

©2018

OLIVER GRAHAM EVANS

ALL RIGHTS RESERVED

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Oliver Graham Evans

May, 2018

MODELING THE LIGHT FIELD IN MACROALGAE AQUACULTURE

Oliver Graham Evans

Thesis

Approved:

Advisor
Dr. Kevin Kreider

Accepted:

Dean of the College
Dr. Linda Subich

Co-Advisor
Dr. Curtis Clemons

Dean of the Graduate School
Dr. Chand Midha

Faculty Reader
Dr. Gerald Young

Date

Department Chair
Dr. Kevin Kreider

ABSTRACT

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

ACKNOWLEDGEMENTS

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

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

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

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

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

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

TABLE OF CONTENTS

	Page
LIST OF TABLES	ix
LIST OF FIGURES	x
CHAPTER	
I. INTRODUCTION	1
1.1 Motivation	1
1.2 Background on Kelp Models	4
1.3 Background on Radiative Transfer	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	34
3.1 Optical Definitions	34
3.2 The Radiative Transfer Equation	38

3.3	Low-Scattering Approximation	41
IV.	NUMERICAL SOLUTION	46
4.1	Discrete Grid	46
4.2	Kelp Distribution	50
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	79
VI.	PRACTICAL CONSIDERATIONS	82
6.1	Parameter Values	82
6.2	Algorithm Parameters	86
6.3	Computational Resources	86
6.4	Rules of Thumb	93
6.5	Comparison to Other Light Models	106
VII.	CONCLUSION	111
	APPENDICES	118
	APPENDIX A. GRID DETAILS	119

APPENDIX B. SYNTHETIC DATA	123
APPENDIX C. MEMORY USAGE	125
APPENDIX D. RAY TRACING ALGORITHM	128
APPENDIX E. FORTRAN CODE	132

LIST OF TABLES

Table	Page
4.1 Breakdown of nonzero matrix elements by derivative case.	64
6.1 Parameter values.	84
6.2 Field measurement data of optical properties in the ocean [22]. 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

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.	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	Absorption coefficient profile in the limiting case of $v_w \gg 1$, $\sigma_l = 0$ and $\sigma_b = 0$ demonstrates the kite-shaped fronds. Note the high gradient near the inner edge of the frond.	30
2.12	Absorption coefficient profile for realistic values of v_w and σ_l , but with $\sigma_b = 0$. Note the large gradient at the origin. By continuing to increase the plotting resolution, the gradient would be found to be unbounded.	31
2.13	A small blur radius of $\sigma_b = 0.1$ m eliminates the derivative singularity at the origin.	32
2.14	Over-blurring kelp distribution with $\sigma_b = 10$ m causes it to lose a great deal of spatial resolution. This should be avoided.	33
4.1	Spatial grid. Discrete quantities are calculated at grid cell centers.	48
4.2	Angular grid at each point in space with poles treated separately.	49
5.1	Code verification for the finite difference solution via the Method of Manufactured Solutions. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. The average absolute difference between the analytical and numerical solutions is shown. A slope of $m = 2$ on a log-log scale demonstrates second order convergence, as expected. This demonstrates the correctness of the code.	76
5.2	mms asym n contourf	77
5.3	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. Ideally, the n -th order approximation should demonstrate $n + 1$ -th order convergence as $b \rightarrow 0$. The first three approximations are reasonably close, but the $n = 3$ approximation converges slower than expected. It is unclear whether this is due to a coding mistake, discretization error, or if there is another cause for the sub-optimal convergence.	78
6.1	mms asym err time, $n_s = 64$, $n_a = 8$	87
6.2	err time collapsed, $n_s = 64$, $n_a = 8$. $\varepsilon t^2 \propto b^n$	88
6.3	mms asym err time, $n_s = 64$, $n_a = 8$	89
6.4	mms asym err time	90

6.5	Memory to store one copy of the finite difference coefficient matrix	91
6.6	Memory to solve the linear system of equations with GMRES restarted every 100 iterations. This seems to require roughly five times the memory required to store the matrix.	92
6.7	Asym real kelp. Blur radius=0.1	93
6.8	asym err vs b a01	94
6.9	asym err vs ab	95
6.10	best n data vs ab	96
6.11	min n data vs ab eps01	97
6.12	min n data vs ab eps005	98
6.13	min n data vs ab eps001	99
6.14	asym err data xi model	100
6.15	asym err vs x all n fit all	101
6.16	asym err vs xi all n fit all	102
6.17	asym err model vs ab all k	103
6.18	nbar model vs ab 3eps	104
6.19	Asym real kelp. Blur radius=0.1	105
6.20	Compare models n=1	106
6.21	Compare models n=2	107
6.22	Compare models n=3	107
6.23	Compare models n=1	108
6.24	Compare models n=2	108
6.25	Compare models n=3	109

6.26	Compare models n=1	109
6.27	Compare models n=2	110
6.28	Compare models n=3	110
A.1	Angular grid	121
C.1	Memory to store one copy of the finite difference coefficient matrix . . .	126
C.2	Memory to solve the linear system of equations with GMRES restarted every 100 iterations. This seems to require roughly five times the memory required to store the matrix.	127

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 [15, 25]. 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) [5, 8, 11, 12].

Furthermore, seaweed cultivation has been proposed as a nutrient remediation technique for natural waters [16]. 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 [30]. 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 [4] for *S. latissima* has been produced and integrated into the SINMOD [29] 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 [1]. In his model, he accounts for solar radiation intensity as a function of time of year and time of day, and refraction on the surface of the water. He uses a simple model for shading, 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* [14], 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 [6]. 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. [20] 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 [10, 31]. 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 [4, 3, 12] 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 [7, 21]; its application to marine biology is fairly recent [18]. 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 ?? AND ??. 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 other 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. In the following sections, it is necessary to convert between Cartesian

and spherical coordinates, which we do using the relations

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

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

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

Then, calculating derivatives from (2.1) yields

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

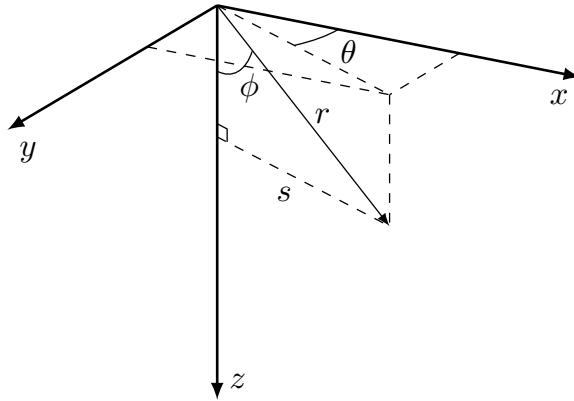


Figure 2.3: Downward-facing right-handed coordinate system with radial distance r from the origin, distance s from the z axis, zenith angle ϕ and azimuthal angle θ .

2.3 Population Distributions

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

2.3.1 Frond Shape

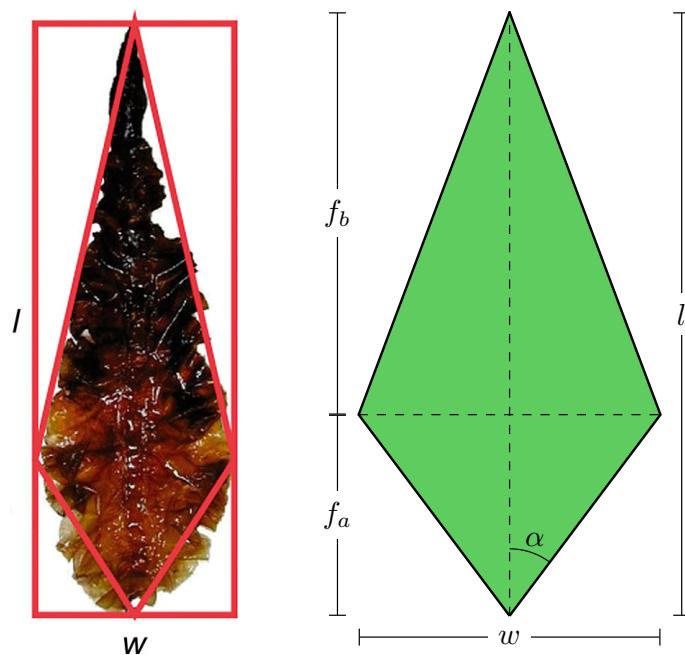


Figure 2.4: Simplified kite-shaped frond.

The frond is assumed to be kite-shaped with length l from base to tip, and width w from left to right. In Figure 2.4, the base is shown at the bottom and the tip is shown at the top. The proximal length is the shortest distance from the base to the diagonal connecting the left and right corners, and is notated as f_a . Likewise,

the distal length, 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

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

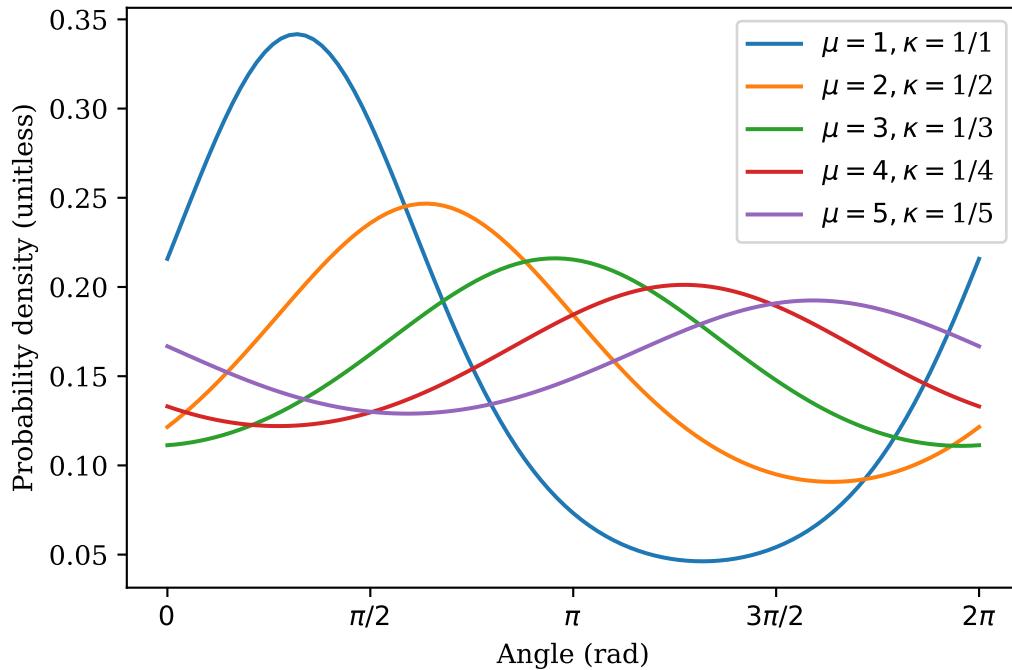


Figure 2.5: von Mises distribution for a variety of parameters.

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

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

which is currently not taken into consideration in this model.

2.4 Spatial Distribution

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

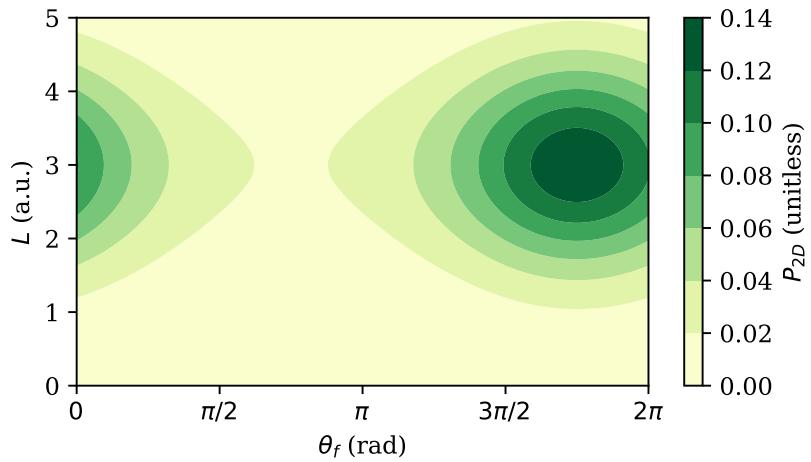


Figure 2.6: 2D length-angle probability distribution with $\theta_w = 7\pi/4$, $v_w = 1$, $\mu_l = 3$, $\sigma_l = 1$.

2.4.1 Rotated Coordinate System

To determine under what conditions a frond will occupy a given point, we begin by describing the shape of the frond in Cartesian coordinates and then convert to polar coordinates. Of primary interest are the edges connected to the frond tip. For convenience, we will use a rotated polar coordinate system (θ', s) such that the line connecting the base to the tip points in the $+y$ direction ($\theta = \pi/2$), with the base at $(0, 0)$. Denote the Cartesian analogue of this coordinate system as (x', y') which

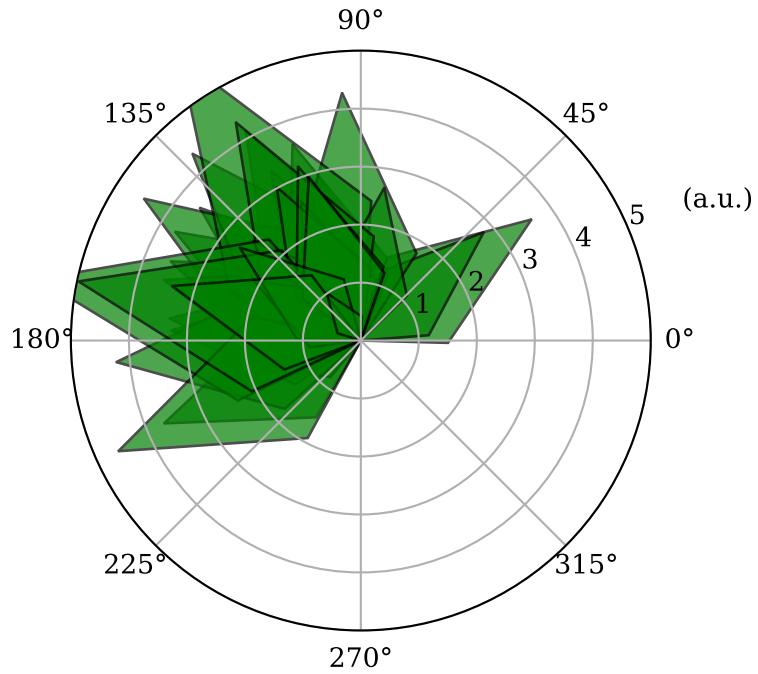


Figure 2.7: A sample of 50 kelp fronds with shape parameters $f_s = 0.5$ and $f_r = 2$ whose lengths are picked from a normal distribution and whose angles are picked from a von Mises distribution.

is related to (θ', s) by

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

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

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

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

2.4.2 Functional Description of Frond Edge

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

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

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

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

where

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

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

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

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.

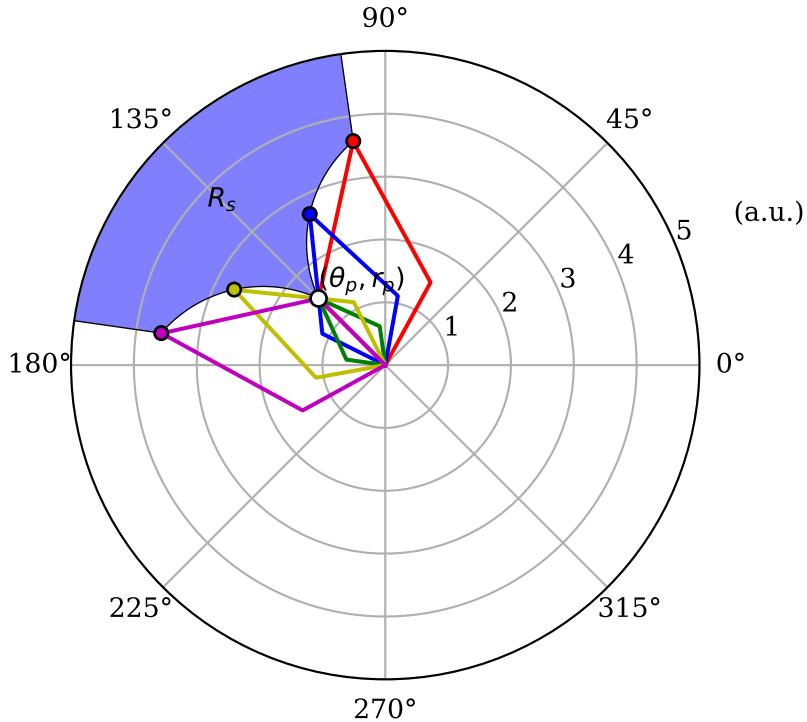


Figure 2.8: Outlines of minimum-length fronds for a variety of angles to occupy the point $(\theta, s) = (3\pi/4, 3/2)$.

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.

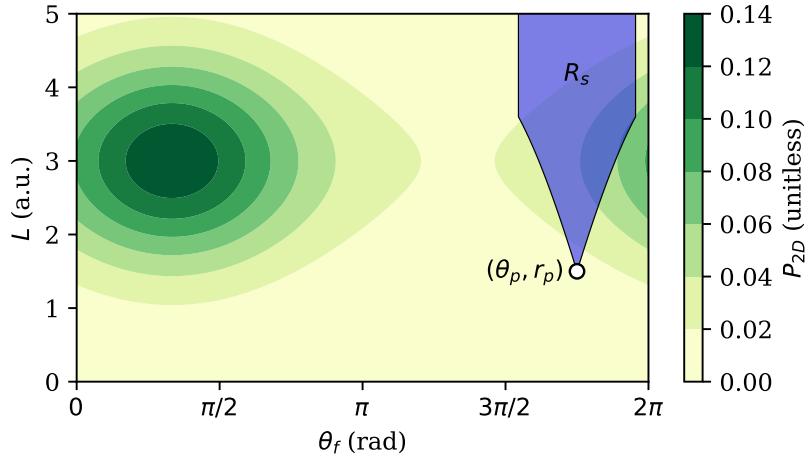


Figure 2.9: Contour plot of $P_{2D}(\theta_f, l)$ overlayed with the region in the θ_f-l plane which results in a frond occupying the point $(\theta, s) = (3\pi/4, 3/2)$.

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) ,

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

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

$$P_k(x, y) = \frac{nt}{dz} \tilde{P}_k(x, y).$$

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 $\rho_n = n/dz$ over the vertical length of the rope with units m^{-1} , in which case

$$P_k(x, y, z) = t\rho_n(z)\tilde{P}_k(x, y, z).$$

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

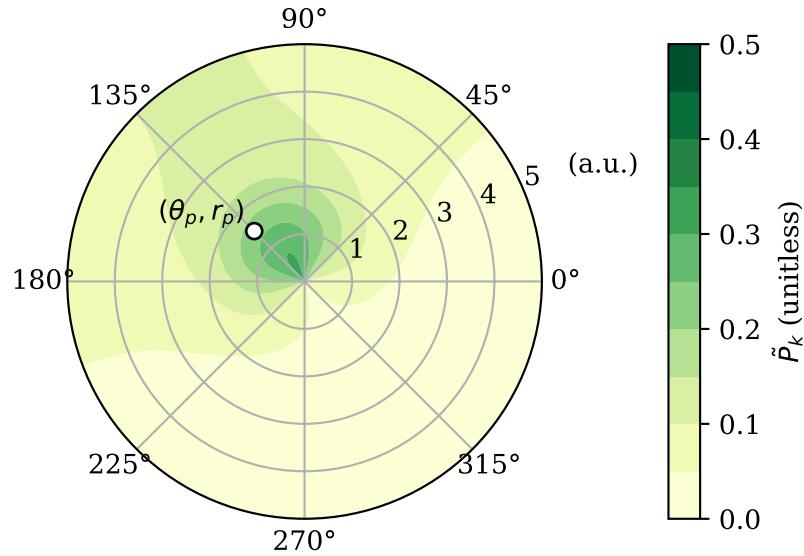


Figure 2.10: Contour plot of the probability of frond occupation sampled at 121 points using $\theta_f = 2\pi/3$, $\eta v_w = 1$.

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

$$k(x; \sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(-\frac{x^2}{2\sigma_b^2}\right). \quad (2.12)$$

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 two-dimensional Gaussian blur in xy -plane to P_k using the kernel

$$k(x, y; \sigma_b) = \frac{1}{\sqrt{2\pi\sigma_b^2}} \exp\left(\frac{x^2 + y^2}{2\sigma_b^2}\right). \quad (2.13)$$

Hence, the blurred solution is

$$P_k^b(x, y, z) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} k(x - x', y - y', z; \sigma_b) P_k(x', y', z) dx' dy'. \quad (2.14)$$

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.

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 representing 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.

Figure 2.11 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.

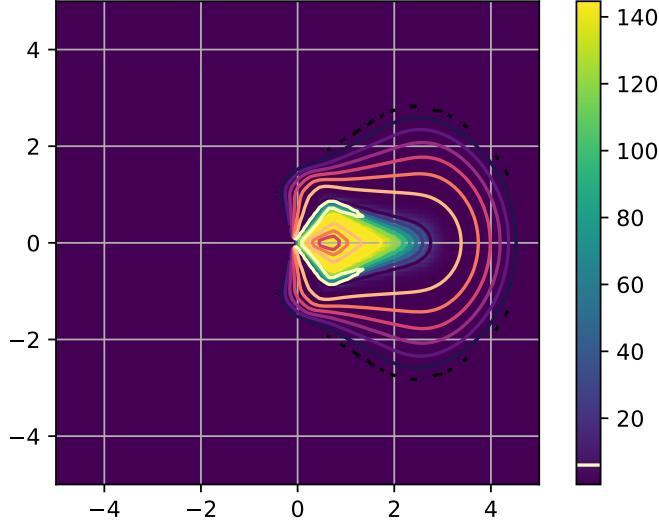


Figure 2.11: Absorption coefficient profile in the limiting case of $v_w \gg 1$, $\sigma_l = 0$ and $\sigma_b = 0$ demonstrates the kite-shaped fronds. Note the high gradient near the inner edge of the frond.

In Figure 2.12, 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 would still cause numerical difficulties for the numerical algorithm.

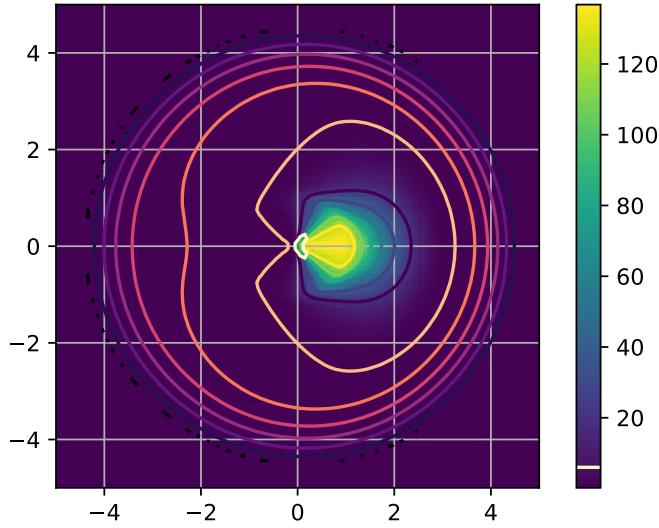


Figure 2.12: Absorption coefficient profile for realistic values of v_w and σ_l , but with $\sigma_b = 0$. Note the large gradient at the origin. By continuing to increase the plotting resolution, the gradient would be found to be unbounded.

Figure 2.13 is similar to Figure 2.12, except it has been post-processed with a small Gaussian blur of radius 10 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.

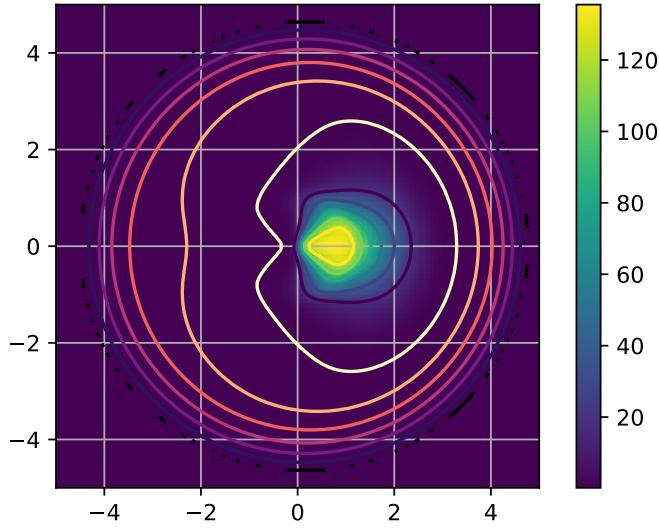


Figure 2.13: A small blur radius of $\sigma_b = 0.1$ m eliminates the derivative singularity at the origin.

In Figure 2.14, 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 2.13. Such a large blur is unlikely to reduce the discretization error, since other factors are contributing to it more significantly.

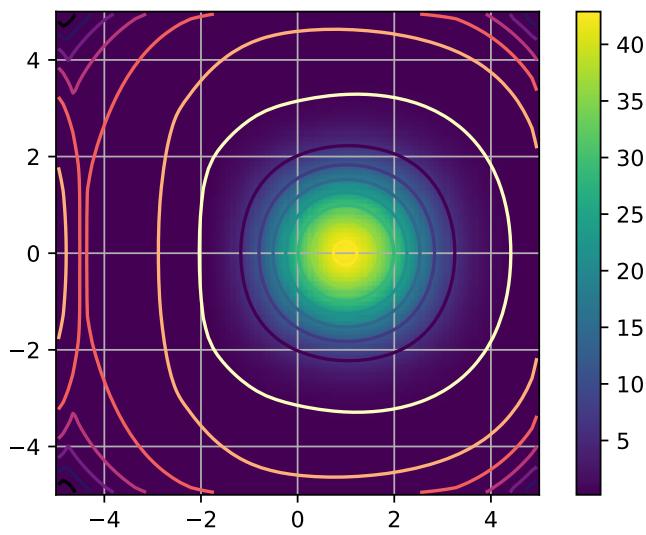


Figure 2.14: Over-blurring kelp distribution with $\sigma_b = 10$ m causes it to lose a great deal of spatial resolution. This should be avoided.

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. 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. 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 [17] 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 exists; this one was chosen for its relative simplicity.

The domain of the radiative transfer equation is embedded in five dimensions: three spatial (x , y , and z) and two angular (azimuthal θ and polar ϕ). The number

of grid cells in each dimension are denoted n_x , n_y , n_z , n_θ , and n_ϕ , with uniform spacings dx , dy , dz , $d\theta$, and $d\phi$ between adjacent grid points.

The following indices are assigned to each dimension:

$$x \rightarrow i,$$

$$y \rightarrow j,$$

$$z \rightarrow k,$$

$$\theta \rightarrow l,$$

$$\phi \rightarrow m.$$

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

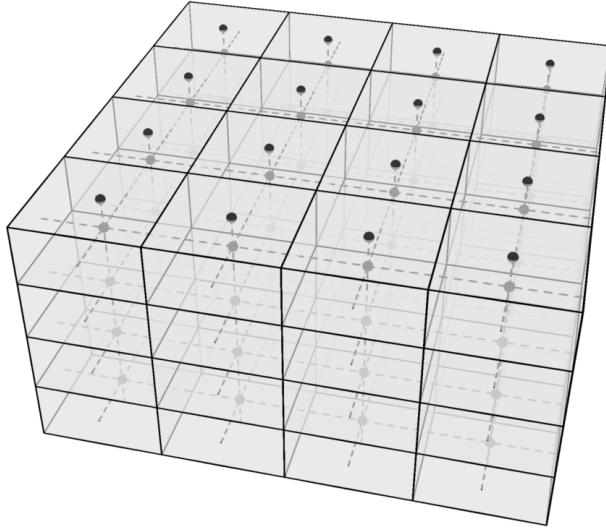


Figure 4.1: Spatial grid. Discrete quantities are calculated at grid cell centers.

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.

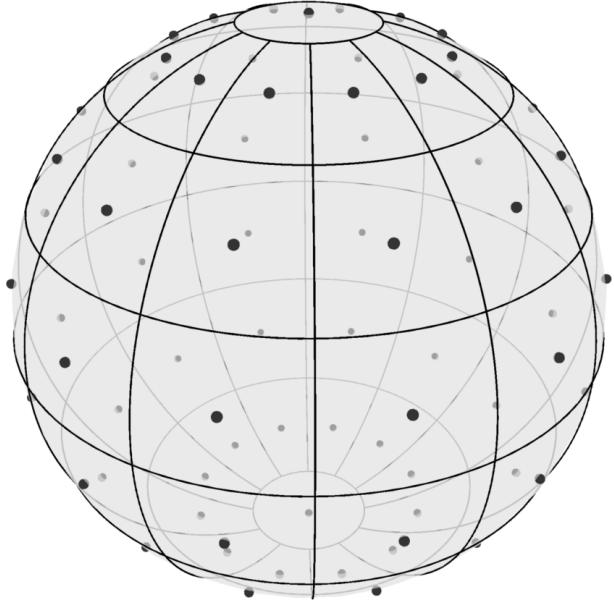


Figure 4.2: Angular grid at each point in space with poles treated separately.

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, 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 [24]. 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 Distributions

The computational crux of calculating the kelp distribution is to integrate $P_{2D}(\theta_f, l)$ 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 builtin 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

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

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

*TODO: numerical blurring details

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 D.

4.3.1 Spatial Quadrature

Define the *spatial characteristic function* as

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

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

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

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

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

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

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

$$\int_{4\pi} \beta(\boldsymbol{\omega}_p \cdot \boldsymbol{\omega}_{p'}) L(\boldsymbol{x}, \boldsymbol{\omega}') d\boldsymbol{\omega} \approx \sum_{p=1}^{n_\omega} \beta_{pp'} |\Omega'| L_{ijkp}. \quad (4.4)$$

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 D. 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

$$u_n(\tilde{s}) = \sum_{\nu=1}^{N-1} \tilde{g}_{n\nu} d_\nu e^{b_\nu}. \quad (4.5)$$

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

$$u_0(\tilde{s}) = f(\boldsymbol{\omega})H(\boldsymbol{\omega} \cdot \hat{z}) \exp\left(-\sum_{j=1}^{N-1} \tilde{a}_j ds_j\right) + \sum_{\nu=1}^{N-1} \tilde{\sigma}_{0\nu} d_\nu e^{b_\nu}. \quad (4.6)$$

4.5 Finite Difference

While the asymptotic solution is valid in the case of low scattering, a more general solution is obtained via finite difference, whereby the derivatives and integrals in the integro-partial differential equation are discretized to differences and sums and evaluated at each grid cell in order to construct a linear system of equations whose solution approximates that of the analytical equation. The price of a general solution, however, is greatly increased computational cost, both in terms of memory and CPU usage.

4.5.1 Discretization

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

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

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

difference is given by

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

and the backward difference by

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

For the upper and lower boundaries, we need an asymmetric finite difference method. In general, the Taylor Series of a function f about x is

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

Truncating after the first few terms, we have

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

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

$$f(x - \frac{\varepsilon}{2}) = f(x) - \frac{f'(x)\varepsilon}{2} + \frac{f''(x)\varepsilon^2}{8} + \mathcal{O}(\varepsilon^3). \quad (4.8)$$

Rearranging (4.7) produces

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

Combining (4.8) with (4.9) gives

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

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

$$f'(x) = \frac{-4f(x - \frac{\varepsilon}{2}) + 3f(x) + f(x + \varepsilon)}{3\varepsilon} + \mathcal{O}(\varepsilon^2). \quad (4.10)$$

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

$$f'(x) = \frac{-f(x - \varepsilon) - 3f(x) + 4f(x + \frac{\varepsilon}{2})}{3\varepsilon} + \mathcal{O}(\varepsilon^2). \quad (4.11)$$

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

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

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 (4.8) 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

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

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 [23], LGMRES [2], IDR [27], and BI-CGSTAB [28].

For the code in this thesis, a parallel implementation of GMRES provided by a Fortran package called LIS (Library of Iterative Solvers) is employed. LIS provides several execution environments: multithreading, multiprocessing, and MPI (Message Passing Interface). 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.

On the other hand, 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.

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 of specific results are surveyed. The Fortran implementations of the finite difference and asymptotic solutions of the radiative transfer equation are verified 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 model outputs and the real behavior. Such differences can be termed *conceptual modelling 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*.

5.2 Verification and Validation

As Roache explains, there are two aspects to building confidence in numerical codes, specifically those which solve PDEs: verification deals with *solving the equations right*, while validation deals with *solving the right equations*. Validation involves comparison with experimental evidence in order to determine that governing equations accurately describe physical phenomenon, and that their solutions match observation. That is, it is the process of checking that conceptual modelling errors are sufficiently small to model the system as accurately as necessary. Validation is an ongoing process; as new experimental data becomes available, the equations can be solved under appropriate conditions in an attempt to replicate the new observations. 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 forms of verification. The first is *code verification*, where the overall implementation of an algorithm is tested, and the the difference between the numerical and analytical solutions is *explicitly measured* at every point in the numerical solution. The same calculation is repeated for several grid sizes, and it is checked that the convergence order as the grid spacing approaches zero matches the theoretical convergence order of the algorithm. The explicit measurement of errors requires that the analytical solution be known, which is generally only possible for some unrealistic or uninteresting set of parameters. If the 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 single specific calculation 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. Error estimation for a specific calculation can be used to calculate error bars to display on a plot to represent discretization error in the solution.

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 $u = f$, 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 the asymptotic region can be achieved for reasonable grid sizes. Since these functions may need to be fairly complicated in order to achieve full generality while meeting the necessary constraints, it is advisable to use a *Computer Algebra System* (CAS) to symbolically compute the source term.

5.3.1 Synthetic Data

In order to verify the numerical implementations described in the previous chapters, it is necessary to first choose a manufactured solution $L(\mathbf{x}, \boldsymbol{\omega})$ to the radiative transfer equation, as well as coefficient functions for the absorption coefficient $a(\mathbf{x})$ and volume scattering function $\beta(\boldsymbol{\omega} \cdot \boldsymbol{\omega}')$. 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 B.

5.3.2 Finite Difference

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

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

resolution, as

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

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

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 works without order-of-accuracy mistakes.

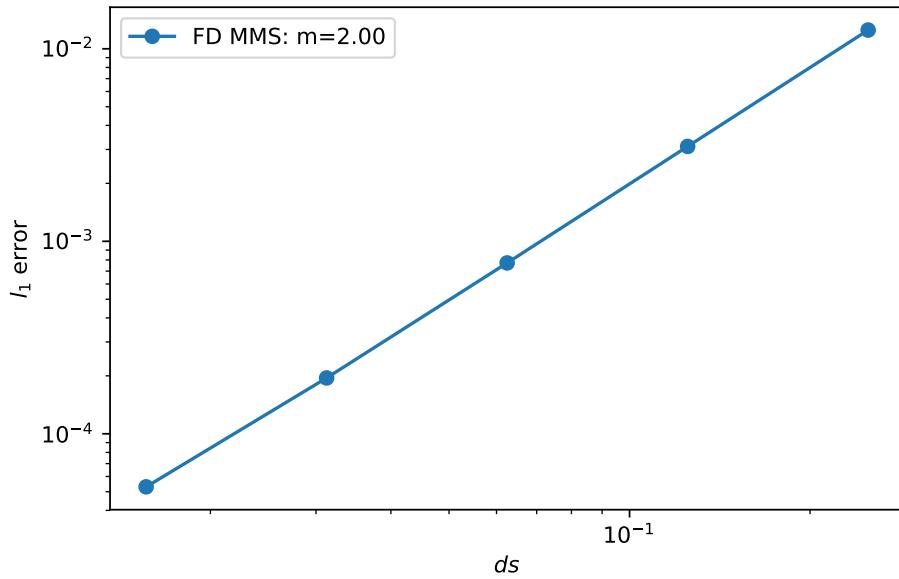


Figure 5.1: Code verification for the finite difference solution via the Method of Manufactured Solutions. Each point represents the same simulation run with a different spatial grid sizes, with the angular grid held constant at $n_a = 8$. The average absolute difference between the analytical and numerical solutions is shown. A slope of $m = 2$ on a log-log scale demonstrates second order convergence, as expected. This demonstrates the correctness of the code.

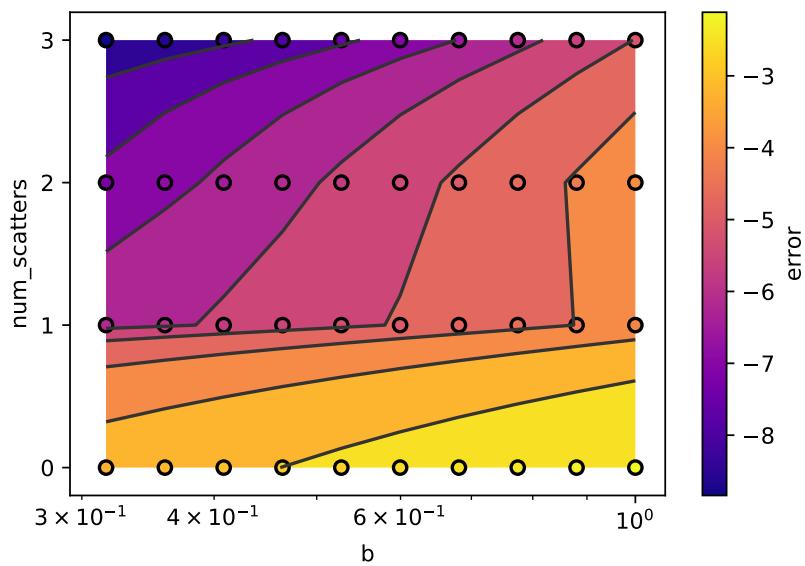


Figure 5.2: mms asym n contourf

5.3.3 Numerical Asymptotics

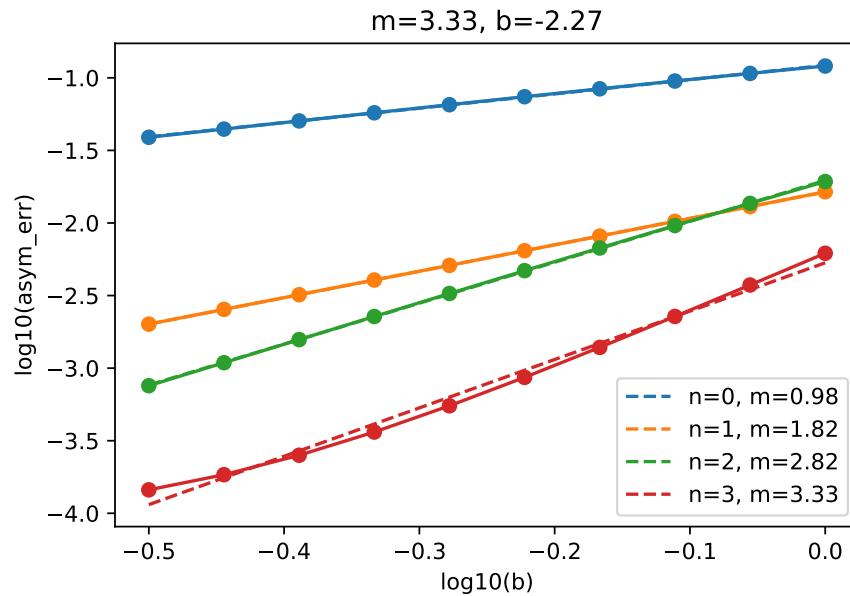


Figure 5.3: Code verification for the numerical asymptotics solution via the Method of Manufactured Solutions. A range of b -values are run, using 0 - 3 terms in the asymptotic series. Ideally, the n -th order approximation should demonstrate $n + 1$ -th order convergence as $b \rightarrow 0$. The first three approximations are reasonably close, but the $n = 3$ approximation converges slower than expected. It is unclear whether this is due to a coding mistake, discretization error, or if there is another cause for the sub-optimal convergence.

5.4 Verification of Calculations

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 stresses on a dam, and is also known as h^2 extrapolation since it was originally applied to a second order method.

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

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

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

Then,

$$\begin{aligned}\phi_1 &= \phi_e + g_2 h_1^2 + O(h^3), \\ \phi_2 &= \phi_e + g_2 h_2^2 + O(h^3).\end{aligned}$$

Solving for g_2 yields

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

so

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

Hence, the approximate continuum solution is

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

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

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 [32] or Ridge [13] 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 CONSIDERATIONS

6.1 Parameter Values

In this chapter, model parameters are discussed. In the case that this model is run in conjunction with a kelp growth model and ocean model, they will provide some of the necessary parameters. Other parameters not coming from the kelp or ocean model can be found in the literature, summarized in Table 6.1 and Table 6.2. Still, some parameters remain which are not well described in the literature.

6.1.1 Simulation Parameters

It is assumed that this model is run together with a kelp growth model such as described in [4], and an ocean model, as in [29]. Both models are assumed to use the same spatial grid, with n_z discrete depth layers of thickness dz_k for $k = 1, \dots, n_z$. It is assumed that the horizontal spacing for both models is quite large, and the light model therefore uses a much finer horizontal resolution, but retains the same vertical resolution as the encompassing calculations. The ocean model provides current speed and direction over depth, which is used in calculating the kelp distribution. The position of the sun and irradiance just below the surface of the water is also provided by the ocean model, which is used to generate the surface radiance boundary

condition. The ocean model should also provide an absorption coefficient for each depth layer, which may vary due to nutrient concentrations and biological specimens such as phytoplankton. The kelp model is expected to provide super-individual data describing the population in each depth layer. Then, (4.1) and (4.2) are used to calculate length and orientation distributions, as described in Section 4.2.1.

6.1.2 Parameters from Literature

Given here is a table of parameter values found in the literature which are used in Chapter V to test this light model. A few comments are in order. No values were available for the absorptance of *Saccharina latissima*, but a value for *Macrocystis pyrifera* was found. The surface irradiance from [4] was given in terms of photons per second, and was converted to W m^{-2} according to (3.1). No data in the literature exist for the frond thickness, so a best estimate is provided.

In [22], very detailed measurements of optical properties in various ocean waters are presented. A few of those measurements are reproduced here, using the same site names as in the original report. There are three categories of water provided: AUTEC is from Tongue of the Ocean, Bahama Islands, and represents very clear, pure water; HAOCE is from offshore southern California, and represents a more average coastal region, likely the most similar to water where kelp cultivation would occur; NUC data is from the San Diego Harbor, and represents very turbid water, likely more so than one would expect to find in a seaweed farm.

Table 6.1: Parameter values.

Parameter Name	Symbol	Value(s)	Citation
Kelp absorptance	A_k	0.8	[9]
Water absorption coefficient	a_w	See Table 6.2	[22]
Scattering coefficient	b	See Table 6.2	[22]
Volume scattering function	β	tabulated	[22, 26],
Frond thickness	t	0.4 mm	estimated
Surface solar irradiance	I_0	50 W m ⁻²	[4]

Table 6.2: Field measurement data of optical properties in the ocean [22]. 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

6.1.3 Frond Alignment Coefficient

The *frond alignment coefficient*, η , describes the dependence of frond alignment on current speed. To the author's knowledge, no such parameter is available in the literature. However, similar measurements have been made in the MACROSEA project by Norvik [19] 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 florescent dye could be applied to the tip of the each frond to make their orientation more easily discernable in the recording. Turn on the flume to several current speeds, recording a video or many snapshots for each. If the fluorescent dye is applied, then a simple peak-finding image processing algorithm can be applied to locate the frond tips. By preprocessing the image to a gray scale such that the color of the dye has the highest intensity, the tip locations are located at local maxima.

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

6.2 Algorithm Parameters

6.3 Computational Resources

6.3.1 CPU Time

32 cores

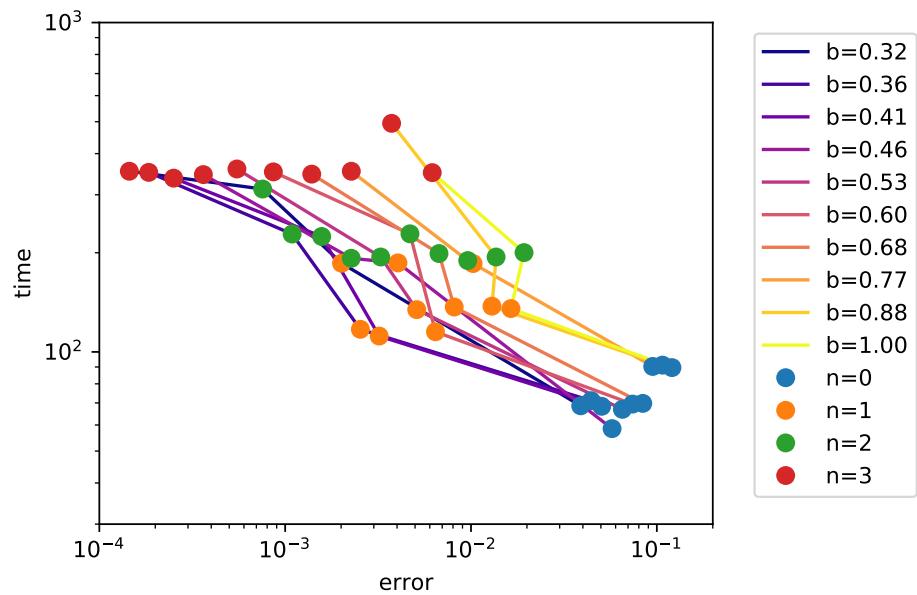


Figure 6.1: mms asym err time, $n_s = 64$, $n_a = 8$

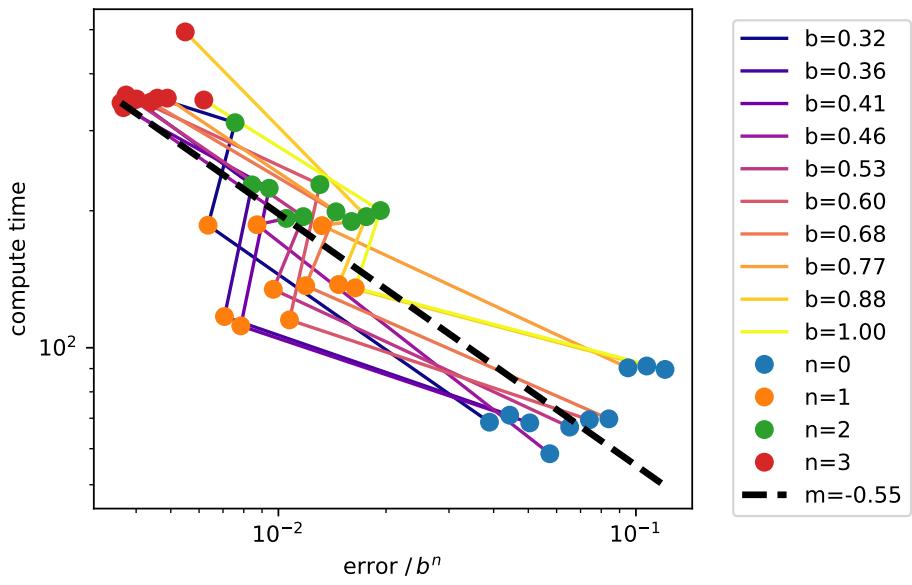


Figure 6.2: err time collapsed, $n_s = 64$, $n_a = 8$. $\varepsilon t^2 \propto b^n$

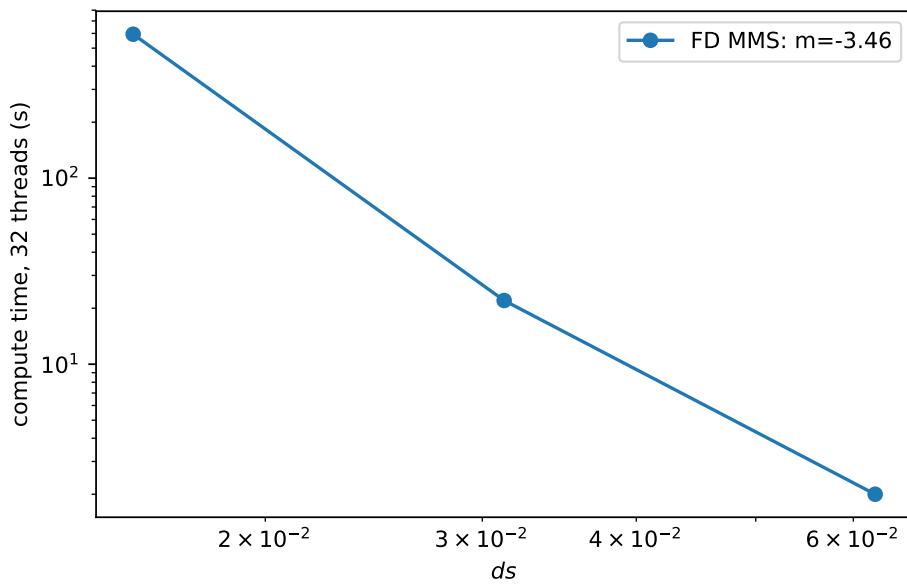


Figure 6.3: mms asym err time, $n_s = 64$, $n_a = 8$

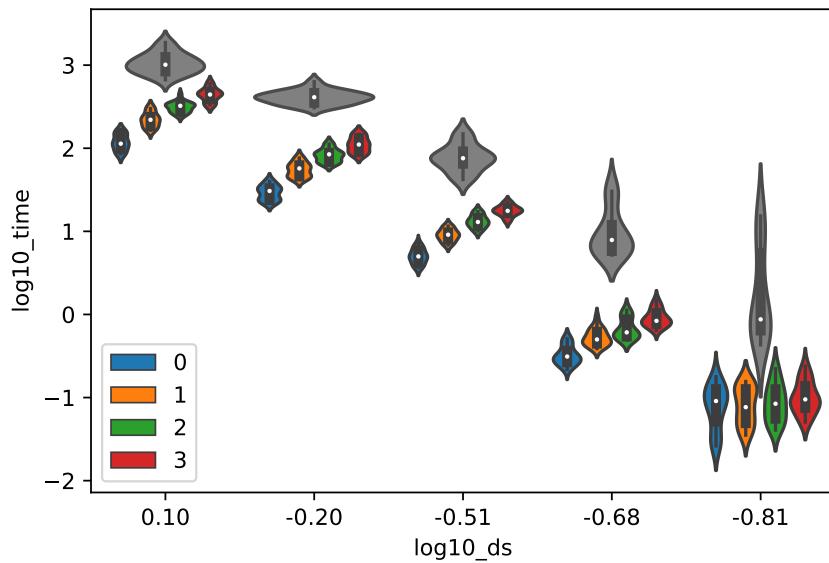


Figure 6.4: mms asym err time

6.3.2 Memory Usage

Single Matrix

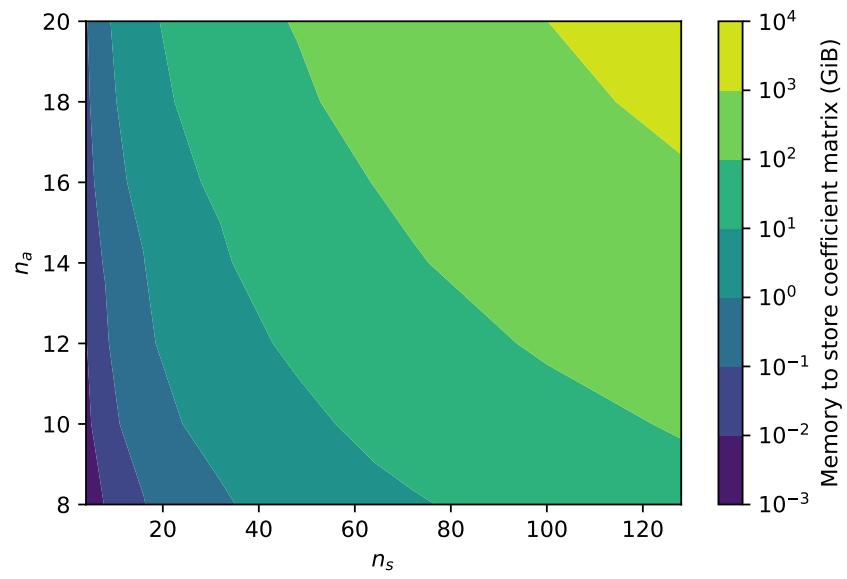


Figure 6.5: Memory to store one copy of the finite difference coefficient matrix

LIS Solution Memory Estimate

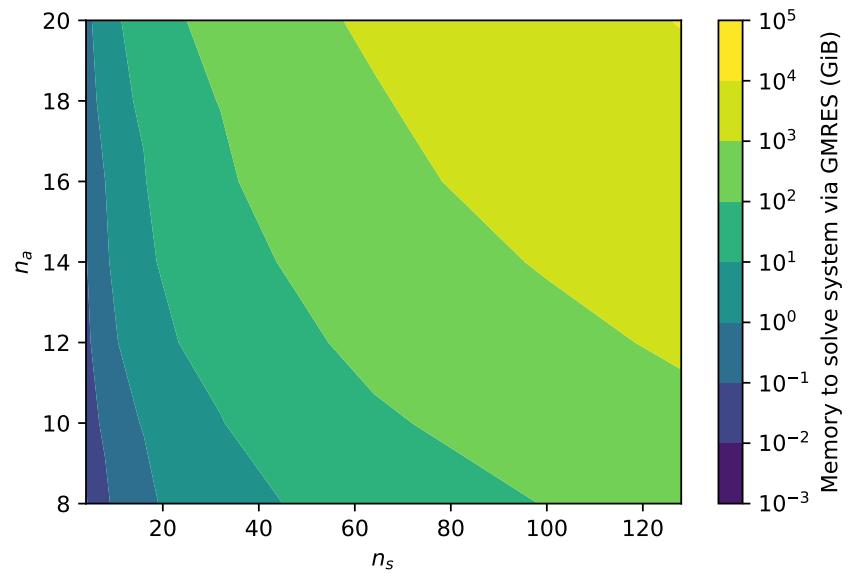


Figure 6.6: Memory to solve the linear system of equations with GMRES restarted every 100 iterations. This seems to require roughly five times the memory required to store the matrix.

6.4 Rules of Thumb

6.4.1 Grid Size and Discretization Error

6.4.2 Optical Conditions for Asymptotics

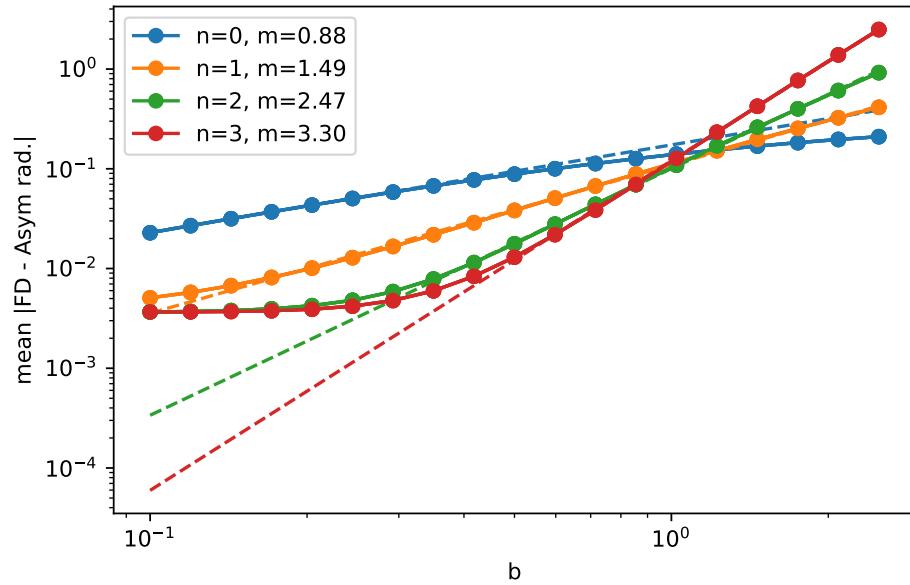


Figure 6.7: Asym real kelp. Blur radius=0.1

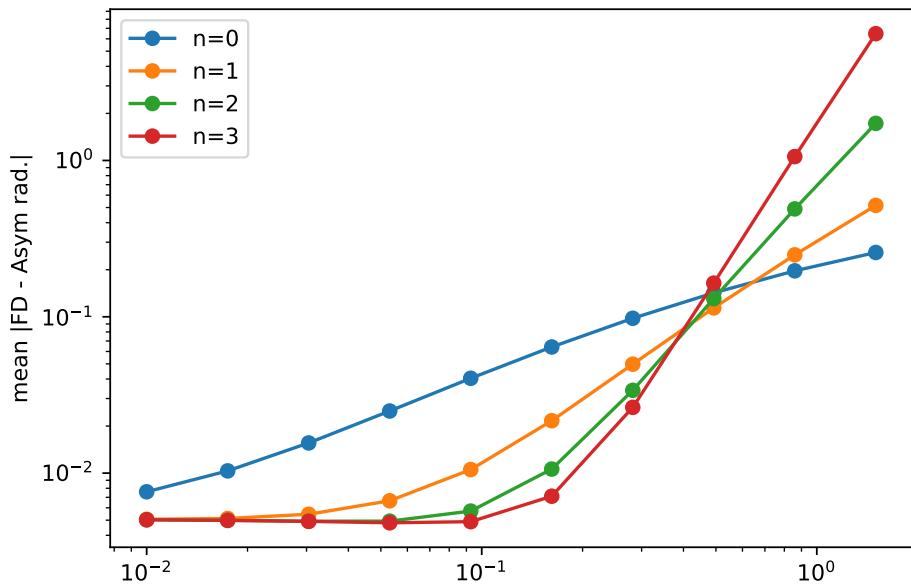


Figure 6.8: asym err vs b a01

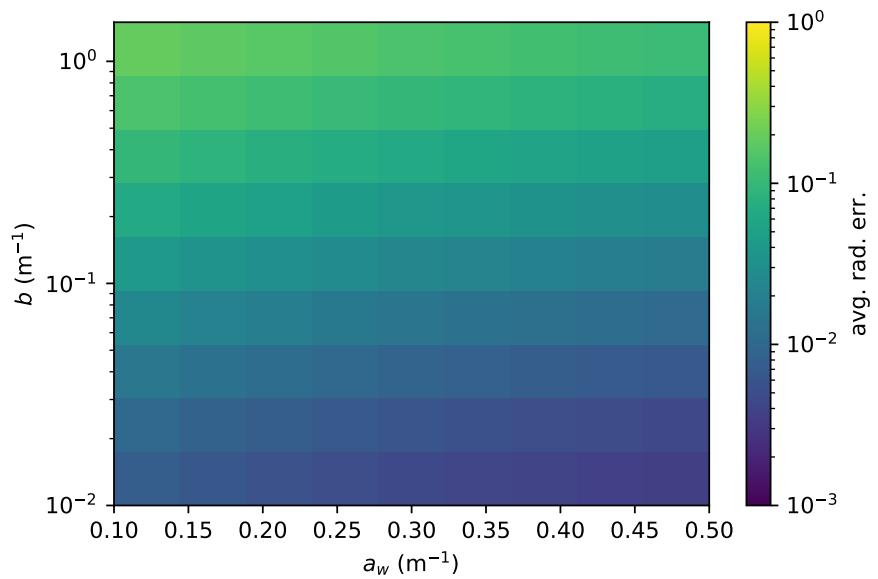


Figure 6.9: asym err vs ab

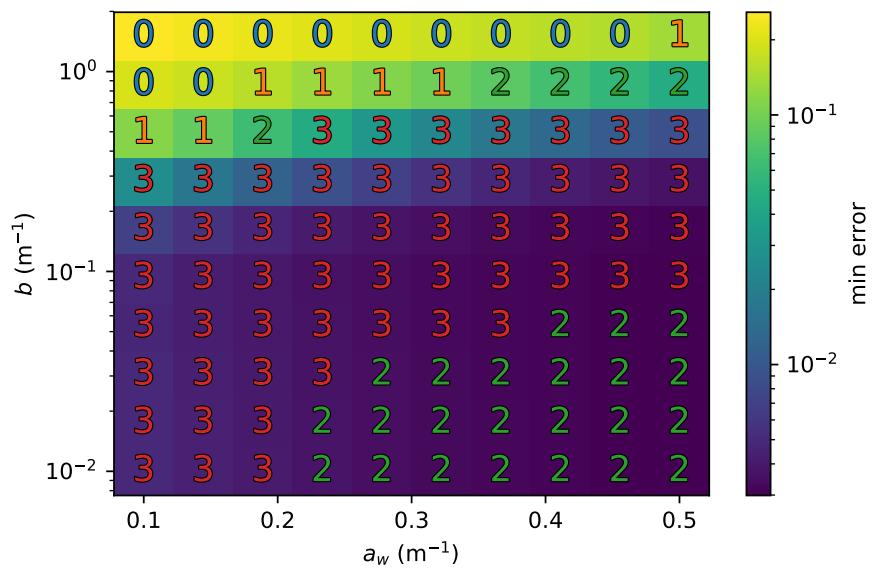


Figure 6.10: best n data vs ab

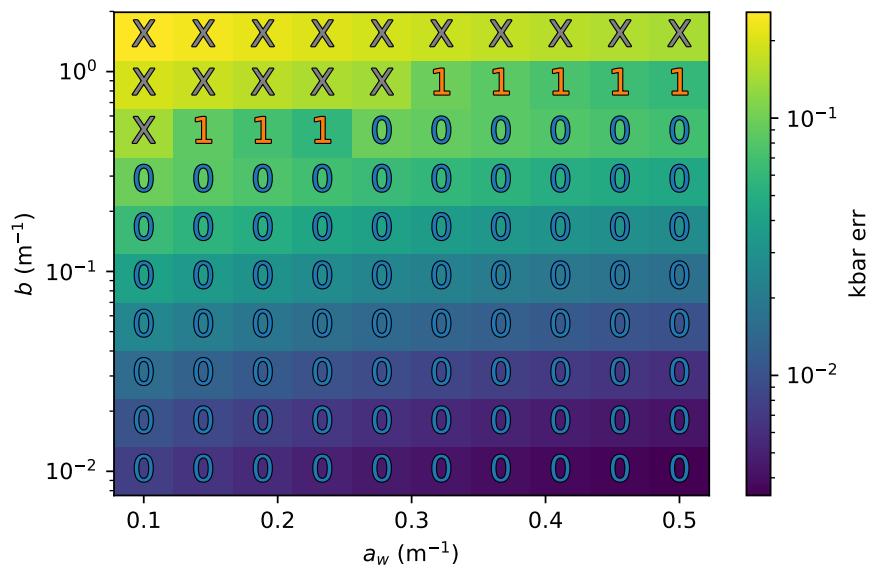


Figure 6.11: min n data vs ab eps01

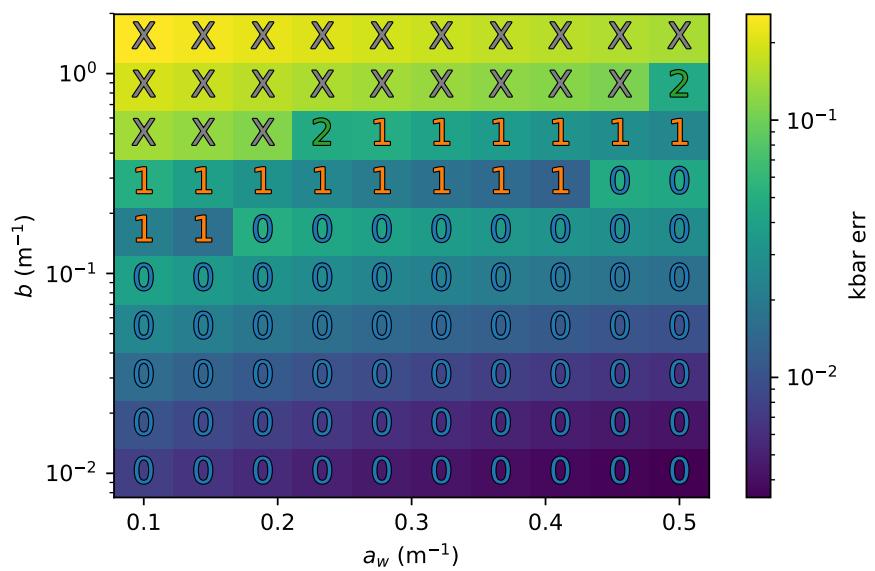


Figure 6.12: min n data vs ab eps005

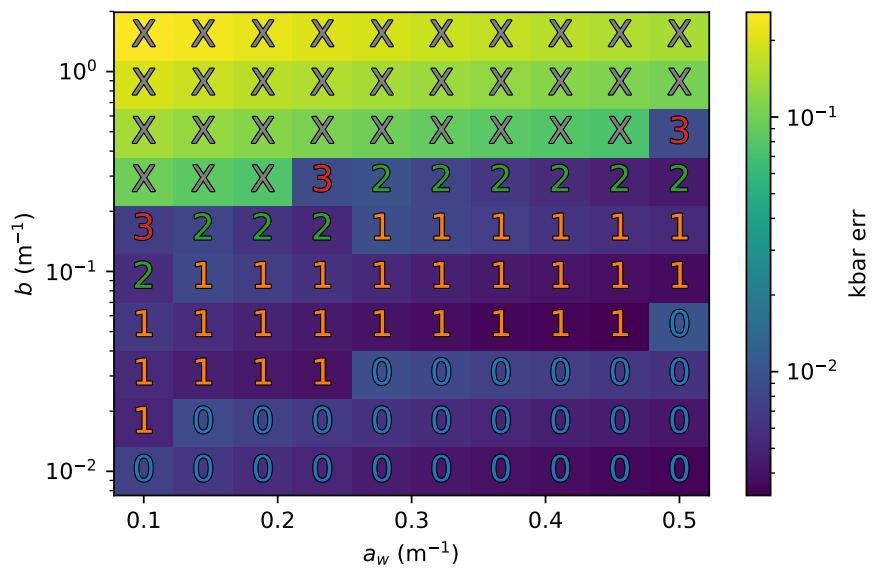


Figure 6.13: min n data vs ab eps001

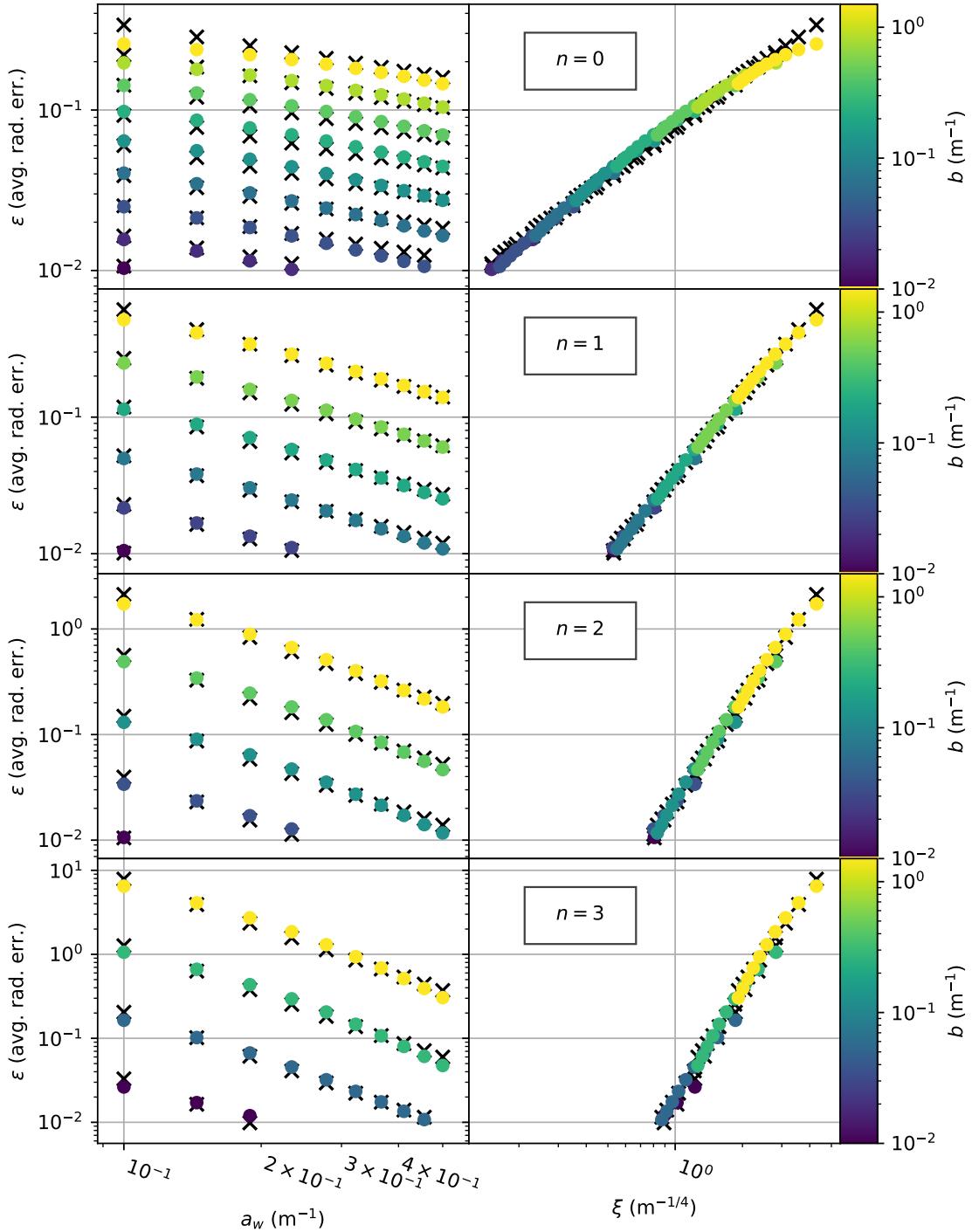


Figure 6.14: asym err data xi model

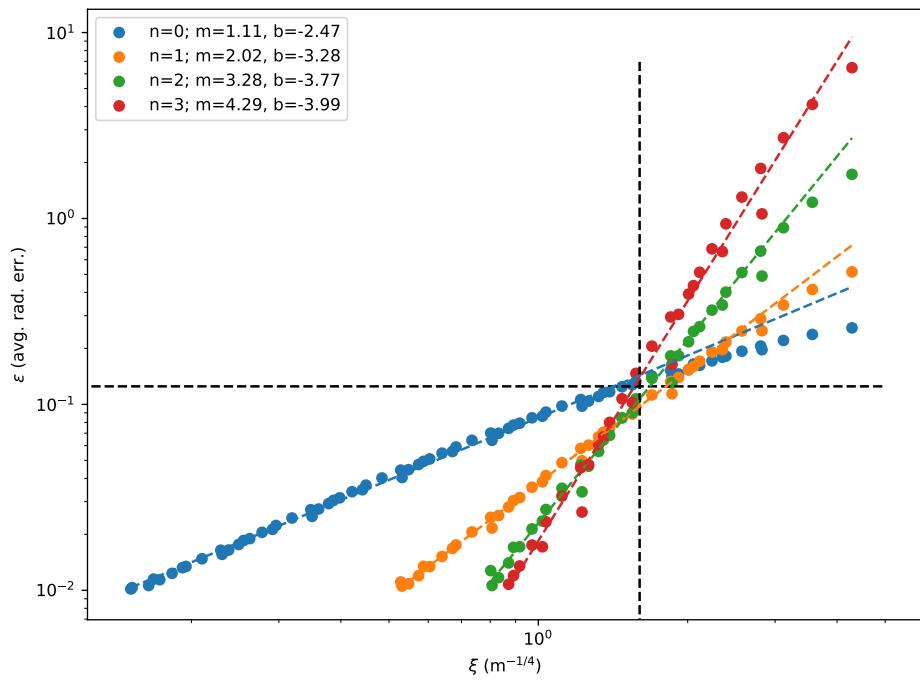


Figure 6.15: asym err vs x all n fit all

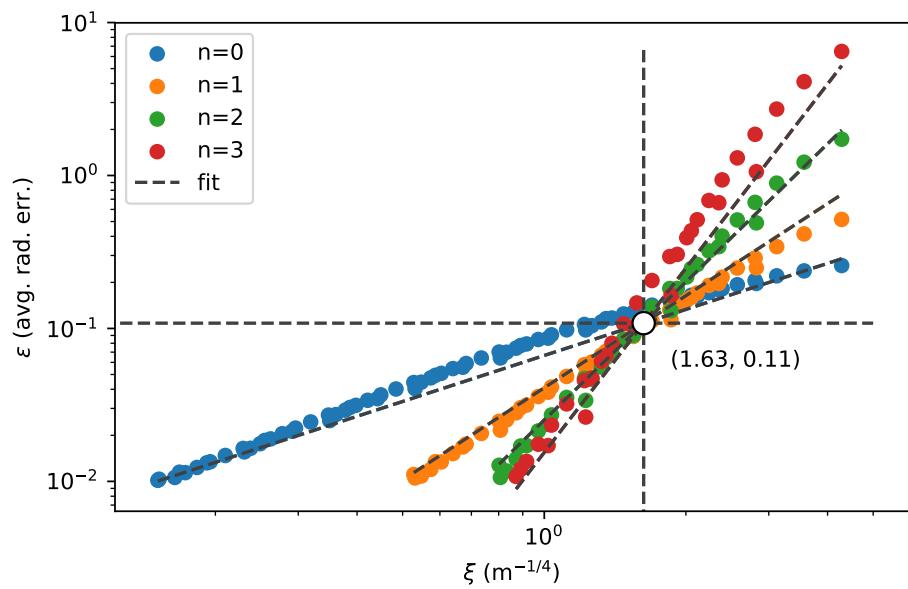


Figure 6.16: asym err vs xi all n fit all

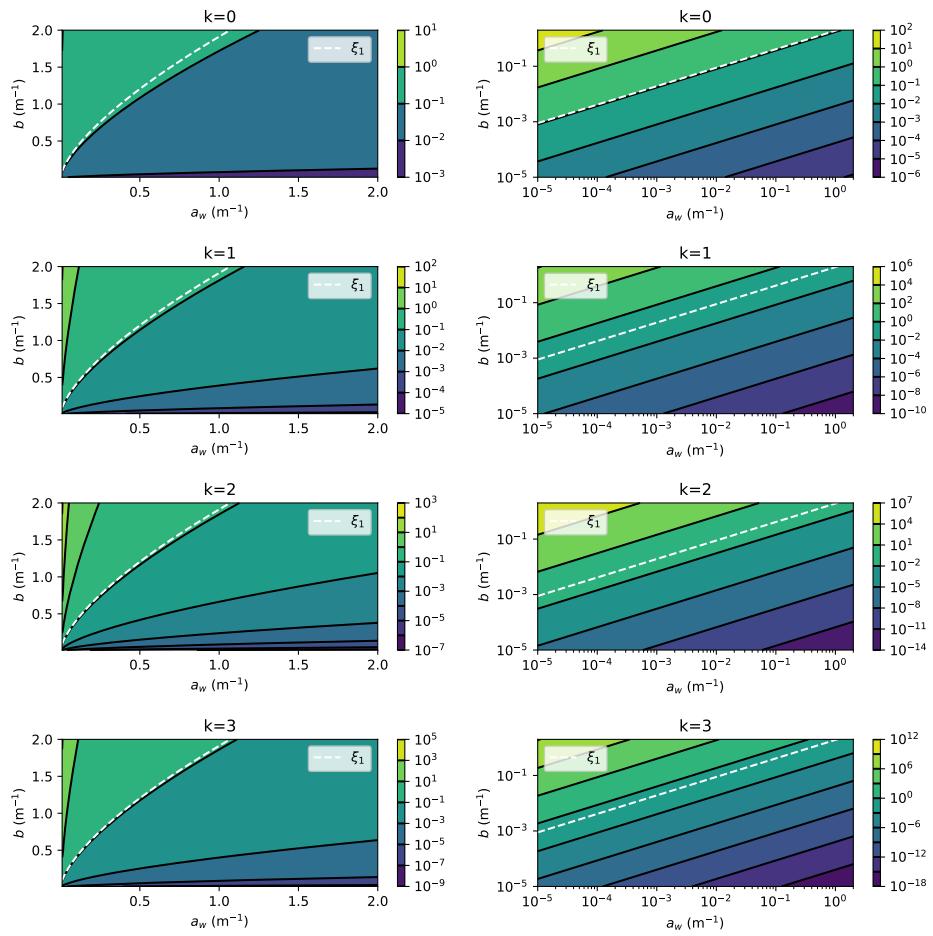


Figure 6.17: asym err model vs ab all k

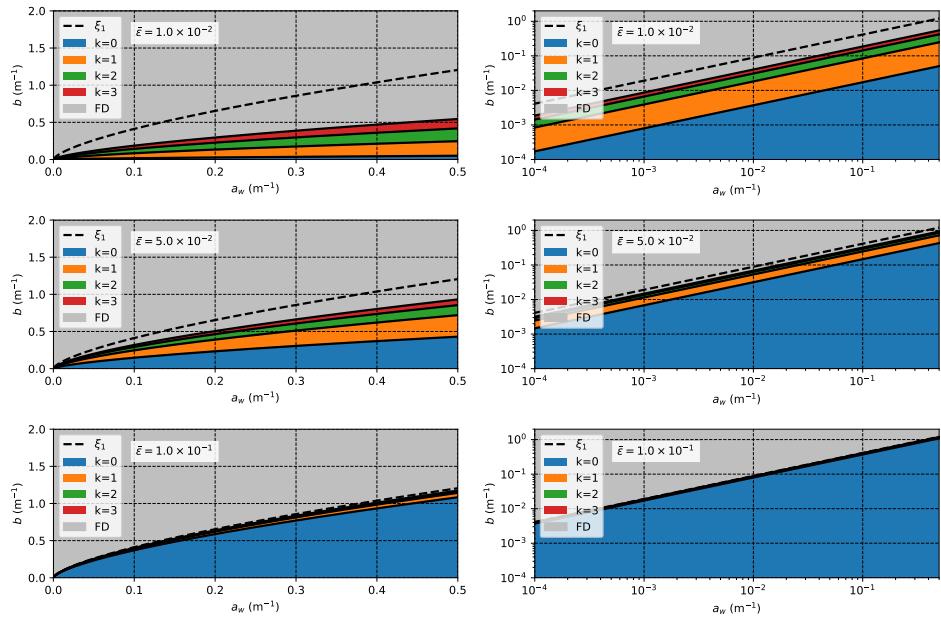


Figure 6.18: nbar model vs ab 3eps

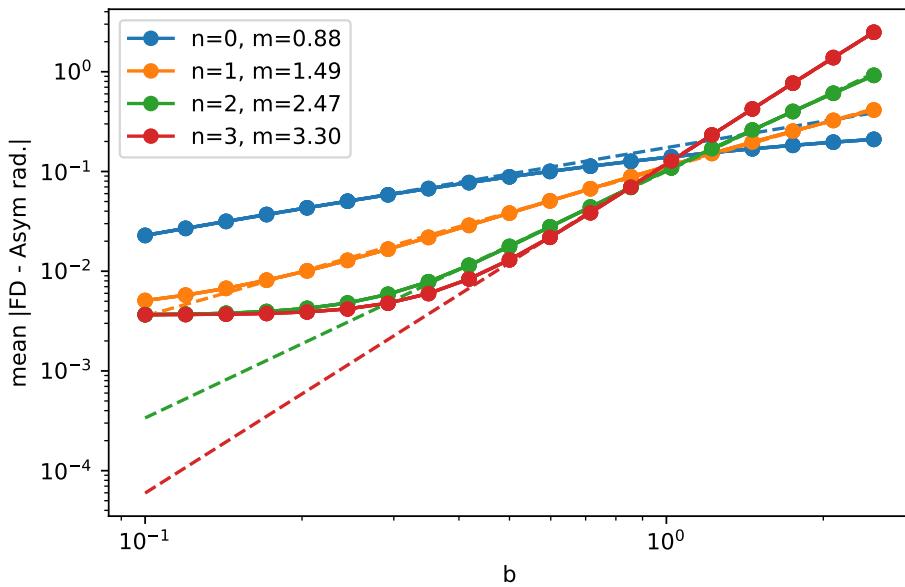


Figure 6.19: Asym real kelp. Blur radius=0.1

6.5 Comparison to Other Light Models

6.5.1 Full 10m

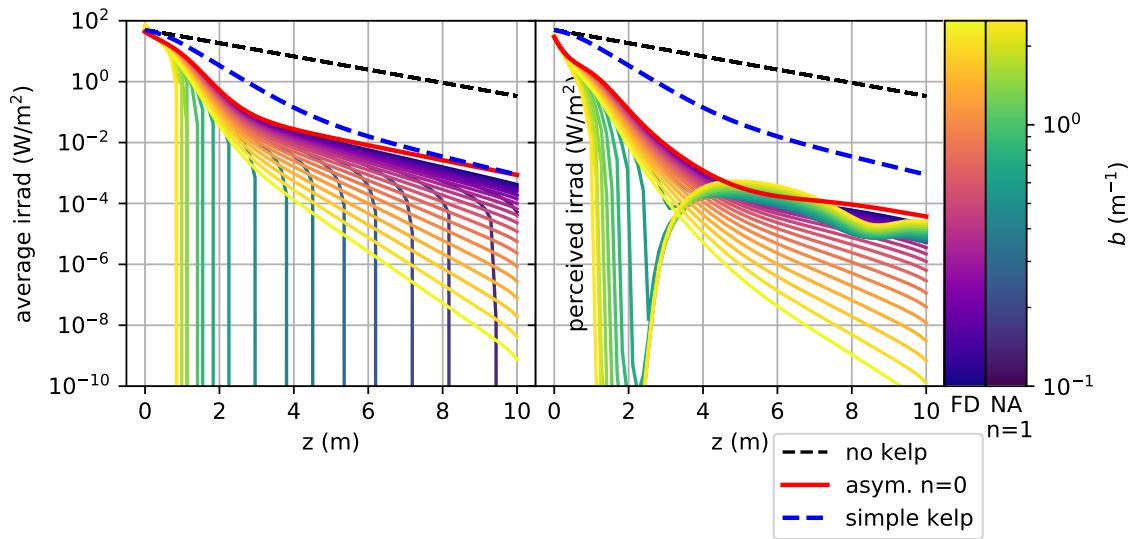


Figure 6.20: Compare models $n=1$

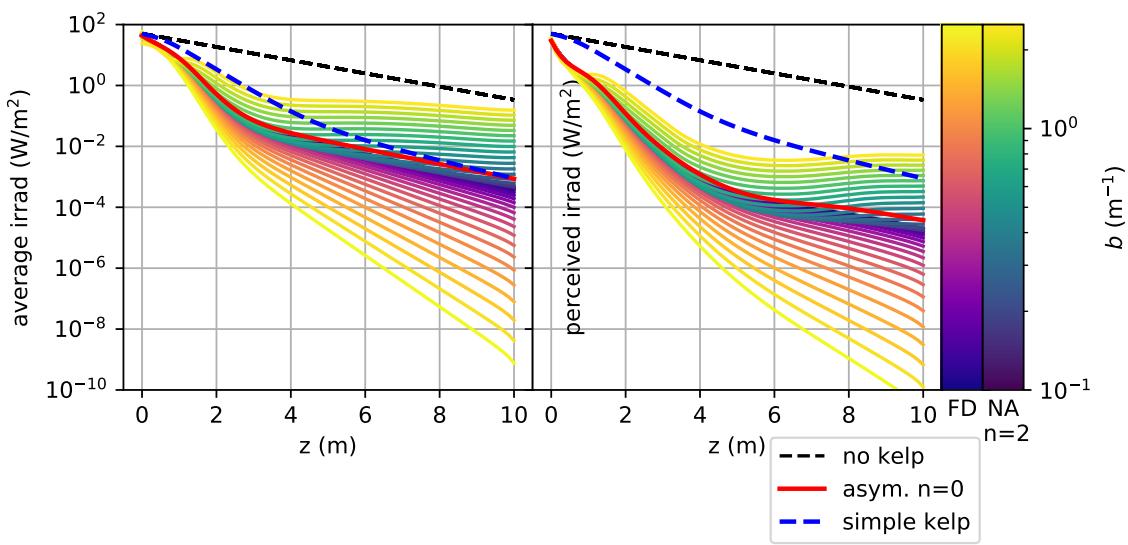


Figure 6.21: Compare models $n=2$

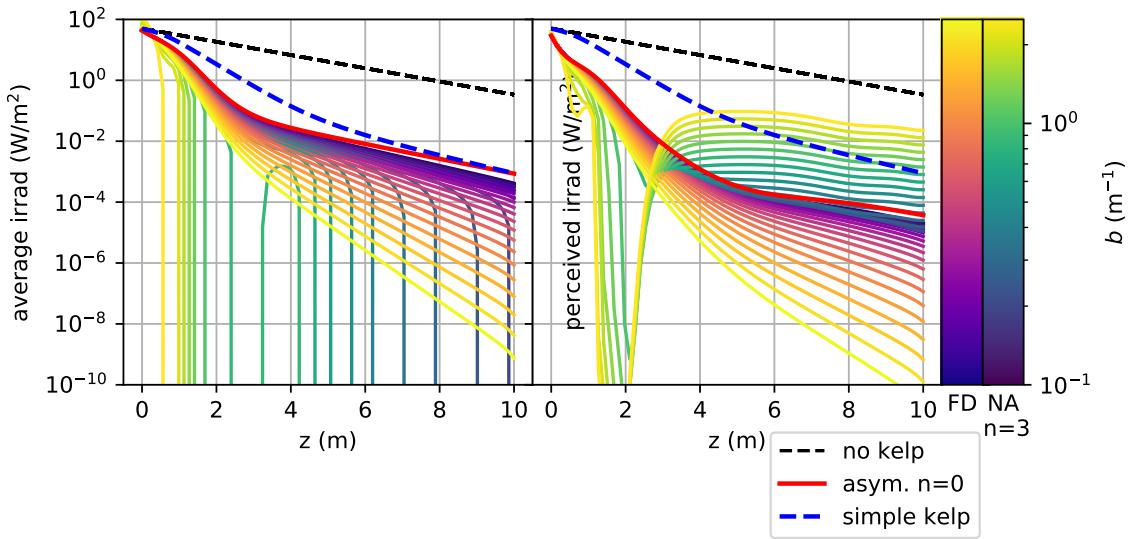


Figure 6.22: Compare models $n=3$

6.5.2 First 4m

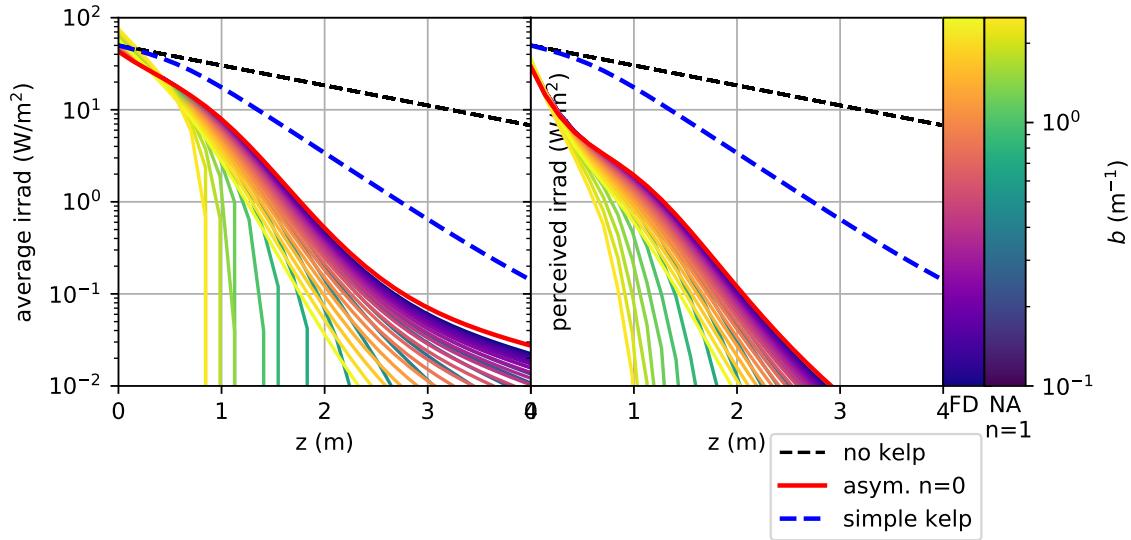


Figure 6.23: Compare models $n=1$

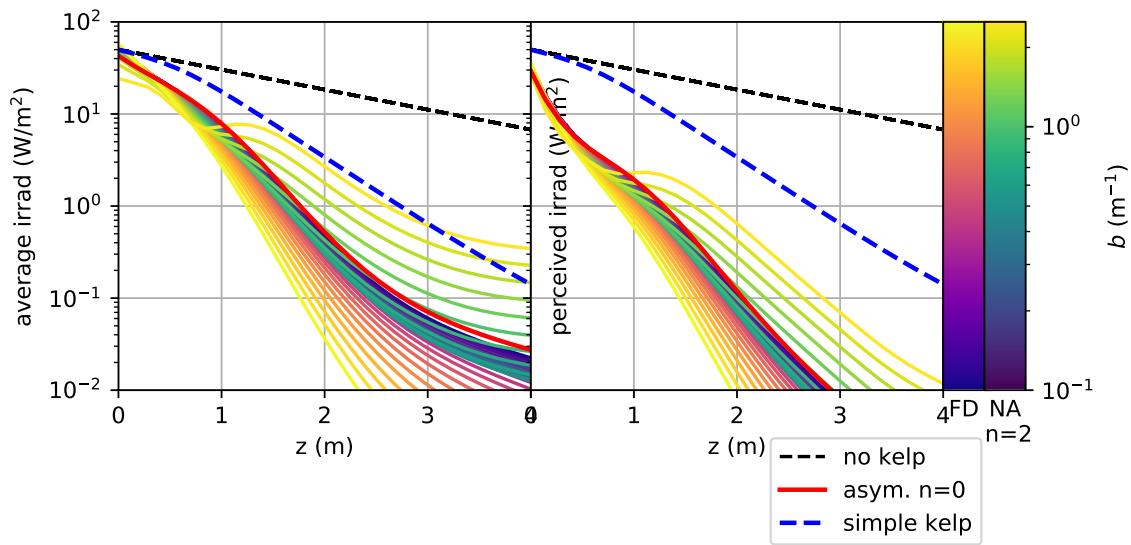


Figure 6.24: Compare models $n=2$

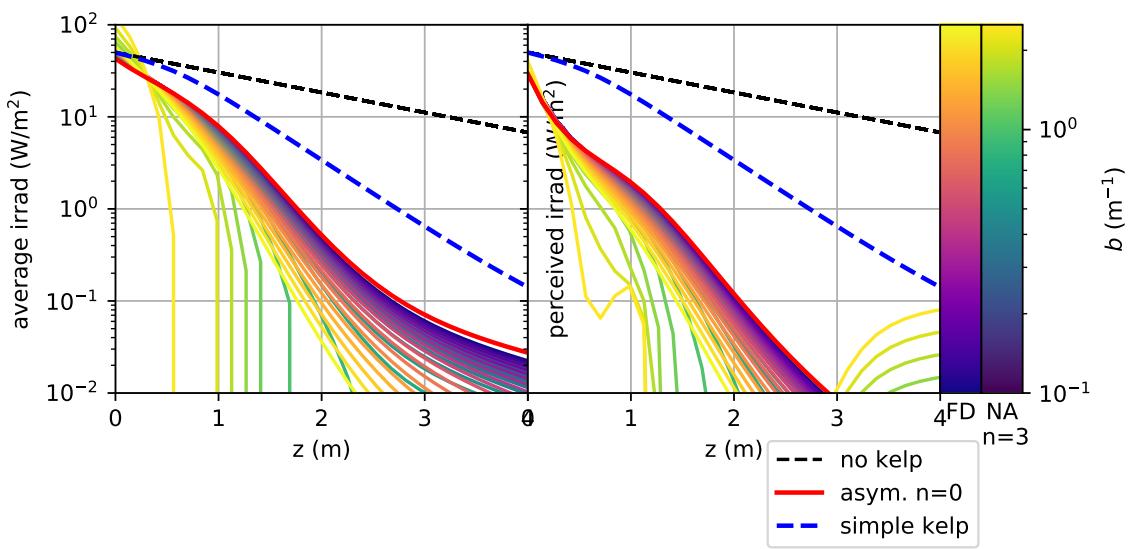


Figure 6.25: Compare models $n=3$

6.5.3 First 2m

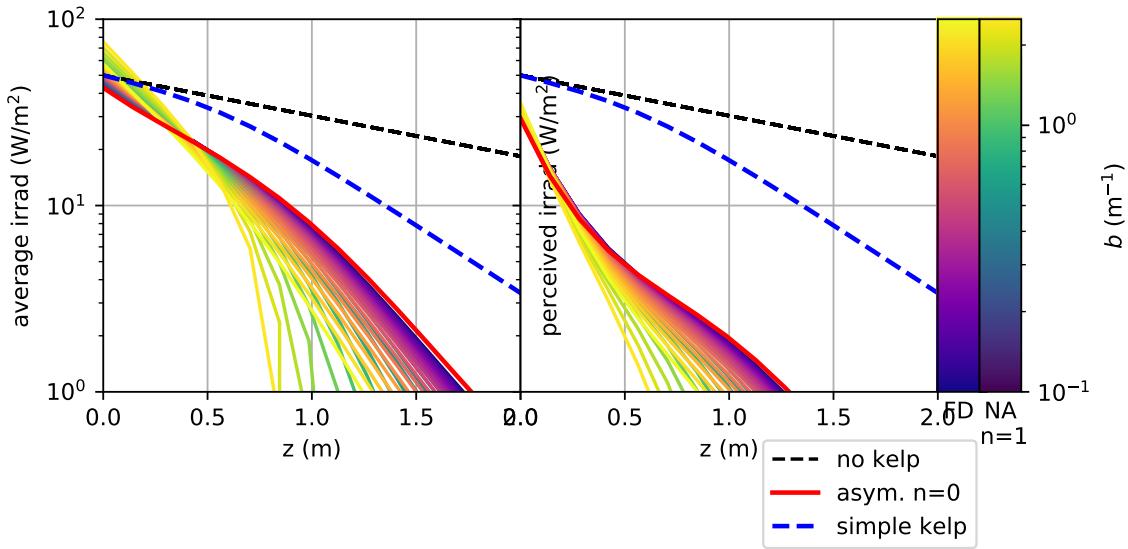


Figure 6.26: Compare models $n=1$

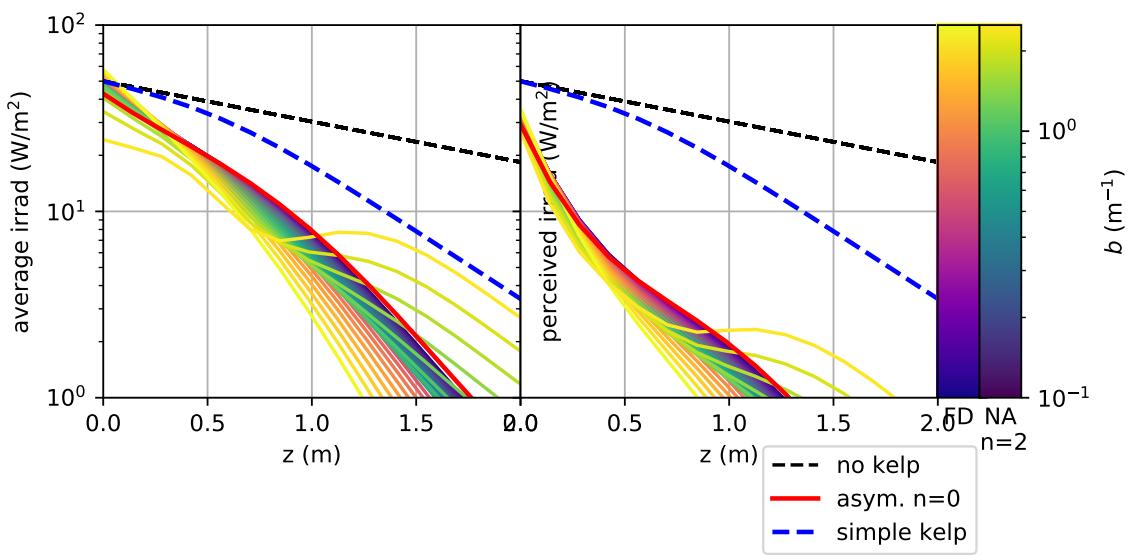


Figure 6.27: Compare models $n=2$

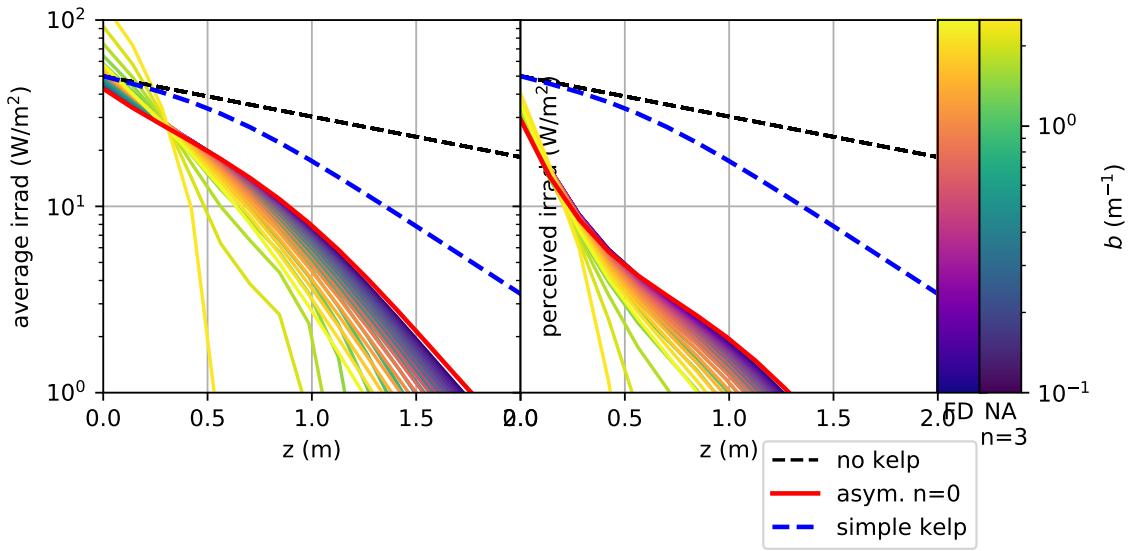


Figure 6.28: Compare models $n=3$

CHAPTER VII

CONCLUSION

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

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

One major simplification in the calculation of the kelp model is the assumption that the fronds are perfectly horizontal. This could be improved in a straightforward way by including some probability distribution for the angular elevation as a function of current speed, similar to the study performed in [19]. The cost of implementing polar rotation is that depth layers are no longer isolated. Rather than integrating the two dimensional length-orientation distribution from Section 2.3.3 to

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

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

BIBLIOGRAPHY

- [1] N. Anderson. A mathematical model for the growth of giant kelp. *Simulation*, 22(4):97–105, 1974.
- [2] A. H. Baker, E. R. Jessup, and T. Manteuffel. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM Journal on Matrix Analysis and Applications*, 26(4):962–984, Jan. 2005.
- [3] O. 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.
- [4] 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.
- [5] V. Brzeski and G. Newkirk. Integrated coastal food production systems a review of current literature. page 17, July 1996.

- [6] 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.
- [7] S. Chandrasekhar. *Radiative Transfer*. Dover, 1960.
- [8] 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.
- [9] M. F. Colombo-Pallotta, E. García-Mendoza, and L. B. Ladah. Photosynthetic Performance, Light Absorption, and Pigment Composition of *Macrocystis Pyrifera* (laminariales, Phaeophyceae) Blades from Different Depths1. *Journal of Phycology*, 42(6):1225–1234, Dec. 2006.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, Feb. 1970.
- [14] G. A. Jackson. Modelling the growth and harvest yield of the giant kelp *Macrocystis pyrifera*. *Marine Biology*, 95(4):611–624, 1987.
- [15] D. G. Jones. Corn-Based Ethanol Production in the United States and the Propensity for Pesticide Use. page 47.
- [16] 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.
- [17] C. Mobley. *Light and Water: Radiative Transfer in Natural Waters*. Academic Press, 1994.
- [18] C. Mobley. Radiative Transfer in the Ocean. In *Encyclopedia of Ocean Sciences*, pages 2321–2330. Elsevier, 2001.
- [19] C. Norvik. Design of Artificial Seaweeds for Assessment of Hydrodynamic Properties of Seaweed Farms. 2017.
- [20] 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.

- [21] 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.
- [22] T. J. Petzold. Volume Scattering Function for Selected Ocean Waters. Technical report, DTIC Document, 1972.
- [23] 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.
- [24] 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.
- [25] T. Searchinger, R. Heimlich, R. A. Houghton, F. Dong, A. Elobeid, J. Fabiosa, S. Tokgoz, D. Hayes, and T.-H. Yu. Use of U.S. Croplands for Biofuels Increases Greenhouse Gases Through Emissions from Land-Use Change. *Science*, 319(5867):1238–1240, Feb. 2008.
- [26] 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.

- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] C. Yu, W. Yao, and X. Bai. Robust Linear Regression: A Review and Comparison. *arXiv:1404.6274 [stat]*, Apr. 2014. arXiv: 1404.6274.

APPENDICES

APPENDIX A

GRID DETAILS

The width of the spatial grid cells in each dimension are

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

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

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

and the cell centers as

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

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

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

Denote the edges as

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

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

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

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

Now, we define the azimuthal angle such that

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

For the sake of periodicity, we need

$$\theta_1 = 0,$$

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

which requires

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

For the polar angle, we similarly let

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

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

$$\phi_1 = 0,$$

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

This gives us

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

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

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

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

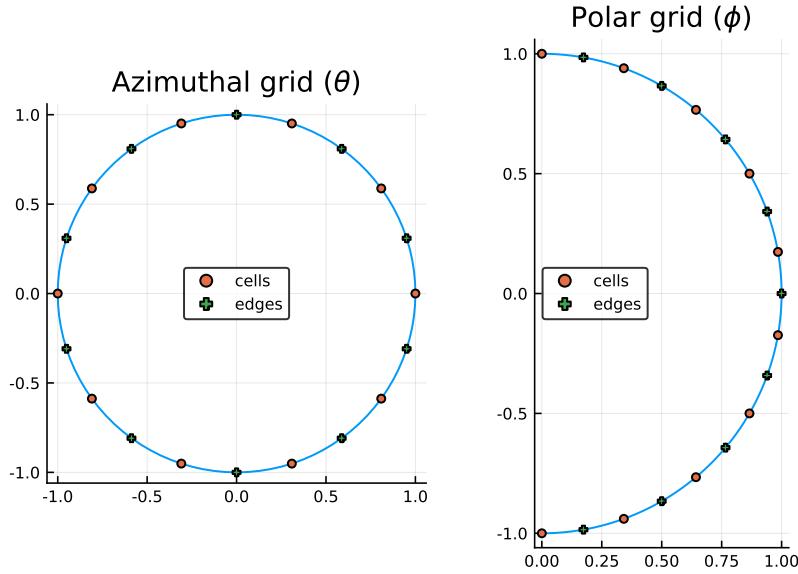


Figure A.1: Angular grid

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

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

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

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

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

Accordingly, define

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

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

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

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

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

For all other angular grid cells,

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

APPENDIX B

SYNTHETIC DATA

The chosen functions are

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

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

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

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, the boundary conditions and

source function are calculated to be

$$f(\theta, \phi) = \alpha (-\gamma + 1) (\sin(\phi + \theta) + 1), \quad (B.4)$$

$$\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 (B.5)$$

APPENDIX C

MEMORY USAGE

The matrix size in memory in bytes is calculated as follows.

```
1 | def calc_size(ns, na):
2 |     nx = ns
3 |     ny = ns
4 |     nz = ns
5 |     ntheta = na
6 |     nphi = na
7 |     nomega = ntheta * (nphi - 2) + 2
8 |
9 |     nnz = nx * ny * nomega * (nz * (6 + nomega)
10 |          - 1)
11 |     return 8*nnz
```

Memory requirement for LIS solution via GMRES with restart=100 is well-estimated by multiplying the above by 5.

Figure C.1: Memory to store one copy of the finite difference coefficient matrix

	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

Figure C.2: Memory to solve the linear system of equations with GMRES restarted every 100 iterations. This seems to require roughly five times the memory required to store the matrix.

	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 D

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then,

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

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

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

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

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

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

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

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.

As the ray traverses the spatial grids, it crosses $N - 2$ spatial grid edges.

Let the nondecreasing path lengths at which these crossings occur be denoted by

$\{s_\nu\}_{\nu=1}^N$, with the convention $s_1 = 0$ and $s_N = \tilde{s}$. This implies that edge ν precedes cell ν . $\{s_\nu\}$ is not strictly increasing if the ray directly intersects a grid corner, which means that multiple edges are traversed at the same path length. Hence, the path lengths through the grid cells, indexed by $\nu = 1, \dots, N - 1$, are

APPENDIX E

FORTRAN CODE

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

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

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

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

```

23      = (/ '.', '..', '..', '..', '/'))
24 ! Temporary path string
25 character(len=maxlength) tmp_path
26 ! Whether Makefile has been found yet
27 logical found
28 ! Path counter
29 integer ii
30 ! Lengths of paths
31 integer, dimension(numpaths) :: pathlengths
32
33 ! OUTPUT:
34 ! getbasedir - relative path to base
35 ! directory
36 ! Will either return '.', '..', or '../..'
37 character(len=maxlength) getbasedir
38
39 ! Determine length of each path
40 pathlengths(1) = 1
41 do ii = 2, numpaths
42     pathlengths(ii) = 2 + 3 * (ii - 2)
43 end do
44
45 ! Loop through paths
46 do ii = 1, numpaths
47     ! Determine this path
48     tmp_path = check_paths(ii)
49
50     ! Check whether Makefile is in this
51     ! directory
52     !write(*,*) 'Checking ', tmp_path(1:
53     !           pathlengths(ii)), ''
54     inquire(file=tmp_path(1:pathlengths(ii))
55             // '/Makefile', exist=found)
56     ! If so, stop. Otherwise, keep looking.
57     if(found) then
58         getbasedir = tmp_path(1:pathlengths(
59                         ii))
60         exit
61     end if
62 end do
63
64 ! If it hasn't been found, then this script
65 ! was probably called
66 ! from outside of the repository.
67 if(.not. found) then
68     write(*,*) 'BASE DIR NOT FOUND.'
69 end if
70
71 end function
72
73 ! Determine array size from min, max and step

```

```

69 ! If alignment is off, array will overstep the
70 ! maximum
71 function bnd2max(xmin,xmax,dx)
72     implicit none
73
74     ! INPUTS:
75     ! xmin - minimum x value in array
76     ! xmax - maximum x value in array (inclusive
77     ! )
78     ! dx - step size
79     double precision, intent(in) :: xmin, xmax,
80             dx
81
82     ! OUTPUT:
83     ! step2max - maximum index of array
84     integer bnd2max
85
86     ! Calculate array size
87     bnd2max = int(ceiling((xmax-xmin)/dx))
88 end function
89
90 ! Create array from bounds and number of
91 ! elements
92 ! xmax is not included in array
93 function bnd2arr(xmin,xmax,imax)
94     implicit none
95
96     ! INPUTS:
97     ! xmin - minimum x value in array
98     ! xmax - maximum x value in array (exclusive
99     ! )
100    double precision, intent(in) :: xmin, xmax
101    ! imax - number of elements in array
102    integer imax
103
104    ! OUTPUT:
105    ! bnd2arr - array to generate
106    double precision, dimension(imax) :: bnd2arr
107
108    ! BODY:
109
110    ! Counter
111    integer ii
112    ! Step size
113    double precision dx
114
115    ! Calculate step size
116    dx = (xmax - xmin) / imax
117
118    ! Generate array
119    do ii = 1, imax
120        bnd2arr(ii) = xmin + (ii-1) * dx

```

```

116     end do
117
118 end function
119
120 function mod1(i, n)
121   implicit none
122   integer i, n, m
123   integer mod1
124
125   m = modulo(i, n)
126
127   if(m .eq. 0) then
128     mod1 = n
129   else
130     mod1 = m
131   end if
132
133 end function mod1
134
135 function sgn_int(x)
136   integer x, sgn_int
137   ! Standard signum function
138   sgn_int = sign(1,x)
139   if(x .eq. 0.) sgn_int = 0
140 end function sgn_int
141
142 function sgn(x)
143   double precision x, sgn
144   ! Standard signum function
145   sgn = sign(1.d0,x)
146   if(x .eq. 0.) sgn = 0
147 end function sgn
148
149 ! Interpolate single point from 1D data
150 function interp(x0,xx,yy,nn)
151   implicit none
152
153   ! INPUTS:
154   ! x0 - x value at which to interpolate
155   double precision, intent(in) :: x0
156   ! xx - ordered x values at which y data is
157   !       sampled
158   ! yy - corresponding y values to interpolate
159   double precision, dimension (nn), intent(in)
160   :: xx,yy
161   ! nn - length of data
162   integer, intent(in) :: nn
163
164   ! OUTPUT:
165   ! interp - interpolated y value
166   double precision interp
167

```

```

166 ! BODY:
167
168 ! Index of lower-adjacent data (xx(i) < x0 <
169 ! xx(i+1))
170 integer ii
171 ! Slope of liine between (xx(ii),yy(ii)) and
172 ! (xx(ii+1),yy(ii+1))
173 double precision mm
174
175 ! If out of bounds , then return endpoint
176 ! value
177 if (x0 < xx(1)) then
178     interp = yy(1)
179 else if (x0 > xx(nn)) then
180     interp = yy(nn)
181 else
182
183     ! Determine ii
184     do ii = 1, nn
185         if (xx(ii) > x0) then
186             ! We've now gone one index too far
187             .
188             exit
189         end if
190     end do
191
192     ! Determine whether we're on the right
193     ! endpoint
194     if(ii-1 < nn) then
195         ! If this is a legitimate
196         ! interpolation , then
197         ! subtract since we went one index too
198         ! far
199         ii = ii - 1
200
201         ! Calculate slope
202         mm = (yy(ii+1) - yy(ii)) / (xx(ii+1) -
203                                     xx(ii))
204
205         ! Return interpolated value
206         interp = yy(ii) + mm * (x0 - xx(ii))
207     else
208         ! If we're actually interpolating the
209         ! right endpoint ,
210         ! then just return it.
211         interp = yy(nn)
212     end if
213
214 end if
215
216
217 end function
218
```

```

209 ! Calculate unshifted position of periodic image
210 ! Assuming xmin, xmax are extreme attainable
211 ! values of x
212 function shift_mod(x, xmin, xmax)
213   double precision x, xmin, xmax
214   double precision mod_part, shift_mod
215   mod_part = mod(x-xmin, xmax-xmin)
216   if(mod_part .ge. 0) then
217     ! In this case, mod_part is distance
218     ! between image & lower bound
219     shift_mod = xmin + mod_part
220   else
221     ! In this case, mod_part is distance
222     ! between image & upper bound
223     shift_mod = xmax + mod_part
224   endif
225 end function shift_mod
226
227 ! Bilinear interpolation on evenly spaced 2D
228 ! grid
229 ! Assume upper endpoint is not included and is
230 ! identical
231 ! to the lower endpoint, which is included.
232 function bilinear_array_periodic(x, y, nx, ny,
233   x_vals, y_vals, fun_vals)
234   implicit none
235   double precision x, y
236   integer nx, ny
237   double precision, dimension(:) :: x_vals,
238   y_vals
239   double precision, dimension(:, :) :: fun_vals
240
241   double precision dx, dy, xmin, ymin
242   integer i0, j0, i1, j1
243   double precision x0, x1, y0, y1
244   double precision z00, z10, z01, z11
245
246   double precision bilinear_array_periodic
247   xmin = x_vals(1)
248   ymin = y_vals(1)
249   dx = x_vals(2) - x_vals(1)
250   dy = y_vals(2) - y_vals(1)
251
252   ! Add 1 for one-indexing
253   i0 = int(floor((x-xmin)/dx))+1
254   j0 = int(floor((y-ymin)/dy))+1
255
256   x0 = x_vals(i0)
257   y0 = y_vals(j0)
258
259   ! Periodic wrap

```

```

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

```

```

301    i1 = i0 + 1
302    j1 = j0 + 1
303
304    ! Bounds checking
305    ! if(i0 .lt. 1) then
306    !   i0 = 1
307    !   i1 = 1
308    ! else if(i1 .gt. nx) then
309    !   i0 = nx
310    !   i1 = nx
311    ! endif
312    ! if(j0 .lt. 1) then
313    !   j0 = 1
314    !   j1 = 1
315    ! else if(j1 .gt. ny) then
316    !   j0 = ny
317    !   j1 = ny
318    ! endif
319
320    x0 = x_vals(i0)
321    x1 = x_vals(i1)
322    y0 = y_vals(j0)
323    y1 = y_vals(j1)
324
325    z00 = fun_vals(i0,j0)
326    z10 = fun_vals(i1,j0)
327    z01 = fun_vals(i0,j1)
328    z11 = fun_vals(i1,j1)
329
330    bilinear_array = bilinear(x, y, x0, y0, x1, y1
331                                , z00, z01, z10, z11)
332 end function bilinear_array
333
334 ! ilinear interpolation of a function of two
335 ! variables
336 ! over a rectangle of points.
337 ! Weight each point by the area of the sub-
338 ! rectangle involving
339 ! the point (x,y) and the point diagonally
340 ! across the rectangle
341
342 function bilinear(x, y, x0, y0, x1, y1, z00, z01
343                 , z10, z11)
344 implicit none
345 double precision x, y
346 double precision x0, y0, x1, y1, z00, z01, z10
347                 , z11
348 double precision a, b, c, d
349 double precision bilinear
350
351 a = (x-x0)*(y-y0)
352 b = (x1-x)*(y-y0)

```

```

346   c = (x-x0)*(y1-y)
347   d = (x1-x)*(y1-y)
348
349   bilinear = (a*z11 + b*z01 + c*z10 + d*z00) / (
350     a + b + c + d)
350 end function bilinear
351
352 ! Integrate using left endpoint rule
353 ! Assuming the right endpoint is not included in
353 ! arr
354 function lep_rule(arr, dx, nn)
355   implicit none
356
357   ! INPUTS:
358   ! arr - array to integrate
359   double precision, dimension(nn) :: arr
360   ! dx - array spacing (mesh size)
361   double precision dx
362   ! nn - length of arr
363   integer, intent(in) :: nn
364
365   ! OUTPUT:
366   ! lep_rule - integral w/ left endpoint rule
367   double precision lep_rule
368
369   ! BODY:
370
371   ! Counter
372   integer ii
373
374   ! Set output to zero
375   lep_rule = 0.0d0
376
377   ! Accumulate integral
378   do ii = 1, nn
379     lep_rule = lep_rule + arr(ii) * dx
380   end do
381
382 end function
383
384 ! Integrate using trapezoid rule
385 ! Assuming both endpoints are included in arr
386 function trap_rule_dx(arr, dx, nn)
387   implicit none
388   double precision, dimension(nn) :: arr
389   double precision dx
390   integer ii, nn
391   double precision trap_rule_dx
392
393   trap_rule_dx = 0.0d0
394
395   do ii=1, nn-1

```

```

396     trap_rule_dx = trap_rule_dx + 0.5d0 * dx *
397             (arr(ii) + arr(ii+1))
398 end do
399
400 end function trap_rule_dx
401
402 ! Integrate using trapezoid rule
403 ! Assuming both endpoints are included in arr
404 function trap_rule_uneven(xx, yy, nn)
405 implicit none
406 double precision, dimension(nn) :: xx
407 double precision, dimension(nn) :: yy
408 integer ii, nn
409 double precision trap_rule_uneven
410
411 trap_rule_uneven = 0.0d0
412
413 do ii=1, nn-1
414     trap_rule_uneven = trap_rule_uneven + 0.5d0
415         * (xx(ii+1)-xx(ii)) * (yy(ii) + yy(ii
416             +1))
417 end do
418
419 end function trap_rule_uneven
420
421 function trap_rule_dx_uneven(dx, yy, nn)
422 implicit none
423 double precision, dimension(nn-1) :: dx
424 double precision, dimension(nn) :: yy
425 integer ii, nn
426 double precision trap_rule_dx_uneven
427
428 trap_rule_dx_uneven = 0.0d0
429
430 do ii=1, nn-1
431     trap_rule_dx_uneven = trap_rule_dx_uneven +
432         0.5d0 * dx(ii) * (yy(ii) + yy(ii+1))
433 end do
434
435 end function trap_rule_dx_uneven
436
437 ! Integrate using midpoint rule
438 ! First and last bins, only use inner half
439 function midpoint_rule_halfends(dx, yy, nn)
440     result(integral)
441 implicit none
442 integer ii, nn
443 double precision, dimension(nn) :: dx, yy
444 double precision integral
445
446 if(nn > 1) then
447     integral = .5d0 * (dx(1)*yy(1) + dx(nn)*yy(
448         nn))
449
450 end function midpoint_rule_halfends

```

```

442 |     do ii=2, nn-1
443 |         integral = integral + dx(ii)*yy(ii)
444 |     end do
445 | else
446 |     integral = 0.d0
447 | end if
448 end function midpoint_rule_halfends
449
450 ! Normalize 1D array and return integral w/ left
451 ! endpoint rule
452 function normalize_dx(arr,dx,nn)
453     implicit none
454
455     ! INPUTS:
456     ! arr - array to normalize
457     double precision, dimension(nn) :: arr
458     ! dx - array spacing (mesh size)
459     double precision dx
460     ! nn - length of arr
461     integer, intent(in) :: nn
462
463     ! OUTPUT:
464     ! normalize - integral before normalization
465     ! (left endpoint rule)
466     double precision normalize_dx
467
468     ! BODY:
469
470     ! Calculate integral
471     normalize_dx = lep_rule(arr,dx,nn)
472
473     ! Normalize array
474     arr = arr / normalize_dx
475
476 end function normalize_dx
477
478 ! Normalize 1D unevenly-spaced array and
479 ! return integral w/ trapezoid rule
480 ! Will not be quite accurate if rightmost
481 ! endpoint is not included
482 ! (Very small for VSF, so not a big deal there)
483 ! Modifies yy in place
484 function normalize_uneven(xx, yy, nn) result(
485     norm)
486     implicit none
487
488     ! INPUTS:
489     ! xx, yy - array values of data to normalize
490     double precision, dimension(nn) :: xx, yy
491     ! nn - length of arr
492     integer, intent(in) :: nn

```

```

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

```

```

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

```

```

581      ! BODY
582
583
584      ! Row counter
585      integer ii
586      ! Final format to use
587      character(len=256) finfmt
588
589      ! Determine string format
590      if(present(fmtstr_in)) then
591          fmtstr = fmtstr_in
592      else
593          fmtstr = 'I10'
594      end if
595
596      ! Generate final format string
597      write(finfo, '(A,I4,A,A)') '(', mm, trim(
598                                     fmtstr), ')'
599
600      ! Loop through rows
601      do ii = 1, nn
602          ! Print one row at a time
603          write(*,finfo) arr(ii,:)
604      end do
605
606      ! Print blank line after
607      write(*,*) ''
608
609      end subroutine print_int_array
610
611      subroutine print_array(arr,nn,mm,fmtstr_in)
612          implicit none
613
614          ! INPUTS:
615          ! arr - array to print
616          double precision, dimension (nn,mm), intent(
617              in) :: arr
618          ! nn - number of data rows in file
619          ! nn - number of data columns in file
620          integer, intent(in) :: nn, mm
621          ! fmtstr - output format (no parentheses,
622          !         don't specify columns)
623          ! e.g. 'E10.2', not '(2E10.2)'
624          character(len=*), optional :: fmtstr_in
625          character(len=256) fmtstr
626
627          ! NO OUTPUTS
628
629          ! BODY
630
631          ! Row counter
632          integer ii

```

```

630 ! Final format to use
631 character(len=256) finfmt
632
633 ! Determine string format
634 if(present(fmtstr_in)) then
635     fmtstr = fmtstr_in
636 else
637     fmtstr = 'ES10.2'
638 end if
639
640 ! Generate final format string
641 write(finfmt,'(A,I4,A,A)') '(', mm, trim(
642             fmtstr), ')'
643
644 ! Loop through rows
645 do ii = 1, nn
646     ! Include row number
647     !write(*,'(I10)', advance='no') ii
648     ! Print one row at a time
649     write(*,finfmt) arr(ii,:)
650 end do
651
652 ! Print blank line after
653 write(*,*) ''
654
655 end subroutine
656
657 ! Write 1D array to file
658 subroutine write_vec(arr,nn,filename,fmtstr_in)
659     implicit none
660
661     ! INPUTS:
662     ! arr - array to print
663     double precision, dimension (nn), intent(in)
664             :: arr
665     ! nn - number of data rows in file
666     ! nn - number of data columns in file
667     integer, intent(in) :: nn
668     ! filename - file to write to
669     character(len=*) filename
670     ! fmtstr - output format (no parentheses,
671             ! don't specify columns)
672     ! e.g. 'E10.2', not '(2E10.2)'
673     character(len=*), optional :: fmtstr_in
674     character(len=256) fmtstr
675
676     ! NO OUTPUTS
677
678     ! BODY
679
680     ! Row counter
681     integer ii

```

```

679      ! Final format to use
680      character(len=256) finfmt
681      ! Dummy file unit to use
682      integer, parameter :: un = 20
683
684      ! Open file for writing
685      open(unit=un, file=trim(filename), status='
686          replace', form='formatted')
687
688      ! Determine string format
689      if(present(fmtstr_in)) then
690          fmtstr = fmtstr_in
691      else
692          fmtstr = 'E10.2'
693      end if
694
695      ! Generate final format string
696      write(finfmt,'(A,A,A)') '(', trim(fmtstr), '
697      ! Loop through rows
698      do ii = 1, nn
699          ! Print entry per row
700          write(un,finfmt) arr(ii)
701      end do
702
703      ! Close file
704      close(unit=un)
705
706  end subroutine
707
708  ! Write 2D array to file
709  subroutine write_array(arr,nn,mm,filename,
710      fmtstr_in)
711      implicit none
712
713      ! INPUTS:
714      ! arr - array to print
715      double precision, dimension (nn,mm), intent(
716          in) :: arr
717      ! nn - number of data rows in file
718      ! nn - number of data columns in file
719      integer, intent(in) :: nn, mm
720      ! filename - file to write to
721      character(len=*) filename
722      ! fmtstr - output format (no parentheses,
723          ! don't specify columns)
724      ! e.g. 'E10.2', not '(2E10.2)'
725      character(len=*), optional :: fmtstr_in
726      character(len=256) fmtstr
727
728      ! NO OUTPUTS

```

```

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

```

sag.f90

```

1 | module sag
2 | use utils

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

```

```

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

```

```

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

```

```

598  subroutine sag_deinit(grid)
599      class(space_angle_grid) :: grid
600      call grid%x%deinit()
601      call grid%y%deinit()
602      call grid%z%deinit()
603      call grid%angles%deinit()
604
605      deallocate(grid%x_factor)
606      deallocate(grid%y_factor)
607  end subroutine sag_deinit
608
609 ! Affine shift on x from [xmin, xmax] to [ymin
610 ! , ymax]
610 subroutine affine_transform(x, xmin, xmax,
611 !     ymin, ymax)
611     double precision x, xmin, xmax, ymin, ymax
612     x = ymin + (ymax-ymin)/(xmax-xmin) * (x-xmin
613     )
613 end subroutine affine_transform
614
615 function num_from_spacing(xmin, xmax, dx)
616     result(n)
616     double precision xmin, xmax, dx
617     integer n
618     n = floor( (xmax - xmin) / dx )
619 end function num_from_spacing
620
621 function spacing_from_num(xmin, xmax, nx)
622     result(dx)
622     double precision xmin, xmax, dx
623     integer nx
624     dx = (xmax - xmin) / dble(nx)
625 end function spacing_from_num
626 end module sag

```

```

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

```

13 | contains
14 |
15 | subroutine generate_grid(xmin, xmax, nx, ymin,
16 |   ymax, ny, zmin, zmax, nz, ntheta, nphi, grid,
17 |   p_kelp)
18 |   double precision xmin, xmax, ymin, ymax, zmin,
19 |   zmax
20 |   integer nx, ny, nz, ntheta, nphi
21 |   type(space_angle_grid) grid
22 |   double precision, dimension(:,:,:),
23 |     allocatable :: p_kelp
24 |
25 |   call grid%set_bounds(xmin, xmax, ymin, ymax,
26 |     zmin, zmax)
27 |   call grid%set_num(nx, ny, nz, ntheta, nphi)
28 |
29 |   allocate(p_kelp(nx,ny,nz))
30 |
31 | end subroutine generate_grid
32 |
33 | subroutine kelp3d_deinit(grid, rope, p_kelp)
34 |   type(space_angle_grid) grid
35 |   type(rope_state) rope
36 |   double precision, dimension(:,:,:),
37 |     allocatable :: p_kelp
38 |   call rope%deinit()
39 |   call grid%deinit()
40 |   deallocate(p_kelp)
41 | end subroutine kelp3d_deinit
42 |
43 | subroutine calculate_kelp_on_grid(grid, p_kelp,
44 |   frond, rope, quadrature_degree, n_images,
45 |   num_threads)
46 |   type(space_angle_grid), intent(in) :: grid
47 |   type(frond_shape), intent(in) :: frond
48 |   type(rope_state), intent(in) :: rope
49 |   type(point3d) point
50 |   integer, intent(in) :: quadrature_degree
51 |   integer, optional :: n_images
52 |   double precision, dimension(grid%x%num, grid%y
53 |     %num, grid%z%num) :: p_kelp

```

```

54 |   type(depth_state) depth
55 |   integer num_threads
56 |
57 |   integer i, j, k, nx, ny, nz
58 |   double precision x, y, z
59 |   ! Number of periodic images
60 |   ! to consider in each horizontal direction
61 |   ! for kelp distribution
62 |   ! n_images=1 => 3x3 meta-grid

```

```

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

```

```

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

```

```

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

```

```

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

```

```

215 !     relative_frond_edge = L / (sin(theta_prime)
216 !     + angular_sign(theta_prime * alpha(fs, fr) *
217 !     cos(theta_prime)))
218 function angular_sign(theta_prime)
219 ! S(\theta')
220 double precision, intent(in) :: theta_prime
221 double precision angular_sign
222 ! This seems to be incorrect in summary.pdf as
223 ! of 9/9/18
224 ! In the report, it's written as sgn(
225 !     theta_prime - pi/2.d0)
226 ! This results in L_min < 0 - not good!
227 angular_sign = sgn(pi/2.d0 - theta_prime)
228 end function angular_sign
229
230 subroutine gaussian_blur_2d(A, sigma, dx, dy, nk
231 , num_threads)
232 ! 2D Gaussian blur (periodic BC) with std
233 ! sigma
234 ! with kernel radius of nk (full size (2*nk+1)
235 ! x(2*nk+1))
236 ! applied to matrix A with element spacings dx
237 ! and dy.
238 double precision, intent(inout), dimension(:, :
239 !) :: A
240 double precision, intent(in) :: sigma, dx, dy
241 ! kernel half width
242 integer, intent(in) :: nk
243 ! kernel full width
244 integer kw
245 integer num_threads
246
247 ! A matrix size
248 integer nx, ny
249
250 ! indices
251 integer i1, j1
252 integer i2, j2
253 integer i, j
254 ! kernel
255 double precision, dimension(:, :, :), allocatable
256 ! :: k
257 ! output matrix
258 double precision, dimension(:, :, :), allocatable
259 ! :: B
260 ! kernel independent variables
261 double precision x, y

```

```

255 | if(sigma > 0) then
256 |   nx = size(A, 1)
257 |   ny = size(A, 2)
258 |
259 |   kw = 2*nk + 1
260 |
261 |   allocate(B(nx, ny))
262 |   allocate(k(kw, kw))
263 |   !write(*,*) 'creating kernel', sigma, nk
264 |   ! Create kernel
265 |   do i1=-nk, nk
266 |     x = i1*dx
267 |     i = i1+nk+1
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 |     )
273 |   end do
274 |
275 |   ! normalize kernel
276 |   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

```

```

301 |     A(:, :) = B(:, :)
302 |     deallocate(k)
303 |     deallocate(B)
304 |     !write(*,*) 'gb2d done.'
305 | end if
306 end subroutine gaussian_blur_2d
307
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

```

```

37   LIS_VECTOR b, x
38   LIS_SOLVER solver
39   LIS_INTEGER ierr
40   character(len=1024) solver_opts
41   logical initx_zeros
42
43   ! Pointer to solver subroutine
44   ! Set to mgmres by default
45   !procedure(solver_interface), pointer, nopass
46   :: solver => mgmres_st
47
48 contains
49   procedure :: init => mat_init
50   procedure :: deinit => mat_deinit
51   procedure :: calculate_size
52   procedure :: set_solver_opts =>
53     mat_set_solver_opts
54   procedure :: set_row => mat_set_row
55   procedure :: assign => mat_assign
56   procedure :: add => mat_add
57   procedure :: assign_rhs => mat_assign_rhs
58   procedure :: add_rhs => mat_add_rhs
59   !procedure :: store_index => mat_store_index
60   !procedure :: find_index => mat_find_index
61   procedure :: set_bc => mat_set_bc
62   procedure :: solve => mat_solve
63   procedure :: get_solver_stats
64   procedure :: ind => mat_ind
65   !procedure :: to_hdf => mat_to_hdf
66   procedure :: attenuate
67   procedure :: angular_integral
68   procedure :: add_source
69
70   ! Derivative subroutines
71   procedure x_cd2
72   procedure x_cd2_first
73   procedure x_cd2_last
74   procedure y_cd2
75   procedure y_cd2_first
76   procedure y_cd2_last
77   procedure z_cd2
78   procedure z_fd2
79   procedure z_bd2
80   procedure z_surface_bc
81   procedure z_bottom_bc
82
83 end type rte_mat
84
85 interface
86   ! Define interface for external procedure
87   ! 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
92 integer(index_kind) :: n_total, nonzero
93 integer, dimension(nonzero) :: row, col
94 double precision, dimension(nonzero) :::
95     data
96 double precision, dimension(nonzero) :: sol
97 double precision, dimension(n_total) :: rhs
98 integer :: maxiter_outer, maxiter_inner
99 double precision :: tol_abs, tol_rel
100 end subroutine solver_interface
101 end interface
102 contains
103
104 subroutine mat_init(mat, grid, iops)
105     class(rte_mat) mat
106     type(space_angle_grid) grid
107     type(optical_properties) iops
108     integer(index_kind) nnz, n_total
109
110     LIS_INTEGER 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 |     call lis_matrix_set_size(mat%A, n_total,
133 |                               n_total, mat% ierr)
134 |     call lis_vector_set_size(mat%b, n_total,
135 |                               n_total, mat% ierr)
136 |     call lis_vector_set_size(mat%x, n_total,
137 |                               n_total, mat% ierr)
138 |
139 |     if (mat% ierr .ne. 0) then
140 |         write(*,*) 'INIT ERR: ', mat% ierr
141 |         call exit(1)
142 |     end if
143 |
144 | ! CSR Format
145 | ! http://www.scipy-lectures.org/advanced/
146 | !      scipy_sparse/csr_matrix.html
147 |     write(*,*) 'Allocate CSR arrays'
148 |     allocate(mat%ptr(n_total+1))
149 |     allocate(mat%col(nnz))
150 |     allocate(mat%data(nnz))
151 |     allocate(mat%surface_vals(grid%angles%nomega
152 |                               ))
153 |     mat%ptr(n_total+1) = nnz
154 | end subroutine mat_init
155 |
156 | subroutine mat_deinit(mat)
157 |     class(rte_mat) mat
158 |
159 |     call lis_matrix_destroy(mat%A, mat% ierr)
160 |     call lis_vector_destroy(mat%b, mat% ierr)
161 |     call lis_vector_destroy(mat%x, mat% ierr)
162 |     call lis_solver_destroy(mat%solver, mat% ierr
163 |                               )
164 |     call lis_finalize(mat% ierr)
165 |
166 |     if (mat% ierr .ne. 0) then
167 |         write(*,*) 'DEINIT ERR: ', mat% ierr
168 |         call exit(1)
169 |     end if
170 |     deallocate(mat%ptr)
171 |     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

```

```

214 |
215 subroutine mat_solve(mat)
216   class(rte_mat) mat
217   character(len=64) init_opt
218
219   ! write(*,*) 'mat%n_total =', mat%n_total
220   ! write(*,*) 'mat%nonzero =', mat%nonzero
221   ! open(unit=1, file='ptr.txt')
222   ! open(unit=2, file='col.txt')
223   ! open(unit=3, file='data.txt')
224   ! write(1,*) mat%ptr
225   ! write(2,*) mat%col
226   ! write(3,*) mat%data
227   ! close(1)
228   ! close(2)
229   ! close(3)
230
231   ! Create matrix
232   write(*,*) 'LIS Set CSR'
233   call lis_matrix_set_csr(mat%nonzero, mat%ptr
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 out"
243   end if
244
245   write(*,*) 'LIS set solver options'
246   write(*,*) 'opt: ', trim(init_opt)
247   call lis_solver_set_option(trim(init_opt), mat%
248     solver, mat% ierr)
249   if(len(trim(mat%solver_opts)) .gt. 0) then
250     write(*,*) 'opt: ', trim(mat%solver_opts)
251     call lis_solver_set_option(trim(mat%
252       solver_opts), mat%solver, mat% ierr)
253   end if
254
255   ! Solve
256   write(*,*) 'LIS Solve'
257   call lis_solve(mat%A, mat%b, mat%x, mat%
258     solver, mat% ierr)
259   write(*,*) 'LIS Solve done'

```

```

257 | end subroutine mat_solve
258 |
259 | subroutine get_solver_stats(mat, lis_iter,
260 |   lis_time, lis_resid)
261 |   class(rte_mat) mat
262 |   integer lis_iter
263 |   double precision lis_time
264 |   double precision lis_resid
265 |   call lis_solver_get_iter(mat%solver,
266 |     lis_iter, mat% ierr)
267 |   call lis_solver_get_time(mat%solver,
268 |     lis_time, mat% ierr)
269 |   call lis_solver_get_residualnorm(mat%solver,
270 |     lis_resid, mat% ierr)
271 | end subroutine get_solver_stats
272 |
273 | subroutine mat_set_solver_opts(mat,
274 |   solver_opts)
275 |   class(rte_mat) mat
276 |   character(len=*) solver_opts
277 |   mat%solver_opts = solver_opts
278 | end subroutine mat_set_solver_opts
279 |
280 | function mat_ind(mat, i, j, k, p) result(ind)
281 |   ! Assuming var ordering: z, x, y, omega
282 |   class(rte_mat) mat
283 |   integer i, j, k, p
284 |   integer(index_kind) ind
285 |
286 |   ind = (i-1) * mat%x_block_size + (j-1) * mat%
287 |     %y_block_size + &
288 |       (k-1) * mat%z_block_size + p * mat%
289 |         omega_block_size
290 | end function mat_ind
291 |
292 | subroutine mat_set_row(mat, ent, row_num)
293 |   ! Start new row for CSR format
294 |   class(rte_mat) mat
295 |   integer(index_kind) ent, row_num
296 |   ! 0-indexing for LIS
297 |   mat%ptr(row_num) = ent - 1
298 | end subroutine mat_set_row
299 |
300 | subroutine mat_assign(mat, ent, val, i, j, k,
301 |   p)
302 |   ! It's assumed that this is the first time
303 |     this entry is defined
304 |   class(rte_mat) mat
305 |   double precision val
306 |   integer i, j, k, p

```

```

299     integer(index_kind) ent
300
301     ! LIS method 2 (LIS manual, p. 19) requires
302     ! 0-indexing
303     mat%col(ent) = mat%ind(i, j, k, p) - 1
304     mat%data(ent) = val
305
306     ent = ent + 1
307   end subroutine mat_assign
308
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
313 class(rte_mat) mat
314 double precision val
315 integer(index_kind) repeat_ent
316
316   ! Entry number where value is already stored
317   mat%data(repeat_ent) = mat%data(repeat_ent)
318   + val
318 end subroutine mat_add
319
320 subroutine mat_assign_rhs(mat, row_num, data)
321   class(rte_mat) mat
322   double precision data
323   integer(index_kind) row_num
324
325   call lis_vector_set_value(LIS_INS_VALUE,
326     row_num, data, mat%b, mat%ierr)
327   if(mat%ierr .ne. 0) then
328     write(*,*) 'RHS ERR: ', mat%ierr
329     call exit(1)
330   end if
331 end subroutine mat_assign_rhs
332
332 subroutine mat_add_rhs(mat, row_num, data)
333   class(rte_mat) mat
334   double precision data
335   integer(index_kind) row_num
336
337   call lis_vector_set_value(LIS_ADD_VALUE,
338     row_num, data, mat%b, mat%ierr)
339   if(mat%ierr .ne. 0) then
340     write(*,*) 'RHS ERR: ', mat%ierr
341     call exit(1)
342   end if
343 end subroutine mat_add_rhs

```

```

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

```

```

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
389   entry
390   ! And adding here is more efficient than a
391   ! conditional
392   ! in the angular loop.
393   class(rte_mat) mat
394   type(index_list) indices
395   integer(index_kind) row_num
396   double precision source_val
397
398   source_val = mat%iops%source_grid(indices%i,
399                                     indices%j, indices%k, indices%p)
400
401   call mat%add_rhs(row_num, source_val)
402 end subroutine add_source
403
404 subroutine x_cd2(mat, indices, ent)
405   class(rte_mat) mat
406   double precision val, dx
407   type(index_list) indices
408   integer i, j, k, p
409   integer(index_kind) ent
410
411   i = indices%i
412   j = indices%j
413   k = indices%k
414   p = indices%p
415
416   dx = mat%grid%x%spacing(1)
417
418   val = mat%grid%angles%sin_phi_p(p) &
419        * mat%grid%angles%cos_theta_p(p) / (2.
420          d0 * dx)
421
422   call mat%assign(ent, -val, i-1, j, k, p)
423   call mat%assign(ent, val, i+1, j, k, p)
424 end subroutine x_cd2
425
426 subroutine x_cd2_first(mat, indices, ent)
427   class(rte_mat) mat
428   double precision val, dx
429   integer nx
430   type(index_list) indices
431   integer i, j, k, p
432   integer(index_kind) ent
433
434   i = indices%i
435   j = indices%j
436   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
442   call mat%assign(ent,-val,nx,j,k,p)
443   call mat%assign(ent,val,i+1,j,k,p)
444 end subroutine x_cd2_first
445
446 subroutine x_cd2_last(mat, indices, ent)
447   class(rte_mat) mat
448   double precision val, dx
449   type(index_list) indices
450   integer i, j, k, p
451   integer(index_kind) ent
452
453   i = indices%i
454   j = indices%j
455   k = indices%k
456   p = indices%p
457
458   dx = mat%grid%x%spacing(1)
459
460   val = mat%grid%angles%sin_phi_p(p) &
461         * mat%grid%angles%cos_theta_p(p) / (2.
462           d0 * dx)
463
464   call mat%assign(ent,-val,i-1,j,k,p)
465   call mat%assign(ent,val,1,j,k,p)
466 end subroutine x_cd2_last
467
468 subroutine y_cd2(mat, indices, ent)
469   class(rte_mat) mat
470   double precision val, dy
471   type(index_list) indices
472   integer i, j, k, p
473   integer(index_kind) ent
474
475   i = indices%i
476   j = indices%j
477   k = indices%k
478   p = indices%p
479
480   dy = mat%grid%y%spacing(1)
481
482   val = mat%grid%angles%sin_phi_p(p) &

```

```

481      * mat%grid%angles%sin_theta_p(p) / (2.
482          d0 * dy)
483
484      call mat%assign(ent,-val,i,j-1,k,p)
485      call mat%assign(ent,val,i,j+1,k,p)
486  end subroutine y_cd2
487
488  subroutine y_cd2_first(mat, indices, ent)
489      class(rte_mat) mat
490      double precision val, dy
491      integer ny
492      type(index_list) indices
493      integer i, j, k, p
494      integer(index_kind) ent
495
496      i = indices%i
497      j = indices%j
498      k = indices%k
499      p = indices%p
500
501      dy = mat%grid%y%spacing(1)
502      ny = mat%grid%y%num
503
504      val = mat%grid%angles%sin_phi_p(p) &
505          * mat%grid%angles%sin_theta_p(p) / (2.
506              d0 * dy)
507
508      call mat%assign(ent,-val,i,ny,k,p)
509      call mat%assign(ent,val,i,j+1,k,p)
510  end subroutine y_cd2_first
511
512  subroutine y_cd2_last(mat, indices, ent)
513      class(rte_mat) mat
514      double precision val, dy
515      type(index_list) indices
516      integer i, j, k, p
517      integer(index_kind) ent
518
519      i = indices%i
520      j = indices%j
521      k = indices%k
522      p = indices%p
523
524      dy = mat%grid%y%spacing(1)
525
526      val = mat%grid%angles%sin_phi_p(p) &
527          * mat%grid%angles%sin_theta_p(p) / (2.
528              d0 * dy)
529
530      call mat%assign(ent,-val,i,j-1,k,p)

```

```

528     call mat%assign(ent, val, i, 1, k, p)
529 end subroutine y_cd2_last
530
531 subroutine z_cd2(mat, indices, ent)
532   class(rte_mat) mat
533   double precision val, dz
534   type(index_list) indices
535   integer i, j, k, p
536   integer(index_kind) ent
537
538   i = indices%i
539   j = indices%j
540   k = indices%k
541   p = indices%p
542
543   dz = mat%grid%z%spacing(indices%k)
544
545   val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
546                                             dz)
547
548   call mat%assign(ent, -val, i, j, k-1, p)
549   call mat%assign(ent, val, i, j, k+1, p)
550 end subroutine z_cd2
551
552 subroutine z_fd2(mat, indices, ent, repeat_ent
553   )
554   ! Has to be called after angular_integral
555   ! Because they both write to the same matrix
556   ! entry
557   ! And adding here is more efficient than a
558   ! conditional
559   ! in the angular loop.
560   class(rte_mat) mat
561   double precision val, val1, val2, val3, dz
562   type(index_list) indices
563   integer i, j, k, p
564   integer(index_kind) ent, repeat_ent
565
566   i = indices%i
567   j = indices%j
568   k = indices%k
569   p = indices%p
570
571   dz = mat%grid%z%spacing(indices%k)
572
573   val = mat%grid%angles%cos_phi_p(p) / (2.d0 *
574                                             dz)
575
576   val1 = -3.d0 * val
577   val2 = 4.d0 * val
578   val3 = -val

```

```

574      call mat%add(repeat_ent, val1)
575      call mat%assign(ent, val2, i, j, k+1, p)
576      call mat%assign(ent, val3, i, j, k+2, p)
577  end subroutine z_fd2
578
579
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
614  subroutine angular_integral(mat, indices, ent)
615    class(rte_mat) mat
616    ! Primed angular integration variables
617    integer pp
618    double precision val
619    type(index_list) indices
620    integer(index_kind) ent
621
622    do pp=1, mat%grid%angles%nomega
623      val = -mat%iops%scat * mat%iops%
624          vsf_integral(indices%p, pp)

```

```

619      call mat%assign(ent, val, indices%i,
620                        indices%j, indices%k, pp)
621    end do
622  end subroutine angular_integral
623
624  subroutine z_surface_bc(mat, indices, row_num,
625                            ent, repeat_ent)
626    class(rte_mat) mat
627    double precision bc_val
628    type(index_list) indices
629    double precision val1, val2, dz
630    integer(index_kind) row_num, ent, repeat_ent
631
632    dz = mat%grid%z%spacing(1)
633
634    val1 = mat%grid%angles%cos_phi_p(indices%p)
635        / (3.d0 * dz)
636    val2 = 3.d0 * val1
637    bc_val = 4.d0 * val1 * mat%surface_vals(
638                  indices%p)
639
640    call mat%assign(ent, val1, indices%i, indices%j
641                    ,2, indices%p)
642    call mat%add(repeat_ent, val2)
643    call mat%assign_rhs(row_num, bc_val)
644  end subroutine z_surface_bc
645
646  subroutine z_bottom_bc(mat, indices, ent,
647                         repeat_ent)
648    class(rte_mat) mat
649    type(index_list) indices
650    double precision val1, val2, dz
651    integer nz
652    integer(index_kind) ent, repeat_ent
653
654    dz = mat%grid%z%spacing(1)
655    nz = mat%grid%z%num
656
657    val1 = -mat%grid%angles%cos_phi_p(indices%p)
658        / (3.d0 * dz)
659    val2 = 3.d0 * val1
660
661    call mat%assign(ent, val1, indices%i, indices%j
662                    ,nz-1, indices%p)
663    call mat%add(repeat_ent, val2)
664  end subroutine z_bottom_bc
665
666  end module rte_sparse_matrices

```

```

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

```

```

47         angle_loop => bottom_angle_loop
48     else
49         angle_loop => interior_angle_loop
50     end if
51
52     indices%i = i
53     if(indices%i .eq. 1) then
54         ddx => x_cd2_first
55     else if(indices%i .eq. mat%grid%x%num
56             ) then
57         ddx => x_cd2_last
58     else
59         ddx => x_cd2
60     end if
61
62     indices%j = j
63     if(indices%j .eq. 1) then
64         ddy => y_cd2_first
65     else if(indices%j .eq. mat%grid%y%num
66             ) then
67         ddy => y_cd2_last
68     else
69         ddy => y_cd2
70     end if
71
72         call angle_loop(mat, indices, ddx,
73                           ddy)
74     end do
75   end do
76 end do
77 !$omp end parallel do
78 end subroutine whole_space_loop
79
80 function calculate_start_ent(grid, indices)
81   result(ent)
82   type(space_angle_grid) grid
83   type(index_list) indices
84   integer(index_kind) ent
85   integer(index_kind) boundary_nnz, interior_nnz
86   integer(index_kind) num_boundary, num_interior
87   integer(index_kind) num_this_x, num_this_z
88
89 ! Nonzero matrix entries for an surface or
! bottom spatial grid cell
! Definitely an integer since nomega is even
boundary_nnz = grid%angles%nomega * (2 * grid%
    angles%nomega + 11) / 2
! Nonzero matrix entries for an interior
! spatial grid cell
interior_nnz = grid%angles%nomega * (grid%
    angles%nomega + 6)

```

```

90 ! Order: z, x, y, omega
91 ! 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 +
    num_this_x
97 ! Calculate number of spatial grid cells of
98     each type which have
99 ! already been traversed up to this point
100 if(indices%k .eq. 1) then
101     num_boundary = num_this_z
102     num_interior = 0
103 else if(indices%k .eq. grid%z%num) then
104     num_boundary = (grid%x%num * grid%y%num) +
        num_this_z
105     num_interior = (grid%z%num-2) * grid%x%num
        * grid%y%num
106 else
107     num_boundary = grid%x%num * grid%y%num
108     num_interior = num_this_z + (indices%k-2) *
        grid%x%num * grid%y%num
109 end if
110
111 ent = num_boundary * boundary_nnz +
    num_interior * interior_nnz + 1
112 end function calculate_start_ent
113
114 function calculate_repeat_ent(ent, p) result(
    repeat_ent)
115     integer p
116     integer(index_kind) ent, repeat_ent
117     ! Entry number for row=mat%ind(i,j,k,p), col=
        mat%ind(i,j,k,p),
118     ! which will be modified multiple times in
        this matrix row
119     repeat_ent = ent + p - 1
120 end function calculate_repeat_ent
121
122 subroutine interior_angle_loop(mat, indices, ddx
    , ddy)
123     type(rte_mat) mat
124     type(index_list) indices
125     procedure(deriv_interface) :: ddx, ddy
126     integer p
127     integer(index_kind) ent, repeat_ent
128     integer(index_kind) row_num
129

```

```

130 ! Determine which matrix row to start at
131 ent = calculate_start_ent(mat%grid, indices)
132 indices%p = 1
133 row_num = mat%ind(indices%i, indices%j,
134     indices%k, indices%p)
135 do p=1, mat%grid%angles%nomega
136     indices%p = p
137     repeat_ent = calculate_repeat_ent(ent, p)
138     call mat%set_row(ent, row_num)
139     call mat%angular_integral(indices, ent)
140     call ddx(mat, indices, ent)
141     call ddy(mat, indices, ent)
142     call mat%z_cd2(indices, ent)
143     call mat%attenuate(indices, repeat_ent)
144     call mat%add_source(indices, row_num)
145     row_num = row_num + 1
146 end do
147 end subroutine
148
149 subroutine surface_angle_loop(mat, indices, ddx,
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 ! Determine which matrix row to start at
158 ent = calculate_start_ent(mat%grid, indices)
159 indices%p = 1
160 row_num = mat%ind(indices%i, indices%j,
161     indices%k, indices%p)
162 ! Downwelling
163 do p=1, mat%grid%angles%nomega / 2
164     indices%p = p
165     repeat_ent = calculate_repeat_ent(ent, p)
166     call mat%set_row(ent, row_num)
167     call mat%angular_integral(indices, ent)
168     call ddx(mat, indices, ent)
169     call ddy(mat, indices, ent)
170     call mat%z_surface_bc(indices, row_num, ent
171         , repeat_ent)
172     call mat%attenuate(indices, repeat_ent)
173     call mat%add_source(indices, row_num)
174     row_num = row_num + 1
175 end do
! Upwelling

```

```

176  do p=mat%grid%angles%nomega/2+1, mat%grid%
177    angles%nomega
178    indices%p = p
179    repeat_ent = calculate_repeat_ent(ent, p)
180    call mat%set_row(ent, row_num)
181    call mat%angular_integral(indices, ent)
182    call ddx(mat, indices, ent)
183    call ddy(mat, indices, ent)
184    call mat%z_fd2(indices, ent, repeat_ent)
185    call mat%attenuate(indices, repeat_ent)
186    call mat%add_source(indices, row_num)
187    row_num = row_num + 1
188  end do
189 end subroutine surface_angle_loop
190
191 subroutine bottom_angle_loop(mat, indices, ddx,
192   ddy)
193 type(rte_mat) mat
194 type(index_list) indices
195 integer p
196 integer(index_kind) row_num, ent, repeat_ent
197 procedure(deriv_interface) :: ddx, ddy
198
199 ! Determine which matrix row to start at
200 ent = calculate_start_ent(mat%grid, indices)
201 indices%p = 1
202 row_num = mat%ind(indices%i, indices%j,
203   indices%k, indices%p)
204
205 ! Downwelling
206 do p=1, mat%grid%angles%nomega/2
207   indices%p = p
208   repeat_ent = calculate_repeat_ent(ent, p)
209   call mat%set_row(ent, row_num)
210   call mat%angular_integral(indices, ent)
211   call ddx(mat, indices, ent)
212   call ddy(mat, indices, ent)
213   call mat%z_bd2(indices, ent, repeat_ent)
214   call mat%attenuate(indices, repeat_ent)
215   call mat%add_source(indices, row_num)
216   row_num = row_num + 1
217 end do
218 ! Upwelling
219 do p=mat%grid%angles%nomega/2+1, mat%grid%
220   angles%nomega
221   indices%p = p
222   repeat_ent = calculate_repeat_ent(ent, p)
223   call mat%set_row(ent, row_num)
224   call mat%angular_integral(indices, ent)
225   call ddx(mat, indices, ent)

```

```

222 |     call ddy(mat, indices, ent)
223 |     call mat%z_bottom_bc(indices, ent,
224 |         repeat_ent)
224 |     call mat%attenuate(indices, repeat_ent)
225 |     call mat%add_source(indices, row_num)
226 |     row_num = row_num + 1
227 |   end do
228 end subroutine bottom_angle_loop
229
230 subroutine gen_matrix(mat, num_threads)
231   type(rte_mat) mat
232   type(index_list) indices
233   integer num_threads
234
235   call indices%init()
236
237   call whole_space_loop(mat, indices,
238     num_threads)
238 ! call surface_space_loop(mat, indices)
239 ! call interior_space_loop(mat, indices)
240 ! call bottom_space_loop(mat, indices)
241 end subroutine gen_matrix
242
243 subroutine rte3d_deinit(mat, iops, light)
244   type(rte_mat) mat
245   type(optical_properties) iops
246   type(light_state) light
247
248   call mat%deinit()
249   call iops%deinit()
250   call light%deinit()
251 end subroutine
252
253 end module rte3d

```

kelp_context.f90

```

1 module kelp_context
2 use sag
3 use prob
4 implicit none
5
6 ! Point in cylindrical coordinates
7 type point3d
8   double precision x, y, z, theta, r
9   contains
10  procedure :: set_cart => point_set_cart
11  procedure :: set_cyl => point_set_cyl
12  procedure :: cartesian_to_polar
13  procedure :: polar_to_cartesian
14 end type point3d

```

```

15 type frond_shape
16   double precision fs, fr, tan_alpha, alpha, ft
17 contains
18   procedure :: set_shape => frond_set_shape
19   procedure :: calculate_angles =>
20     frond_calculate_angles
21 end type frond_shape
22
23 type rope_state
24   integer nz
25   double precision, dimension(:), allocatable
26     :: frond_lengths, frond_stds, num_fronds,
27       water_speeds, water_angles
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

```

```

58   procedure :: calc_vsf_on_grid
59   procedure :: deinit => iop_deinit
60   procedure :: vsf_from_function
61 end type optical_properties
62
63 type boundary_condition
64   double precision IO, decay, theta_s, phi_s
65   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
76   function bc_gaussian(bc, theta, phi)
77     class(boundary_condition) bc
78     double precision theta, phi, diff
79     double precision bc_gaussian
80     diff = angle_diff_3d(theta, phi, bc%theta_s,
81                           bc%phi_s)
82     bc_gaussian = exp(-bc%decay * diff)
83   end function bc_gaussian
84
85   subroutine bc_init(bc, grid, theta_s, phi_s,
86                      decay, IO)
87     class(boundary_condition) bc
88     type(space_angle_grid) grid
89     double precision theta_s, phi_s, decay, IO
90     integer p
91     double precision theta, phi
92     double precision bc_norm
93     double precision, allocatable, dimension(:)
94       :: whole_bc_grid
95     integer nomega
96
97     nomega = grid%angles%nomega
98
99     allocate(bc%bc_grid(nomega/2))
100    allocate(whole_bc_grid(nomega))
101
102    bc%theta_s = theta_s
103    bc%phi_s = phi_s
104    bc%decay = decay
105    bc%IO = IO
106
107    ! Only set BC for downwelling light
108    do p=1, nomega/2

```

```

105     theta = grid%angles%theta_p(p)
106     phi = grid%angles%phi_p(p)
107     bc%bc_grid(p) = bc%bc_gaussian(theta, phi
108     )
109   end do
110
111 ! Normalize
112 ! Use 'whole_bc_grid' because angular
113 ! integration
114 ! subroutine requires all angles to have
115 ! values,
116 ! but 'bc%bc_grid' only has downwelling
117 ! values.
118 ! Use zeros for upwelling.
119 whole_bc_grid(1:nomega/2) = bc%bc_grid
120 whole_bc_grid(nomega/2+1:nomega) = 0
121 bc_norm = grid%angles%integrate_points(
122     whole_bc_grid)
123 bc%bc_grid = bc%I0 * bc%bc_grid / bc_norm
124 end subroutine bc_init
125
126 subroutine bc_deinit(bc)
127   class(boundary_condition) bc
128   deallocate(bc%bc_grid)
129 end subroutine
130
131 subroutine point_set_cart(point, x, y, z)
132   class(point3d) :: point
133   double precision x, y, z
134   point%x = x
135   point%y = y
136   point%z = z
137   call point%cartesian_to_polar()
138 end subroutine point_set_cart
139
140 subroutine point_set_cyl(point, theta, r, z)
141   class(point3d) :: point
142   double precision theta, r, z
143   point%theta = theta
144   point%r = r
145   point%z = z
146   call point%polar_to_cartesian()
147 end subroutine point_set_cyl
148
149 subroutine polar_to_cartesian(point)
150   class(point3d) :: point
151   point%x = point%r*cos(point%theta)
152   point%y = point%r*sin(point%theta)
153 end subroutine polar_to_cartesian

```

```

150 | subroutine cartesian_to_polar(point)
151 |   class(point3d) :: point
152 |   point%r = sqrt(point%x**2 + point%y**2)
153 |   point%theta = atan2(point%y, point%x)
154 | end subroutine cartesian_to_polar
155 |
156 | subroutine frond_set_shape(frond, fs, fr, ft)
157 |   class(frond_shape) frond
158 |   double precision fs, fr, ft
159 |   frond%fs = fs
160 |   frond%fr = fr
161 |   frond%ft = ft
162 |   call frond%calculate_angles()
163 | end subroutine frond_set_shape
164 |
165 | subroutine frond_calculate_angles(frond)
166 |   class(frond_shape) frond
167 |   frond%tan_alpha = 2.d0*frond%fs*frond%fr /
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

```

```

194 subroutine iop_zero_source(iops)
195   class(optical_properties) iops
196
197   iops%source_grid = 0
198 end subroutine iop_zero_source
199
200 subroutine iop_set_source(iops, source)
201   class(optical_properties) iops
202   double precision, dimension(:,:,:,:) :: source
203
204   iops%source_grid = source
205 end subroutine iop_set_source
206
207 subroutine calculate_coef_grids(iops, p_kelp)
208   class(optical_properties) iops
209   double precision, dimension(:,:,:,:) :: p_kelp
210
211   integer k
212
213   ! Allow water IOPs to vary over depth
214   do k=1, iops%grid%z%num
215     iops%abs_grid(:,:,k) = (iops%abs_kelp -
216       iops%abs_water(k)) * p_kelp(:,:,k) +
217       iops%abs_water(k)
218   end do
219
220 end subroutine calculate_coef_grids
221
222 subroutine load_vsf(iops, filename, fmtstr)
223   class(optical_properties) :: iops
224   character(len=*) :: filename, fmtstr
225   double precision, dimension(:, :),
226     allocatable :: tmp_2d_arr
227   integer num_rows, num_cols, skip_lines_in
228
229   ! First column is the angle at which the
230   ! measurement is taken
231   ! Second column is the value of the VSF at
232   ! that angle
233   num_rows = iops%num_vsf
234   num_cols = 2
235   skip_lines_in = 1 ! Ignore comment on first
236   ! line
237
238   allocate(tmp_2d_arr(num_rows, num_cols))
239
240   tmp_2d_arr = read_array(filename, fmtstr,
241     num_rows, num_cols, skip_lines_in)
242   iops%vsf_angles = tmp_2d_arr(:,1)

```

```

237 |     iops%vsf_vals = tmp_2d_arr(:,2)
238 |
239 |     ! write(*,*) 'vsf_angles = ', iops%
240 |     ! write(*,*) 'vsf_vals = ', iops%vsf_vals
241 |
242 |     ! Pre-evaluate for all pair of angles
243 |     call iops%calc_vsf_on_grid()
244 end subroutine load_vsf
245
246 function eval_vsf(iops, theta)
247     class(optical_properties) iops
248     double precision theta
249     double precision eval_vsf
250     ! No need to set vsf(0) = 0.
251     ! It's the area under the curve that matters
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   depth%frond_length = rope%frond_lengths(
284     depth_layer)
285   depth%frond_std = rope%frond_stds(
286     depth_layer)
287   depth%water_speeds = rope%water_speeds(
288     depth_layer)
289   depth%water_angles = rope%water_angles(
290     depth_layer)
291   depth%num_fronds = rope%num_fronds(
292     depth_layer)
293   depth%depth_layer = depth_layer
294   depth%depth = grid%z%vals(depth_layer)
295 end subroutine set_depth
296
297 function length_distribution_cdf(depth, L)
298   result(output)
299 ! C_L(L)
300 class(depth_state) depth
301 double precision L, L_mean, L_std
302 double precision output
303
304 L_mean = depth%frond_length
305 L_std = depth%frond_std
306
307 call normal_cdf(L, L_mean, L_std, output)
308 end function length_distribution_cdf
309
310 function angle_distribution_pdf(depth, theta_f)
311   result(output)
312 ! P_{\theta_f}(\theta_f)
313 class(depth_state) depth
314 double precision theta_f, v_w, theta_w
315 double precision output
316 double precision diff
317
318 v_w = depth%water_speeds
319 theta_w = depth%water_angles
320
321 ! von_mises_pdf is only defined on [-pi, pi]
322 ! So take difference of angles and input
323 ! into
324 ! von_mises dist. centered & x=0.
325
326 diff = angle_diff_2d(theta_f, theta_w)
327
328 call von_mises_pdf(diff, 0.d0, v_w, output)
329 end function angle_distribution_pdf
330
331 function angle_mod(theta) result(mod_theta)
332   ! Shift theta to the interval [-pi, pi]

```

```

325 ! which is where von_mises_pdf is defined.
326
327 double precision theta, mod_theta
328
329 mod_theta = mod(theta + pi, 2.d0*pi) - pi
330 end function angle_mod
331
332 function angle_diff_2d(theta1, theta2) result(
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     diff = pi - abs(abs(modt1-modt2) - pi)
349 end function angle_diff_2d
350
351
352 function angle_diff_3d(theta, phi, theta_prime,
353     , phi_prime) result(diff)
354     ! Angle between two vectors in spherical
355     ! coordinates
356     double precision theta, phi, theta_prime,
357     phi_prime
358     double precision alpha, diff
359
360     ! Faster, but produces lots of NaNs (at
361     ! least in Python)
362     !alpha = sin(theta)*sin(theta_prime)*cos(
363     !    theta-theta_prime) + cos(phi)*cos(
364     !    phi_prime)
365
366     ! Slower, but more accurate
367     alpha = (sin(phi)*sin(phi_prime) &
368             * (cos(theta)*cos(theta_prime) + sin(theta
369                 )*sin(theta_prime)) &
370             + cos(phi)*cos(phi_prime))
371
372     ! Avoid out-of-bounds errors due to rounding
373     alpha = min(1.d0, alpha)
374     alpha = max(-1.d0, alpha)
375

```

```

365     diff = acos(alpha)
366   end function angle_diff_3d
367
368   subroutine vsf_from_function(iops, func)
369     class(optical_properties) iops
370     double precision, external :: func
371     integer i
372     type(angle_dim) :: angle1d
373
374     call angle1d%set_bounds(-1.d0, 1.d0)
375     call angle1d%set_num(iops%num_vsf)
376     call angle1d%assign_legendre()
377
378     iops%vsf_angles(:) = acos(angle1d%vals(:))
379     do i=1, iops%num_vsf
380       iops%vsf_vals(i) = func(iops%vsf_angles(i))
381     end do
382
383     call iops%calc_vsf_on_grid()
384
385     call angle1d%deinit()
386   end subroutine vsf_from_function
387
388   subroutine calc_vsf_on_grid(iops)
389     class(optical_properties) iops
390     double precision th, ph, thp, php
391     integer p, pp
392     integer nomega
393     double precision norm
394
395     nomega = iops%grid%angles%nomega
396
397     do p=1, nomega
398       th = iops%grid%angles%theta_p(p)
399       ph = iops%grid%angles%phi_p(p)
400       do pp=1, nomega
401         thp = iops%grid%angles%theta_p(pp)
402         php = iops%grid%angles%phi_p(pp)
403         iops%vsf(p, pp) = iops%eval_vsf(
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

```

```

411      ! from cell pp (2nd ind.) into direction
412      !          p (1st ind.).
413      iops%vsf_integral(p, :) = iops%vsf(p, :)
414      &
415      * iops%grid%angles%area_p(:)
416      !write(*,*) 'vsf_integral (beta_pp)', p,
417      ' = ', iops%vsf_integral(p, :)
418  end do
419 end subroutine calc_vsf_on_grid
420
421 subroutine iop_deinit(iops)
422   class(optical_properties) iops
423   deallocate(iops%vsf_angles)
424   deallocate(iops%vsf_vals)
425   deallocate(iops%vsf)
426   deallocate(iops%vsf_integral)
427   deallocate(iops%abs_water)
428   deallocate(iops%abs_grid)
429   deallocate(iops%source_grid)
430
431 end subroutine iop_deinit
432
433 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

```

```

23   end type light_state
24
25 contains
26
27 ! Init for use with mat
28 subroutine light_init(light, mat)
29   class(light_state) light
30   type(rte_mat) mat
31   integer nx, ny, nz, nomega
32
33   light%mat = mat
34   light%grid = mat%grid
35
36   nx = light%grid%x%num
37   ny = light%grid%y%num
38   nz = light%grid%z%num
39   nomega = light%grid%angles%nomega
40
41   allocate(light%irradiance(nx, ny, nz))
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      ! write(*,*) 'ln =', ln
77      ! write(*,*) 'gn =', gn
78
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

```

```

117      call lis_vector_get_value(light%
118        mat%x, index, &
119          light%radiance(i,j,k,p),
120          light%mat% ierr)
121      if(light%mat% ierr .ne. 0) then
122        write(*,*) 'EXTRACT ERROR:', 
123          light%mat% ierr
124      end if
125      index = index + 1
126    end do
127  end do
128 end subroutine calculate_radiance
129
130 subroutine light_calculate_irradiance(light)
131   class(light_state) light
132   integer i, j, k
133   integer nx, ny, nz
134   double precision, dimension(light%grid%
135     angles%omega) :: tmp_rad
136
137   nx = light%grid%x%num
138   ny = light%grid%y%num
139   nz = light%grid%z%num
140
141   do i=1, nx
142     do j=1, ny
143       do k=1, nz
144         ! Use temporary array to avoid
145         ! creating one
146         ! implicitly at every spatial grid
147         ! point
148         tmp_rad = light%radiance(i,j,k,:)
149         light%irradiance(i,j,k) = &
150           light%grid%angles%
151             integrate_points(tmp_rad)
152       end do
153     end do
154   end do
155
156 end subroutine light_calculate_irradiance
157
158 ! subroutine light_to_hdf(light, radfile,
159 !   irradfile)
160 !   class(light_state) light
161 !   character(len=*) radfile
162 !   character(len=*) irradfile
163 !
164 !   call hdf_write_radianc(radfile, light%
165 !     radiance, light%grid)

```

```

159 !     call hdf_write_irradiance(irradfile, light%
160 !     irradiance, light%grid)
161 ! end subroutine light_to_hdf
162
163 subroutine light_deinit(light)
164   class(light_state) light
165
166   deallocate(light%irradiance)
167   deallocate(light%radiance)
168 end subroutine light_deinit
169 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)

```

```

32
33     allocate(path_length(max_cells+1))
34     allocate(path_spacing(max_cells))
35     allocate(a_tilde(max_cells))
36     allocate(gn(max_cells))
37     allocate(rad_scatter(grid%x%num, grid%y%num,
38                           grid%z%num, grid%angles%nomega))
39
40     write(*,*) 'before'
41     write(*,*) 'min radiance =', minval(radiance)
42     write(*,*) 'max radiance =', maxval(radiance)
43     write(*,*) 'mean radiance =', sum(radiance)/
44                   size(radiance)
45
46     write(*,*) 'advect source + bc'
47     bc_flag = .true.
48     call advect_light( &
49                       grid, iops, source, radiance, &
50                       path_length, path_spacing, &
51                       a_tilde, gn, bc_flag, num_threads, bc)
52
53     write(*,*) 'after'
54     write(*,*) 'min radiance =', minval(radiance)
55     write(*,*) 'max radiance =', maxval(radiance)
56     write(*,*) 'mean radiance =', sum(radiance)/
57                   size(radiance)
58
59     rad_scatter = radiance
60     bb = iops%scat
61
62     do n=1, num_scatters
63         write(*,*) 'scatter #', n
64         call scatter(grid, iops, source,
65                      rad_scatter, path_length, path_spacing
66                      , a_tilde, gn, num_threads)
67         radiance = radiance + bb**n * rad_scatter
68     end do
69
70     write(*,*) 'asymptotics complete'
71
72     deallocate(path_length)
73     deallocate(path_spacing)
74     deallocate(a_tilde)
75     deallocate(gn)
76     deallocate(rad_scatter)

```

```

72  end subroutine
73      calculate_asymptotic_light_field
74
74 subroutine
75     calculate_asymptotic_light_field_expanded_source
76     (&
77         grid, bc, iops, source, &
78         source_expansion, radiance, &
79         num_scatters, num_threads)
80     type(space_angle_grid) grid
81     type(boundary_condition) bc
82     type(optical_properties) iops
83     double precision, dimension(:,:,:,:,:) :::
84         radiance
85     double precision, dimension(:,:,:,:,:) :::
86         source_expansion
87     integer num_scatters
88     integer nx, ny, nz, nomega
89     integer max_cells
90     integer n
91     logical bc_flag
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'

```

```

112 bc_flag = .true.
113 call advect_light( &
114     grid, iops, source_expansion(:,:,:,:,1)
115     , radiance, &
116     path_length, path_spacing, &
117     a_tilde, gn, bc_flag, num_threads, bc)
118 ! Disable BC for scattering advection
119 bc_flag = .false.
120
121 rad_scatter = radiance
122 bb = iops%scat
123
124 do n=1, num_scatters
125     write(*,*) 'scatter #', n
126     call calculate_scatter_source(grid, iops,
127         rad_scatter, source, scatter_integral
128         , num_threads)
129     source = source + source_expansion
130         (:,:,:,:,n+1)
131     call advect_light(grid, iops, source,
132         rad_scatter, path_length, path_spacing
133         , a_tilde, gn, bc_flag, num_threads)
134
135     radiance = radiance + bb**n * rad_scatter
136
137 end do
138
139 write(*,*) 'asymptotics complete'
140
141 deallocate(path_length)
142 deallocate(path_spacing)
143 deallocate(a_tilde)
144 deallocate(gn)
145 deallocate(rad_scatter)
146 deallocate(scatter_integral)
147 end subroutine
148     calculate_asymptotic_light_field_expanded_source
149
150 ! Add attenuated surface light to existing
151     radiance
152 subroutine advect_surface_bc(&
153     i, j, k, p, radiance, &
154     path_spacing, num_cells, a_tilde, bc)
155     type(boundary_condition) bc
156     double precision, dimension(:,:,:,:,:) :::
157         radiance
158     double precision, dimension(:) :::
159         path_spacing, a_tilde
160     integer i, j, k, p
161     integer num_cells

```

```

152     double precision atten
153
154     atten = sum(path_spacing(1:num_cells) *
155                  a_tilde(1:num_cells))
156     ! Avoid underflow
157     if(atten .lt. 100.d0) then
158         radiance(i,j,k,p) = radiance(i,j,k,p) +
159                     bc%bc_grid(p) * exp(-atten)
160     end if
161     end subroutine advect_surface_bc
162
163     ! Perform one scattering event
164     subroutine scatter(grid, iops, source,
165                        rad_scatter, path_length, path_spacing,
166                        a_tilde, gn, num_threads)
167     type(space_angle_grid) grid
168     type(optical_properties) iops
169     double precision, dimension(:,:,:,:,:) :::
170                 rad_scatter, source
171     double precision, dimension(:,:,:,:),
172                 allocatable :: scatter_integral
173     double precision, dimension(:) :::
174                 path_length, path_spacing, a_tilde, gn
175     integer nx, ny, nz, nomega
176     logical bc_flag
177     integer num_threads
178
179     nx = grid%x%num
180     ny = grid%y%num
181     nz = grid%z%num
182     nomega = grid%angles%nomega
183     bc_flag = .false.
184
185     allocate(scatter_integral(nx, ny, nz, nomega
186                               ))
187
188     call calculate_scatter_source(grid, iops,
189                                 rad_scatter, source, scatter_integral,
190                                 num_threads)
191     call advect_light(grid, iops, source,
192                      rad_scatter, path_length, path_spacing,
193                      a_tilde, gn, bc_flag, num_threads)
194
195     deallocate(scatter_integral)
196     end subroutine scatter
197
198     ! Calculate source from no-scatter or previous
199     ! scattering layer
200     subroutine calculate_scatter_source(grid, iops
201                                         , rad_scatter, source, scatter_integral,
202                                         num_threads)

```

```

188 | type(space_angle_grid) grid
189 | type(optical_properties) iops
190 | double precision, dimension(:,:,:,:,:) :: 
191 |     rad_scatter
192 | double precision, dimension(:,:,:,:,:) :: 
193 |     source
194 | double precision, dimension(:,:,:,:,:) :: 
195 |     scatter_integral
196 | type(index_list) indices
197 | integer nx, ny, nz, nomega
198 | integer i, j, k, p
199 | integer num_threads
200 |
201 | nx = grid%x%num
202 | ny = grid%y%num
203 | nz = grid%z%num
204 | nomega = grid%angles%nomega
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 | !$omp end parallel do
231 |
232 | source(:,:,:,:,:) = -rad_scatter(:,:,:,:,:) +
233 |     scatter_integral(:,:,:,:,:)

```

```

230
231     write(*,*) 'source min: ', minval(source)
232     write(*,*) 'source max: ', maxval(source)
233     write(*,*) 'source mean: ', sum(source)/size
234             (source)
235 end subroutine calculate_scatter_source
236
237 subroutine calculate_scatter_integral(iops,
238         rad_scatter, scatter_integral, indices)
239     type(optical_properties) iops
240     double precision, dimension(:,:,:,:,:) :::
241             rad_scatter, scatter_integral
242     type(index_list) indices
243
244     scatter_integral(indices%i,indices%j,indices
245             %k,indices%p) &
246             = sum(iops%vsf_integral(indices%p, :) &
247             * rad_scatter(&
248                 indices%i,&
249                 indices%j,&
250                 indices%k,:))
251
252 end subroutine calculate_scatter_integral
253
254 subroutine advect_light(grid, iops, source,
255         rad_scatter, path_length, path_spacing,
256         a_tilde, gn, 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

```

```

269      do p=1, grid%angles%nomega
270        call integrate_ray(grid, iops,
271          source,&
272            rad_scatter, path_length,
273              path_spacing,&
274                a_tilde, gn, i, j, k, p,
275                  bc_flag, bc)
276      end do
277    end do
278  end do
279 !$omp end parallel do
280 end subroutine advect_light
281 ! New algorithm, double integral over
282   piecewise-constant 1d funcs
283 subroutine integrate_ray(grid, iops, source,
284   rad_scatter, path_length, path_spacing,
285   a_tilde, gn, i, j, k, p, bc_flag, bc)
286 type(space_angle_grid) :: grid
287 type(optical_properties) :: iops
288 double precision, dimension(:,:,:,:,:) :: source
289 double precision, dimension(:,:,:,:,:) :: rad_scatter
290 integer :: i, j, k, p
291 ! The following are only passed to avoid
292   unnecessary allocation
293 double precision, dimension(:) :: path_length, path_spacing, a_tilde, gn
294 logical bc_flag
295 type(boundary_condition), intent(in),
296   optional :: bc
297 integer num_cells
298 call traverse_ray(grid, iops, source, i, j,
299   k, p, path_length, path_spacing, a_tilde,
300     gn, num_cells)
301 rad_scatter(i,j,k,p) =
302   calculate_ray_integral(num_cells,
303     path_length, path_spacing, a_tilde, gn)
304 if(bc_flag .and. p .le. grid%angles%nomega
305 /2) then
306   call advect_surface_bc(&
307     i, j, k, p, rad_scatter, &
308       path_spacing, num_cells, &
309         a_tilde, bc)
310 end if
311
```

```

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

```

```

347      ! In this case, so much absorption has
348      ! occurred
349      ! previously on the path that we don't
350      ! need
351      ! to continue, and we might get underflow
352      ! if we do.
353      if(bi .lt. -100.d0) then
354          di_exp_bi = 0.d0
355      else
356
357          ! Without this conditional, overflow
358          ! occurs.
359          ! Which is unnecessary, because large
360          ! absorption
361          ! means very small light added to the
362          ! ray
363          ! at this grid cell.
364          if(a_tilde(i) .lt. cutoff) then
365              if(a_tilde(i) .eq. 0) then
366                  di_exp_bi = ds(i) * exp(bi)
367              else
368                  ! In an attempt to avoid
369                  ! overflow
370                  ! and reduce compute time,
371                  ! I'm combining exponentials.
372                  ! di*exp(bi) -> di_exp.bi
373                  di_exp.bi = (exp(a_tilde(i)*s(i)
374                      +1) + bi) - exp(a_tilde(i)*s(
375                          i) + bi))/a_tilde(i)
376              end if
377              integral = integral + gn(i)*
378                  di_exp.bi
379          end if
380      end if
381  end do
382
383  end function calculate_ray_integral
384
385  ! Calculate maximum number of cells a path
386  ! through the grid could take
387  ! This is a loose upper bound
388  function calculate_max_cells(grid) result(
389      max_cells)
390      type(space_angle_grid) :: grid
391      integer :: max_cells
392      double precision dx, dy, zrange, phi_middle
393
394      ! Angle that will have the longest ray
395      phi_middle = grid%angles%phi(grid%angles%
396          nphi/2)
397      dx = grid%x%spacing(1)
398      dy = grid%y%spacing(1)

```

```

386     zrange = grid%z%maxval - grid%z%minval
387
388     max_cells = grid%z%num + ceiling((1/dx+1/dy)
389         *zrange*tan(phi_middle))
390 end function calculate_max_cells
391
392 ! Traverse from surface or bottom to point (xi
393     , yj, zk)
394 ! in the direction omega_p, extracting path
395     lengths (ds) and
396 ! function values (f) along the way,
397 ! as well as number of cells traversed (n).
398 subroutine traverse_ray(grid, iops, source, i,
399     j, k, p, s_array, ds, a_tilde, gn,
400     num_cells)
401     type(space_angle_grid) :: grid
402     type(optical_properties) :: iops
403     double precision, dimension(:,:,:,:,:) :: source
404     integer :: i, j, k, p
405     double precision, dimension(:) :: s_array,
406     ds, a_tilde, gn
407     integer :: num_cells
408
409     integer t
410     double precision p0x, p0y, p0z
411     double precision p1x, p1y, p1z
412     double precision z0
413     double precision s_tilde, s
414     integer dir_x, dir_y, dir_z
415     integer shift_x, shift_y
416     integer cell_x, cell_y, cell_z
417     integer edge_x, edge_y
418     integer first_x, last_x, first_y, last_y,
419         last_z
420     double precision s_next_x, s_next_y,
421         s_next_z, s_next
422     double precision x_factor, y_factor,
423         z_factor
424     double precision ds_x, ds_y
425     double precision, dimension(grid%z%num) :: ds_z
426     double precision smx, smy
427
428     ! Divide by these numbers to get path
429     separation
430     ! from separation in individual dimensions
431     x_factor = grid%angles%sin_phi_p(p) * grid%
432         angles%cos_theta_p(p)
433     y_factor = grid%angles%sin_phi_p(p) * grid%
434         angles%sin_theta_p(p)
435     z_factor = grid%angles%cos_phi_p(p)

```

```

424      ! Destination point
425      p1x = grid%x%vals(i)
426      p1y = grid%y%vals(j)
427      p1z = grid%z%vals(k)
428
429      ! write(*,*) 'START PATH.'
430      ! write(*,*) 'ijk = ', i, j, k
431
432      ! Direction
433      if(p .le. grid%angles%nomega/2) then
434          ! Downwelling light originates from
435          ! 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
445      ! destination
446      ! (sign is correct for upwelling and
447      ! downwelling)
448      s_tilde = (p1z - z0)/grid%angles%cos_phi_p(p
449
450      ! Path spacings between edge intersections
451      ! in each dimension
452      ! Set to 2*s_tilde if infinite in this
453      ! dimension so that it's unreachable
454      ! (e.g., if ray is parallel to x axis, then
455      ! no x intersection will occur.)
456      ! Assume x & y spacings are uniform,
457      ! so it's okay to just use the first value.
458      if(x_factor .eq. 0) then
459          ds_x = 2*s_tilde
460      else
461          ds_x = abs(grid%x%spacing(1)/x_factor)
462      end if
463      if(y_factor .eq. 0) then
464          ds_y = 2*s_tilde
465      else
466          ds_y = abs(grid%y%spacing(1)/y_factor)
467      end if
468
469      ! This one is an array because z spacing can
470      ! vary
471      ! z_factor should never be 0,
472      ! because the ray is then horizontal
473      ! and infinite in length.

```

```

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

```

```

512     if(dir_y .gt. 0) then
513         first_y = 1
514         last_y = grid%y%num
515     else
516         first_y = grid%y%num
517         last_y = 1
518     end if
519     if(dir_z .gt. 0) then
520         last_z = grid%z%num
521     else
522         last_z = 1
523     end if
524
525     ! Calculate periodic images
526     smx = shift_mod(p0x, grid%x%minval, grid%x%
527                     maxval)
527     smy = shift_mod(p0y, grid%y%minval, grid%y%
528                     maxval)
529
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                     - smy) / y_factor
560     else
561         ! Otherwise, just compare to next cell

```

```

552     s_next_y = (grid%y%edges(edge_y) - smy) /
553         y_factor
554 end if
555
556 s_next_z = ds_z(cell_z)
557
558 ! Initialize path
559 s = 0.d0
560 s_array(1) = 0.d0
561
562 ! Start with t=0 so that we can increment
563     before storing,
564 ! so that t will be the number of grid cells
565     at the end of the loop.
566 t = 0
567
568 ! s is the beginning of the current cell,
569 ! s_next is the end of the current cell.
570 do while (s .lt. s_tilde)
571     ! Move cell counter
572     t = t + 1
573
574     ! Extract function values
575     a_tilde(t) = iops%abs_grid(cell_x, cell_y
576             , cell_z)
577     gn(t) = source(cell_x, cell_y, cell_z, p)
578
579     !write(*,*) ''
580     !write(*,*) 's_next_x = ', s_next_x
581     !write(*,*) 's_next_y = ', s_next_y
582     !write(*,*) 's_next_z = ', s_next_z
583     !write(*,*) 'theta, phi =', grid%angles%
584             theta_p(p)*180.d0/pi, grid%angles%
585             phi_p(p)*180.d0/pi
586     !write(*,*) 's = ', s, '/', s_tilde
587     !write(*,*) 'cell_z =', cell_z, '/', grid
588             %z%num
589     !write(*,*) 's_next_z =', s_next_z
590     !write(*,*) 'last_z =', last_z
591     !write(*,*) 'new'
592
593     ! Move to next cell in path
594     if(s_next_x .le. min(s_next_y, s_next_z))
595         then
596             ! x edge is closest
597             s_next = s_next_x
598
599             ! Increment indices (periodic)
600             cell_x = mod1(cell_x + dir_x, grid%x%
601                         num)

```

```

593     edge_x = mod1(edge_x + dir_x, grid%x%
594             num)
595
596     ! x intersection after the one at s=
597             s_next
598     s_next_x = s_next + ds_x
599
600     else if (s_next_y .le. min(s_next_x,
601             s_next_z)) then
602         ! y edge is closest
603         s_next = s_next_y
604
605         ! Increment indices (periodic)
606         cell_y = mod1(cell_y + dir_y, grid%y%
607             num)
608         edge_y = mod1(edge_y + dir_y, grid%y%
609             num)
610
611         ! y intersection after the one at s=
612             s_next
613         s_next_y = s_next + ds_y
614
615     else if(s_next_z .le. min(s_next_x,
616             s_next_y)) then
617         ! z edge is closest
618         s_next = s_next_z
619
620         ! Increment indices
621         cell_z = cell_z + dir_z
622
623         ! write(*,*) 'z edge, s_next =', s_next
624
625         ! z intersection after the one at s=
626             s_next
627         if(dir_z * (last_z - cell_z) .gt. 0)
628             then
629                 ! Only look ahead if we aren't at
630                     the end
631                 s_next_z = s_next + ds_z(cell_z)
632             else
633                 ! Otherwise, no need to continue.
634                 ! this is our final destination.
635                 ! exit
636                 s_next_z = 2*s_tilde
637                 ! write(*,*) 'end. s_next_z =',
638                         s_next_z
639             end if
640
641         end if
642
643         ! Cut off early if this is the end

```

```

633      ! This will be the last cell traversed if
634      s_next >= s_tilde
635      s_next = min(s_tilde, s_next)
636      ! Store path length
637      s_array(t+1) = s_next
638      ! Extract path length from same cell as
639      ! function vals
640      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, &

```

```

31      depth_spacing, &
32      ! SOLVER PARAMETERS
33      nx, &
34      ny, &
35      nz, &
36      ntheta, &
37      nphi, &
38      num_scatters, &
39      ! FINAL RESULTS
40      perceived_irrad, &
41      avg_irrad)
42
43      implicit none
44
45      ! OPTICAL PROPERTIES
46      integer, intent(in) :: nx, ny, nz, ntheta,
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

```

```

69   double precision, intent(in) ::  
    frond_thickness  
70   ! Ratio of length to width (0,infty)  
71   double precision, intent(in) ::  
    frond_aspect_ratio  
72   ! Rescaled position of greatest width (0=  
    base, 1=tip)  
73   double precision, intent(in) ::  
    frond_shape_ratio  
74  
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  
! Number of fronds in each depth layer  
104  double precision, dimension(:, allocatable  
      :: num_fronds

```

```

106   double precision, dimension(:,:,:,:) ,
107      allocatable :: p_kelp
108
109      write(*,*) 'Light calculation'
110
111      allocate(pop_length_means(nz))
112      allocate(pop_length_stds(nz))
113      allocate(num_fronds(nz))
114      allocate(p_kelp(nx, ny, nz))
115
116      xmin = -rope_spacing/2
117      xmax = rope_spacing/2
118
119      ymin = -rope_spacing/2
120      ymax = rope_spacing/2
121
122      zmin = 0.d0
123      zmax = sum(depth_spacing)
124
125      write(*,*) 'Grid'
126      call grid%set_bounds(xmin, xmax, ymin, ymax,
127                             zmin, zmax)
128      call grid%set_num(nx, ny, nz, ntheta, nphi)
129      call grid%init()
130      !call grid%set_uniform_spacing_from_num()
131      call grid%z%set_spacing_array(depth_spacing)
132
133      call rope%init(grid)
134
135      write(*,*) 'Rope'
136      ! Calculate kelp distribution
137      call calculate_length_dist_from_superinds( &
138        nz, &
139        num_si, &
140        si_area, &
141        si_ind, &
142        frond_aspect_ratio, &
143        num_fronds, &
144        pop_length_means, &
145        pop_length_stds)
146
147      rope%frond_lengths = pop_length_means
148      rope%frond_stds = pop_length_stds
149      rope%num_fronds = num_fronds
150      rope%water_speeds = current_speeds
151      rope%water_angles = current_angles
152
153      write(*,*) 'frond_lengths = ', rope%
154                  frond_lengths
155      write(*,*) 'frond_stds = ', rope%frond_stds

```

```

153 |     write(*,*) 'num_fronds   =', rope%num_fronds
154 |     write(*,*) 'water_speeds  =', rope%
155 |         water_speeds
156 |     write(*,*) 'water_angles =', rope%
157 |         water_angles
158 |
159 |     write(*,*) 'Frond'
160 | ! INIT FROND
161 | call frond%set_shape(frond_shape_ratio,
162 |     frond_aspect_ratio, frond_thickness)
163 | ! CALCULATE KELP
164 | quadrature_degree = 5
165 | call calculate_kelp_on_grid(grid, p_kelp,
166 |     frond, rope, quadrature_degree)
167 | ! INIT IOPS
168 | iops%num_vsf = num_vsf
169 | call iops%init(grid)
170 | write(*,*) 'IOPs'
171 | iops%abs_kelp = absorptance_kelp /
172 |     frond_thickness
173 | iops%abs_water = abs_water
174 | iops%scat = scat
175 |
176 | !write(*,*) 'iop init'
177 | !iops%vsf_angles = vsf_angles
178 | !iops%vsf_vals = vsf_vals
179 | call iops%load_vsf(vsf_file, fmtstr)
180 |
181 | ! load_vsf already calls calc_vsf_on_grid
182 | !call iops%calc_vsf_on_grid()
183 | call iops%calculate_coef_grids(p_kelp)
184 |
185 | !write(*,*) 'BC'
186 | max_rad = 1.d0 ! Doesn't matter because we'
187 |     11 rescale
188 | decay = 1.d0 ! Does matter, but maybe not
189 |     much. Determines drop-off from angle
190 | call bc%init(grid, solar_zenith,
191 |     solar_azimuthal, decay, max_rad)
192 | ! Rescale surface radiance to match surface
193 |     irradiance
194 | bc%bc_grid = bc%bc_grid * surface_irrad /
195 |     grid%angles%integrate_points(bc%bc_grid)
196 |
197 | write(*,*) 'bc'
198 | write(*,*) bc%bc_grid
199 |
200 | ! write(*,*) 'bc'
201 | ! do i=1, grid%y%num
202 |     write(*,'(10F15.3)') bc%bc_grid(i,:)

```

```

193 ! end do
194
195 call light%init_grid(grid)
196
197 write(*,*) 'Scatter'
198 call calculate_light_with_scattering(grid,
199      bc, iops, light%radiance, num_scatters)
200
201 write(*,*) 'Irrad'
202 call light%calculate_irradiance()
203
204 ! Calculate output variables
205 call calculate_average_irradiance(grid,
206      light, avg_irrad)
207 call calculate_perceived_irradiance(grid,
208      p_kelp, &
209      perceived_irrad, light%irradiance)
210
211 ! write(*,*) 'vsf_angles = ', iops%vsf_angles
212 ! write(*,*) 'vsf_vals = ', iops%vsf_vals
213 ! write(*,*) 'vsf_norm = ', grid%
214      integrate_angle_2d(iops%vsf(1,1,:,:))
215
216 ! write(*,*) 'abs_water = ', abs_water
217 ! write(*,*) 'scat_water = ', scat_water
218 write(*,*) 'kelp'
219 write(*,*) p_kelp(:,:,:)
220 write(*,*) 'ft =', frond%ft
221
222 write(*,*) 'irrad'
223 write(*,*) light%irradiance
224
225 write(*,*) 'avg_irrad = ', avg_irrad
226 write(*,*) 'perceived_irrad = ',
227      perceived_irrad
228
229 write(*,*) 'deinit'
230 call bc%deinit()
231 ! write(*,*) 'a'
232 call iops%deinit()
233 ! write(*,*) 'b'
234 call light%deinit()
235 ! write(*,*) 'c'
236 call rope%deinit()
237 ! write(*,*) 'd'
238 call grid%deinit()
239 ! write(*,*) 'e'
240
241 deallocate(pop_length_means)
242 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, &
250         frond_aspect_ratio, &
251         num_fronds, &
252         pop_length_means, &
253         pop_length_stds)
254
255 implicit none
256
257 ! Number of depth levels
258 integer, intent(in) :: nz
259 ! Number of Superindividuals in each depth
260 ! level
261 integer, intent(in) :: num_si
262 ! si_area(i,j) = area of superindividual j
263 ! at depth i
264 double precision, dimension(nz, num_si),
265 ! intent(in) :: si_area
266 ! si_area(i,j) = number of individuals
267 ! represented by superindividual j at depth
268 ! i
269 double precision, dimension(nz, num_si),
270 ! intent(in) :: si_ind
271 double precision, intent(in) :::
272 ! frond_aspect_ratio
273
274 double precision, dimension(nz), intent(out)
275 ! :: num_fronds
276 ! Population mean area at each depth level
277 double precision, dimension(nz), intent(out)
278 ! :: pop_length_means
279 ! Population area standard deviation at each
280 ! depth level
281 double precision, dimension(nz), intent(out)
282 ! :: pop_length_stds
283
284 ! -----
285
286 integer i, k
287 ! Numerators for mean and std
288 double precision mean_num, std_num
289 ! Convert area to length

```

```

278     double precision , dimension(num_si) :: si_length
279
280     do k=1, nz
281         mean_num = 0.d0
282         std_num = 0.d0
283         num_fronds(k) = 0
284
285         do i=1, num_si
286             si_length(i) = sqrt(2.d0*
287                               frond_aspect_ratio*si_area(k,i))
288             mean_num = mean_num + si_length(i)
289             num_fronds(k) = num_fronds(k) + si_ind
290                 (k,i)
291         end do
292
293         pop_length_means(k) = mean_num /
294             num_fronds(k)
295
296         do i=1, num_si
297             std_num = std_num + (si_length(i) -
298                               pop_length_means(k))**2
299         end do
300
301     end subroutine calculate_length_dist_from_superinds
302
303     subroutine calculate_average_irradiance(grid,
304                                              light, avg_irrad)
305     type(space_angle_grid) grid
306     type(light_state) light
307     real, dimension(:) :: avg_irrad
308     integer k, nx, ny, nz
309
310     nx = grid%x%num
311     ny = grid%y%num
312     nz = grid%z%num
313
314     do k=1, nz
315         avg_irrad(k) = real(sum(light%irradiance
316                               (:,:,k)) / nx / ny)
317     end do
318 end subroutine calculate_average_irradiance
319
320 subroutine calculate_perceived_irradiance(grid,
321                                             p_kelp, &

```

```

319      perceived_irrad, irradiance)
320      type(space_angle_grid) grid
321      double precision, dimension(:,:,:) :: p_kelp
322      real, dimension(:) :: perceived_irrad
323      double precision, dimension(:,:,:) :::
324          irradiance
325      double precision total_kelp
326      integer center_i1, center_i2, center_j1,
327          center_j2
328
329      integer k
330
331      ! Calculate the average irradiance
332          experienced over the frond.
333      ! Has same units as irradiance.
334      ! If no kelp, then just take the irradiance
335          at the center
336      ! of the grid.
337      do k=1, grid%z%num
338          total_kelp = sum(p_kelp(:,:,k))
339          if(total_kelp .eq. 0) then
340              center_i1 = int(ceiling(grid%x%num /
341                  2.d0))
342              center_j1 = int(ceiling(grid%y%num /
343                  2.d0))
344              ! For even grid, use average of center
345                  two cells
346              ! For odd grid, just use center cell
347              if(mod(grid%x%num, 2) .eq. 0) then
348                  center_i2 = center_i1 + 1
349              else
350                  center_i2 = center_i1
351              end if
352              if(mod(grid%y%num, 2) .eq. 0) then
353                  center_j2 = center_j1 + 1
354              else
355                  center_j2 = center_j1
356              end if
357
358          ! Irradiance at the center of the grid
359          ! (at the rope)
360          perceived_irrad(k) = real(sum(
361              irradiance( &
362                  center_i1:center_i2, &
363                  center_j1:center_j2, k)) &
364                  / ((center_i2-center_i1+1) * (
365                      center_j2-center_j1+1)))
366
367      else
368          ! Average irradiance weighted by kelp
369          distribution

```

```
359 |     perceived_irrad(k) = real( &
360 |         sum(p_kelp(:,:,k)*irradiance(:,:,k
361 |             )) &
361 |             / total_kelp)
362 |     end if
363 |   end do
364 |
365 | end subroutine calculate_perceived_irradiance
366 |
367 |end module light_interface
```