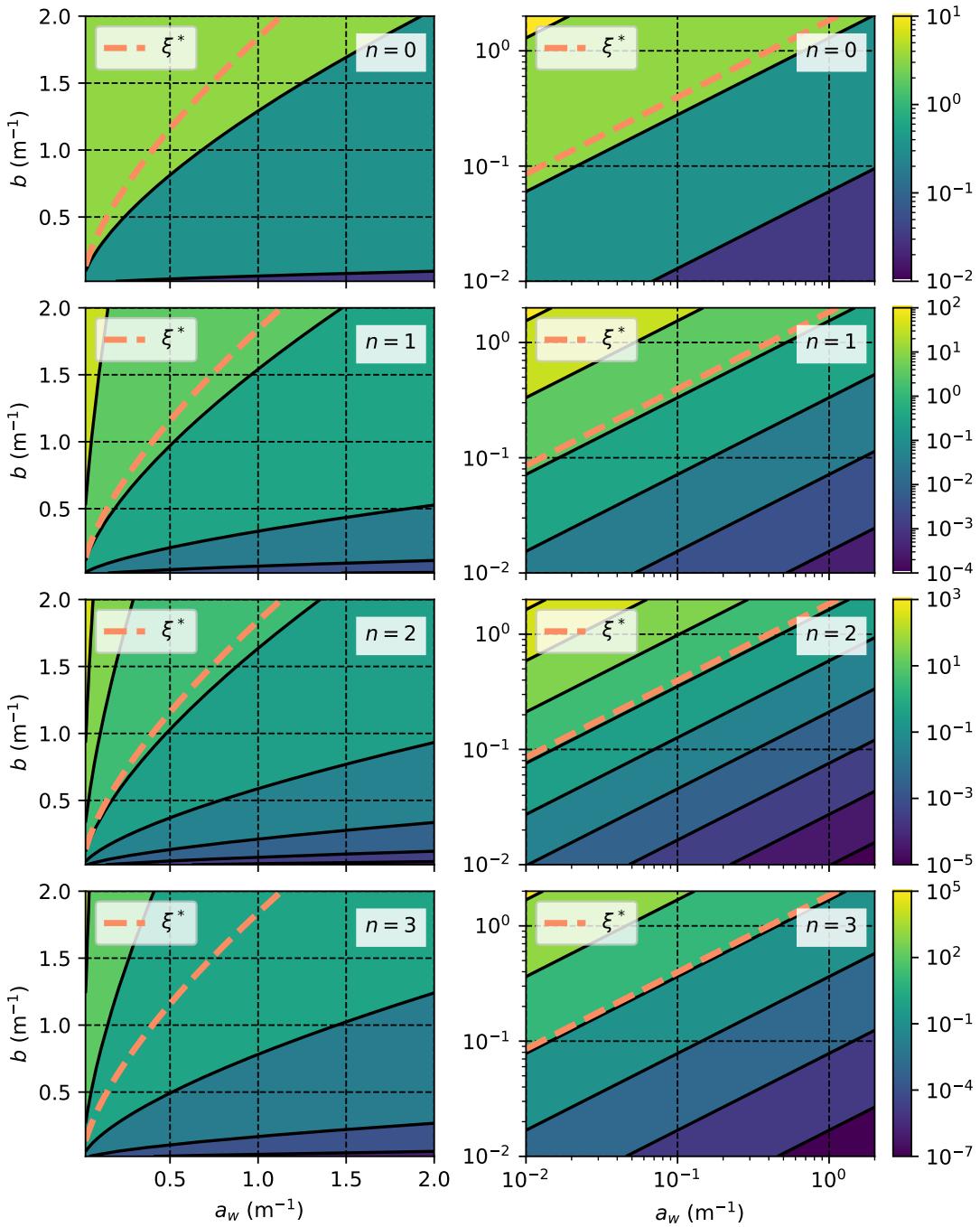


Figure 6.12: Predicted truncation error for a range of a_w and b values. n varies over plot rows. Linear–linear scale on the left, log–log scale on the right. The asymptotic series converges only for (a_w, b) values to the right of the ξ^* contour.

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 be determined by solving Equation (6.9) 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

$$\bar{n} = \text{floor} \left(\frac{\ln \bar{\varepsilon} - \ln \varepsilon^*}{\ln \xi - \ln \xi^*} \right). \quad (6.13)$$

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 $\bar{n} \rightarrow \infty$ 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 (6.13). 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.

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 $n = \bar{n}(a_w, b)$.

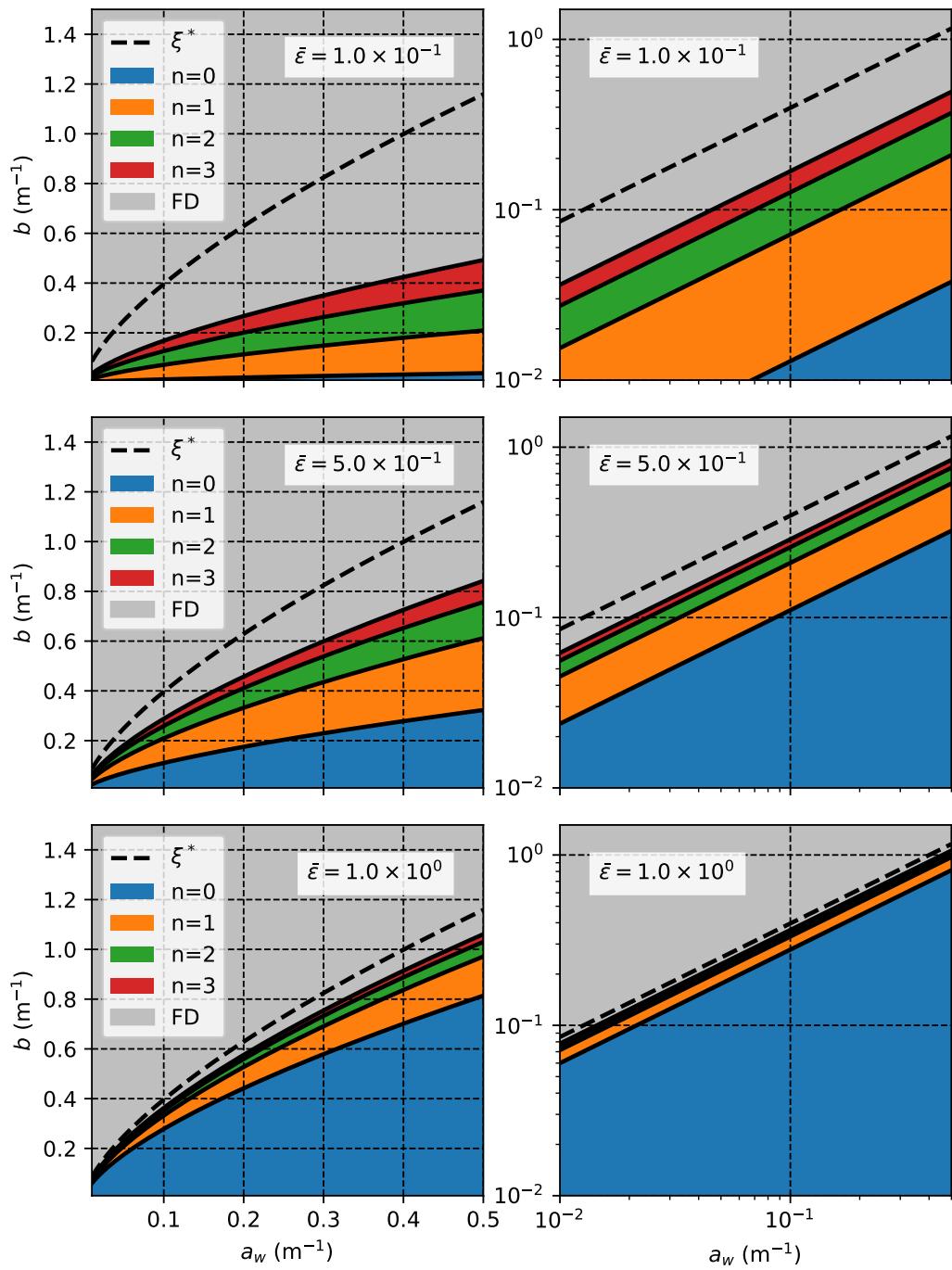


Figure 6.13: Recommended $n = \bar{n}$ value as a function of (a_w, b) to achieve the truncation error criteria $\bar{\varepsilon} = 0.1, 0.5, 1.0 \text{W/m}^2$ for the three plot rows.

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 used. The irradiance in this case is simply

$$I(z) = \exp(-a_w z). \quad (6.14)$$

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

$$\frac{dI}{dz} = - \left(a_w + k_{\text{kelp}}(z) \right), \quad (6.15)$$

$$k_{\text{kelp}}(z) = -\ln(1 - A_k(1 - (1 - AD)^{\rho_f(z)})), \quad (6.16)$$

where A_k is the kelp absorptance, D is the number of vertical kelp ropes per horizontal m², $A(z)$ is the average area of the kelp fronds per meter vertical rope, and ρ_f is the number of kelp fronds per vertical meter of rope. Then,

$$D = \frac{1}{(x_{\max} - x_{\min})(y_{\max} - y_{\min})}, \quad (6.17)$$

and the mean frond A is calculated from the mean frond length according to Equation (2.3) as

$$A = \frac{\mu_l^2}{2f_r}. \quad (6.18)$$

Once $k_{\text{kelp}}(z)$ is calculated according to Equation (6.16), then the solution of Equation (6.15) is a simplified version of Equation (3.10) with $\tilde{a}(z) = a_w + k_{\text{kelp}}(z)$ and $\tilde{\sigma}_0 = 0$. The solution is therefore the analogous simplification of Equation (3.11), namely

$$I(z) = I_0 \exp \left(-a_w z - \int_0^z k_{\text{kelp}}(z') dz' \right). \quad (6.19)$$

6.5.2 Description of Comparison

In Figure 6.14, 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 [?]. 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; they 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 6.14. 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 greater depths.

6.5.4 Comparison: Total Radiant Flux

Figure 6.15 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* (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 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 full spatial kelp distribution is known. The large degree of these differences in model predictions highlight the importance of considering this spatial distribution.

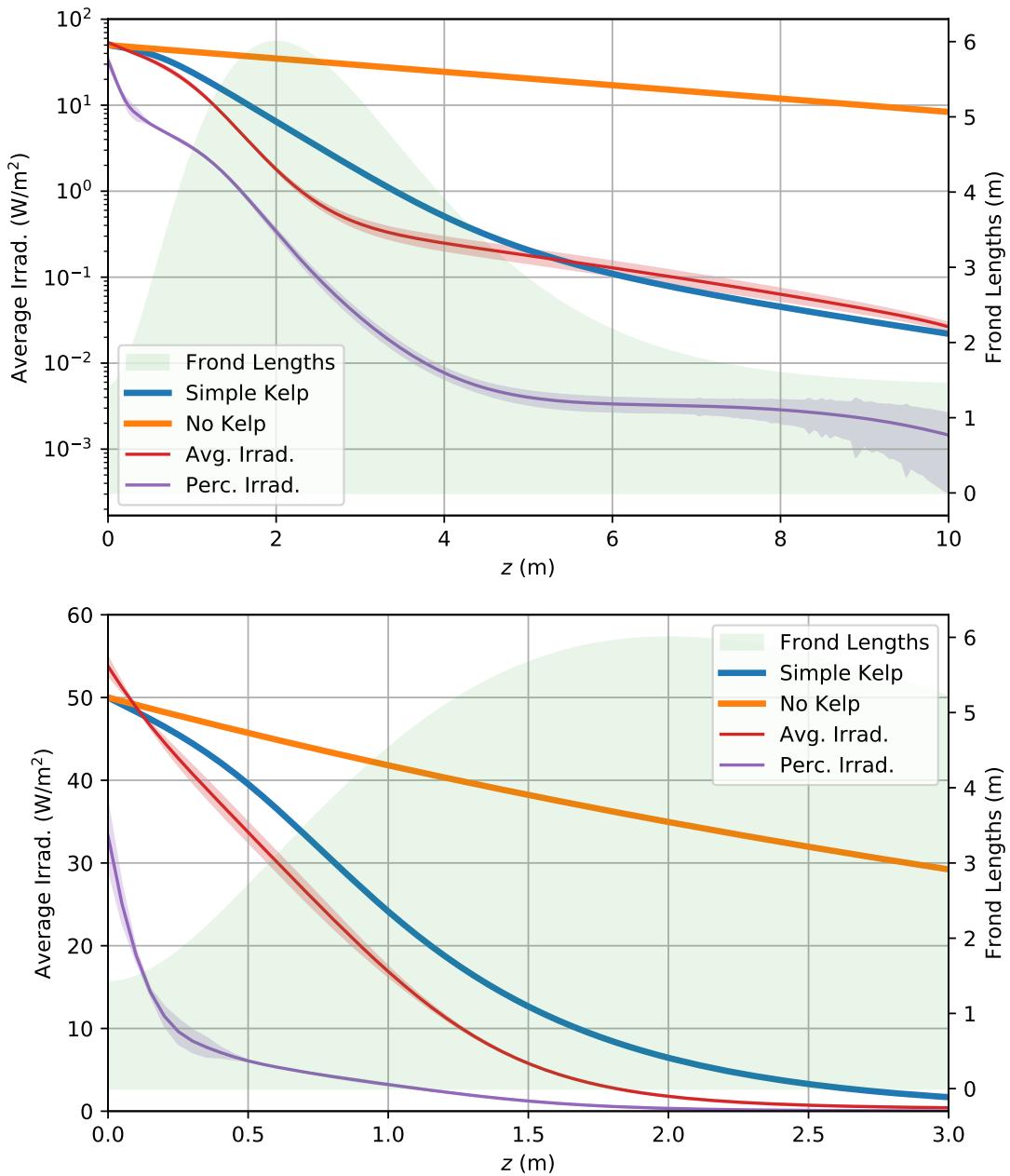


Figure 6.14: Average irradiance and perceived irradiance from finite difference compared to simpler light models for the case of coastal California water (HAOCE11) with realistic kelp growth. Frond lengths over depth are shown on the right-hand axis. Discretization error is shown by error bands for perc. irrad. and avg. irrad. Note the different scales in the two plots.

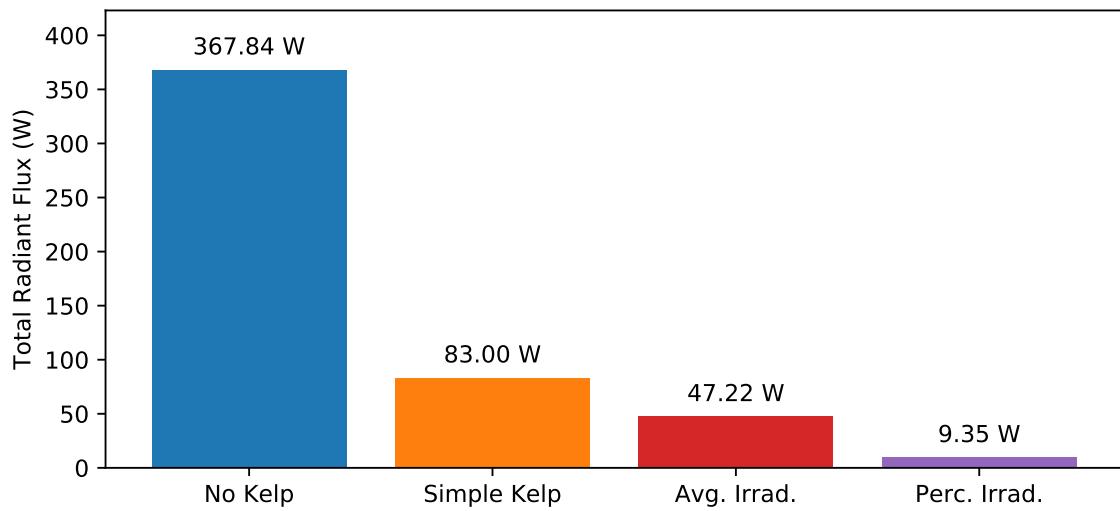


Figure 6.15: Total radiant flux in Watts through kelp predicted by each model.

The magnitude of the differences highlight the importance of considering the three-dimensional spatial kelp distribution.

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 computationally cheaper 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 ex-

pected, 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. Each of these assumptions are vast simplifications from the real system, and leave room for future improvement.

- All fronds in the population are congruent kites of equal 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.

- Absorption coefficient of the aquatic medium is constant within a depth layer.
- All fronds have the same absorptance.
- The scattering coefficient is constant everywhere and equal for both kelp and water.
- Only the volumetric optical effects, not surface effects, of the kelp 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 (2.4.4). 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 $b(\mathbf{x}, \bar{b})$ 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 (3.12) 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 (3.12) 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 multi-threading 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” [6, 2, 5]. 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 with 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. 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] P. J. Roache. *Verification and validation in computational science and engineering*. Hermosa, Albuquerque, NM, 1998.
- [33] P. J. Roache. Verification of Codes and Calculations. *AIAA Journal*, 36(5):696–702, May 1998.
- [34] P. J. Roache. Building PDE codes to be verifiable and validatable. *Computing in Science Engineering*, 6(5):30–38, Sept. 2004.
- [35] 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.
- [36] Y. Saad and M. H. Schultz. GMRES: a Generalized Minimal Residual algorithm for solving nonsymmetric linear systems. Research Report YALEU/DCS/RR-254, Yale University, May 1985.

- [37] M. Scheffer, J. Baveco, D. DeAngelis, K. Rose, and E. van Nes. Super-individuals a simple solution for modelling large populations on an individual basis. *Eco-logical Modelling*, 80:161–170, Mar. 1994.
- [38] 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.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.

- [43] 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.
- [44] C. Yu, W. Yao, and X. Bai. Robust Linear Regression: A Review and Comparison. *arXiv:1404.6274 [stat]*, Apr. 2014. arXiv: 1404.6274.

APPENDICES

APPENDIX A

GRID DETAILS

The width of the spatial grid cells in each dimension are

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

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

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

and the cell centers as

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

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

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

Denote the edges as

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

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

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

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

Now, we define the azimuthal angle such that

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

For the sake of periodicity, we need

$$\theta_1 = 0,$$

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

which requires

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

For the polar angle, we similarly let

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

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

$$\phi_1 = 0,$$

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

This gives us

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

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

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

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

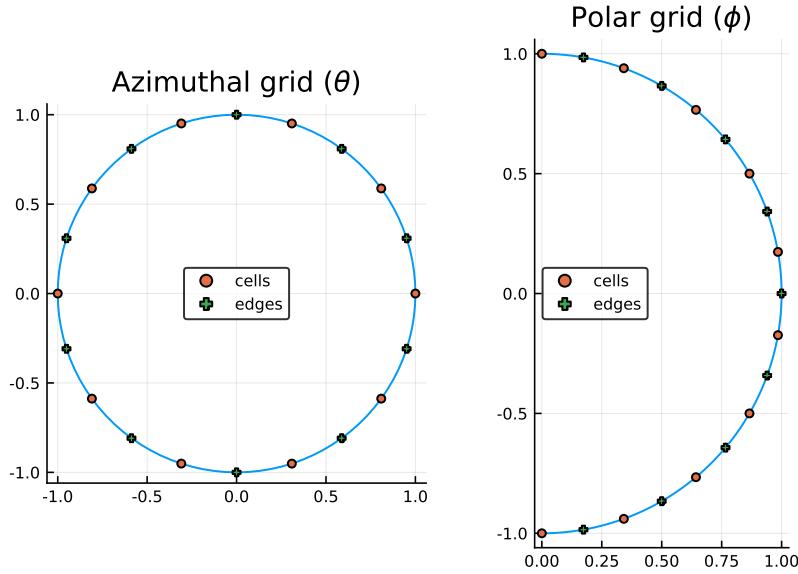


Figure A.1: Angular grid

Note that while θ has its final edge following its final center, this is not the case for ϕ , as seen in Figure A.1. Because angles are indexed by a single integer p , there is a one-to-one relationship between an integer p and a pair (l, m) . The relationships are as follows:

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

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

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

Accordingly, define

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

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

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

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

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

For all other angular grid cells,

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

APPENDIX B

RAY TRACING ALGORITHM

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

Consider a grid center $\mathbf{p}_1 = (p_{1x}, p_{1y}, p_{1z})$ and a corresponding path $\mathbf{l}(\mathbf{x}_1, \omega, s)$. To find the location of discontinuities in the integrand, we first calculate the distance from its origin, $\mathbf{p}_0 = \mathbf{x}_0(\mathbf{p}_1, \omega) = (p_{0x}, p_{0y}, p_{0z})$ (as in (3.3)) to grid edges in each dimension separately. Given

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

simply to keep 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 \mathbf{p}_0 . That is,

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

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

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

Then,

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

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

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

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

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

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

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

For each grid cell, we check the path lengths required to cross the next x , y , and z edge–planes. Then, we move to the next grid cell in whichever dimension is crossed soonest.

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

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

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

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

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

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

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

$n_s \backslash n_a$	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.

$n_a \backslash n_s$	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 up-to-date version of this code can be found online at <https://github.com/OliverEvans96/kelp>.

```
utils.f90
1 ! General utilities which might be useful in
2     other settings
3 module utils
4 implicit none
5
6 ! Constants
7 double precision, parameter :: pi = 4.D0 * datan
8     (1.D0)
9
10 contains
11
12 ! Determine base directory relative to current
13 ! directory
14 ! by looking for Makefile, which is in the base
15 ! dir
16 ! Assuming that this is executed from within the
17 ! git repo.
18 function getbasedir()
19     implicit none
20
21     ! INPUTS:
22     ! Number of paths to check
23     integer, parameter :: numpaths = 3
24     ! Maximum length of path names
25     integer, parameter :: maxlenlength = numpaths *
26         2 - 1
27     ! Paths to check for Makefile
28     character(len=maxlength), parameter,
29         dimension(numpaths) :: check_paths &
30         = (/ '.', '..', '..', '..', '..' /)
31     ! Temporary path string
32     character(len=maxlength) tmp_path
33     ! Whether Makefile has been found yet
34     logical found
```

```

28      ! Path counter
29      integer ii
30      ! Lengths of paths
31      integer, dimension(numpaths) :: pathlengths
32
33      ! OUTPUT:
34      ! getbasedir - relative path to base
35      !   directory
36      ! Will either return '.', '..', or '../..'
37      character(len=maxlength) getbasedir
38
39      ! Determine length of each path
40      pathlengths(1) = 1
41      do ii = 2, numpaths
42          pathlengths(ii) = 2 + 3 * (ii - 2)
43      end do
44
45      ! Loop through paths
46      do ii = 1, numpaths
47          ! Determine this path
48          tmp_path = check_paths(ii)
49
50          ! Check whether Makefile is in this
51          !   directory
52          !write(*,*) 'Checking ', tmp_path(1:
53          !   pathlengths(ii)), ''
54          inquire(file=tmp_path(1:pathlengths(ii)))
55          !// '/Makefile', exist=found)
56          ! If so, stop. Otherwise, keep looking.
57          if(found) then
58              getbasedir = tmp_path(1:pathlengths(
59                  ii))
60              exit
61          end if
62      end do
63
64      ! If it hasn't been found, then this script
65      !   was probably called
66      ! from outside of the repository.
67      if(.not. found) then
68          write(*,*) 'BASE DIR NOT FOUND.'
69      end if
70
71  end function
72
73      ! Determine array size from min, max and step
74      ! If alignment is off, array will overstep the
75      !   maximum
76  function bnd2max(xmin,xmax,dx)
77      implicit none
78
```

```

73      ! INPUTS:
74      ! xmin - minimum x value in array
75      ! xmax - maximum x value in array (inclusive
76          )
76      ! dx - step size
77      double precision, intent(in) :: xmin, xmax,
78          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
87      ! 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
94          )
95      double precision, intent(in) :: xmin, xmax
96      ! imax - number of elements in array
97      integer imax
98
99      ! OUTPUT:
100     ! bnd2arr - array to generate
101     double precision, dimension(imax) :: bnd2arr
102
103     ! BODY:
104
105     ! Counter
106     integer ii
107     ! Step size
108     double precision dx
109
110     ! Calculate step size
111     dx = (xmax - xmin) / imax
112
113     ! Generate array
114     do ii = 1, imax
115         bnd2arr(ii) = xmin + (ii-1) * dx
116     end do
117
118  end function
119
120  function mod1(i, n)

```

```

121      implicit none
122      integer i, n, m
123      integer mod1
124
125      m = modulo(i, n)
126
127      if(m .eq. 0) then
128          mod1 = n
129      else
130          mod1 = m
131      end if
132
133  end function mod1
134
135  function sgn_int(x)
136      integer x, sgn_int
137      ! Standard signum function
138      sgn_int = sign(1,x)
139      if(x .eq. 0.) sgn_int = 0
140  end function sgn_int
141
142  function sgn(x)
143      double precision x, sgn
144      ! Standard signum function
145      sgn = sign(1.d0,x)
146      if(x .eq. 0.) sgn = 0
147  end function sgn
148
149  ! Interpolate single point from 1D data
150  function interp(x0,xx,yy,nn)
151      implicit none
152
153      ! INPUTS:
154      ! x0 - x value at which to interpolate
155      double precision, intent(in) :: x0
156      ! xx - ordered x values at which y data is
157      ! sampled
158      ! yy - corresponding y values to interpolate
159      double precision, dimension (nn), intent(in)
160      :: xx,yy
161      ! nn - length of data
162      integer, intent(in) :: nn
163
164      ! OUTPUT:
165      ! interp - interpolated y value
166      double precision interp
167
168      ! BODY:
169
170      ! Index of lower-adjacent data (xx(i) < x0 <
171      ! xx(i+1))
172      integer ii

```

```

170 ! Slope of liine between (xx(ii),yy(ii)) and
171     (xx(ii+1),yy(ii+1))
172 double precision mm
173 ! If out of bounds , then return endpoint
174 value
175 if (x0 < xx(1)) then
176     interp = yy(1)
177 else if (x0 > xx(nn)) then
178     interp = yy(nn)
179 else
180
181     ! Determine ii
182 do ii = 1, nn
183     if (xx(ii) > x0) then
184         ! We've now gone one index too far
185         .
186         exit
187     end if
188 end do
189
190 ! Determine whether we're on the right
191 ! endpoint
192 if(ii-1 < nn) then
193     ! If this is a legitimate
194     ! interpolation , then
195     ! subtract since we went one index too
196     ! far
197     ii = ii - 1
198
199 ! Calculate slope
200 mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
201     xx(ii))
202
203 ! Return interpolated value
204 interp = yy(ii) + mm * (x0 - xx(ii))
205 else
206     ! If we're actually interpolating the
207     ! right endpoint,
208     ! then just return it.
209     interp = yy(nn)
210 end if
211
212 end if
213
214 end function
215
216 ! Calculate unshifted position of periodic image
217 ! Assuming xmin , xmax are extreme attainable
218 ! values of x
219 function shift_mod(x, xmin, xmax)
220     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
217 |             between image & lower bound
218 |         shift_mod = xmin + mod_part
219 |     else
220 |         ! In this case, mod_part is distance
221 |             between image & upper bound
222 |         shift_mod = xmax + mod_part
223 |     endif
224 | end function shift_mod
225 |
226 ! Bilinear interpolation on evenly spaced 2D
227 ! grid
228 ! Assume upper endpoint is not included and is
229 ! identical
230 ! to the lower endpoint, which is included.
231 function bilinear_array_periodic(x, y, nx, ny,
232     x_vals, y_vals, fun_vals)
233 implicit none
234 double precision x, y
235 integer nx, ny
236 double precision, dimension(:) :: x_vals,
237     y_vals
238 double precision, dimension(:, :) :: fun_vals
239
240 double precision bilinear_array_periodic
241
242 xmin = x_vals(1)
243 ymin = y_vals(1)
244 dx = x_vals(2) - x_vals(1)
245 dy = y_vals(2) - y_vals(1)
246
247 ! Add 1 for one-indexing
248 i0 = int(floor((x-xmin)/dx))+1
249 j0 = int(floor((y-ymin)/dy))+1
250
251 x0 = x_vals(i0)
252 y0 = y_vals(j0)
253
254 ! Periodic wrap
255 if(i0 .lt. nx) then
256     i1 = i0 + 1
257     x1 = x_vals(i1)
258 else
259     i1 = 1

```

```

259     x1 = x_vals(nx) + dx
260   endif
261
262   if(j0 .lt. ny) then
263     j1 = j0 + 1
264     y1 = y_vals(j1)
265   else
266     j1 = 1
267     y1 = y_vals(ny) + dy
268   endif
269
270   z00 = fun_vals(i0,j0)
271   z10 = fun_vals(i1,j0)
272   z01 = fun_vals(i0,j1)
273   z11 = fun_vals(i1,j1)
274
275   bilinear_array_periodic = bilinear(x, y, x0,
276                                       y0, x1, y1, z00, z01, z10, z11)
276 end function bilinear_array_periodic
277
278 ! Bilinear interpolation on evenly spaced 2D
279 ! grid
280 ! Assume upper and lower endpoints are included
280 function bilinear_array(x, y, x_vals, y_vals,
281                         fun_vals)
281 implicit none
282 double precision x, y
283 double precision, dimension(:) :: x_vals,
283         y_vals
284 double precision, dimension(:, :) :: fun_vals
285
286 double precision dx, dy, xmin, ymin
287 integer i0, j0, i1, j1
288 double precision x0, x1, y0, y1
289 double precision z00, z10, z01, z11
290
291 double precision bilinear_array
292
293 xmin = x_vals(1)
294 ymin = y_vals(1)
295 dx = x_vals(2) - x_vals(1)
296 dy = y_vals(2) - y_vals(1)
297
298 ! Add 1 for one-indexing
299 i0 = int(floor((x-xmin)/dx))+1
300 j0 = int(floor((y-ymin)/dy))+1
301 i1 = i0 + 1
302 j1 = j0 + 1
303
304 ! Bounds checking
305 ! if(i0 .lt. 1) then

```

```

306   !      i0 = 1
307   !      i1 = 1
308   ! else if(i1 .gt. nx) then
309   !      i0 = nx
310   !      i1 = nx
311   ! endif
312   ! if(j0 .lt. 1) then
313   !      j0 = 1
314   !      j1 = 1
315   ! else if(j1 .gt. ny) then
316   !      j0 = ny
317   !      j1 = ny
318   ! endif
319
320   x0 = x_vals(i0)
321   x1 = x_vals(i1)
322   y0 = y_vals(j0)
323   y1 = y_vals(j1)
324
325   z00 = fun_vals(i0,j0)
326   z10 = fun_vals(i1,j0)
327   z01 = fun_vals(i0,j1)
328   z11 = fun_vals(i1,j1)
329
330   bilinear_array = bilinear(x, y, x0, y0, x1, y1
331   , z00, z01, z10, z11)
332 end function bilinear_array
333
334 ! ilinear interpolation of a function of two
335 ! variables
336 ! over a rectangle of points.
337 ! Weight each point by the area of the sub-
338 ! rectangle involving
339 ! the point (x,y) and the point diagonally
340 ! across the rectangle
341
342 function bilinear(x, y, x0, y0, x1, y1, z00, z01
343   , z10, z11)
344 implicit none
345 double precision x, y
346 double precision x0, y0, x1, y1, z00, z01, z10
347   , z11
348 double precision a, b, c, d
349 double precision bilinear
350
351 a = (x-x0)*(y-y0)
352 b = (x1-x)*(y-y0)
353 c = (x-x0)*(y1-y)
354 d = (x1-x)*(y1-y)
355
356 bilinear = (a*z11 + b*z01 + c*z10 + d*z00) / (
357   a + b + c + d)

```

```

350 | end function bilinear
351 |
352 | ! Integrate using left endpoint rule
353 | ! Assuming the right endpoint is not included in
354 | arr
354 | function lep_rule(arr,dx,nn)
355 |   implicit none
356 |
357 |   ! INPUTS:
358 |   ! arr - array to integrate
359 |   double precision, dimension(nn) :: arr
360 |   ! dx - array spacing (mesh size)
361 |   double precision dx
362 |   ! nn - length of arr
363 |   integer, intent(in) :: nn
364 |
365 |   ! OUTPUT:
366 |   ! lep_rule - integral w/ left endpoint rule
367 |   double precision lep_rule
368 |
369 |   ! BODY:
370 |
371 |   ! Counter
372 |   integer ii
373 |
374 |   ! Set output to zero
375 |   lep_rule = 0.0d0
376 |
377 |   ! Accumulate integral
378 |   do ii = 1, nn
379 |     lep_rule = lep_rule + arr(ii) * dx
380 |   end do
381 |
382 | end function
383 |
384 | ! Integrate using trapezoid rule
385 | ! Assuming both endpoints are included in arr
386 | function trap_rule_dx(arr, dx, nn)
387 |   implicit none
388 |   double precision, dimension(nn) :: arr
389 |   double precision dx
390 |   integer ii, nn
391 |   double precision trap_rule_dx
392 |
393 |   trap_rule_dx = 0.0d0
394 |
395 |   do ii=1, nn-1
396 |     trap_rule_dx = trap_rule_dx + 0.5d0 * dx *
397 |                   (arr(ii) + arr(ii+1))
398 |   end do
399 |
400 | end function trap_rule_dx

```

```

401 ! Integrate using trapezoid rule
402 ! Assuming both endpoints are included in arr
403 function trap_rule_uneven(xx, yy, nn)
404   implicit none
405   double precision, dimension(nn) :: xx
406   double precision, dimension(nn) :: yy
407   integer ii, nn
408   double precision trap_rule_uneven
409
410   trap_rule_uneven = 0.0d0
411
412   do ii=1, nn-1
413     trap_rule_uneven = trap_rule_uneven + 0.5d0
414     * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii
415       +1))
416   end do
417 end function trap_rule_uneven
418
419 function trap_rule_dx_uneven(dx, yy, nn)
420   implicit none
421   double precision, dimension(nn-1) :: dx
422   double precision, dimension(nn) :: yy
423   integer ii, nn
424   double precision trap_rule_dx_uneven
425
426   trap_rule_dx_uneven = 0.0d0
427
428   do ii=1, nn-1
429     trap_rule_dx_uneven = trap_rule_dx_uneven +
430       0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
431   end do
432 end function trap_rule_dx_uneven
433
434 ! Integrate using midpoint rule
435 ! First and last bins, only use inner half
436 function midpoint_rule_halfends(dx, yy, nn)
437   result(integral)
438   implicit none
439   integer ii, nn
440   double precision, dimension(nn) :: dx, yy
441   double precision integral
442
443   if(nn > 1) then
444     integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(
445       nn))
446
447     do ii=2, nn-1
448       integral = integral + dx(ii)*yy(ii)
449     end do
450   else
451     integral = 0.d0
452   end if

```

```

448 | end function midpoint_rule_halfends
449 |
450 ! Normalize 1D array and return integral w/ left
451 | endpoint rule
451 function normalize_dx(arr, dx, nn)
452   implicit none
453
454   ! INPUTS:
455   ! arr - array to normalize
456   double precision, dimension(nn) :: arr
457   ! dx - array spacing (mesh size)
458   double precision dx
459   ! nn - length of arr
460   integer, intent(in) :: nn
461
462   ! OUTPUT:
463   ! normalize - integral before normalization
463   ! (left endpoint rule)
464   double precision normalize_dx
465
466   ! BODY:
467
468   ! Calculate integral
469   normalize_dx = lep_rule(arr, dx, nn)
470
471   ! Normalize array
472   arr = arr / normalize_dx
473
474 end function normalize_dx
475
476 ! Normalize 1D unevenly-spaced array and
477 ! return integral w/ trapezoid rule
478 ! Will not be quite accurate if rightmost
478 | 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(
481   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 (
491   ! left endpoint rule)
492   double precision norm
493
494   ! BODY:

```

```

495      ! Calculate integral
496      ! PERHAPS WE SHOULD USE TRAPEZOID RULE
497      norm = trap_rule_uneven(xx, yy, nn)
498
499      ! Normalize array
500      yy(:) = yy(:) / norm
501
502
503  end function normalize_uneven
504
505  ! Read 2D array from file
506  function read_array(filename,fmtstr,nn,mm,
507    skiplines_in)
508    implicit none
509
510    ! INPUTS:
511    ! filename - path to file to be read
512    ! fmtstr - input format (no parentheses, don
513    !   't specify columns)
514    ! e.g. 'E10.2', not '(2E10.2)'
515    character(len=*), intent(in) :: filename,
516      fmtstr
517    ! nn - Number of data rows in file
518    ! mm - number of data columns in file
519    integer, intent(in) :: nn, mm
520    ! skiplines - optional - number of lines to
521    !   skip from header
522    integer, optional :: skiplines_in
523    integer skiplines
524
525    ! OUTPUT:
526    double precision, dimension(nn,mm) :: read_array
527
528    ! BODY:
529
530    ! Row counter
531    integer ii
532    ! File unit number
533    integer, parameter :: un = 10
534    ! Final format to use
535    character(len=256) finfmt
536
537    ! Generate final format string
538    write(finfmt,'(A,I1,A,A)') '(', mm, fmtstr,
539    ')'
540
541    ! Print message
542    !write(*,*) 'Reading data from "', trim(
543      filename), '",'
544    !write(*,*) 'using format "', trim(finfmt),
545      '",'

```

```

539      ! Open file
540      open(unit=un, file=trim(filename), status='
541          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
548              ! line
549              read(un,*)
550          end do
551      else
552          skiplines = 0
553      end if
554
555      ! Loop through lines
556      do ii = 1, nn
557          ! Read one row at a time
558          read(unit=un, fmt=trim(finfmt))
559          read_array(ii,:)
560      end do
561
562      ! Close file
563      close(unit=un)
564
565  end function
566
567  ! Print 2D array to stdout
568  subroutine print_int_array(arr,nn,mm,fmtstr_in)
569      implicit none
570
571      ! INPUTS:
572      ! arr - array to print
573      integer, dimension (nn,mm), intent(in) :: arr
574      ! nn - number of data rows in file
575      ! nn - number of data columns in file
576      integer, intent(in) :: nn, mm
577      ! fmtstr - output format (no parentheses, don't
578      ! specify columns)
579      ! e.g. 'E10.2', not '(2E10.2)'
580      character(len=*), optional :: fmtstr_in
581      character(len=256) fmtstr
582
583      ! NO OUTPUTS
584
585      ! BODY
586      ! Row counter
587      integer ii
588      ! 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(
598   fmtstr), ')'
599
600 ! Loop through rows
601 do ii = 1, nn
602   ! Print one row at a time
603   write(*,finfmt) arr(ii,:)
604 end do
605
606 ! Print blank line after
607 write(*,*) ''
608
609 end subroutine print_int_array
610
611 subroutine print_array(arr,nn,mm,fmtstr_in)
612   implicit none
613
614   ! INPUTS:
615   ! arr - array to print
616   double precision, dimension (nn,mm), intent(
617     in) :: arr
618   ! nn - number of data rows in file
619   ! nn - number of data columns in file
620   integer, intent(in) :: nn, mm
621   ! fmtstr - output format (no parentheses,
622   !           don't specify columns)
623   ! e.g. 'E10.2', not '(2E10.2)'
624   character(len=*), optional :: fmtstr_in
625   character(len=256) fmtstr
626
627   ! NO OUTPUTS
628
629   ! BODY
630
631   ! Row counter
632   integer ii
633   ! Final format to use
634   character(len=256) finfmt
635
636   ! Determine string format
637   if(present(fmtstr_in)) then
638     fmtstr = fmtstr_in

```

```

636      else
637          fmtstr = 'ES10.2'
638      end if
639
640      ! Generate final format string
641      write(finfmt,'(A,I4,A,A)') '(', mm, trim(
642          fmtstr), ')'
643
644      ! Loop through rows
645      do ii = 1, nn
646          ! Include row number
647          !write(*,'(I10)', advance='no') ii
648          ! Print one row at a time
649          write(*,finfmt) arr(ii,:)
650      end do
651
652      ! Print blank line after
653      write(*,*) ''
654
655  end subroutine
656
657  ! Write 1D array to file
658  subroutine write_vec(arr,nn,filename,fmtstr_in)
659      implicit none
660
661      ! INPUTS:
662      ! arr - array to print
663      double precision, dimension (nn), intent(in)
664          :: arr
665          ! nn - number of data rows in file
666          ! nn - number of data columns in file
667          integer, intent(in) :: nn
668          ! filename - file to write to
669          character(len=*) filename
670          ! fmtstr - output format (no parentheses,
671          ! don't specify columns)
672          ! e.g. 'E10.2', not '(2E10.2)'
673          character(len=*), optional :: fmtstr_in
674          character(len=256) fmtstr
675
676      ! NO OUTPUTS
677
678      ! BODY
679
680      ! Row counter
681      integer ii
682      ! Final format to use
683      character(len=256) finfmt
684      ! Dummy file unit to use
685      integer, parameter :: un = 20
686
687      ! Open file for writing

```

```

685 |     open(unit=un, file=trim(filename), status='
686 |         replace', form='formatted')
687 |
688 |     ! Determine string format
689 |     if(present(fmtstr_in)) then
690 |         fmtstr = fmtstr_in
691 |     else
692 |         fmtstr = 'E10.2'
693 |     end if
694 |
695 |     ! Generate final format string
696 |     write(finfmt,'(A,A,A)') '(', trim(fmtstr), '
697 |     )
698 |
699 |     ! Loop through rows
700 |     do ii = 1, nn
701 |         ! Print entry per row
702 |         write(un,finfmt) arr(ii)
703 |     end do
704 |
705 |     ! Close file
706 |     close(unit=un)
707 |
708 | ! Write 2D array to file
709 subroutine write_array(arr,nn,mm,filename,
710   fmtstr_in)
711   implicit none
712
713   ! INPUTS:
714   ! arr - array to print
715   double precision, dimension (nn,mm), intent(
716     in) :: arr
717   ! nn - number of data rows in file
718   ! nn - number of data columns in file
719   integer, intent(in) :: nn, mm
720   ! filename - file to write to
721   character(len=*) filename
722   ! fmtstr - output format (no parentheses,
723   ! don't specify columns)
724   ! e.g. 'E10.2', not '(2E10.2)'
725   character(len=*), optional :: fmtstr_in
726   character(len=256) fmtstr
727
728   ! NO OUTPUTS
729
730   ! BODY
731   ! Row counter
732   integer ii
733   ! 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='
738     replace', form='formatted')
739
740   ! Determine string format
741   if(present(fmtstr_in)) then
742     fmtstr = fmtstr_in
743   else
744     fmtstr = 'E10.2'
745   end if
746
747   ! Generate final format string
748   write(finfmt, '(A,I4,A,A)') '(', mm, trim(
749     fmtstr), ')'
750
751   ! Loop through rows
752   do ii = 1, nn
753     ! Print one row at a time
754     write(un,finfmt) arr(ii,:)
755   end do
756
757   ! Close file
758   close(unit=un)
759
760 end subroutine
761
762 subroutine zeros(x, n)
763   implicit none
764   integer n, i
765   double precision, dimension(n) :: x
766
767   do i=1, n
768     x(i) = 0
769   end do
770 end subroutine zeros
771
772 end module

```

sag.f90

```

1 module sag
2 use utils
3 use fastgl
4
5 implicit none
6
7 ! Spatial grids do not include upper endpoints.
8 ! Angular grids do include upper endpoints.

```

```

9 ! Both include lower endpoints.
10
11 ! To use:
12 ! call grid%set_bounds(...)
13 ! call grid%set_num(...) (or set_uniform_spacing
14 ! )
15 ! call grid%init()
16 ! ...
17 ! call grid%deinit()
18
19 !integer, parameter :: pi = 3.141592653589793D
20 !+00
21
22 type index_list
23     integer i, j, k, p
24 contains
25     procedure :: init => index_list_init
26     procedure :: print => index_list_print
27 end type index_list
28
29 type angle2d
30     integer ntheta, nphi, nomega
31     double precision dtheta, dphi
32     double precision, dimension(:), allocatable
33         :: theta, phi, theta_edge, phi_edge
34     double precision, dimension(:), allocatable
35         :: theta_p, phi_p, theta_edge_p,
36             phi_edge_p
37     double precision, dimension(:), allocatable
38         :: cos_theta, sin_theta, cos_phi, sin_phi
39     double precision, dimension(:), allocatable
40         :: cos_theta_edge, sin_theta_edge,
41             cos_phi_edge, sin_phi_edge
42     double precision, dimension(:), allocatable
43         :: cos_theta_p, sin_theta_p, cos_phi_p,
44             sin_phi_p
45     double precision, dimension(:), allocatable
46         :: cos_theta_edge_p, sin_theta_edge_p,
47             cos_phi_edge_p, sin_phi_edge_p
48     double precision, dimension(:), allocatable
49         :: area_p
50 contains
51     procedure :: set_num => angle_set_num
52     procedure :: phat, lhat, mhat
53     procedure :: init => angle_init ! Call after
54         set_num
55     procedure :: integrate_points =>
56         angle_integrate_points
57     procedure :: integrate_func =>
58         angle_integrate_func
59     procedure :: deinit => angle_deinit
60 end type angle2d

```

```

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

```

```

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

```

```

135      m = ceiling(dble(p-1)/dble(angles%ntheta)
136          ) + 1
137    end if
138  end function mhat
139
140  function phat(angles, l, m) result(p)
141    class(angle2d) :: angles
142    integer l, m, p
143
144    if(m .eq. 1) then
145      p = 1
146    else if(m .eq. angles%nphi) then
147      p = angles%nomega
148    else
149      p = (m-2)*angles%ntheta + l + 1
150    end if
151  end function phat
152
153  subroutine angle_init(angles)
154    class(angle2d) :: angles
155    integer l, m, p
156    double precision area
157
158    ! TODO: CONSIDER REMOVING non-p
159    allocate(angles%theta(angles%ntheta))
160    allocate(angles%phi(angles%nphi))
161    allocate(angles%theta_edge(angles%ntheta))
162    allocate(angles%phi_edge(angles%nphi-1))
163    allocate(angles%theta_p(angles%nomega))
164    allocate(angles%phi_p(angles%nomega))
165    allocate(angles%theta_edge_p(angles%nomega))
166    allocate(angles%phi_edge_p(angles%nomega))
167    allocate(angles%cos_theta_p(angles%nomega))
168    allocate(angles%sin_theta_p(angles%nomega))
169    allocate(angles%cos_phi_p(angles%nomega))
170    allocate(angles%sin_phi_p(angles%nomega))
171    allocate(angles%cos_theta(angles%nomega))
172    allocate(angles%sin_theta(angles%nomega))
173    allocate(angles%cos_phi(angles%nomega))
174    allocate(angles%sin_phi(angles%nomega))
175    allocate(angles%cos_theta_edge(angles%ntheta
176        ))
177    allocate(angles%sin_theta_edge(angles%ntheta
178        ))
179    allocate(angles%cos_phi_edge(angles%nphi-1))
180    allocate(angles%sin_phi_edge(angles%nphi-1))
181    allocate(angles%cos_theta_edge_p(angles%
182        nomega))

```

```

180 |     allocate(angles%sin_theta_edge_p(angles%
181 |             nomega))
182 |     allocate(angles%cos_phi_edge_p(angles%nomega
183 |             -1))
184 |     allocate(angles%sin_phi_edge_p(angles%nomega
185 |             -1))
186 |     allocate(angles%area_p(angles%nomega))
187 |
188 | ! Calculate spacing
189 | angles%dtheta = 2.d0*pi/dble(angles%ntheta)
190 | angles%dphi = pi/dble(angles%nphi-1)
191 |
192 | ! Create grids
193 | do l=1, angles%ntheta
194 |     angles%theta(l) = dble(l-1)*angles%dtheta
195 |     angles%cos_theta(l) = cos(angles%theta(l))
196 |     angles%sin_theta(l) = sin(angles%theta(l))
197 |     angles%theta_edge(l) = dble(l-0.5d0)*
198 |         angles%dtheta
199 |     angles%cos_theta_edge(l) = cos(angles%
200 |             theta_edge(l))
201 |     angles%sin_theta_edge(l) = sin(angles%
202 |             theta_edge(l))
203 | end do
204 |
205 | do m=1, angles%nphi
206 |     angles%phi(m) = dble(m-1.d0)*angles%dphi
207 |     angles%cos_phi(m) = cos(angles%phi(m))
208 |     angles%sin_phi(m) = sin(angles%phi(m))
209 |     if(m<angles%nphi) then
210 |         angles%phi_edge(m) = dble(m-0.5d0)*
211 |             angles%dphi
212 |         angles%cos_phi_edge(m) = cos(angles%
213 |             phi_edge(m))
214 |         angles%sin_phi_edge(m) = sin(angles%
215 |             phi_edge(m))
216 |     end if
217 | end do
218 |
219 | ! Create p arrays
220 | do m=2, angles%nphi-1
221 |     area = angles%dtheta *
222 |             (angles%cos_phi_edge(m-1) - angles%
223 |                 %cos_phi_edge(m))
224 |     do l=1, angles%ntheta
225 |         p = angles%phat(l, m)
226 |
227 |         angles%theta_p(p) = angles%theta(l)

```

```

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

```

```

253 |     angles%sin_theta_edge_p(p) = sin(angles%
254 |         theta_edge_p(p))
255 |     angles%cos_phi_edge_p(p) = cos(angles%
256 |         phi_edge_p(p))
257 |     angles%sin_phi_edge_p(p) = sin(angles%
258 |         phi_edge_p(p))
259 |     angles%area_p(p) = area
260 |
261 | ! South Pole
262 | p = angles%nomega
263 | m = angles%nphi
264 | angles%theta_p(p) = angles%theta(l)
265 | angles%theta_edge_p(p) = angles%theta_edge(l)
266 | angles%phi_p(p) = angles%phi(m)
267 | angles%cos_theta_p(p) = cos(angles%theta_p(p))
268 | angles%sin_theta_p(p) = sin(angles%theta_p(p))
269 | angles%cos_phi_p(p) = cos(angles%phi_p(p))
270 | angles%sin_phi_p(p) = sin(angles%phi_p(p))
271 | angles%area_p(p) = area
272 | end subroutine angle_init
273 |
274 | ! Integrate function given function values at
275 | ! grid cells
276 | function angle_integrate_points(angles,
277 |     func_vals) result(integral)
278 |     class(angle2d) :: angles
279 |     double precision, dimension(angles%nomega)
280 |         :: func_vals
281 |     double precision integral
282 |     integer p
283 |
284 |     integral = 0.d0
285 |
286 |     do p=1, angles%nomega
287 |         integral = integral + angles%area_p(p) *
288 |             func_vals(p)
289 |     end do
290 |
291 | end function angle_integrate_points
292 |
293 | function angle_integrate_func(angles,
294 |     func_callable) result(integral)
295 |     class(angle2d) :: angles
296 |     double precision, external :: func_callable
297 |     double precision, dimension(:), allocatable
298 |         :: func_vals
299 |     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
305   deallocate(func_vals)
306 end function angle_integrate_func
307
308 subroutine angle_deinit(angles)
309   class(angle2d) :: angles
310   deallocate(angles%theta)
311   deallocate(angles%phi)
312   deallocate(angles%theta_edge)
313   deallocate(angles%phi_edge)
314   deallocate(angles%theta_p)
315   deallocate(angles%phi_p)
316   deallocate(angles%theta_edge_p)
317   deallocate(angles%phi_edge_p)
318   deallocate(angles%cos_theta)
319   deallocate(angles%sin_theta)
320   deallocate(angles%cos_phi)
321   deallocate(angles%sin_phi)
322   deallocate(angles%cos_theta_p)
323   deallocate(angles%sin_theta_p)
324   deallocate(angles%cos_phi_p)
325   deallocate(angles%sin_phi_p)
326   deallocate(angles%cos_theta_edge)
327   deallocate(angles%sin_theta_edge)
328   deallocate(angles%cos_phi_edge)
329   deallocate(angles%sin_phi_edge)
330   deallocate(angles%cos_theta_edge_p)
331   deallocate(angles%sin_theta_edge_p)
332   deallocate(angles%cos_phi_edge_p)
333   deallocate(angles%sin_phi_edge_p)
334   deallocate(angles%area_p)
335
336
337   !!! ANGLE 1D !!!
338

```

```

339 subroutine angle_set_bounds(angle, minval,
340     maxval)
341     class(angle_dim) :: angle
342     double precision minval, maxval
343     angle%minval = minval
344     angle%maxval = maxval
345 end subroutine angle_set_bounds
346
347 subroutine angle1d_set_num(angle, num)
348     class(angle_dim) :: angle
349     integer num
350     angle%num = num
351 end subroutine angle1d_set_num
352
353 subroutine angle1d_assign_linspace(angle)
354     class(angle_dim) :: angle
355     double precision spacing
356     integer i
357
358     spacing = (angle%maxval - angle%minval) /
359         dble(angle%num)
360     do i=1, angle%num
361         angle%vals(i) = (i-1) * spacing
362     end do
363 end subroutine angle1d_assign_linspace
364
365 ! To calculate  $\int_{x_{\min}}^{x_{\max}} f(x) dx$  :
366 ! int = prefactor * sum(weights * f(roots))
367 subroutine assign_legendre(angle)
368     class(angle_dim) :: angle
369     double precision root, weight, theta
370     integer i
371     ! glpair produces both x and theta, where x=
372         cos(theta). We'll throw out theta.
373
374     allocate(angle%vals(angle%num))
375     allocate(angle%weights(angle%num))
376     allocate(angle%sin(angle%num))
377     allocate(angle%cos(angle%num))
378
379     ! Prefactor for integration
380     ! From change of variables
381     angle%prefactor = (angle%maxval - angle%
382         minval) / 2.d0
383
384     do i = 1, angle%num
385         call glpair(angle%num, i, theta, weight,
386             root)
387         call affine_transform(root, -1.d0, 1.d0,
388             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
392 ! Gauss-Legendre quadrature
393
394 function angle1d_integrate_func(angle,
395   func_callable) result(integral)
396   class(angle_dim) :: angle
397   double precision, external :: func_callable
398   double precision, dimension(:), allocatable
399   :: func_vals
400   double precision integral
401   integer i
402
403   allocate(func_vals(angle%num))
404
405   do i=1, angle%num
406     func_vals(i) = func_callable(angle%vals(i))
407   end do
408
409   integral = angle%integrate_points(func_vals)
410
411   deallocate(func_vals)
412 end function angle1d_integrate_func
413
414 ! Integrate function given function values
415 ! sampled at legendre theta values
416
417 function angle1d_integrate_points(angle,
418   func_vals) result(integral)
419   class(angle_dim) :: angle
420   double precision, dimension(angle%num) ::
421   func_vals
422   double precision integral
423
424   integral = angle%prefactor * sum(angle%
425   weights * func_vals)
426 end function angle1d_integrate_points
427
428 subroutine angle1d_deinit(angle)
429   class(angle_dim) :: angle
430   deallocate(angle%vals)
431   deallocate(angle%weights)
432   deallocate(angle%sin)
433   deallocate(angle%cos)

```

```

426 | end subroutine angle1d_deinit
427 |
428 |
429 | !! SPACE !!
430 |
431 | ! Integrate function given function values
432 |     sampled at even grid points
433 | function space_integrate_points(space,
434 |     func_vals) result(integral)
435 |     class(space_dim) :: space
436 |     double precision, dimension(space%num) :: :
437 |         func_vals
438 |     double precision integral
439 |
440 |     ! Encapsulate actual method for easy
441 |         switching
442 |     integral = space%trapezoid_rule(func_vals)
443 |
444 | end function space_integrate_points
445 |
446 | function trapezoid_rule(space, func_vals)
447 |     result(integral)
448 |     class(space_dim) :: space
449 |     double precision, dimension(space%num) :: :
450 |         func_vals
451 |     double precision integral
452 |
453 |     integral = 0.5d0 * sum(func_vals * space%
454 |         spacing)
455 | end function
456 |
457 | subroutine space_set_bounds(space, minval,
458 |     maxval)
459 |     class(space_dim) :: space
460 |     double precision minval, maxval
461 |     space%minval = minval
462 |     space%maxval = maxval
463 | end subroutine space_set_bounds
464 |
465 | subroutine space_set_num(space, num)
466 |     class(space_dim) :: space
467 |     integer num
468 |     space%num = num
469 | end subroutine space_set_num
470 |
471 | subroutine space_set_uniform_spacing(space,
472 |     spacing)
473 |     class(space_dim) :: space
474 |     double precision spacing
475 |     integer k
476 |     do k=1, space%num

```

```

468     space%spacing(k) = spacing
469   end do
470 end subroutine space_set_uniform_spacing
471
472 subroutine space_set_spacing_array(space,
473   spacing)
474   class(space_dim) :: space
475   double precision, dimension(space%num) :: spacing
476   space%spacing = spacing
477 end subroutine space_set_spacing_array
478
479 subroutine assign_linspace(space)
480   class(space_dim) :: space
481   double precision spacing
482   integer i
483
484   allocate(space%vals(space%num))
485   allocate(space%edges(space%num))
486   allocate(space%spacing(space%num))
487
488   spacing = spacing_from_num(space%minval,
489     space%maxval, space%num)
490   call space%set_uniform_spacing(spacing)
491
492   do i=1, space%num
493     space%edges(i) = space%minval + dble(i-1)
494       * space%spacing(i)
495     space%vals(i) = space%minval + dble(i-0.5
496       d0) * space%spacing(i)
497   end do
498
499 end subroutine assign_linspace
500
501 subroutine set_uniform_spacing_from_num(space)
502   ! Create evenly spaced grid (linspace)
503   class(space_dim) :: space
504   double precision spacing
505
506   spacing = spacing_from_num(space%minval,
507     space%maxval, space%num)
508   call space%set_uniform_spacing(spacing)
509
510 end subroutine set_uniform_spacing_from_num
511
512 ! subroutine set_num_from_spacing(space)
513 !   class(space_dim) :: space
514 !   !space%num = num_from_spacing(space%minval
515 !     , space%maxval, space%spacing)
516 !
517 ! end subroutine set_num_from_spacing

```

```

512
513 subroutine space_deinit(space)
514   class(space_dim) :: space
515   deallocate(space%vals)
516   deallocate(space%edges)
517   deallocate(space%spacing)
518 end subroutine space_deinit
519
520 !! SAG !!
521
522 subroutine sag_set_bounds(grid, xmin, xmax,
523   ymin, ymax, zmin, zmax)
524   class(space_angle_grid) :: grid
525   double precision xmin, xmax, ymin, ymax,
526     zmin, zmax
527
528   call grid%x%set_bounds(xmin, xmax)
529   call grid%y%set_bounds(ymin, ymax)
530   call grid%z%set_bounds(zmin, zmax)
531 end subroutine sag_set_bounds
532
533 subroutine sag_set_uniform_spacing(grid, dx,
534   dy, dz)
535   class(space_angle_grid) :: grid
536   double precision dx, dy, dz
537
538   call grid%x%set_uniform_spacing(dx)
539   call grid%y%set_uniform_spacing(dy)
540   call grid%z%set_uniform_spacing(dz)
541
542 end subroutine sag_set_uniform_spacing
543
544 subroutine sag_set_num(grid, nx, ny, nz,
545   ntheta, nphi)
546   class(space_angle_grid) :: grid
547   integer nx, ny, nz, ntheta, nphi
548
549   call grid%x%set_num(nx)
550   call grid%y%set_num(ny)
551   call grid%z%set_num(nz)
552
553   call grid%angles%set_num(ntheta, nphi)
554
555 end subroutine sag_set_num
556
557 subroutine sag_init(grid)
558   class(space_angle_grid) :: grid
559
560   call grid%x%assign_linspace()
561   call grid%y%assign_linspace()
562   call grid%z%assign_linspace()
563
564   call grid%angles%init()
565   call grid%calculate_factors()

```

```

558 | end subroutine sag_init
559 |
560 | subroutine sag_calculate_factors(grid)
561 |   ! Factors by which depth difference is
562 |   ! multiplied
563 |   ! in order to calculate distance traveled in
564 |   ! the
565 |   ! (x, y) direction along a ray in the (theta
566 |   , phi)
567 |   ! direction
568 |   class(space_angle_grid) :: grid
569 |   integer p, nomega
570 |   double precision theta, phi
571 |
572 |   nomega = grid%angles%nomega
573 |
574 |   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(
584 |   grid)
585 |   class(space_angle_grid) :: grid
586 |   call grid%x%set_uniform_spacing_from_num()
587 |   call grid%y%set_uniform_spacing_from_num()
588 |   call grid%z%set_uniform_spacing_from_num()
589 | end subroutine
590 |   sag_set_uniform_spacing_from_num
591 |
592 | ! subroutine sag_set_num_from_spacing(grid)
593 | !   class(space_angle_grid) :: grid
594 | !   call grid%x%set_num_from_spacing()
595 | !   call grid%y%set_num_from_spacing()
596 | !   call grid%z%set_num_from_spacing()
597 |
598 | ! end subroutine sag_set_num_from_spacing
599 |
600 | subroutine sag_deinit(grid)
601 |   class(space_angle_grid) :: grid
602 |   call grid%x%deinit()
603 |   call grid%y%deinit()
604 |   call grid%z%deinit()
605 |   call grid%angles%deinit()

```

```

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

```

kelp3d.f90

```

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

```

```

16   double precision xmin, xmax, ymin, ymax, zmin,
17       zmax
18   integer nx, ny, nz, ntheta, nphi
19   type(space_angle_grid) grid
20   double precision, dimension(:,:,:),
21       allocatable :: p_kelp
22
23   call grid%set_bounds(xmin, xmax, ymin, ymax,
24       zmin, zmax)
25   call grid%set_num(nx, ny, nz, ntheta, nphi)
26
27   allocate(p_kelp(nx,ny,nz))
28
29 end subroutine generate_grid
30
31 subroutine kelp3d_deinit(grid, rope, p_kelp)
32   type(space_angle_grid) grid
33   type(rope_state) rope
34   double precision, dimension(:,:,:),
35       allocatable :: p_kelp
36   call rope%deinit()
37   call grid%deinit()
38   deallocate(p_kelp)
39 end subroutine kelp3d_deinit
40
41 subroutine calculate_kelp_on_grid(grid, p_kelp,
42   frond, rope, quadrature_degree, n_images,
43   num_threads)
44   type(space_angle_grid), intent(in) :: grid
45   type(frond_shape), intent(in) :: frond
46   type(rope_state), intent(in) :: rope
47   type(point3d) point
48   integer, intent(in) :: quadrature_degree
49   integer, optional :: n_images
50   double precision, dimension(grid%x%num, grid%y
51       %num, grid%z%num) :: p_kelp
52   type(depth_state) depth
53   integer num_threads
54
55   integer i, j, k, nx, ny, nz
56   double precision x, y, z
57   ! Number of periodic images
58   ! to consider in each horizontal direction
59   ! for kelp distribution
60   ! n_images=1 => 3x3 meta-grid
61   ! n_images=2 => 5x5 meta-grid (only necessary
62       for very dense kelp ropes)
63   integer im_i, im_j
64   double precision x_width, y_width
65
66   x_width = grid%x%maxval - grid%x%minval

```

```

59     y_width = grid%y%maxval - grid%y%minval
60
61     if(.not. present(n_images)) then
62         n_images = 1
63     end if
64
65     nx = grid%x%num
66     ny = grid%y%num
67     nz = grid%z%num
68
69     p_kelp(:,:,:,:) = 0
70
71 !$omp parallel do default(shared) private(x,y,
72     z) &
73 !$omp firstprivate(point,depth) &
74 !$omp private(i,j,k,im_i,im_j) shared(nx,ny,nz
75     ,n_images) &
76 !$omp shared(frond,rope,grid,quadrature_degree
77     ) &
78 !$omp shared(p_kelp,x_width,y_width) &
79 !$omp num_threads(num_threads) collapse(3) &
80 !$omp schedule(dynamic, 10) ! 10 grid points
81     per thread
82 do k=1, nz
83     do i=1, nx
84         do j=1, ny
85             z = grid%z%vals(k)
86             call depth%set_depth(rope, grid, k)
87             do im_i=-n_images, n_images
88                 x = im_i*x_width + grid%x%vals(i)
89                 do im_j=-n_images, n_images
90                     y = im_j*y_width + grid%y%vals(j)
91                     call point%set_cart(x, y, z)
92                     p_kelp(i, j, k) = p_kelp(i,j,k) +
93                         kelp_proportion(point, frond
94                             , grid, depth,
95                             quadrature_degree)
96             end do
97         end do
98     end do
99 end do
100 !$omp end do
101 end subroutine calculate_kelp_on_grid
102
103 subroutine shading_region_limits(theta_low_lim,
104     theta_high_lim, point, frond)
105     type(point3d), intent(in) :: point
106     type(frond_shape), intent(in) :: frond
107     double precision, intent(out) :: theta_low_lim
108     , 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,
108   quadrature_degree)
109 ! P_s(theta_p, r_p) - This is the proportion of
110   the population of this depth layer which can
111   be found in this Cartesian grid cell.
112 type(point3d), intent(in) :: point
113 type(frond_shape), intent(in) :: frond
114 type(depth_state), intent(in) :: depth
115 integer, intent(in) :: quadrature_degree
116 double precision prob_kelp
117 double precision theta_low_lim, theta_high_lim
118
119 call shading_region_limits(theta_low_lim,
120   theta_high_lim, point, frond)
121 prob_kelp = integrate_ps(theta_low_lim,
122   theta_high_lim, quadrature_degree, point,
123   frond, depth)
124 end function prob_kelp
125
126 function kelp_proportion(point, frond, grid,
127   depth, quadrature_degree)
128 ! This is the proportion of the volume of the
129   Cartesian grid cell occupied by kelp
130 type(point3d), intent(in) :: point
131 type(frond_shape), intent(in) :: frond
132 type(depth_state), intent(in) :: depth
133 type(space_angle_grid), intent(in) :: grid
134 integer, intent(in) :: quadrature_degree
135 double precision p_k, n, t, dz
136 double precision kelp_proportion
137
138 n = depth%num_fronds
139 dz = grid%z%spacing(depth%depth_layer)
140 t = frond%ft
141 !write(*,*) 'KELP PROPORTION'
142 !write(*,*) 'n=', n
143 !write(*,*) 'dz=', dz
144 !write(*,*) 't=', t
145 !write(*,*) 'coef=', n*t/dz
146 p_k = prob_kelp(point, frond, depth,
147   quadrature_degree)
148 kelp_proportion = n*t/dz * p_k
149 end function kelp_proportion

```

```

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

```

```

181   length_part = 1 - depth%
182   length_distribution_cdf(l_min)
183   ps_integrand = angular_part * length_part
184 end function ps_integrand
185
186 function min_shading_length(theta_f, point,
187   frond) result(l_min)
188 ! L_min(\theta)
189 type(point3d), intent(in) :: point
190 type(frond_shape), intent(in) :: frond
191 double precision, intent(in) :: theta_f
192 double precision l_min
193 double precision tpp
194 double precision frond_frac
195 ! tpp === theta_p_prime
196 tpp = point%theta - theta_f + pi / 2.d0
197 frond_frac = 2.d0 * frond%fr / (1.d0 + frond%
198   fs)
199 l_min = point%r * (sin(tpp) + angular_sign(tpp)
200   ) * frond_frac * cos(tpp))
201 end function min_shading_length
202
203 function frond_edge(theta, theta_f, L, fs, fr)
204 ! r_f(\theta)
205 ! double precision, intent(in) :: theta,
206 !   theta_f, L, fs, fr
207 ! double precision, intent(out) :: frond_edge
208 !
209 ! frond_edge = relative_frond_edge(theta -
210 !   theta_f + pi/2.d0)
211 !
212 end function frond_edge
213
214 function relative_frond_edge(theta_prime, L,
215   fs, fr)
216 ! r_f'(\theta')
217 ! double precision, intent(in) :: theta_prime,
218 !   L, fs, fr
219 ! double precision, intent(out) :::
220 !   relative_frond_edge
221 !
222 ! relative_frond_edge = L / (sin(theta_prime)
223 !   + angular_sign(theta_prime * alpha(fs, fr) *
224 !   cos(theta_prime)))
225 ! end function relative_frond_edge
226
227 function angular_sign(theta_prime)
228 ! 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
224  ! of 9/9/18
225  ! In the report, it's written as sgn(
226  !   theta_prime - pi/2.d0)
227  ! This results in L_min < 0 - not good!
228  angular_sign = sgn(pi/2.d0 - theta_prime)
229 end function angular_sign
230
231 subroutine gaussian_blur_2d(A, sigma, dx, dy, nk
232   , num_threads)
233  ! 2D Gaussian blur (periodic BC) with std
234  !   sigma
235  ! with kernel radius of nk (full size (2*nk+1)
236  !   x(2*nk+1))
237  ! applied to matrix A with element spacings dx
238  !   and dy.
239  double precision, intent(inout), dimension(:, :
240   ) :: A
241  double precision, intent(in) :: sigma, dx, dy
242  ! kernel half width
243  integer, intent(in) :: nk
244  ! kernel full width
245  integer kw
246  integer num_threads
247
248  ! A matrix size
249  integer nx, ny
250
251  ! indices
252  integer i1, j1
253  integer i2, j2
254  integer i, j
255  ! kernel
256  double precision, dimension(:, :, :), allocatable
257   :: k
258  ! output matrix
259  double precision, dimension(:, :, :), allocatable
260   :: B
261  ! kernel independent variables
262  double precision x, y
263
264  if(sigma > 0) then
265    nx = size(A, 1)
266    ny = size(A, 2)
267
268    kw = 2*nk + 1
269
270    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
268     do j1=-nk, nk
269       y = j1*dy
270       j = j1+nk+1
271       k(i,j) = exp(-(x**2+y**2)/(2*sigma**2))
272     end do
273   end do
274   ! normalize kernel
275   k = k / sum(k)
277
278   !write(*,*) 'convolving'
279   ! convolve
280   !$omp parallel do default(private) private(x
281   ,y) &
282   !$omp private(i,j,i1,j1,i2,j2) shared(nx,ny,
283   nk,kw) &
284   !$omp shared(A,B,k) &
285   !$omp num_threads(num_threads) collapse(2) &
286   !$omp schedule(dynamic, 10) ! 10 grid points
287   ! per thread
288   do i1=1, nx
289     do j1=1, ny
290       B(i1, j1) = 0
291       do i2=1, kw
292         do j2=1, kw
293           i = mod1(i1 - nk + i2 - 1, nx)
294           j = mod1(j1 - nk + j2 - 1, ny)
295           B(i1, j1) = B(i1, j1) + k(i2, j2)
296             * A(i, j)
297         end do
298       end do
299     end do
300   !$omp end parallel do
301   !write(*,*) 'done convolving'
302
303   ! Update original matrix
304   A(:, :) = B(:, :)
305   deallocate(k)
306   deallocate(B)
307   !write(*,*) 'gb2d done.'
308 end if
309 end subroutine gaussian_blur_2d

```

```
308 |end module kelp3d
```

```
rte_sparse_matrices.f90
1 module rte_sparse_matrices
2 use sag
3 use kelp_context
4 use mgmres
5 use type_consts
6 !use hdf5_utils
7 ! Use 64-bit integers for LIS
8 ! Necessary for FD solution w/ large matrices
9 #define LONG_LONG
10 #include "lispf.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,
26             z_block_size, omega_block_size
27     double precision, dimension(:), allocatable
28             :: surface_vals
29
30     ! CSR format
31     ! http://www.scipy-lectures.org/advanced/
32     !         scipy_sparse/csr_matrix.html
33     ! with LIS method 2 (LIS manual, p.19)
34     integer(index_kind), dimension(:),
35             allocatable :: ptr, col
36     double precision, dimension(:), allocatable
37             :: data
38
39     ! Lis Matrix and vectors
40     LIS_MATRIX A
41     LIS_VECTOR b, x
42     LIS_SOLVER solver
43     LIS_INTEGER ierr
44     character(len=1024) solver_opts
45     logical initx_zeros
46
47     ! Pointer to solver subroutine
```

```

44 ! Set to mgmres by default
45 !procedure(solver_interface), pointer, nopass
46     :: solver => mgmres_st
47 contains
48     procedure :: init => mat_init
49     procedure :: deinit => mat_deinit
50     procedure :: calculate_size
51     procedure :: set_solver_opts =>
52         mat_set_solver_opts
53     procedure :: set_row => mat_set_row
54     procedure :: assign => mat_assign
55     procedure :: add => mat_add
56     procedure :: assign_rhs => mat_assign_rhs
57     procedure :: add_rhs => mat_add_rhs
58     !procedure :: store_index => mat_store_index
59     !procedure :: find_index => mat_find_index
60     procedure :: set_bc => mat_set_bc
61     procedure :: solve => mat_solve
62     procedure :: get_solver_stats
63     procedure :: ind => mat_ind
64     !procedure :: to_hdf => mat_to_hdf
65     procedure :: attenuate
66     procedure :: angular_integral
67     procedure :: add_source
68
69     ! Derivative subroutines
70     procedure x_cd2
71     procedure x_cd2_first
72     procedure x_cd2_last
73     procedure y_cd2
74     procedure y_cd2_first
75     procedure y_cd2_last
76     procedure z_cd2
77     procedure z_fd2
78     procedure z_bd2
79     procedure z_surface_bc
80     procedure z_bottom_bc
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,
87         row, col, data, &
88         sol, rhs, maxiter_outer, maxiter_inner,
89         &
90         tol_abs, tol_rel)
91     use type_consts

```

```

90   integer(index_kind) :: n_total, nonzero
91   integer, dimension(nonzero) :: row, col
92   double precision, dimension(nonzero) :::
93   data
94   double precision, dimension(nonzero) :: sol
95   double precision, dimension(n_total) :: rhs
96   integer :: maxiter_outer, maxiter_inner
97   double precision :: tol_abs, tol_rel
98 end subroutine solver_interface
99 end interface
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,
129                         mat% ierr)
129   call lis_vector_create(comm_world, mat%b,
130                         mat% ierr)
130   call lis_vector_create(comm_world, mat%x,
131                         mat% ierr)
132
132   call lis_matrix_set_size(mat%A, n_total,
133                           n_total, mat% ierr)
133   call lis_vector_set_size(mat%b, n_total,
134                           n_total, mat% ierr)

```

```

134 |     call lis_vector_set_size(mat%x, n_total,
135 |                               n_total, mat% ierr)
136 |     call lis_vector_set_all(0.0d0, mat%x, mat%
137 |                               ierr)
138 |     call lis_vector_set_all(0.0d0, mat%b, mat%
139 |                               ierr)
140 |     if(mat% ierr .ne. 0) then
141 |         write(*,*) 'INIT ERR: ', mat% ierr
142 |         call exit(1)
143 |     end if
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 |                           )
163 |   call lis_finalize(mat% ierr)
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 * ( (
185         nz-1) * (6 + ntheta * nphi) + 1)
186     mat%nonzero = nx * ny * nomega * (nz * (
187         nomega + 6) - 1)
188     mat%n_total = nx * ny * nz * nomega
189     write(*,*) 'nnz = ', mat%nonzero
190     write(*,*) 'n_total = ', mat%n_total
191
192     !mat%theta_block_size = 1
193     !mat%phi_block_size = mat%theta_block_size *
194         ntheta
195     mat%omega_block_size = 1
196     mat%y_block_size = int(mat%omega_block_size *
197         nomega)
198     mat%x_block_size = int(mat%y_block_size * ny
199         )
200     mat%z_block_size = int(mat%x_block_size * nx
201         )
202
203 end subroutine calculate_size
204
205 subroutine mat_to_hdf(mat, filename)
206 !     class(rte_mat) mat
207 !     character(len=*) filename
208 !     call write_coo(filename, mat%row, mat%col,
209 !         mat%data, mat%nonzero)
210 ! end subroutine mat_to_hdf
211
212 subroutine mat_set_bc(mat, bc)
213     class(rte_mat) mat
214     class(boundary_condition) bc
215     integer p
216
217     do p=1, mat%grid%angles%nomega/2
218         mat%surface_vals(p) = bc%bc_grid(p)
219     end do
220 end subroutine mat_set_bc
221
222 subroutine mat_solve(mat)
223     class(rte_mat) mat
224     character(len=64) init_opt
225
226     ! write(*,*) 'mat%n_total = ', mat%n_total
227     ! 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
234 , mat%col, mat%data, mat%A, mat% ierr)
235 write(*,*) 'LIS Assemble'
236 call lis_matrix_assemble(mat%A, mat% ierr)
237
238 ! Set solver options
239 if(mat%initx_zeros) then
240     init_opt = "-initx_zeros true -print out"
241 else
242     init_opt = "-initx_zeros false -print
243         out"
244 end if
245
246 write(*,*) 'LIS set solver options'
247 write(*,*) 'opt: ', trim(init_opt)
248 call lis_solver_set_option(trim(init_opt), mat%
249     solver, mat% ierr)
250 if(len(trim(mat%solver_opts)) .gt. 0) then
251     write(*,*) 'opt: ', trim(mat%solver_opts)
252     call lis_solver_set_option(trim(mat%
253         solver_opts), mat%solver, mat% ierr)
254 end if
255
256 ! Solve
257 write(*,*) 'LIS Solve'
258 call lis_solve(mat%A, mat%b, mat%x, mat%
259     solver, mat% ierr)
260 write(*,*) 'LIS Solve done'
261
262 end subroutine mat_solve
263
264 subroutine get_solver_stats(mat, lis_iter,
265     lis_time, lis_resid)
266 class(rte_mat) mat
267 integer lis_iter
268 double precision lis_time

```

```

263      double precision lis_resid
264
265      call lis_solver_get_iter(mat%solver,
266          lis_iter, mat% ierr)
266      call lis_solver_get_time(mat%solver,
267          lis_time, mat% ierr)
267      call lis_solver_get_residualnorm(mat%solver,
268          lis_resid, mat% ierr)
268  end subroutine get_solver_stats
269
270  subroutine mat_set_solver_opts(mat,
271      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%
283          %y_block_size + &
283          (k-1) * mat%z_block_size + p * mat%
283          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,
295      p)
295      ! It's assumed that this is the first time
295      ! 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
301      ! 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
310 ! already been assigned
311 ! and you'd like to add this value to the
312 ! existing value.
313
314 class(rte_mat) mat
315 double precision val
316 integer(index_kind) repeat_ent
317
318 ! Entry number where value is already stored
319 mat%data(repeat_ent) = mat%data(repeat_ent)
320     + val
321 end subroutine mat_add
322
323 subroutine mat_assign_rhs(mat, row_num, data)
324     class(rte_mat) mat
325     double precision data
326     integer(index_kind) row_num
327
328     call lis_vector_set_value(LIS_INS_VALUE,
329         row_num, data, mat%b, mat%ierr)
330     if(mat%ierr .ne. 0) then
331         write(*,*) 'RHS ERR: ', mat%ierr
332         call exit(1)
333     end if
334 end subroutine mat_assign_rhs
335
336 subroutine mat_add_rhs(mat, row_num, data)
337     class(rte_mat) mat
338     double precision data
339     integer(index_kind) row_num
340
341     call lis_vector_set_value(LIS_ADD_VALUE,
342         row_num, data, mat%b, mat%ierr)
343     if(mat%ierr .ne. 0) then
344         write(*,*) 'RHS ERR: ', mat%ierr
345         call exit(1)
346     end if
347 end subroutine mat_add_rhs
348
349 ! subroutine mat_store_index(mat, row_num,
350 !     col_num)
351 ! ! Remember where we stored information for
352 ! this matrix element
353 ! class(rte_mat) mat
354 ! integer row_num, col_num
355 ! mat%index_map(row_num, col_num) = mat%ent

```

```

349 ! end subroutine
350
351 ! function mat_find_index(mat, row_num,
352 !   col_num) result(index)
352 !   ! Find the position in row, col, data
353 !   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
360 !   index
361 !   ! do index=1, mat%ent-1
362 !   !   if( (mat%row(index) .eq. row_num) .
362 !       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
370 ! entry
371 ! And adding here is more efficient than a
371 ! 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,
379 !           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
388 ! entry
389 ! And adding here is more efficient than a
389 ! 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,
397 |                                         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.
417           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.
440          d0 * dx)
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.
461               d0 * dx)
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.
482               d0 * dy)
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.
505           d0 * dy)
506
507   call mat%assign(ent,-val,i,ny,k,p)
508   call mat%assign(ent,val,i,j+1,k,p)
509 end subroutine y_cd2_first
510
511 subroutine y_cd2_last(mat, indices, ent)
512   class(rte_mat) mat
513   double precision val, dy
514   type(index_list) indices
515   integer i, j, k, p
516   integer(index_kind) ent
517
518   i = indices%i
519   j = indices%j
520   k = indices%k
521   p = indices%p
522
523   dy = mat%grid%y%spacing(1)
524
525   val = mat%grid%angles%sin_phi_p(p) &
526       * mat%grid%angles%sin_theta_p(p) / (2.
527           d0 * dy)
528
529   call mat%assign(ent,-val,i,j-1,k,p)
530   call mat%assign(ent,val,i,1,k,p)
531 end subroutine y_cd2_last
532
533 subroutine z_cd2(mat, indices, ent)
534   class(rte_mat) mat
535   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 *
546    dz)
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   )
553 ! Has to be called after angular_integral
554 ! Because they both write to the same matrix
555 ! entry
556 ! And adding here is more efficient than a
557 ! conditional
558 ! in the angular loop.
559 class(rte_mat) mat
560 double precision val, val1, val2, val3, dz
561 type(index_list) indices
562 integer i, j, k, p
563 integer(index_kind) ent, repeat_ent
564
565 i = indices%i
566 j = indices%j
567 k = indices%k
568 p = indices%p
569
570 dz = mat%grid%z%spacing(indices%k)
571
572 val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
573   dz)
574
575 val1 = -3.d0 * val
576 val2 = 4.d0 * val
577 val3 = -val
578
579 call mat%add(repeat_ent, val1)
580 call mat%assign(ent,val2,i,j,k+1,p)
581 call mat%assign(ent,val3,i,j,k+2,p)
582 end subroutine z_fd2
583

```

```

580 subroutine z_bd2(mat, indices, ent, repeat_ent
581   )
582   ! Has to be called after angular_integral
583   ! Because they both write to the same matrix
584   ! entry
585   ! And adding here is more efficient than a
586   ! conditional
587   ! in the angular loop.
588 class(rte_mat) mat
589 double precision val, val1, val2, val3, dz
590 type(index_list) indices
591 integer i, j, k, p
592 integer(index_kind) ent, repeat_ent
593
594 i = indices%i
595 j = indices%j
596 k = indices%k
597 p = indices%p
598
599 dz = mat%grid%z%spacing(indices%k)
600
601 val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
602   dz)
603
604 val1 = 3.d0 * val
605 val2 = -4.d0 * val
606 val3 = val
607
608 call mat%add(repeat_ent, val1)
609 call mat%assign(ent, val2, i, j, k-1, p)
610 call mat%assign(ent, val3, i, j, k-2, p)
611 end subroutine z_bd2
612
613 subroutine angular_integral(mat, indices, ent)
614   class(rte_mat) mat
615   ! Primed angular integration variables
616   integer pp
617   double precision val
618   type(index_list) indices
619   integer(index_kind) ent
620
621   do pp=1, mat%grid%angles%nomega
622     val = -mat%iops%scat * mat%iops%
623       vsf_integral(indices%p, pp)
624     call mat%assign(ent, val, indices%i,
625       indices%j, indices%k, pp)
626   end do
627 end subroutine angular_integral
628
629 subroutine z_surface_bc(mat, indices, row_num,
630   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)
633   / (3.d0 * dz)
634   val2 = 3.d0 * val1
635   bc_val = 4.d0 * val1 * mat%surface_vals(
636   indices%p)
637
638   call mat%assign(ent, val1, indices%i, indices%j
639   , 2, indices%p)
640   call mat%add(repeat_ent, val2)
641   call mat%assign_rhs(row_num, bc_val)
642 end subroutine z_surface_bc
643
644 subroutine z_bottom_bc(mat, indices, ent,
645   repeat_ent)
646   class(rte_mat) mat
647   type(index_list) indices
648   double precision val1, val2, dz
649   integer nz
650   integer(index_kind) ent, repeat_ent
651
652   dz = mat%grid%z%spacing(1)
653   nz = mat%grid%z%num
654
655   val1 = -mat%grid%angles%cos_phi_p(indices%p)
656   / (3.d0 * dz)
657   val2 = 3.d0 * val1
658
659   call mat%assign(ent, val1, indices%i, indices%j
660   , nz-1, indices%p)
661   call mat%add(repeat_ent, val2)
662 end subroutine z_bottom_bc
663
664 end module rte_sparse_matrices

```

rte3d.f90

```

1 module rte3d
2 use kelp_context
3 use rte_sparse_matrices
4 use light_context
5 use type_consts
6 implicit none
7

```

```

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

```

```

54         ddx => x_cd2_first
55     else if(indices%i .eq. mat%grid%x%num
56             ) then
57         ddx => x_cd2_last
58     else
59         ddx => x_cd2
60     end if
61
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
65             ) then
66         ddy => y_cd2_last
67     else
68         ddy => y_cd2
69     end if
70
70     call angle_loop(mat, indices, ddx,
71                     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)
78   result(ent)
79   type(space_angle_grid) grid
80   type(index_list) indices
81   integer(index_kind) ent
82   integer(index_kind) boundary_nnz, interior_nnz
83   integer(index_kind) num_boundary, num_interior
84   integer(index_kind) num_this_x, num_this_z
85
85   ! Nonzero matrix entries for an surface or
86   ! bottom spatial grid cell
87   ! Definitely an integer since nomega is even
88   boundary_nnz = grid%angles%nomega * (2 * grid%
89   ! Nonzero matrix entries for an interior
89   ! spatial grid cell
90   interior_nnz = grid%angles%nomega * (grid%
91   ! Order: z, x, y, omega
92   ! Total number traversed so far in each
92   ! spatial category
93   ! row
94   num_this_x = indices%j - 1
95   ! depth layer

```

```

96  num_this_z = (indices%i - 1) * grid%y%num +
97      num_this_x
98
99      ! Calculate number of spatial grid cells of
100     ! each type which have
101     ! already been traversed up to this point
102  if(indices%k .eq. 1) then
103      num_boundary = num_this_z
104      num_interior = 0
105  else if(indices%k .eq. grid%z%num) then
106      num_boundary = (grid%x%num * grid%y%num) +
107          num_this_z
108      num_interior = (grid%z%num-2) * grid%x%num
109          * grid%y%num
110  else
111      num_boundary = grid%x%num * grid%y%num
112      num_interior = num_this_z + (indices%k-2) *
113          grid%x%num * grid%y%num
114  end if
115
116  ent = num_boundary * boundary_nnz +
117      num_interior * interior_nnz + 1
118 end function calculate_start_ent
119
120 function calculate_repeat_ent(ent, p) result(
121     repeat_ent)
122     integer p
123     integer(index_kind) ent, repeat_ent
124     ! Entry number for row=mat%ind(i,j,k,p), col=
125     ! mat%ind(i,j,k,p),
126     ! which will be modified multiple times in
127     ! this matrix row
128     repeat_ent = ent + p - 1
129 end function calculate_repeat_ent
130
131 subroutine interior_angle_loop(mat, indices, ddx
132     , ddy)
133     type(rte_mat) mat
134     type(index_list) indices
135     procedure(deriv_interface) :: ddx, ddy
136     integer p
137     integer(index_kind) ent, repeat_ent
138     integer(index_kind) row_num
139
140     ! Determine which matrix row to start at
141     ent = calculate_start_ent(mat%grid, indices)
142     indices%p = 1
143     row_num = mat%ind(indices%i, indices%j,
144         indices%k, indices%p)
145
146     do p=1, mat%grid%angles%nomega

```

```

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

```

```

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,
191 |     ddy)
192 type(rte_mat) mat
193 type(index_list) indices
194 integer p
195 integer(index_kind) row_num, ent, repeat_ent
196 procedure(deriv_interface) :: ddx, ddy
197 ! 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,
201 |     indices%k, indices%p)
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%
217 |     angles%nomega
218 |     indices%p = p
219 |     repeat_ent = calculate_repeat_ent(ent, p)
220 |     call mat%set_row(ent, row_num)
221 |     call mat%angular_integral(indices, ent)
222 |     call ddx(mat, indices, ent)
223 |     call ddy(mat, indices, ent)
224 |     call mat%z_bottom_bc(indices, ent,
225 |         repeat_ent)
226 |     call mat%attenuate(indices, repeat_ent)
227 |     call mat%add_source(indices, row_num)
228 |     row_num = row_num + 1
229 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,
238 |     num_threads)
239 | ! call surface_space_loop(mat, indices)
240 | ! call interior_space_loop(mat, indices)
241 | ! call bottom_space_loop(mat, indices)
242 | end subroutine gen_matrix
243 |
244 | subroutine rte3d_deinit(mat, iops, light)
245 |   type(rte_mat) mat
246 |   type(optical_properties) iops
247 |   type(light_state) light
248 |
249 |   call mat%deinit()
250 |   call iops%deinit()
251 |   call light%deinit()
252 | end subroutine
253 |

```

```

      kelp_context.f90
1 | module kelp_context
2 | use sag
3 | use prob
4 | implicit none
5 |
6 | ! Point in cylindrical coordinates
7 | type point3d
8 |   double precision x, y, z, theta, r
9 | contains
10 |   procedure :: set_cart => point_set_cart
11 |   procedure :: set_cyl => point_set_cyl
12 |   procedure :: cartesian_to_polar
13 |   procedure :: polar_to_cartesian
14 | end type point3d
15 |
16 | type frond_shape
17 |   double precision fs, fr, tan_alpha, alpha, ft
18 | contains
19 |   procedure :: set_shape => frond_set_shape
20 |   procedure :: calculate_angles =>
21 |     frond_calculate_angles

```

```

22
23 type rope_state
24   integer nz
25   double precision, dimension(:), allocatable
26     :: frond_lengths, frond_stds, num_fronds,
27       water_speeds, water_angles
28 contains
29   procedure :: init => rope_init
30   procedure :: deinit => rope_deinit
31 end type rope_state
32
33 type depth_state
34   double precision frond_length, frond_std,
35     num_fronds, water_speeds, water_angles,
36     depth
37   integer depth_layer
38 contains
39   procedure :: set_depth
40   procedure :: length_distribution_cdf
41   procedure :: angle_distribution_pdf
42 end type depth_state
43
44 type optical_properties
45   integer num_vsf
46   type(space_angle_grid) grid
47   double precision, dimension(:), allocatable
48     :: vsf_angles, vsf_vals
49   double precision, dimension(:), allocatable
50     :: abs_water
51   double precision abs_kelp, scat
52   ! On x, y, z grid - including water & kelp.
53   double precision, dimension(:,:,:,:),
54     allocatable :: abs_grid
55   double precision, dimension(:,:,:,:,:),
56     allocatable :: source_grid
57   ! On theta, phi, theta_prime, phi_prime grid
58   double precision, dimension(:,:,:), allocatable
59     :: vsf, vsf_integral
60 contains
61   procedure :: init => iop_init
62   procedure :: calculate_coef_grids
63   procedure :: zero_source => iop_zero_source
64   procedure :: set_source => iop_set_source
65   procedure :: load_vsf
66   procedure :: eval_vsf
67   procedure :: calc_vsf_on_grid
68   procedure :: deinit => iop_deinit
69   procedure :: vsf_from_function
70 end type optical_properties
71
72 type boundary_condition
73   double precision I0, decay, theta_s, phi_s
74   type(space_angle_grid) grid

```

```

66   double precision, dimension(:), allocatable
67   :: bc_grid
68 contains
69   procedure :: bc_gaussian
70   procedure :: init => bc_init
71   procedure :: deinit => bc_deinit
72 end type boundary_condition
73
74 contains
75   function bc_gaussian(bc, theta, phi)
76     class(boundary_condition) bc
77     double precision theta, phi, diff
78     double precision bc_gaussian
79     diff = angle_diff_3d(theta, phi, bc%theta_s,
80                           bc%phi_s)
81     bc_gaussian = exp(-bc%decay * diff)
82   end function bc_gaussian
83
84   subroutine bc_init(bc, grid, theta_s, phi_s,
85                      decay, I0)
86     class(boundary_condition) bc
87     type(space_angle_grid) grid
88     double precision theta_s, phi_s, decay, I0
89     integer p
90     double precision theta, phi
91     double precision bc_norm
92     double precision, allocatable, dimension(:)
93     :: whole_bc_grid
94     integer nomega
95
96     nomega = grid%angles%nomega
97
98     allocate(bc%bc_grid(nomega/2))
99     allocate(whole_bc_grid(nomega))
100
101    bc%theta_s = theta_s
102    bc%phi_s = phi_s
103    bc%decay = decay
104    bc%I0 = I0
105
106    ! Only set BC for downwelling light
107    do p=1, nomega/2
108      theta = grid%angles%theta_p(p)
109      phi = grid%angles%phi_p(p)
110      bc%bc_grid(p) = bc%bc_gaussian(theta, phi)
111    )
112  end do
113
114  ! Normalize

```

```

111 ! Use 'whole_bc_grid' because angular
112   integration
113 ! subroutine requires all angles to have
114   values,
115 ! but 'bc%bc_grid' only has downwelling
116   values.
117 ! Use zeros for upwelling.
118 whole_bc_grid(1:nomega/2) = bc%bc_grid
119 whole_bc_grid(nomega/2+1:nomega) = 0
120 bc_norm = grid%angles%integrate_points(
121   whole_bc_grid)
122 bc%bc_grid = bc%I0 * bc%bc_grid / bc_norm
123 end subroutine bc_init
124
125 subroutine bc_deinit(bc)
126   class(boundary_condition) bc
127   deallocate(bc%bc_grid)
128 end subroutine
129
130 subroutine point_set_cart(point, x, y, z)
131   class(point3d) :: point
132   double precision x, y, z
133   point%x = x
134   point%y = y
135   point%z = z
136   call point%cartesian_to_polar()
137 end subroutine point_set_cart
138
139 subroutine point_set_cyl(point, theta, r, z)
140   class(point3d) :: point
141   double precision theta, r, z
142   point%theta = theta
143   point%r = r
144   point%z = z
145   call point%polar_to_cartesian()
146 end subroutine point_set_cyl
147
148 subroutine polar_to_cartesian(point)
149   class(point3d) :: point
150   point%x = point%r*cos(point%theta)
151   point%y = point%r*sin(point%theta)
152 end subroutine polar_to_cartesian
153
154 subroutine cartesian_to_polar(point)
155   class(point3d) :: point
156   point%r = sqrt(point%x**2 + point%y**2)
157   point%theta = atan2(point%y, point%x)
158 end subroutine cartesian_to_polar
159
160 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 /
168     (1.d0 + frond%fs)
169   frond%alpha = atan(frond%tan_alpha)
170 end subroutine
171
172 subroutine iop_init(iops, grid)
173   class(optical_properties) iops
174   type(space_angle_grid) grid
175
176   iops%grid = grid
177
178   ! Assume that these are preallocated and
179   ! passed to function
180   ! Nevermind, don't assume this.
181   allocate(iops%abs_water(grid%z%num))
182
183   ! Assume that these must be allocated here
184   ! NOTE: vsf_angles are defined on [0, pi].
185   ! (not on [-1, 1])
186   allocate(iops%vsf_angles(iops%num_vsf))
187   allocate(iops%vsf_vals(iops%num_vsf))
188   allocate(iops%vsf(grid%angles%nomega, grid%
189     angles%nomega))
190   allocate(iops%vsf_integral(grid%angles%
191     nomega, grid%angles%nomega))
192   allocate(iops%abs_grid(grid%x%num, grid%y%
193     num, grid%z%num))
194   allocate(iops%source_grid(grid%x%num, grid%y%
195     %num, grid%z%num, grid%angles%nomega))
196
197   call iops%zero_source()
198 end subroutine iop_init
199
200 subroutine iop_zero_source(iops)
201   class(optical_properties) iops
202
203   iops%source_grid = 0
204 end subroutine iop_zero_source
205
206 subroutine iop_set_source(iops, source)

```

```

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 -
216 |             iops%abs_water(k)) * p_kelp(:,:,k) +
217 |             iops%abs_water(k)
218 |     end do
219 |
220 | end subroutine calculate_coef_grids
221 |
222 | subroutine load_vsf(iops, filename, fmtstr)
223 |     class(optical_properties) :: iops
224 |     character(len=*) :: filename, fmtstr
225 |     double precision, dimension(:,:), allocatable :: tmp_2d_arr
226 |     integer num_rows, num_cols, skiplines_in
227 |
228 |     ! First column is the angle at which the
229 |     ! measurement is taken
230 |     ! Second column is the value of the VSF at
231 |     ! that angle
232 |     num_rows = iops%num_vsf
233 |     num_cols = 2
234 |     skiplines_in = 1 ! Ignore comment on first
235 |     ! line
236 |
237 |     allocate(tmp_2d_arr(num_rows, num_cols))
238 |
239 |     tmp_2d_arr = read_array(filename, fmtstr,
240 |         num_rows, num_cols, skiplines_in)
241 |     iops%vsf_angles = tmp_2d_arr(:,1)
242 |     iops%vsf_vals = tmp_2d_arr(:,2)
243 |
244 |     ! write(*,*) 'vsf_angles = ', iops%
245 |     ! vsf_angles
246 |     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
247 |
248 |     ! 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
252   , not the value.
253   eval_vsf = interp(theta, iops%vsf_angles,
254                      iops%vsf_vals, iops%num_vsf)
255
256 end function eval_vsf
257
258 subroutine rope_init(rope, grid)
259   class(rope_state) :: rope
260   type(space_angle_grid) :: grid
261
262   rope%nz = grid%z%num
263   allocate(rope%frond_lengths(rope%nz))
264   allocate(rope%frond_stds(rope%nz))
265   allocate(rope%water_speeds(rope%nz))
266   allocate(rope%water_angles(rope%nz))
267   allocate(rope%num_fronds(rope%nz))
268 end subroutine rope_init
269
270 subroutine rope_deinit(rope)
271   class(rope_state) rope
272   deallocate(rope%frond_lengths)
273   deallocate(rope%frond_stds)
274   deallocate(rope%water_speeds)
275   deallocate(rope%water_angles)
276   deallocate(rope%num_fronds)
277 end subroutine rope_deinit
278
279 subroutine set_depth(depth, rope, grid,
280                      depth_layer)
281   class(depth_state) depth
282   type(rope_state) rope
283   type(space_angle_grid) grid
284   integer depth_layer
285
286   depth%frond_length = rope%frond_lengths(
287     depth_layer)
288   depth%frond_std = rope%frond_stds(
289     depth_layer)
290   depth%water_speeds = rope%water_speeds(
291     depth_layer)

```

```

286   depth%water_angles = rope%water_angles(
287     depth_layer)
288   depth%num_fronds = rope%num_fronds(
289     depth_layer)
290   depth%depth_layer = depth_layer
291   depth%depth = grid%z%vals(depth_layer)
292 end subroutine set_depth
293
294 function length_distribution_cdf(depth, L)
295   result(output)
296   ! C_L(L)
297   class(depth_state) depth
298   double precision L, L_mean, L_std
299   double precision output
300
301   L_mean = depth%frond_length
302   L_std = depth%frond_std
303
304   call normal_cdf(L, L_mean, L_std, output)
305 end function length_distribution_cdf
306
307 function angle_distribution_pdf(depth, theta_f)
308   result(output)
309   ! P_{\theta_f}(\theta_f)
310   class(depth_state) depth
311   double precision theta_f, v_w, theta_w
312   double precision output
313   double precision diff
314
315   v_w = depth%water_speeds
316   theta_w = depth%water_angles
317
318   ! von_mises_pdf is only defined on [-pi, pi]
319   ! So take difference of angles and input
320   ! into
321   ! von_mises dist. centered & x=0.
322
323   diff = angle_diff_2d(theta_f, theta_w)
324
325   call von_mises_pdf(diff, 0.d0, v_w, output)
326 end function angle_distribution_pdf
327
328 function angle_mod(theta) result(mod_theta)
329   ! Shift theta to the interval [-pi, pi]
330   ! which is where von_mises_pdf is defined.
331
332   double precision theta, mod_theta
333
334   mod_theta = mod(theta + pi, 2.d0*pi) - pi
335 end function angle_mod

```

```

332 | function angle_diff_2d(theta1, theta2) result(
333 |   diff)
334 |   ! Shortest difference between two angles
335 |   ! which may be
336 |   ! in different periods.
337 |   double precision theta1, theta2, diff
338 |   double precision modt1, modt2
339 |
340 |   ! Shift to [0, 2*pi]
341 |   modt1 = mod(theta1, 2*pi)
342 |   modt2 = mod(theta2, 2*pi)
343 |
344 |   ! https://gamedev.stackexchange.com/
345 |   ! questions/4467/comparing-angles-and-
346 |   ! working-out-the-difference
347 |
348 | end function angle_diff_2d
349 |
350 | function angle_diff_3d(theta, phi, theta_prime,
351 |   , phi_prime) result(diff)
352 |   ! Angle between two vectors in spherical
353 |   ! coordinates
354 |   double precision theta, phi, theta_prime,
355 |   phi_prime
356 |   double precision alpha, diff
357 |
358 |   ! Faster, but produces lots of NaNs (at
359 |   ! least in Python)
360 |   !alpha = sin(theta)*sin(theta_prime)*cos(
361 |   !     theta-theta_prime) + cos(phi)*cos(
362 |   !     phi_prime)
363 |
364 |   ! Slower, but more accurate
365 |   alpha = (sin(phi)*sin(phi_prime) &
366 |   * (cos(theta)*cos(theta_prime) + sin(theta)
367 |   !     )*sin(theta_prime)) &
368 |   + cos(phi)*cos(phi_prime))
369 |
370 |   ! Avoid out-of-bounds errors due to rounding
371 |   alpha = min(1.d0, alpha)
372 |   alpha = max(-1.d0, alpha)
373 |
374 |   diff = acos(alpha)
375 | end function angle_diff_3d
376 |
377 | subroutine vsf_from_function(iops, func)
378 |   class(optical_properties) iops
379 |   double precision, external :: func
380 |   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(
404         angle_diff_3d(th,ph,thp,php))
405     end do
406
407     ! Normalize each row of VSF (midpoint
408     ! rule)
409     norm = sum(iops%vsf(p,:) * iops%grid%
410               angles%area_p(:))
411     iops%vsf(p,:) = iops%vsf(p,:) / norm
412
413     ! % / meter light scattered
414     ! from cell pp (2nd ind.) into direction
415     ! p (1st ind.).
416     iops%vsf_integral(p, :) = iops%vsf(p, :)
417     &
418     * iops%grid%angles%area_p(:)
419     !write(*,*) 'vsf_integral (beta_pp)', p,
420     ' = ', 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

```

1 module light_context
2 ! Use 64-bit integers for LIS
3 ! Necessary for FD solution w/ large matrices
4 #define LONG__LONG
5 #include "lisf.h"
6 use sag
7 use rte_sparse_matrices
8 !use hdf5
9 implicit none
10
11 type light_state
12   double precision, dimension(:,:,:),
13     allocatable :: irradiance
14   double precision, dimension(:,:,:,:),
15     allocatable :: radiance
16   type(space_angle_grid) :: grid
17   type(rte_mat) :: mat
18   contains
19     procedure :: init => light_init
20     procedure :: init_grid => light_init_grid
21     procedure :: calculate_radiance
22     procedure :: calculate_irradiance =>
23       light_calculate_irradiance
24     procedure :: deinit => light_deinit
25     !procedure :: to_hdf => light_to_hdf
26   end type light_state
27
28 contains
29
30   ! Init for use with mat
31   subroutine light_init(light, mat)
32     class(light_state) light
33     type(rte_mat) mat

```

```

31     integer nx, ny, nz, nomega
32
33     light%mat = mat
34     light%grid = mat%grid
35
36     nx = light%grid%x%num
37     ny = light%grid%y%num
38     nz = light%grid%z%num
39     nomega = light%grid%angles%nomega
40
41     allocate(light%irradiance(nx, ny, nz))
42     allocate(light%radiance(nx, ny, nz, nomega))
43 end subroutine light_init
44
45 ! Init for use without mat
46 subroutine light_init_grid(light, grid)
47   class(light_state) light
48   type(space_angle_grid) grid
49   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,
75   !                           gn)
76
77   ! write(*,*) 'ln =', ln
78   ! write(*,*) 'gn =', gn
79   index = 1

```

```

80
81      write(*,*) 'Set matrix values'
82      write(*,*) 'index_kind = ', storage_size(
83          index)
84      write(*,*) 'lis_kind = ', storage_size(
85          lis_test_int)
86      ! Set initial guess from provided radiance
87      ! Traverse solution vector in order
88      ! so as to avoid calculating index
89      do k=1, nz
90          do i=1, nx
91              do j=1, ny
92                  do p=1, nomega
93                      call lis_vector_set_value(
94                          LIS_INS_VALUE, index, &
95                          light%radiance(i,j,k,p),
96                          light%mat%x, light%mat%
97                          ierr)
98                      if(light%mat%ierr .ne. 0) then
99                          write(*,*) 'IG ERROR:', light
100                         %mat%ierr
101                     end if
102
103                     index = index + 1
104                 end do
105             end do
106         end do
107     end do
108
109     !call light%mat%initial_guess()
110
111     ! Solve (LIS)
112     write(*,*) 'Solve matrix'
113     call light%mat%solve()
114
115     index = 1
116
117     write(*,*) 'Extract solution'
118     ! Extract solution
119     do k=1, nz
120         do i=1, nx
121             do j=1, ny
122                 do p=1, nomega
123                     call lis_vector_get_value(light%
124                         mat%x, index, &
125                         light%radiance(i,j,k,p),
126                         light%mat%ier
127                     if(light%mat%ier .ne. 0) then
128                         write(*,*) 'EXTRACT ERROR:', light%mat%ier
129                     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%
134     angles%nomega) :: tmp_rad
135
136   nx = light%grid%x%num
137   ny = light%grid%y%num
138   nz = light%grid%z%num
139
140   do i=1, nx
141     do j=1, ny
142       do k=1, nz
143         ! Use temporary array to avoid
144         ! creating one
145         ! implicitly at every spatial grid
146         ! point
147         tmp_rad = light%radiance(i,j,k,:)
148         light%irradiance(i,j,k) = &
149           light%grid%angles%
150             integrate_points(tmp_rad)
151       end do
152     end do
153   end do
154
155 end subroutine light_calculate_irradiance
156
157 ! subroutine light_to_hdf(light, radfile,
158 !   irradfile)
159 !   class(light_state) light
160 !   character(len=*) radfile
161 !   character(len=*) irradfile
162 !
163 !   call hdf_write_radiance(radfile, light%
164 !     radiance, light%grid)
165 !   call hdf_write_irradiance(irradfile, light%
166 !     irradiance, light%grid)
167 ! end subroutine light_to_hdf
168
169 subroutine light_deinit(light)
170   class(light_state) light
171
172   deallocate(light%irradiance)

```

```

166 |     deallocate(light%radiance)
167 | end subroutine light_deinit
168 | end module

asymptotics.f90
1 | module asymptotics
2 |   use kelp_context
3 |   !use rte_sparse_matrices
4 |   !use light_context
5 |   implicit none
6 |   contains
7 |
8 |   subroutine calculate_asymptotic_light_field(
9 |     grid, bc, iops, source, radiance,
10 |     num_scatters, num_threads)
11 |     type(space_angle_grid) grid
12 |     type(boundary_condition) bc
13 |     type(optical_properties) iops
14 |     double precision, dimension(:,:,:,:,:) :: radiance
15 |     double precision, dimension(:,:,:,:,:), allocatable :: rad_scatter
16 |     double precision, dimension(:,:,:,:,:) :: source
17 |     integer num_scatters
18 |     integer nx, ny, nz, nomega
19 |     integer max_cells
20 |     integer n
21 |     logical bc_flag
22 |     integer num_threads
23 |
24 |     double precision bb
25 |
26 |     double precision, dimension(:), allocatable :: path_length, path_spacing, a_tilde, gn
27 |
28 |     nx = grid%x%num
29 |     ny = grid%y%num
30 |     nz = grid%z%num
31 |     nomega = grid%angles%nomega
32 |
33 |     max_cells = calculate_max_cells(grid)
34 |
35 |     allocate(path_length(max_cells+1))
36 |     allocate(path_spacing(max_cells))
37 |     allocate(a_tilde(max_cells))
38 |     allocate(gn(max_cells))
39 |     allocate(rad_scatter(grid%x%num, grid%y%num,
40 |                           grid%z%num, grid%angles%nomega))

```

```

39 |     write(*,*) 'before'
40 |     write(*,*) 'min radiance =', minval(radiance
41 |             )
42 |     write(*,*) 'max radiance =', maxval(radiance
43 |             )
44 |     write(*,*) 'mean radiance =', sum(radiance)/
45 |             size(radiance)
46 |
47 |     write(*,*) 'advect source + bc'
48 |     bc_flag = .true.
49 |     call advect_light( &
50 |             grid, iops, source, radiance, &
51 |             path_length, path_spacing, &
52 |             a_tilde, gn, bc_flag, num_threads, bc)
53 |
54 |     write(*,*) 'after'
55 |     write(*,*) 'min radiance =', minval(radiance
56 |             )
57 |     write(*,*) 'max radiance =', maxval(radiance
58 |             )
59 |     write(*,*) 'mean radiance =', sum(radiance)/
60 |             size(radiance)
61 |
62 |     rad_scatter = radiance
63 |     bb = iops%scat
64 |
65 |     do n=1, num_scatters
66 |         write(*,*) 'scatter #', n
67 |         call scatter(grid, iops, source,
68 |             rad_scatter, path_length, path_spacing
69 |             , a_tilde, gn, num_threads)
70 |         radiance = radiance + bb**n * rad_scatter
71 |     end do
72 |
73 |     write(*,*) 'asymptotics complete'
74 |
75 |     deallocate(path_length)
76 |     deallocate(path_spacing)
77 |     deallocate(a_tilde)
78 |     deallocate(gn)
79 |     deallocate(rad_scatter)
80 | end subroutine
81 |         calculate_asymptotic_light_field
82 |
83 | subroutine
84 |     calculate_asymptotic_light_field_expanded_source
85 |     (&
86 |         grid, bc, iops, source, &
87 |         source_expansion, radiance, &
88 |         num_scatters, num_threads)

```

```

78 | type(space_angle_grid) grid
79 | type(boundary_condition) bc
80 | type(optical_properties) iops
81 | double precision, dimension(:,:,:,:,:) :: 
82 |     radiance
83 | double precision, dimension(:,:,:,:,:) :: 
84 |     source_expansion
85 | integer num_scatters
86 | integer nx, ny, nz, nomega
87 | integer max_cells
88 | integer n
89 | logical bc_flag
90 | integer num_threads
91 |
92 | double precision bb
93 |
94 | double precision, dimension(:,), allocatable
95 |     :: path_length, path_spacing, a_tilde, gn
96 | double precision, dimension(:,:,:,:),
97 |     allocatable :: source
98 | double precision, dimension(:,:,:,:),
99 |     allocatable :: rad_scatter
100 | double precision, dimension(:,:,:,:),
101 |     allocatable :: scatter_integral
102 |
103 | nx = grid%x%num
104 | ny = grid%y%num
105 | nz = grid%z%num
106 | nomega = grid%angles%nomega
107 |
108 | max_cells = calculate_max_cells(grid)
109 |
110 | allocate(path_length(max_cells+1))
111 | allocate(path_spacing(max_cells))
112 | allocate(a_tilde(max_cells))
113 | allocate(gn(max_cells))
114 | allocate(rad_scatter(grid%x%num, grid%y%num,
115 |     grid%z%num, grid%angles%nomega))
116 | allocate(scatter_integral(nx, ny, nz, nomega
117 | ))
118 |
119 | write(*,*) 'advect source + bc'
120 | bc_flag = .true.
121 | call advect_light( &
122 |     grid, iops, source_expansion(:,:,:,:,:,1)
123 |     , radiance, &
124 |     path_length, path_spacing, &
125 |     a_tilde, gn, bc_flag, num_threads, bc)
126 | ! Disable BC for scattering advection
127 | bc_flag = .false.
128 |
129 |

```

```

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

```

```

159 end subroutine advect_surface_bc
160
161 ! Perform one scattering event
162 subroutine scatter(grid, iops, source,
163     rad_scatter, path_length, path_spacing,
164     a_tilde, gn, num_threads)
165 type(space_angle_grid) grid
166 type(optical_properties) iops
167 double precision, dimension(:,:,:,:,:) :::
168     rad_scatter, source
169 double precision, dimension(:,:,:,:,:) :::
170     allocatable :: scatter_integral
171 double precision, dimension(:) :::
172     path_length, path_spacing, a_tilde, gn
173 integer nx, ny, nz, nomega
174 logical bc_flag
175 integer num_threads
176
177 nx = grid%x%num
178 ny = grid%y%num
179 nz = grid%z%num
180 nomega = grid%angles%nomega
181 bc_flag = .false.
182
183 allocate(scatter_integral(nx, ny, nz, nomega
184     ))
185
186 call calculate_scatter_source(grid, iops,
187     rad_scatter, source, scatter_integral,
188     num_threads)
189 call advect_light(grid, iops, source,
190     rad_scatter, path_length, path_spacing,
191     a_tilde, gn, bc_flag, num_threads)
192
193 deallocate(scatter_integral)
194 end subroutine scatter
195
196 ! Calculate source from no-scatter or previous
197 ! scattering layer
198 subroutine calculate_scatter_source(grid, iops
199     , rad_scatter, source, scatter_integral,
200     num_threads)
201 type(space_angle_grid) grid
202 type(optical_properties) iops
203 double precision, dimension(:,:,:,:,:) :::
204     rad_scatter
205 double precision, dimension(:,:,:,:,:) :::
206     source
207 double precision, dimension(:,:,:,:,:) :::
208     scatter_integral
209 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      !$omp parallel do default(none) private(
207          indices) &
208          !$omp private(i,j,k,p) shared(nx,ny,nz,
209              nomega) &
210          !$omp shared(iops, rad_scatter,
211              scatter_integral) &
212          !$omp num_threads(num_threads) collapse(2)
213
214      do k=1, nz
215          do i=1, nx
216              indices%k = k
217              indices%i = i
218              do j=1, ny
219                  indices%j = j
220                  do p=1, nomega
221                      indices%p = p
222                      call calculate_scatter_integral
223                          (&
224                              iops, rad_scatter,&
225                              scatter_integral,&
226                              indices)
227                  end do
228              end do
229          end do
230      end do
231      !$omp end parallel do
232
233      source(:,:,:,:) = -rad_scatter(:,:,:,:)
234      source(:,:,:,:) = scatter_integral(:,:,:,:)
235
236      write(*,*) 'source min: ', minval(source)
237      write(*,*) 'source max: ', maxval(source)
238      write(*,*) 'source mean: ', sum(source)/size
239          (source)
240
241      end subroutine calculate_scatter_source

```

```

237 subroutine calculate_scatter_integral(iops,
238     rad_scatter, scatter_integral, indices)
239     type(optical_properties) iops
240     double precision, dimension(:,:,:,:) :: 
241         rad_scatter, scatter_integral
242     type(index_list) indices
243
244     scatter_integral(indices%i,indices%j,indices
245         %k,indices%p) &
246         = sum(iops%vsf_integral(indices%p, :) &
247             * rad_scatter(&
248                 indices%i,&
249                 indices%j,&
250                 indices%k,:))
251
252 end subroutine calculate_scatter_integral
253
254 subroutine advect_light(grid, iops, source,
255     rad_scatter, path_length, path_spacing,
256     a_tilde, gn, bc_flag, num_threads, bc)
257     type(space_angle_grid) grid
258     type(optical_properties) iops
259     double precision, dimension(:,:,:,:) :: 
260         rad_scatter, source
261     double precision, dimension(:) :: 
262         path_length, path_spacing, a_tilde, gn
263     logical bc_flag
264     type(boundary_condition), intent(in),
265         optional :: bc
266     integer i, j, k, p
267     integer num_threads
268
269 !$omp parallel do default(none) &
270 !$omp private(i,j,k,p) &
271 !$omp shared(rad_scatter,source,grid,iops,
272     bc_flag,bc) &
273 !$omp private(path_length,path_spacing,
274     a_tilde,gn) &
275 !$omp num_threads(num_threads) collapse(2)
276 do k=1, grid%z%num
277     do i=1, grid%x%num
278         do j=1, grid%y%num
279             do p=1, grid%angles%omega
280                 call integrate_ray(grid, iops,
281                     source,&
282                     rad_scatter, path_length,
283                         path_spacing,&
284                         a_tilde, gn, i, j, k, p,
285                         bc_flag, bc)
286         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
281 ! piecewise-constant 1d funcs
282 subroutine integrate_ray(grid, iops, source,
283 rad_scatter, path_length, path_spacing,
284 a_tilde, gn, i, j, k, p, bc_flag, bc)
285 type(space_angle_grid) :: grid
286 type(optical_properties) :: iops
287 double precision, dimension(:,:,:,:,:) :: source
288 double precision, dimension(:,:,:,:,:) :: rad_scatter
289 integer :: i, j, k, p
290 ! The following are only passed to avoid
291 ! unnecessary allocation
292 double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn
293 logical bc_flag
294 type(boundary_condition), intent(in),
295 optional :: bc
296
297 integer num_cells
298
299 call traverse_ray(grid, iops, source, i, j,
300 k, p, path_length, path_spacing, a_tilde,
301 gn, num_cells)
302 rad_scatter(i,j,k,p) =
303     calculate_ray_integral(num_cells,
304     path_length, path_spacing, a_tilde, gn)
305
306 if(bc_flag .and. p .le. grid%angles%nomega
307 /2) then
308     call advect_surface_bc(&
309         i, j, k, p, rad_scatter, &
310         path_spacing, num_cells, &
311         a_tilde, bc)
312 end if
313
314 ! if((i .eq. 1) &
315 !     .and. (j .eq. 1) &
316 !     !.and. (k .eq. grid%z%num/2) &
317 !     .and. ( (
318 !         (p .eq. 1) .or. (p .eq. grid%angles%
319 !             nomega) &
320 !             )) then

```

```

311   !      write(*,*) 'ray (', i, ', ', j, ', ', k
312   !      , ', ', p, ')
313   !      write(*,*) 'num_cells = ', num_cells
314   !      write(*,*) 'path_spacing:'
315   !      write(*,*) path_spacing(1:num_cells)
316   !      write(*,*) 'path_length:'
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,
327   ds, a_tilde, gn) result(integral)
327 ! Double integral which accumulates all
328 ! scattered light along the path
328 ! via an angular integral and attenuates it
329 ! by integrating along the path
329 integer :: num_cells
330 double precision, dimension(num_cells) :: ds
330   , 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
337 ! 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
347 ! occurred
348 ! previously on the path that we don't
348 ! need
349 ! to continue, and we might get underflow
349 ! if we do.
350 if(bi .lt. -100.d0) then
351   di_exp_bi = 0.d0
352 else

```

```

353      ! Without this conditional, overflow
354      ! occurs.
355      ! Which is unnecessary, because large
356      ! absorption
357      ! means very small light added to the
358      ! ray
359      ! at this grid cell.
360      if(a_tilde(i) .lt. cutoff) then
361          if(a_tilde(i) .eq. 0) then
362              di_exp_bi = ds(i) * exp(bi)
363          else
364              ! In an attempt to avoid
365              ! overflow
366              ! and reduce compute time,
367              ! I'm combining exponentials.
368              ! di*exp(bi) -> di_exp_bi
369              di_exp_bi = (exp(a_tilde(i)*s(i
370                  +1) + bi) - exp(a_tilde(i)*s(
371                      i) + bi))/a_tilde(i)
372          end if
373          integral = integral + gn(i)*
374              di_exp_bi
375      end if
376  end if
377 end do
378
379 end function calculate_ray_integral
380
381 ! Calculate maximum number of cells a path
382 ! through the grid could take
383 ! This is a loose upper bound
384 function calculate_max_cells(grid) result(
385     max_cells)
386 type(space_angle_grid) :: grid
387 integer :: max_cells
388 double precision dx, dy, zrange, phi_middle
389
390 ! Angle that will have the longest ray
391 phi_middle = grid%angles%phi(grid%angles%
392     nphi/2)
393 dx = grid%x%spacing(1)
394 dy = grid%y%spacing(1)
395 zrange = grid%z%maxval - grid%z%minval
396
397 max_cells = grid%z%num + ceiling((1/dx+1/dy)
398     *zrange*tan(phi_middle))
399 end function calculate_max_cells
400
401 ! Traverse from surface or bottom to point (xi
402     , yj, zk)

```

```

392 ! in the direction omega_p, extracting path
393   lengths (ds) and
394 ! function values (f) along the way,
395 ! as well as number of cells traversed (n).
396 subroutine traverse_ray(grid, iops, source, i,
397   j, k, p, s_array, ds, a_tilde, gn,
398   num_cells)
399   type(space_angle_grid) :: grid
400   type(optical_properties) :: iops
401   double precision, dimension(:,:,:,:,:) :: source
402   integer :: i, j, k, p
403   double precision, dimension(:) :: s_array,
404   ds, a_tilde, gn
405   integer :: num_cells
406
407   integer t
408   double precision p0x, p0y, p0z
409   double precision p1x, p1y, p1z
410   double precision z0
411   double precision s_tilde, s
412   integer dir_x, dir_y, dir_z
413   integer shift_x, shift_y
414   integer cell_x, cell_y, cell_z
415   integer edge_x, edge_y
416   integer first_x, last_x, first_y, last_y,
417     last_z
418   double precision s_next_x, s_next_y,
419     s_next_z, s_next
420   double precision x_factor, y_factor,
421     z_factor
422   double precision ds_x, ds_y
423   double precision, dimension(grid%z%num) :: ds_z
424   double precision smx, smy
425
426   ! Divide by these numbers to get path
427   ! separation
428   ! from separation in individual dimensions
429   x_factor = grid%angles%sin_phi_p(p) * grid%
430     angles%cos_theta_p(p)
431   y_factor = grid%angles%sin_phi_p(p) * grid%
432     angles%sin_theta_p(p)
433   z_factor = grid%angles%cos_phi_p(p)
434
435   ! Destination point
436   p1x = grid%x%vals(i)
437   p1y = grid%y%vals(j)
438   p1z = grid%z%vals(k)
439
440   !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 =>
477     ! increasing. -1 => decreasing.
478     dir_x = int(sgn(p1x-p0x))
479     dir_y = int(sgn(p1y-p0y))
480
481     ! Shifts
482     ! Conversion from cell_inds to edge_inds
483     ! merge is fortran's ternary operator
484     shift_x = merge(1,0,dir_x>0)
485     shift_y = merge(1,0,dir_y>0)
486
487     ! Indices for cell containing origin point
488     cell_x = floor((p0x-grid%x%minval)/grid%x%
489                     spacing(1)) + 1
490     cell_y = floor((p0y-grid%y%minval)/grid%y%
491                     spacing(1)) + 1
492
493     ! x and y may be in periodic image, so shift
494     ! back.
495     cell_x = mod1(cell_x, grid%x%num)
496     cell_y = mod1(cell_y, grid%y%num)
497
498     ! z starts at top or bottom depending on
499     ! direction.
500     if(dir_z > 0) then
501         cell_z = 1
502     else
503         cell_z = grid%z%num
504     end if
505
506     ! Edge indices preceding starting cells
507     edge_x = mod1(cell_x + shift_x, grid%x%num)
508     edge_y = mod1(cell_y + shift_y, grid%y%num)
509
510     ! First and last cells in each
511     if(dir_x .gt. 0) then
512         first_x = 1
513         last_x = grid%x%num
514     else
515         first_x = grid%x%num
516         last_x = 1
517     end if
518     if(dir_y .gt. 0) then
519         first_y = 1
520         last_y = grid%y%num
521     else
522         first_y = grid%y%num
523         last_y = 1
524     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%
527 |                         maxval)
528 |     smy = shift_mod(p0y, grid%y%minval, grid%y%
529 |                         maxval)
530 |     ! Path length to next edge plane in each
531 |     ! dimension
532 |     if(abs(x_factor) .lt. 1.d-10) then
533 |         ! Will never cross, so set above total
534 |         path length
535 |         s_next_x = 2*s_tilde
536 |     else if(cell_x .eq. last_x) then
537 |         ! If starts out at last cell, then
538 |         compare to periodic image
539 |         s_next_x = (grid%x%edges(first_x) + dir_x
540 |                         * (grid%x%maxval - grid%x%minval)&
541 |                         - smx) / x_factor
542 |     else
543 |         ! Otherwise, just compare to next cell
544 |         s_next_x = (grid%x%edges(edge_x) - smx) /
545 |                         x_factor
546 |     end if
547 |
548 |     ! Path length to next edge plane in each
549 |     ! dimension
550 |     if(abs(y_factor) .lt. 1.d-10) then
551 |         ! Will never cross, so set above total
552 |         path length
553 |         s_next_y = 2*s_tilde
554 |     else if(cell_y .eq. last_y) then
555 |         ! If starts out at last cell, then
556 |         compare to periodic image
557 |         s_next_y = (grid%y%edges(first_y) + dir_y
558 |                         * (grid%y%maxval - grid%y%minval)&

```

```

559      s_array(1) = 0.d0
560
561      ! Start with t=0 so that we can increment
562          ! before storing,
563          ! so that t will be the number of grid cells
564          ! 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
573              , cell_z)
573          gn(t) = source(cell_x, cell_y, cell_z, p)
574
575          !write(*,*) ''
576          !write(*,*) 's_next_x = ', s_next_x
577          !write(*,*) 's_next_y = ', s_next_y
578          !write(*,*) 's_next_z = ', s_next_z
579          !write(*,*) 'theta, phi =', grid%angles%
580              theta_p(p)*180.d0/pi, grid%angles%
581                  phi_p(p)*180.d0/pi
580          !write(*,*) 's = ', s, '/', s_tilde
581          !write(*,*) 'cell_z =', cell_z, '/', grid
582              %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))
588              then
589                  ! x edge is closest
590                  s_next = s_next_x
591
591                  ! Increment indices (periodic)
592                  cell_x = mod1(cell_x + dir_x, grid%x%
593                      num)
593                  edge_x = mod1(edge_x + dir_x, grid%x%
594                      num)
594
595                  ! x intersection after the one at s=
596                      s_next
596                  s_next_x = s_next + ds_x
597
598          else if (s_next_y .le. min(s_next_x,
598              s_next_z)) then

```

```

599      ! y edge is closest
600      s_next = s_next_y
601
602      ! Increment indices (periodic)
603      cell_y = mod1(cell_y + dir_y, grid%y%
604                      num)
604      edge_y = mod1(edge_y + dir_y, grid%y%
605                      num)
605
606      ! y intersection after the one at s=
607          s_next
608      s_next_y = s_next + ds_y
609
610      else if(s_next_z .le. min(s_next_x,
611                                  s_next_y)) then
611          ! z edge is closest
612          s_next = s_next_z
613
614          ! Increment indices
615          cell_z = cell_z + dir_z
616
617          ! write(*,*) 'z edge, s_next =', s_next
618
619          if(dir_z * (last_z - cell_z) .gt. 0)
620              then
621                  ! Only look ahead if we aren't at
622                  ! the end
623                  s_next_z = s_next + ds_z(cell_z)
624
625          else
626              ! Otherwise, no need to continue.
627              ! this is our final destination.
628              ! exit
629              s_next_z = 2*s_tilde
630              ! write(*,*) 'end. s_next_z =',
631                  s_next_z
632
633      end if
634
635
636      ! Cut off early if this is the end
637      ! This will be the last cell traversed if
638          s_next >= s_tilde
639      s_next = min(s_tilde, s_next)
640
641      ! Store path length
642      s_array(t+1) = s_next
643
644      ! Extract path length from same cell as
645          function vals
646      ds(t) = s_next - s

```

```

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

```

light_interface.f90

```

1 module light_interface
2   use rte3d
3   use kelp3d
4   use asymptotics
5   implicit none
6
7 contains
8   subroutine full_light_calculations( &
9     ! OPTICAL PROPERTIES
10    absorptance_kelp, & ! NOT THE SAME AS
11    ABSORPTION COEFFICIENT
12    abs_water, &
13    scat, &
14    num_vsf, &
15    vsf_file, &
16    ! SUNLIGHT
17    solar zenith, &
18    solar azimuthal, &
19    surface_irrad, &
20    ! KELP &
21    num_si, &
22    si_area, &
23    si_ind, &
24    frond_thickness, &
25    frond_aspect_ratio, &
26    frond_shape_ratio, &
27    ! WATER CURRENT
28    current_speeds, &
29    current_angles, &
30    ! SPACING
31    rope_spacing, &
32    depth_spacing, &
33    ! SOLVER PARAMETERS
34    nx, &
35    ny, &
36    nz, &
37    ntheta, &
38    nphi, &
39    num_scatters, &
40    ! FINAL RESULTS
41    perceived_irrad, &

```

```

41    avg_irrad)
42
43    implicit none
44
45    ! OPTICAL PROPERTIES
46    integer, intent(in) :: nx, ny, nz, ntheta,
47        nphi
48    ! Absorption and scattering coefficients
49    double precision, intent(in) :::
50        absorptance_kelp, scat
51    double precision, dimension(nz), intent(in)
52        :: abs_water
53    ! Volume scattering function
54    integer, intent(in) :: num_vsf
55    character(len=*) :: vsf_file
56    !double precision, dimension(num_vsf),
57        intent(int) :: vsf_angles
58    !double precision, dimension(num_vsf),
59        intent(int) :: vsf_vals
60
61    ! SUNLIGHT
62    double precision, intent(in) :: solar_zenith
63    double precision, intent(in) :::
64        solar_azimuthal
65    double precision, intent(in) :::
66        surface_irrad
67
68    ! KELP
69    ! Number of Superindividuals in each depth
70        level
71    integer, intent(in) :: num_si
72    ! si_area(i,j) = area of superindividual j
73        at depth i
74    double precision, dimension(nz, num_si),
75        intent(in) :: si_area
76    ! si_ind(i,j) = number of individuals
77        represented by superindividual j at depth
78        i
79    double precision, dimension(nz, num_si),
80        intent(in) :: si_ind
81    ! Thickness of each frond
82    double precision, intent(in) :::
83        frond_thickness
84    ! Ratio of length to width (0,infty)
85    double precision, intent(in) :::
86        frond_aspect_ratio
87    ! Rescaled position of greatest width (0=
88        base, 1=tip)
89    double precision, intent(in) :::
90        frond_shape_ratio
91
92
93
94

```

```

75 ! WATER CURRENT
76 double precision, dimension(nz), intent(in)
    :: current_speeds
77 double precision, dimension(nz), intent(in)
    :: current_angles
78
79 ! SPACING
80 double precision, intent(in) :: rope_spacing
81 double precision, dimension(nz), intent(in)
    :: depth_spacing
82 ! SOLVER PARAMETERS
83 integer, intent(in) :: num_scatters
84
85 ! FINAL RESULT
86 real, dimension(nz), intent(out) :: 
    avg_irrad, perceived_irrad
87
88 ! -----
89
90 double precision xmin, xmax, ymin, ymax,
    zmin, zmax
91 character(len=5), parameter :: fmtstr = 'E13
    .4'
92 !double precision, dimension(num_vsf) :: 
    vsf_angles, vsf_vals
93 double precision max_rad, decay
94 integer quadrature_degree
95
96 type(space_angle_grid) grid
97 type(optical_properties) iops
98 type(light_state) light
99 type(rope_state) rope
100 type(frond_shape) frond
101 type(boundary_condition) bc
102
103 double precision, dimension(:, allocatable
    :: pop_length_means, pop_length_stds
104 ! Number of fronds in each depth layer
105 double precision, dimension(:, allocatable
    :: num_fronds
106 double precision, dimension(:,:,:),
    allocatable :: p_kelp
107
108 write(*,*) 'Light calculation'
109
110 allocate(pop_length_means(nz))
111 allocate(pop_length_stds(nz))
112 allocate(num_fronds(nz))
113 allocate(p_kelp(nx, ny, nz))
114
115 xmin = -rope_spacing/2

```

```

116     xmax = rope_spacing/2
117
118     ymin = -rope_spacing/2
119     ymax = rope_spacing/2
120
121     zmin = 0.d0
122     zmax = sum(depth_spacing)
123
124     write(*,*) 'Grid'
125     call grid%set_bounds(xmin, xmax, ymin, ymax,
126                            zmin, zmax)
126     call grid%set_num(nx, ny, nz, ntheta, nphi)
127     call grid%init()
128 !call grid%set_uniform_spacing_from_num()
129     call grid%z%set_spacing_array(depth_spacing)
130
131     call rope%init(grid)
132
133     write(*,*) 'Rope'
134 ! Calculate kelp distribution
135     call calculate_length_dist_from_superinds( &
136          nz, &
137          num_si, &
138          si_area, &
139          si_ind, &
140          frond_aspect_ratio, &
141          num_fronds, &
142          pop_length_means, &
143          pop_length_stds)
144
145     rope%frond_lengths = pop_length_means
146     rope%frond_stds = pop_length_stds
147     rope%num_fronds = num_fronds
148     rope%water_speeds = current_speeds
149     rope%water_angles = current_angles
150
151     write(*,*) 'frond_lengths = ', rope%
152           frond_lengths
153     write(*,*) 'frond_stds = ', rope%frond_stds
154     write(*,*) 'num_fronds = ', rope%num_fronds
155     write(*,*) 'water_speeds = ', rope%
156           water_speeds
157     write(*,*) 'water_angles = ', rope%
158           water_angles
159
159     write(*,*) 'Frond'
160 ! INIT FROND
161     call frond%set_shape(frond_shape_ratio,
162                           frond_aspect_ratio, frond_thickness)
160 ! CALCULATE KELP
161     quadrature_degree = 5

```

```

162 |     call calculate_kelp_on_grid(grid, p_kelp,
163 |         frond, rope, quadrature_degree)
164 | ! INIT IOPS
165 | iops%num_vsf = num_vsf
166 | call iops%init(grid)
167 | write(*,*) 'IOPs'
168 | iops%abs_kelp = absorptance_kelp /
169 |     frond_thickness
170 | iops%abs_water = abs_water
171 | iops%scat = scat
172 |
173 | !write(*,*) 'iop init'
174 | !iops%vsf_angles = vsf_angles
175 | !iops%vsf_vals = vsf_vals
176 | call iops%load_vsf(vsf_file, fmtstr)
177 |
178 | ! load_vsf already calls calc_vsf_on_grid
179 | !call iops%calc_vsf_on_grid()
180 | call iops%calculate_coef_grids(p_kelp)
181 |
182 | !write(*,*) 'BC'
183 | max_rad = 1.d0 ! Doesn't matter because we'
184 |     ll rescale
185 | decay = 1.d0 ! Does matter, but maybe not
186 |     much. Determines drop-off from angle
187 | call bc%init(grid, solar_zenith,
188 |     solar_azimuthal, decay, max_rad)
189 | ! Rescale surface radiance to match surface
190 |     irradiance
191 | bc%bc_grid = bc%bc_grid * surface_irrad /
192 |     grid%angles%integrate_points(bc%bc_grid)
193 |
194 | write(*,*) 'bc'
195 | write(*,*) bc%bc_grid
196 |
197 | ! write(*,*) 'bc'
198 | ! do i=1, grid%y%num
199 |     write(*,'(10F15.3)') bc%bc_grid(i,:)
200 | ! end do
201 |
202 | call light%init_grid(grid)
203 |
204 | write(*,*) 'Scatter'
205 | call calculate_light_with_scattering(grid,
206 |     bc, iops, light%radiance, num_scatters)
207 |
208 | write(*,*) 'Irrad'
209 | call light%calculate_irradiance()
210 |
211 | ! Calculate output variables

```

```

204 |     call calculate_average_irradiance(grid,
205 |         light, avg_irrad)
206 |     call calculate_perceived_irradiance(grid,
207 |         p_kelp, &
208 |             perceived_irrad, light%irradiance)
209 |
210 |     !write(*,*) 'vsf_angles = ', iops%vsf_angles
211 |     !write(*,*) 'vsf_vals = ', iops%vsf_vals
212 |     !write(*,*) 'vsf_norm = ', grid%
213 |         integrate_angle_2d(iops%vsf(1,1,:,:))
214 |
215 |     ! write(*,*) 'abs_water = ', abs_water
216 |     ! write(*,*) 'scat_water = ', scat_water
217 |     write(*,*) 'kelp '
218 |     write(*,*) p_kelp(:,:, :)
219 |     write(*,*) 'ft =', frond%ft
220 |
221 |     write(*,*) 'irrad'
222 |     write(*,*) light%irradiance
223 |
224 |     write(*,*) 'deinit',
225 |     call bc%deinit()
226 |     !write(*,*) 'a'
227 |     call iops%deinit()
228 |     !write(*,*) 'b'
229 |     call light%deinit()
230 |     !write(*,*) 'c'
231 |     call rope%deinit()
232 |     !write(*,*) 'd'
233 |     call grid%deinit()
234 |     !write(*,*) 'e'
235 |
236 |     deallocate(pop_length_means)
237 |     deallocate(pop_length_stds)
238 |     deallocate(num_fronds)
239 |     deallocate(p_kelp)
240 |
241 |     !write(*,*) 'done'
242 end subroutine full_light_calculations
243
244 subroutine
245     calculate_length_dist_from_superinds( &
246     nz, &
247     num_si, &
248     si_area, &
249     si_ind, &

```

```

249   frond_aspect_ratio, &
250   num_fronds, &
251   pop_length_means, &
252   pop_length_stds)
253
254   implicit none
255
256   ! Number of depth levels
257   integer, intent(in) :: nz
258   ! Number of Superindividuals in each depth
259   ! level
260   integer, intent(in) :: num_si
261   ! si_area(i,j) = area of superindividual j
262   ! at depth i
263   double precision, dimension(nz, num_si),
264   ! intent(in) :: si_area
265   ! si_area(i,j) = number of individuals
266   ! represented by superindividual j at depth
267   ! i
268   double precision, dimension(nz, num_si),
269   ! intent(in) :: si_ind
270   double precision, intent(in) :::
271   ! frond_aspect_ratio
272
273   double precision, dimension(nz), intent(out)
274   ! :: num_fronds
275   ! Population mean area at each depth level
276   double precision, dimension(nz), intent(out)
277   ! :: pop_length_means
278   ! Population area standard deviation at each
279   ! depth level
280   double precision, dimension(nz), intent(out)
281   ! :: pop_length_stds
282
283   ! -----
284
285   integer i, k
286   ! Numerators for mean and std
287   double precision mean_num, std_num
288   ! Convert area to length
289   double precision, dimension(num_si) :::
290   ! si_length
291
292   do k=1, nz
293     mean_num = 0.d0
294     std_num = 0.d0
295     num_fronds(k) = 0
296
297     do i=1, num_si
298       si_length(i) = sqrt(2.d0*
299         frond_aspect_ratio*si_area(k,i))
300     mean_num = mean_num + si_length(i)
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
999

```

```

288         num_fronds(k) = num_fronds(k) + si_ind
289             (k,i)
290     end do
291     pop_length_means(k) = mean_num /
292         num_fronds(k)
293     do i=1, num_si
294         std_num = std_num + (si_length(i) -
295             pop_length_means(k))**2
296     end do
297     pop_length_stds(k) = std_num / (
298         num_fronds(k) - 1)
299 end do
300
301 end subroutine
302     calculate_length_dist_from_superinds
303
304 subroutine calculate_average_irradiance(grid,
305     light, avg_irrad)
306     type(space_angle_grid) grid
307     type(light_state) light
308     real, dimension(:) :: avg_irrad
309     integer k, nx, ny, nz
310
311     nx = grid%x%num
312     ny = grid%y%num
313     nz = grid%z%num
314
315     do k=1, nz
316         avg_irrad(k) = real(sum(light%irradiance
317             (:,:,k)) / nx / ny)
318     end do
319 end subroutine calculate_average_irradiance
320
321 subroutine calculate_perceived_irradiance(grid
322     , p_kelp, &
323         perceived_irrad, irradiance)
324     type(space_angle_grid) grid
325     double precision, dimension(:,:,:,:) :: p_kelp
326     real, dimension(:) :: perceived_irrad
327     double precision, dimension(:,:,:,:) ::
328         irradiance
329     double precision total_kelp
330     integer center_i1, center_i2, center_j1,
331         center_j2
332
333     integer k

```

```

329      ! Calculate the average irradiance
330      ! experienced over the frond.
331      ! Has same units as irradiance.
332      ! If no kelp, then just take the irradiance
333      ! at the center
334      ! of the grid.
335      do k=1, grid%z%num
336          total_kelp = sum(p_kelp(:,:,:,k))
337          if(total_kelp .eq. 0) then
338              center_i1 = int(ceiling(grid%x%num /
339                           2.d0))
340              center_j1 = int(ceiling(grid%y%num /
341                           2.d0))
342              ! For even grid, use average of center
343              ! two cells
344              ! For odd grid, just use center cell
345              if(mod(grid%x%num, 2) .eq. 0) then
346                  center_i2 = center_i1 + 1
347              else
348                  center_i2 = center_i1
349              end if
350              if(mod(grid%y%num, 2) .eq. 0) then
351                  center_j2 = center_j1 + 1
352              else
353                  center_j2 = center_j1
354              end if
355
356              ! Irradiance at the center of the grid
357              ! (at the rope)
358              perceived_irrad(k) = real(sum(
359                  irradiance( &
360                  center_i1:center_i2, &
361                  center_j1:center_j2, k)) &
362                  / ((center_i2-center_i1+1) * (
363                  center_j2-center_j1+1)))
364
365      else
366          ! Average irradiance weighted by kelp
367          ! distribution
368          perceived_irrad(k) = real( &
369              sum(p_kelp(:,:,:,k)*irradiance(:,:,:,
370                  k)) &
371                  / total_kelp)
372      end if
373  end do
374
375  end subroutine calculate_perceived_irradiance
376
377 end module light_interface

```