1-1-2011

Wind Turbine Sound Propagation Using A Finite-Difference Time-Domain Method

Daniel Wrobel
Ryerson University

Wind Turbine Sound Propagation using a Finite-Difference Time-Domain Method

by

Daniel Wrobel

Bachelor of Engineering - Aerospace Engineering, Ryerson, 2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Applied Science

in the Program of

Aerospace Engineering

Toronto, Ontario, Canada, 2011

I hereby declare that I am the sole author of this thesis.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Wind Turbine Sound Propagation using a Finite-Difference Time-Domain Method

Master of Applied Science 2011

Daniel Wrobel

Aerospace Engineering

Ryerson University

Abstract

Energy usage is on the rise in both Canada and the United States. Because of this, there is a growing demand and strain on the current infrastructure. More importantly though, there is a strong demand for the use of renewable energy sources to meet this demand. One of the most popular renewable energy sources at this time is the wind turbine. In Ontario, there are plans to implement a significant number of them throughout the province. There are concerns though from residents in the vinicity of them that they cause too much noise, as well as health issues. However, some argue that these complaints stem from incorrectly calculated setback distances due to the lack of use of a detailed sound propagation model. In this study, a sound propagation model was developed using a Finite-Difference Time-Domain method, for a three dimensional computational domain, and simulated using data for a Siemens SWT-2.3-101 wind turbine. The simulations produced data of the sound propagation characteristics of each emitted wave, for each tested case. The model was developed as a starting point and building block for the eventual use in simulations of large domains and complex flow phenomena.

Acknowledgements

Most importantly, I would like to thank my advisor, Jason Lassaline, whose guidance and support throughout this project helped me develop a greater understanding of the subject area. Without his assistance, this project would not have been possible.

My sincere thanks to those who have taught and helped me throughout my undergraduate and graduate degrees; your unique approaches to many of the courses I studied assisted me in developing an appreciation and understanding of the fundamentals, which helped me immensely while working on this project.

Computations were performed on the gpc supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto [43].

Computations were also performed on the orca supercomputer at the SHARCNET HPC Consortium. SHARCNET is a partner of the Compute Canada national HPC platform, and is the largest high performance computing consortium in Canada, including 17 universities, colleges and research institutes across southwestern, central and northern Ontario.

Dedication

To my parents, Bob and Vicky, who encouraged me to continue my education and who helped me get through a difficult time after completing my undergraduate degree. Also to my friends, who have given me support and have been there for me.

Contents

List of Tables

List of Figures

List of Appendices

List of Symbols

The following nomenclature is used throughout this text with common S.I. units given if applicable.

Alphanumeric Symbols

A	amplitude
a	order of convergence
b	mass bouyancy
$\tilde{c}$	speed of sound
c	ambient speed of sound
Δx	mesh spacing in x
Δy	mesh spacing in y
Δy	mesh spacing in z
$\mathbf{F}$	force acting on medium
f	frequency
f_1	fine grid sound power level
f_2	normal grid sound power level
f_3	coarse grid sound power level
F_s	safety factor
$\mathbf{g}$	gravity
GCI	grid convergence index
k	wave number
k_0	wave number
k_1	incidence wave number
k_2	transmission wave number
k_c	complex wavenumber
L	length variation in density
L_P	sound power level
M	mach number
m	grid convergence ratio
N	number of grid nodes per wavelength
N_x	number of nodes in x
N_y	number of nodes in y

N_z	number of nodes in z
$\tilde{\mathbf{P}}$	pressure
P	ambient pressure
p	pressure perturbation
P_{ref}	reference pressure
P_{RMS}	root mean square pressure
$\dot{Q}$	monopole strength per unit volume
$\dot{q}$	monopole strength
Q	in-flow rate density
q	volumetric flow rate
Δr	grid spacing
$\tilde{S}$	medium entropy
S	ambient entropy
s	entropy perturbation
s_c	structure constant/effective density factor
t	time
$\mathbf{v}$	ambient velocity
$\tilde{\mathbf{v}}$	medium velocity vector
$\mathbf{w}$	velocity perturbation
w_i	acoustic wave velocity
x_{max}	maximum domain dimension in x
x_{min}	minimum domain dimension in x
x_i	position of observer in space
y_{max}	maximum domain dimension in y
y_{min}	minimum domain dimension in y
z_{max}	maximum domain dimension in z
z_{min}	minimum domain dimension in z
Z_c	characteristic impedance

Greek Symbols

γ	heat capacity ratio
κ_e	effective bulk modulus
κ_p	bulk modulus of air in pores
λ	wavelength
ω	porosity
ρ_p	density of air in pores
σ	static flow resistivity
θ_i	angle of incidence
θ_t	angle of transmission
Λ	turbulent eddy size
ω	angular frequency

ψ	source term
τ	retarded time
θ_1	angle of incidence
θ_2	transmission angle
η	density perturbation
∇	vector gradient
ρ	ambient density
$\tilde{\rho}$	density

Chapter 1

Introduction

The energy demands of Canada and the United States are increasing every year. To alleviate the increasing demand, there are a number of traditional methods to produce energy on a mass scale. These include oil, gas, and coal burning plants, as well as nuclear power. However, in recent years, there has been a strong push towards to use of renewable energy sources. Perhaps the most familiar renewable energy source in Canada, and in particular Southern Ontario, is Niagara Falls. Nevertheless, there are other potential sources of energy which can utilized. In Ontario, solar energy appears to have a bright future. However, wind energy has seen a significant increase in use as well. Being a renewable energy resource relying on wind, it has a significant potential in Ontario.

The use of wind turbines has caused some controversy though, in the form of complaints from residents living in the vicinity of them. Many of these residents have complained that the turbines produce too much noise, and that they also cause health problems. Nonetheless, the amount of noise produced by the turbines has been decreased since their inception with design changes. Further, to alleviate the noise issues, setback distances have been established by governing bodies to aid builders in positioning the turbines in relation to homes. Using the setback distances however, residents still complain of noise and health issues. Therefore, it would be logical to assume that perhaps the setback distance is not quite adequate. In addition, the methods used to calculate the setback distance may also need to be re-evaluated.

Hence, it was worthwhile to consider a more complex model to examine the sound propagation characteristics of wind turbines. Work has been done in these areas, but the methods used to obtain the results (Finite Volume Method (FVM) and Finite Element Method (FEM)) have not always been satisfactory, according the respective authors. Therefore, this study considered the use of a Finite-Difference, Time-Domain (FDTD) method for developing a sound propagation model to predict noise from wind turbines. From past studies, several authors have metioned that this method is recommended for complex and detailed simulations, and that future work is recommended to be done using it in wave propagation situations. Hence, to aid in the development of the model, it was subdivided into two separate areas: the development of the domain and sound propagation model with absorbing boundaries (using the FDTD method), and the development of the sound generation model.

To explore the applicability of the FDTD method to wind turbine applications, geometrical data was obtained for a Siemens SWT-2.3-101 wind turbine; this turbine is the most commonly installed in Ontario. Simulations were performed by using a variety of different ground conditions, as well as source and flow situations. The results that were obtained clearly showed the effects varying ground conditions and flow profiles on absorption and the propagating sound wave from the source, as well as on the differences in propagation between using a monopole and dipole to model a wind turbine. As expected, the ground conditions with lower porosity values tended to be absorbed less, in comparison to those with higher porosities. As well, it was determined that the monopole sources were less effective at radiating sound away in comparison to the dipoles. The dipole was able to maintain a Sound Pressure Level (SPL) which was less affected by the absorption characteristics, in comparison to the monopole. Finally, in using differing flow conditions, it was determined that the effects of the flow profile affected the SPL of the sound being emitted; that is, for a uniform flow, the SPL tended to be higher at the ground, in comparison to a power law flow. As well, it was evident that the flow velocity had an effect on the sound propagation.

1.1 Organization

This document has been divided into four chapters, which contain the main matter of the study. The first chapter is the introduction and outlines the current energy situation in Ontario, as well as some of the issues relating to the development of wind turbines in the province. The section concludes by introducing the current method of calculating setback distances used in Ontario, as well as discussing existing developments in sound propagation models for wind turbines.

The second chapter introduces the methodology which was used to develop a new sound prediction model. It outlines the configuration of the sound propagation, boundary condition, and sound generation models which were used to produce data.

The third chapter provides an analysis of the results which were obtained from the study. The chapter also includes the validation which was performed to test the correct operation of the model.

The fourth chapter concludes the document with a discussion of the results. It also provides a discussion of how the results and study itself could be improved upon.

1.2 Background

Energy production is a very important topic in engineering. Research is constantly being carried out to develop new forms of energy production, as the worlds energy needs are changing every year [9]. In using renewable energy sources such as wind turbines however, there are some compromises; perhaps the most significant of these is annoyance [3], which tends to spread a negative general perception of the technology. The numerical sound propagation model which was developed in this paper can be used to alleviate the adverse effects and potentially solve the issues relating to noise around turbine installations.

In Ontario, there is currently an installed capacity of approximately 36,000 Megawatts (MW), with

4,600 MW [24] of potentially imported power being available. Of this energy reserve, the largest contribution is from nuclear power. Although this reserve can definitely supply Ontario with an abundance of electricity, some significant usage milestones have been reached in the last decade; in August 2006 a record summer peak need of 27,005 MW was reached, and in December 2004 a record winter peak of 24,979 MW was reached [24]. To deal with current and future demand, Ontario has proposed to put into effect 5,000 MW of additional renewable energy production. This would be comprised of solar, hydro, wind, biomass, biogas, and landfill gas derived energy [24]. It should be noted that a significant amount of energy is produced in the United States as well; this amount is expected to increase to meet demand [47]. One can therefore logically conclude that the future demand for energy is a significant issue in both the United States and Canada. It is logical to assume that as demand increases in Canada, the United States follows, or vice versa, which can lead to supply and demand issues.

1.2.1 IESO and its role in Ontario

The governing body of energy production standards in the province is known as the Independant Electricity System Operator (IESO) (originally known as the Independant Electricity Market Operator (IMO) until 2005). A not-for-profit organization, it was initially put into place in 1998 following the deregulation of the electricity market in Ontario [7]. Before the de-regulation, the provinces electricity needs were handled by Ontario Hydro. Furthermore, with the establishment of the IMO, Ontario followed suite with other power providers in North America to establish an integrated energy network composed of several different independent operators (ie. Hydro Quebec, New York Indepedant System Operator, Indepedant System Operator of New England, etc, each comprising a part of the North American power grid) (Fig. 1.1) [8]. In this province, IESO essentially oversees Ontario Power Generation (OPG), Hydro One (HO), and other local distribution companies. To handle the research and analysis required to assess system performance and determine future demand, in 2004, with the introduction of Bill 100 - Electrical Restructing Act in Ontario, the Ontario Power Authority (OPA) was created [14]. It essentially oversees long term power supply needs and is also in place to ensure that an energy conservation culture in Ontario is followed.

1.2.2 Wind Turbine Development

The use of wind turbines to produce energy is seen as a very promising technology now and into the future [36]; this is largely because wind is a renewable resource. In Ontario, there is a very strong demand for wind turbines. In fact, a mandate has been set by the IESO and the government for the future of energy production in the province. In this mandate, the IESO states that the goal of it and the government are to eliminate coal-fired power plants, and to focus significant efforts on producing substantially more power through more environmentally-friendly means [2]. Further, this 'green' energy plan calls for exceptional developments in wind turbines; by 2025, the country alone could have wind turbines produce nearly 20 percent of all energy [61]. Ontario has indeed already seen a small increase in wind turbine development, increasing from approximately one percent in 2007, to three percent of all renewable energy produced in 2009 [39]. Shown in Figure 1.2 below is one of the projected amounts of

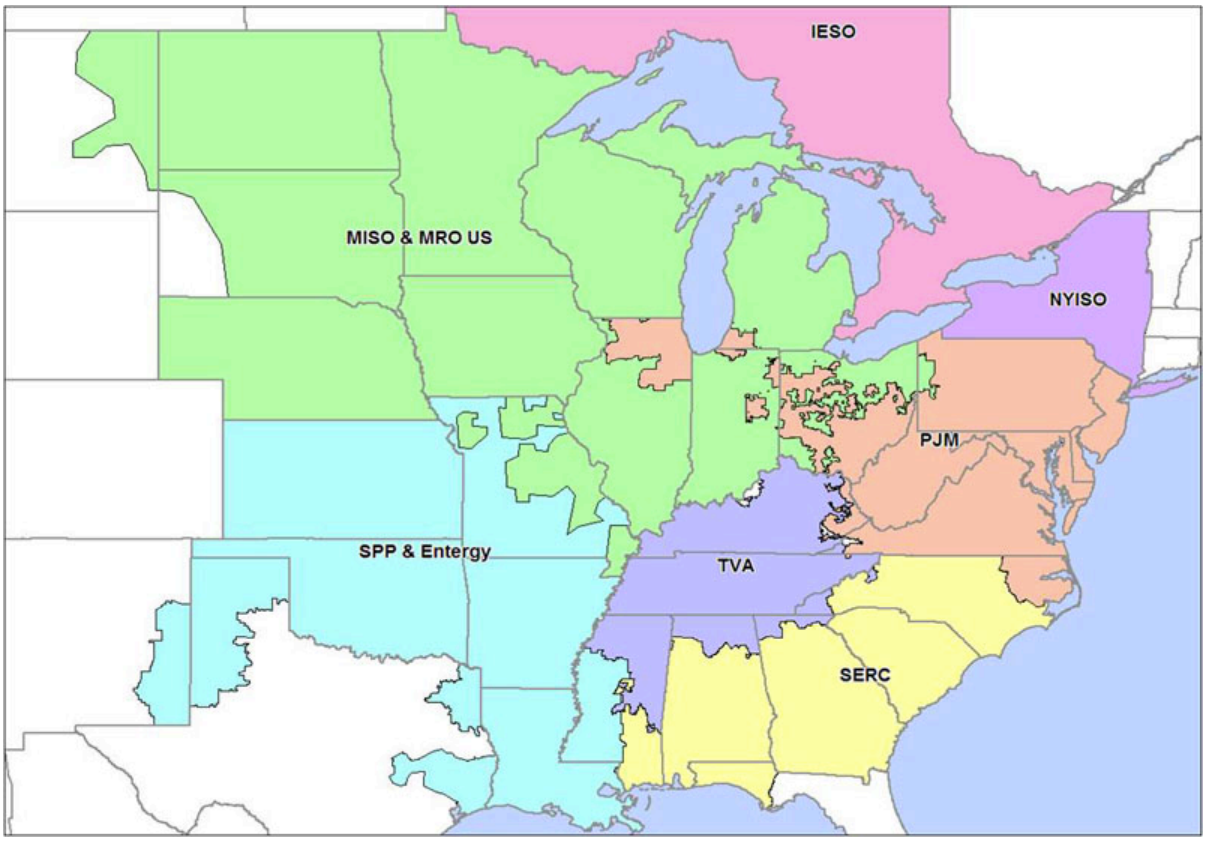


Figure 1.1: *IESO Boundaries and Partner Operators [12]*

development set to take place into the future.

Clearly, the province of Ontario is focused on harnessing renewable energy into the future. Throughout the province there are currently approximately 1,100 MW of installed capacity, with about 475 MW being produced at a given time on average [10]. The locations in Ontario which have had installations take place and which are producing energy are listed in Table 1.1.

Two projects in Ontario which are currently drawing ongoing attention are those proposed by Southpoint Wind. Both of the projects would be a first in Ontario, developing an offshore wind turbine farm, along the shorelines of Lake Erie and Lake St. Clair. The first project, slated to generate 30 MW of electricity, was to be developed between the Leamington and Kingsville townships in Essex county. Although there was no specific location determined, there were several proposed to the residents in the area [5]. The second project, slated to generate a significantly larger 1,400 MW of electricity, has potentially more locations due to the amount of power being generated. The areas proposed included sites near Leamington, Amherstburg, Lakeshore, and Chatham-Kent counties [4]. However, it should be noted that at the time of publication of its report, the projects had not yet been approved by the

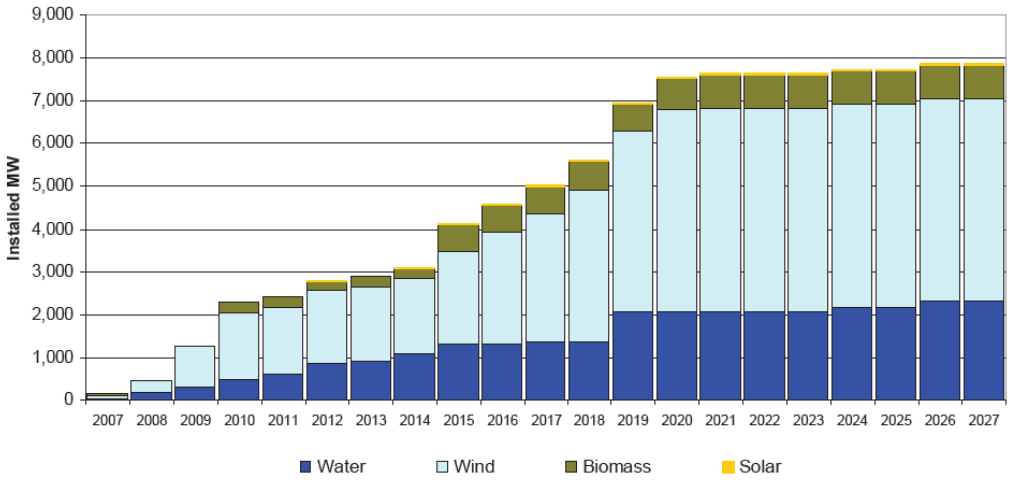


Figure 1.2: *Future Renewable Energy Production in Ontario [39]*

Name/Location	Capacity [MW]	Operational Date
Amaranth I - Township of Melancthon	67.5	March 2006
Amaranth II - Township of Melancthon	132	November 2008
Kingsbridge I - Huron County	39.6	March 2006
Port Alma- Port Alma	101.2	October 2008
Port Burwell - Norfolk and Elgin Counties	99	May 2006
Prince I - Sault Ste. Marie	99	September 2006
Prince II - Sault Ste. Marie	90	November 2006
Ripley - Township of Huron-Kinloss	76	December 2007
Underwood - Bruce County	181.5	February 2009
Wolfe Island - Township of Frontenac Islands	197.8	June 2009

Table 1.1: *Installed Wind Farms in Ontario [10].*

government and Southpoint Wind was still waiting for approval to proceed [11]. Some other projects being developed are listed in Table 1.2.

However, Ontario is not the only region which is taking advantage of renewable energy production through wind turbines. Projects have been developed across the United States. As well, in Europe, wind turbines have become a significant source of energy production [47]. As a further incentive for those who have turbines installed nearby their homes, compensation is usually offered [28].

There are drawbacks however to the development of wind turbines. Some of the strongest issues that have been proposed by critics of the technology deal with the effects of the turbines on their surroundings; namely, the wildlife and the general publics health. Studies have suggested that although the energy production aspect would not affect the local surroundings, there could potentially be an effect from vibrations due to the spinning blades [5]. Furthermore, although not stated in the SouthPoint study, residents have complained in various installation locations that they have become ill and that there are

Name/Location	Capacity [MW]	Operational Date
Bow Lake Phase I	20	Q2 2012
Chatham Wind Project	99.4	Q1 2011
Comber East Wind Project	82.8	Q3 2011
Comber West Wind Project	82.8	Q3 2011
Conestoga Wind Energy Centre I	69	Q4 2011
Greenwich Wind Farm	98.9	Q3 2011
Gosfield Wind Project	50.6	Q4 2011
McLean's Mountain Wind Farm I	50	Q3 2011
McLean's Mountain Wind Farm II	10	Q3 2011
Pointe Aux Roche Wind	48.6	Q3 2011
Raleigh Wind Centre	78	Q1 2011
Summerhaven Wind Energy Centre	125	Q1 2012
Talbot Wind Farm	98.9	Q1 2011

Table 1.2: *Wind Farm Development Projects in Ontario [10].*

adverse health effects from turbines; at this time however, studies have been performed which state there are no health effects, but rather a potential annoyance to residents [3]. Another effect, although mentioned that it would not be a factor in the development plan drafted by SouthPoint Wind, is in regards to bird strikes. They can potentially fly into the spinning blades; however, the birds learn to recognize that a new structure has been erected in a certain place and tend to avoid it after that [4]. A potential hazard also exists with bats; it has been suggested that their fatalities near turbines may be related to barotrauma that occurs to their being in close proximity to the blades when hunting for insects. As the bats approach the blades, they enter a region of low pressure which causes a significant pressure drop inside their lungs, leading to internal haemorrhaging [18].

From the issues which have been bought forth in regards to wind turbines, unwanted noise is one of the strongest complaints because sound plays an important role in the lives of many people everyday. Sound itself can be seen described as a pressure wave causing a perturbation in a medium at rest. To interpret sound, a vibration in our ear drum occurs which is then recognized by the brain as what we 'hear'. How 'loudly' one perceives that sound can be described by an amplitude, and its pitch by a frequency [54]. As seen below (Fig. 1.3), the frequency is related to the compression or expansion of the wave. Compressed regions (compression) signify an increase in density, while expanded regions (rarefaction) show a decrease in density. In relation to wind turbines, a general negative perception of noise from them tends to increase complaints and annoyance in comparison to those who take a positive outlook on the technology [3].

Turbines themselves are made up of several different components and typically have a lifespan of 20 to 30 years; these main components include the rotor and shaft, gearbox, generator, control system, tower, support, and foundation. An example of an installation at Erie Shores is shown in Figure 1.4. In its entirety, the completed assembly weighs approximately 200 tonnes. The turbine works by converting wind (kinetic) energy to electrical energy; the amount of available energy depends on the wind speed

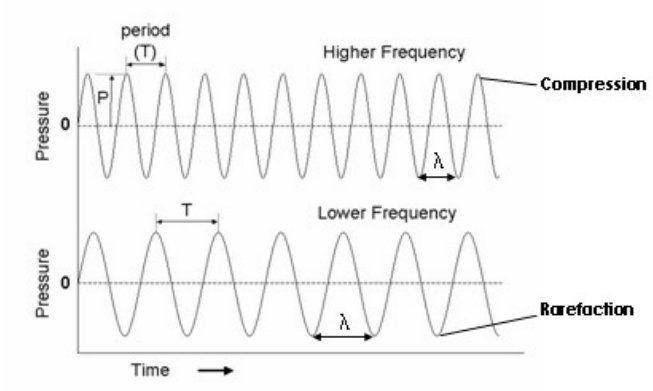


Figure 1.3: *A Sound Wave* [15]

and density of the air. A significant advantage of using wind energy is essentially that it is a pollution free alternative to other energy sources, is a renewable resource in abundance, and typically caries a financial incentive for the land owner on which the turbine is installed. For the turbines installed at Erie Shores, a cut-in speed of approximately nine kilometres per hour at the rotor hub is needed to generate a usable amount of electricity [13].

1.2.3 Development Issues

Before wind turbine development takes place, a build site will undergo different assessments to try to minimize the impact on the surroundings and determine the benefits of installing in that location [28]. In Ontario, upon completing the assesments, a setback distance is used to determine how far away the turbine is to be built from any nearby receiver. The currently established setback distance is 550 m [6]; the value has been established by the government depending on what type of renewable energy project is installed [6](Table 1.6). This is seen by some as being too small; a recently settled case in such a dispute involved the Ontario provincial government vs. Ian Hanna (March 2011). Hanna objected to the established setback distance. He claimed that medical studies show that there are ill-health effects from the low-frequency noise turbines emit (causing issues such as sleep deprivation, stress, and depression). In addition, Hanna also went as far as saying that Ontario's environment minister ignored scientific evidence of such issues resulting from turbine installations [30].

Upon going to court however, Hanna's claim was dismissed. The established panel of judges claimed that, "The ministerial review included science-based evidence, such as reports of the WHO and the opinions of acoustical engineering experts". Despite the outcome, Hanna and his group of supporters pledge to focus efforts on challenging other newly developing wind farms. In particular, Chatham-Kent, an installation of eight turbines rated to produce 20 MW total output at peak capacity [30].

Nevertheless, wind turbine installations may face continued opposition due to the negative perception of the technology by many people. In addition, in a recent announcement of new subsidized wind farm projects, the government only approved four out of a potential forty development projects [30].

Figure 1.4: *Erie Shores Wind Turbine Farm*

1.2.4 ISO Sound Propagation Model

The noise issues brought forth earlier are a consequence of a loaded rotating blade in a moving fluid; the problem however exists in that the currently available tools are not always able to accurately and effectively determine the sound propagation characteristics from an installation. Therefore, the distances currently used for separating the turbines from residences may not be sufficient [33]. The method currently used in determining turbine placement and setback distances in Ontario is performed by using International Organization for Standardization (ISO) 9613-2: Acoustics - Attenuation of sound during propagation outdoors and the Environmental Protection Act - 359/09. These documents are used as a guideline when developing new turbine farms in Ontario; the former containing guidelines for determining setback distances and sound characteristics at a receiver location, and the latter containing additional guidelines for receiving approval for a renewable energy project. The ISO document was developed as a summary of the standards ISO 1996 and ISO 3740 to produce a comprehensive method of describing noise in outdoor environments as a result of various sources. ISO 9613-2 can be applied to a variety of general situations and it determines sound pressure levels based on octave bands [1]. The octave bands are defined as the space between frequencies; the upper limit of a band is two times the lower limit. The

bands that are considered are shown below (Table 1.3).

Octave	Frequency [Hz]
1	63
2	125
3	250
4	500
5	1000
6	2000
7	4000
8	8000

Table 1.3: *Octave Bands*

To measure sound, different weighting schemes are used; this is because our perception of sound is not linear. By using a particular weighting scheme, a different range of frequencies are considered to better approximate the perception of sound. Because a sound can be percieved differently, there are multiple weighting schemes that can be used (Fig. 1.5). In Figure 1.5, each of the weighting schemes are labelled 'A', 'B', 'C', and 'D'. For humans ears, sounds can be perceived from approximately 16 Hertz (Hz) to 16 Kilohertz (kHz) [46]. The A-weighted scheme (dB(A)) measures sound in a range which humans ears are most accustomed to [36].

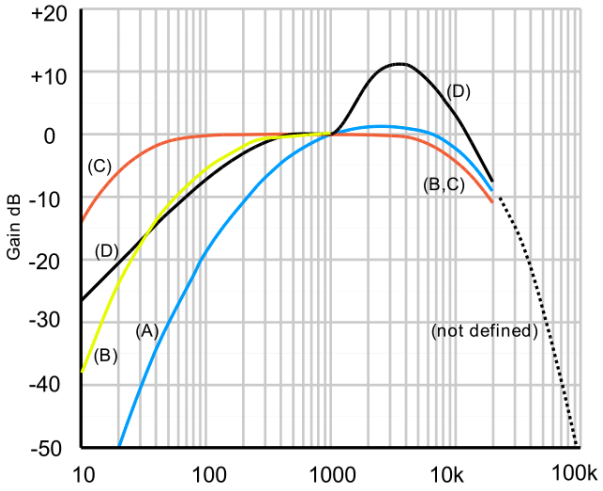


Figure 1.5: *Sound Power Level Measurements Curves* [56]

ISO 9613-2 relies on point sources to determine predictions of noise levels. If one encounters a variety of point sources within an area however, it is acceptable to combine these points into a cell. These combined sources can then be represented as an individual point which can be used for analysis. To determine the amount of noise at the receiver, it is necessary to calculate each of the individual octave bands. A directivity correction is also applied based on how much deviation takes place in the

direction of propagation. An attenuation value is applied based on the amount of sound lost between the source and receiver. The different types of attenuation used are listed in Table 1.4.

Attenuation Type	**Description**
Divergence	Accounts for the spherical spreading from the source.
Atmospheric	Results due to absorption of sound from the atmosphere.
Ground (limited to flat ground, horizontally oriented or sloped)	Sound reflections and absorption takes place depending on the type of surface over which it propagates.
Barrier	The attenuation resulting from barriers in place between the source and receiver. The sound is assumed to travel either through a vertically or horizontally oriented edge.
Miscellaneous	Defines additional attenuation criteria in addition to the primary ones; consists of attenuation from foliage, industrial sites, and housing.

Table 1.4: *ISO 9613-2 Attenuation Types* [1]

Upon determining the attenuation, it is then possible to calculate the true sound pressure level for a given octave; these bands are combined to form a sound. This process needs to be completed for each point source. Furthermore, corrections can be performed which allow one to encompass a wide variety of meteorological conditions.

However, there are several constraints on the use of the ISO method:

1. The sound propagation from the noise source take place under a ground-based temperature inversion.
2. It is not possible to take into account sound attenuation over water.
3. The method cannot be expected to work on any given day; the conditions for which it encompasses are averaged over independent situations.
4. The estimated errors in the average receiver sound pressure levels and pure tone levels maybe larger than for A-weighted levels of broadband noise sources.

There are specific conditions which must be satisfied when using the ISO model. If the ambient conditions differ from those listed in Table 1.5, then the results of the sound propagation model may not be as accurate as one would hope. To obtain the most accurate results, one should ensure the testing conditions satisfy those defined in the ISO model as close as possible.

Condition	**Limits**
Wind Direction	Must be within 45 degrees of a line connecting the point source and the receiver.
Wind Speed	1 to 5 m/s
Height of Measurement	3 to 11 m

Table 1.5: *ISO 9613-2 Conditions for Propagation* [1]

Class	Location	Capacity [kW]	Greatest Sound Power [dB(A)]
1	At a location where no part of a wind turbine is located in direct contact with surface water other than in a wetland.	<= 3	Any output
2	At a location where no part of a wind turbine is located in direct contact with surface water other than in a wetland.	>3, <50	Any output
3	At a location where no part of a wind turbine is located in direct contact with surface water other than in a wetland.	>=50	<102
4	At a location where no part of a wind turbine is located in direct contact with surface water other than in a wetland.	>=50	>= 102
5	At a location where one or more parts of a wind turbine is located in direct contact with surface water other than in a wetland.	Any output	Any output

Table 1.6: *Ontario Wind Turbine Installation Sound Power Limits* [6]

One of the largest disadvantages of the ISO method, is it does not give a detailed visualization or description of the overall noise produced from the source as it is geometrically based and does not rely on any type of CFD code. It is a rather simple method of obtaining a general prediction of sound propagation under favourable propagation conditions. It is an emperical method based on emperical data, which have been developed for a set of particular situations.

In dealing with noise propagation in a real-world scenario, distance and noise emission standards have been established by governments as to how much of a buffer zone can exist between the source and receiver. In Ontario, guidelines have been established based on the type of installation to be put into effect based on the facility location, capacity, and sound power level output (Table 1.6). These guidelines exist in the Environmental Protection Act - 359/09. Further, based on the established classifications, setback distances have also been set. A setback distance of 120 m exists for Class I and II installations, while 550 m is used for Classes III,IV, and V [6].

1.2.5 Past development

Significant research has been performed in developing sound propagation models to be used in situations where wind turbines are in place. Below are some methods which have been developed using different techniques to predict sound.

In 'A 3D Parabolic Equation Method for Wind Turbine Noise Propagation in Moving Inhomogenous Atmosphere' [22], a Parabolic Equation (PE) method was developed using the Helmholtz equation in an inhomogenous atmosphere with rigid boundaries. Although the equation is developed in cartesian coordinates, it was reformulated in the cylindrical domain and then solved using the Crank Nicholson (CN) time marching scheme, while assuming incompressible flow. The PE method however required a

significant amount of computational resources in three dimensions. In addition, several simplifications were introduced into the setup to simplify the problem to allow only the use of symmetry, a single test frequency, and not including a density gradient for the domain.

A study was performed in 'An Aerodynamic Noise Propagation Model for Wind Turbines' [65], on turbines which are installed offshore and in open/flat ground conditions. The analysis involved using varying ground conditions as well as air attentuation effects. A detailed noise generation model was used, describing the noise of the airfoil/blade. It took into account geometrical spreading, sound directivity, air absorption, effects of terrain, temperature, and the effects of wind. However, geometrical sound ray theory was used as the propagation model which is not as accurate as the FDTD method, as mentioned by the author in their work; the developed method is better suited for quick and simple simulations.

Finite element methods were used in 'Finite Element Simulation of Sound Propagation Concerning Meteorological Condition' [49]. The study was useful in that it presented situations in which propagation was coupled with atmospheric effects such as wind and temperature gradients. Further, two different solution methods were considered (using a Fast Fourier Program (FFP) or a PE solver). While the FFP was considered, it is normally used for oceanic studies as it is able to deal with multiple frequency bands with ease, but only works for situations entailing a homogenous domain. The PE method was considered to be a better choice, but was said to be computationally expensive. As well, propagation characteristics were only calculated in one direction as opposed to both. Nevertheless, the methods employed are more accurate then employing a simple ray theory method. Ultimately, the method of solution used the Galerkin finite element spatial discretization and backward Euler time marching technique, with a focus on analysis of low frequency propagation. The final results produced were in a two dimensional domain, using a flat plate oscillator as a source. The results also assumed the wind profile did not change during propagation.

The methods used in 'Integrated Numerical Method for the Prediction of Wind Turbine Noise and the Long Range Propagation' [58], utilized a considerably detailed model for sound generation, taking in account each of the specific noise mechanisms along a rotating blade (though mechanical noise was not considered, aerodynamic sources included low frequency, turbulent in-flow, and airfoil self-noise). As well, a spherical oscillator was used for the noise source. Unlike other models, the authors proposed using a database of pre-determined values for the sound generation model, to reduce the amount of computational time. The sound generation model data would then be fed into the propagation model, which would apply any type of attenuation or dispersion effects. Although the other considered studies in this paper utilized flat surfaces, tests using GIS data from a complex terrain mesh were tested. With the use of the complex terrain, the author made mention that it would be beneficial to test in the future using an FDTD setup. Unfortunately, this study used a propagation model utlizing ray theory, leading to a very simplified analysis despite the accurate source data. In addition, the model did not include any wind or a temperature gradient.

Ray theory was used again in 'Noise Propagation Issues in Wind Energy Applications' [53] to simplify analysis. Validation was performed against real-world turbine data for a specific ground condition (wet grass). It was concluded that for complex terrains, simple models of propagation cannot be used to obtain accurate and detailed results.

In 'Wind Turbine Noise Propagation over Flat Ground: Measurements and Predictions' [26], several different models were examined to see how they compared to empirical data using different solution methods (geometrical ray theory, and using a PE solver with the CN time marching method). The study also considered varying ground conditions based on different values of a ground resistivity coefficient. The author went on to mention the potential significant benefits of using a FDTD in comparison to others for future work.

In considering the work of previous authors, several of the studies indicate that the use of the FDTD method would indeed produce the most accurate results and would be of benefit to study. Furthermore, by implementing rigid and absorbing boundary conditions in a three dimensional setting, as well as a detailed sound generation model, one could develop a detailed model for noise propagation downstream from a source.

Chapter 2

Methodology

This section was divided into three subsections for convenience; the first section considers the sound propagation model, the second considers the sound generation model, while the third goes over the code implementation and development. The three steps led to the development of the completed prediction model. Before introducing the main matter however, a brief overview of the FDTD method is given.

The FDTD method was developed in 1966 by Kane Yee; it was originally formulated to solve the Maxwell equation. Over the years, the use of the FDTD method has been increasing; this is in part due to the increasing amount of computing power available. It is seen as a fast method of solving equations of N Degrees of Freedom (DOF), where each iteration requires a number of operations of order O(N). Futhermore, the method is able to accurately solve problems involving models such as those involving inhomogenous media, as well as those using an unbounded domain requiring the implementation of absorbing boundary conditions (from absorbing media to Perfectly Matched Layers) [27].

Nevertheless, there are some shortcomings to using the FDTD method. Gedney [27], presents a comprehensive list of the drawbacks. Perhaps the largest of the disadvantages involves the need to discretize the entire domain in question. The entire discretization may result in extra computational effort being exhausted, instead of focusing on the area under study; if using a three dimensional domain (which increases DOF significantly) with a highly accurate model and small timestep, the computational expense in terms of memory will be drastic. Another disadvantage of using a FDTD method deals with the broadband solution it gives; this may be of issue if limited information is only available about a narrowband, potentially requiring an estimate. As well, when using the method in particular with a large source strength, lengthy run times may result as narrowband resonances may take a significant period of time to decay, or may not decay at all. Finally, it should be mentioned that the use of a FDTD method requires the use of an orthogonal grid to encompass the domain.

2.1 Implementation

It was necessary to specify a volume in which the domain of the problem being solved would be defined (Fig. 2.1). For the purposes of this study, a rectangular-prism shaped Control Volume (CV) was used, being represented by a structured staggered grid. The grid itself could be varied based on the users desire; however, it was advisable to keep the spacing between nodes equal in all three coordinate directions (ie. $\Delta x = \Delta y = \Delta z$) to maintain equivalent spatial accuracy in all directions.

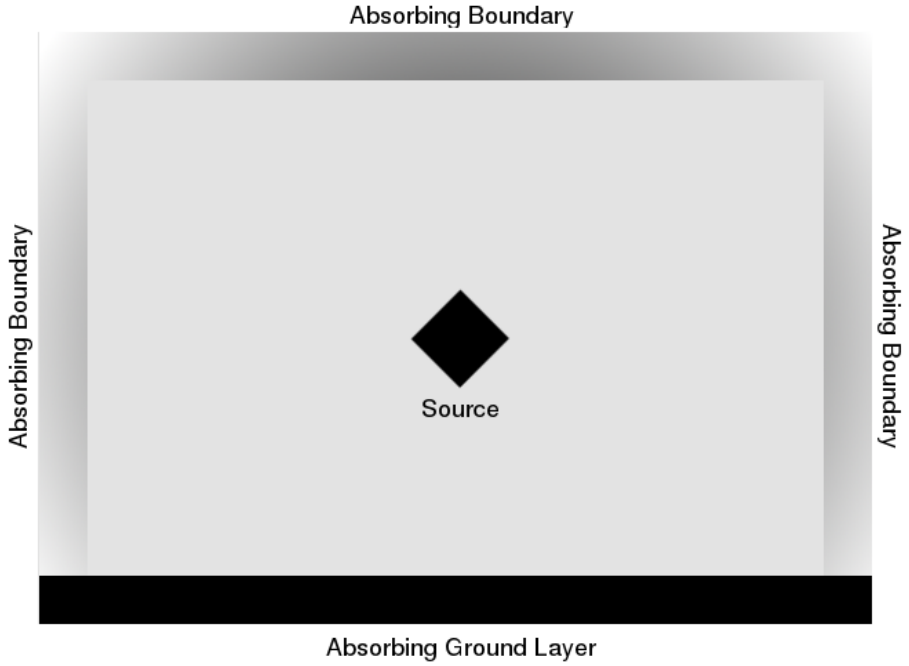


Figure 2.1: *Control Volume Cross-Section*

As a real world example, the setup used was such that it resembled a space of air, where the ground plane was defined as being rigid or semi-permeable, while the other boundaries of the cube were taken as being absorbing.

2.1.1 Domain Configuration

In Ostashev et al. [50], a method of modelling the sound propagation from a source was discussed. That method was expanded upon in this paper. The use of a FDTD technique is emphasized over that of parabolic or frequency domain equations, due to their simpler solution, yet more accurate results and wider range of applications. In the two final sets of equations developed, the first makes no assumptions, while the second ignores terms proportional to the divergence of the medium velocity and the gradient of the ambient atmospheric pressure (as the change in the medium velocity and ambient pressure is not significant).

The development of the algorithm begins with the fundamental equations of fluid dynamics: the continuity equation, momentum equation, adiabatic equation of motion, and equation of state, respectively,

$$\left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{\rho} + \tilde{\rho} \nabla \cdot \tilde{\mathbf{v}} = \tilde{\rho} Q \tag{2.1}$$

$$\left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{\mathbf{v}} + \frac{\nabla \tilde{P}}{\tilde{\rho}} - \mathbf{g} = \frac{\mathbf{F}}{\tilde{\rho}} \tag{2.2}$$

$$\left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{S} = 0 \tag{2.3}$$

$$\tilde{P} = \tilde{P}(\tilde{\rho}, \tilde{S}) \tag{2.4}$$

In the above equations, $\tilde{P}$ is the pressure, $\tilde{\rho}$ is density, $\tilde{v}$ is the velocity, and $\tilde{S}$ is the entropy. A perturbation method is used to linearize the above equations, where

$$\tilde{P} = P + p, \tag{2.5}$$

$$\tilde{\rho} = \rho + \eta, \tag{2.6}$$

$$\tilde{\mathbf{v}} = \mathbf{v} + \mathbf{w}, \tag{2.7}$$

$$\tilde{S} = S + s. \tag{2.8}$$

The P, ρ, $\mathbf{v}$, and S represent the values of the ambient conditions. The variables p, η, $\mathbf{w}$, and s respresent the perturbations due to a propagating sound wave. It should be noted that the term 'perturbation' has the same meaning as an 'acoustic' quantity; they can both be used interchangeably. Substituting the above into the fundamental equations of fluid dynamics, applying an argument of scale to eliminate higher order terms, and setting $(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla) = \frac{d}{dt}$, one would obtain

$$\frac{d\eta}{dt} + (\mathbf{w} \cdot \nabla)\rho + \rho \nabla \cdot \mathbf{w} + \eta \nabla \cdot \mathbf{v} = \rho Q, \tag{2.9}$$

$$\frac{d\mathbf{w}}{dt} + (\mathbf{w} \cdot \nabla)\mathbf{v} + \frac{\nabla p}{\rho} - \frac{\eta \nabla P}{\rho^2} = \frac{\mathbf{F}}{\rho}, \tag{2.10}$$

$$\frac{ds}{dt} + (\mathbf{w} \cdot \nabla) S = 0, \tag{2.11}$$

$$p = \eta c^2 + hs. \tag{2.12}$$

The h term above is defined by $\frac{\partial P(\rho,S)}{\partial S}$. The above equations describe the most general case of sound propagation in a moving inhomogenous medium with only one component; one should note, however, that the above equations do differ from those used for aeroacoustics by way of the sound sources used. In the equations above, the **F** and Q terms represent known sources of force and in-flow rate density respectively, where in aeroacoustics the sources have to be calculated from the flow functions and are part of the ambient flow. Finally, in some cases it may be desirable to assume that the entropy is constant throughout the medium under study. This may not be done if one considers a stratified medium where the ambient density varies such that it is smaller than the acoustic wavelength or varies significantly with a change in height; the wavelength can be defined as $\lambda = \frac{c}{f}$, with frequency f. Continuing with the derivation, a set of three coupled equations were produced to solve for velocity, pressure, and density, in a moving medium with an arbitrary equation of state. Hence, first using the equation of state and applying an operator to both sides

$$\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla. \tag{2.13}$$

Using the adiabatic equation of motion, one can then obtain

$$\left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{P} = \tilde{c}^2 \left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{\rho}. \tag{2.14}$$

Finally, applying the continuity equation,

$$\left(\frac{\partial}{\partial t} + \tilde{\mathbf{v}} \cdot \nabla\right) \tilde{\mathbf{P}} + \tilde{c}^2 \tilde{\rho} \nabla \cdot \tilde{\mathbf{v}} = \tilde{c}^2 \tilde{\rho} Q. \tag{2.15}$$

Now, it is necessary to perform linearization on the above, but due to the sound speed term, it is necessary to perform several intermediary steps before proceeding. It is first required to determine the value of the speed of sound with first order accuracy using the perturbations defined earlier to bring it into line with acoustic quantities. Defining the speed of sound term as

$$\tilde{c}^2 = \frac{\partial \tilde{P}(\tilde{\rho}, \tilde{S})}{\partial \tilde{\rho}}. \tag{2.16}$$

Then, proceeding to decompose the sound speed into a Taylor series while keeping the perturbations of density and entropy in the first order

$$\tilde{c}^2 = \frac{\partial}{\partial \rho} P(\rho + \eta, S + s) = \frac{\partial P(\rho, S)}{\partial \rho} + \frac{\partial^2 P(\rho, S)}{\partial \rho^2} \eta + \frac{\partial^2 P(\rho, S)}{\partial \rho \partial S} s. \tag{2.17}$$

One can take notice that the first term on the right, c^2, is equal to the ambient sound speed. The second and third term can be taken as the perturbations in the sound speed. These assumptions, and using α and β as placeholders, lead to the development of

$$\tilde{c}^2 = c^2 + (\beta\eta + \alpha s). \tag{2.18}$$

Now, using the above and isolating the second term on the right, one can substitute for the entropy

perturbation s, using the linearized equation of state developed earlier

$$s = \frac{p - c^2\eta}{h}. \tag{2.19}$$

Upon performing the substitution, one obtains a formula for the sound speed perturbation

$$(c^2)' = (\beta - \frac{\alpha c^2}{h})\eta + \frac{\alpha p}{h}. \tag{2.20}$$

The linearization of equation 2.15 can now take place, noting that $\tilde{c}^2 = c^2 + (c^2)'$. This is done using the same perturbations used for the linearization of the equations of fluid dynamics earlier. Upon completing the linearization, one would find that

$$\frac{dp}{dt} + \rho c^2 \nabla \cdot \mathbf{w} + \mathbf{w} \cdot \nabla P + \left[\rho\beta + c^2\left(1 - \frac{\alpha\rho}{h}\right)\right]\eta + \left(\frac{\alpha\rho}{h}\right) p \nabla \cdot \mathbf{v} = \rho c^2 Q \tag{2.21}$$

The above equation 2.21 as well as 2.9 and 2.10 make a set of three coupled equations with no approximations. This set of equations can be used for sound propagation prediction in a moving inhomogenous atmosphere using FDTD techniques. The equations themselves are used to solve for perturbations in pressure, velocity, and density.

The three coupled equations produced above however, can be reduced to a simpler set of two coupled equations if the ambient velocity is taken to be less than the sound speed. This is done by using the linearized momentum equation and equation 2.21, in terms of pressure change with time. This allows one to obtain two equations, one to solve for pressure perturbations, the other for velocity perturbations. However, some assumptions must be made. Using [41], it can be shown that

$$\nabla \cdot \mathbf{v} \sim \frac{v^3}{c^2 L}, \tag{2.22}$$

Where L represents the variation in density. Using the above, the terms proportional to $\nabla \cdot \mathbf{v}$ can be eliminated to the order of $\frac{v^2}{c^2}$. The terms which are proportional to ∇P, can also be eliminated as they are proportional to $\frac{v}{c}$. Finally, because the developed equations for this paper are used with acoustic quantities, the effects of gravity can be omitted. Therefore, the equation for pressure equation above (eq. 2.21) and the linearized momentum equation can be written as

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla\right) p + \rho c^2 \nabla \cdot \mathbf{w} = \rho c^2 Q, \tag{2.23}$$

$$\left(\frac{\partial}{\partial t} + \mathbf{v} \cdot \nabla\right) \mathbf{w} + (\mathbf{w} \cdot \nabla)\mathbf{v} + \frac{\nabla p}{\rho} = \frac{\mathbf{F}}{\rho}. \tag{2.24}$$

The two above equations are coupled and can be used to solve for the acoustic pressure and velocity. They are simpler and require less information (no need for P, α, β, and h) to use in comparison to the three coupled equations presented earlier. They can be solved using FDTD techniques and can be used for predicting sound in a moving inhomogenous atmosphere. Before the equations could be solved

numerically however, it was necessary to isolate the partial derivatives with respect to time; the terms on the right of the equality could then be approximated using finite-differences. In total, there are four equations to solve due to the three dimensional nature of the problem at hand

$$\frac{\partial p}{\partial t} = \rho c^2 Q - \rho c^2 \left(\frac{\partial w_x}{\partial x} + \frac{\partial w_y}{\partial y} + \frac{\partial w_z}{\partial z} \right) - \left(v_x \frac{\partial p}{\partial x} + v_y \frac{\partial p}{\partial y} + v_z \frac{\partial p}{\partial z} \right), \tag{2.25}$$

$$\frac{\partial w_x}{\partial t} = F_x b_x - \frac{\partial p_x}{\partial x} b_x - \left(v_x \frac{\partial w_x}{\partial x} + v_y \frac{\partial w_x}{\partial y} + v_z \frac{\partial w_x}{\partial z} \right) - \left(w_x \frac{\partial v_x}{\partial x} + w_y \frac{\partial v_x}{\partial y} + w_z \frac{\partial v_x}{\partial z} \right), \tag{2.26}$$

$$\frac{\partial w_y}{\partial t} = F_y b_y - \frac{\partial p_y}{\partial y} b_y - \left(v_x \frac{\partial w_y}{\partial x} + v_y \frac{\partial w_y}{\partial y} + v_z \frac{\partial w_y}{\partial z} \right) - \left(w_x \frac{\partial v_y}{\partial x} + w_y \frac{\partial v_y}{\partial y} + w_z \frac{\partial v_y}{\partial z} \right), \tag{2.27}$$

$$\frac{\partial w_z}{\partial t} = F_z b_z - \frac{\partial p_z}{\partial z} b_z - \left(v_x \frac{\partial w_z}{\partial x} + v_y \frac{\partial w_z}{\partial y} + v_z \frac{\partial w_z}{\partial z} \right) - \left(w_x \frac{\partial v_z}{\partial x} + w_y \frac{\partial v_z}{\partial y} + w_z \frac{\partial v_z}{\partial z} \right). \tag{2.28}$$

The b term above represents the mass bouyancy. To solve the above equations, finite differences were applied to the spatial derivatives; the approximations to each of the derivatives are provided in Appendix A.

Grid Configuration

The grid configuration and subsequent spatial finite-differences were setup in a staggered configuration. Using this setup, there were essentially multiple grids in use at once; the acoustic pressure values (p) were calculated and stored at whole integer nodes ($x = i\Delta x$,$y = j\Delta y$,$z = k\Delta z$), while the acoustic velocity values were calculated and stored at offset nodes ($x = (i + \frac{1}{2})\Delta x$,$y = j\Delta y$,$z = k\Delta z$ for w_x; $x = i\Delta x$,$y = (j + \frac{1}{2})\Delta x$,$z = k\Delta z$ for w_y; and $x = i\Delta x$,$y = j\Delta y$,$z = (k + \frac{1}{2})\Delta z$ for w_z). In addition to the acoustic pressure being stored at the pressure nodes, the mass buoyancy b, bulk modulus κ, and source Q were stored there as well. The ambient flow velocity v_x and source term F_x were stored at the velocity perturbation w_x nodes in the x-direction. The flow velocity v_y and source term F_y were stored at the velocity perturbation w_y nodes in the y-direction, while the flow velocity v_z and source term F_z were stored at the velocity perturbation w_z nodes in the z-direction.

An example of a staggered grid configuration for the main quantities of pressure and velocity perturbation is shown below for a two dimensional case (Fig. 2.2). It should be noted that the staggered grid configuration is typically used in wave propagation problems in non-moving media, and also provides a convenient way of applying finite-differences [63]. Although the figure illustrates an example in two dimensions, the developed approximations for this study were solved in three dimensions. Further, due to the use of a computer code array (which is unable to specify 'half' nodes), values were stored internally at regular indices. For example, the variable w_x stored at $i + \frac{1}{2}$ on the Wilson and Liu grid, appears at i on this study's array. Likewise w_y and w_z, stored at $j + \frac{1}{2}$ and $k + \frac{1}{2}$ on the Wilson and Liu grid, were

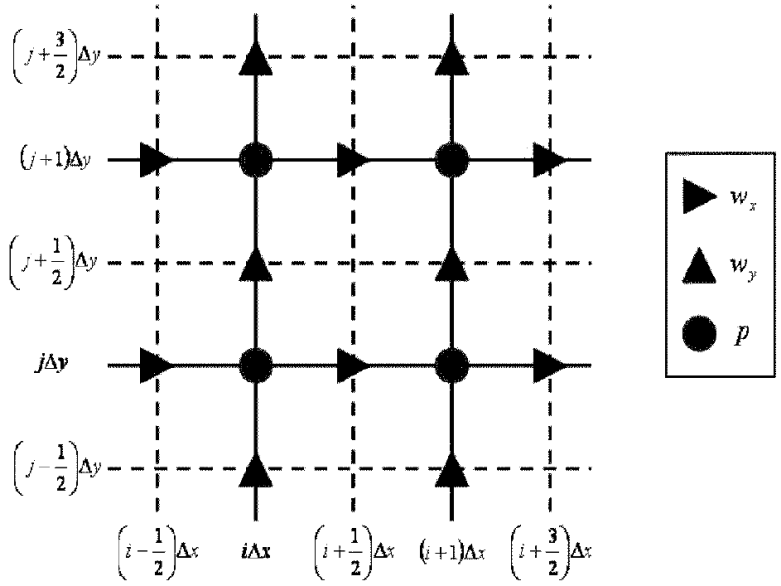


Figure 2.2: *Staggered Grid Configuration* [63]

stored at j and k. The approximations given in Appendix A to the equations of sound propagation take into account the change in indicies between the method used by Wilson and Liu, and the one used in the development of this studies computer code.

2.1.2 Boundary Conditions

In this study, two different types of boundaries were considered, a rigid boundary and an absorbing boundary. The absorbing boundaries represented each of the outer faces of the domain, while the remaining ground layer could be varied between being absorbing or fully rigid. A thorough analysis was done by Wilson and Liu [63] and was used as a starting point in this implementation. A rigid boundary can be represented by a hard surface such as a wall, barrier, or building. In terms of acoustics, using a rigid boundary causes the acoustic particle velocity to disappear perpendicular to the hard surface. To implement the rigid boundary condition, the following were ensured

1. The medium below the rigid boundary was made the same as that above.
2. The same source as above the rigid boundary was used below; because the two sources are equal and emitting in opposite directions, they should cancel at the boundary (the acoustic particle velocities should vanish).

For example, taking into account a rigid surface in place at the ground boundary, one needs to ensure that any vertical velocity components on the grid along the boundary are zero (w_y and v_y). In terms of the x-components of velocity, above the boundary they are reflected (w_x and v_x). The pressure values

stored at the nodes are defined slightly differently; along the boundary a single value is defined, while above and below the values mimic each other; that is, if the surface is defined at y = (j-1/2)Δy, the pressure values at $(j-1)\Delta y$ were set equal to those at $(j)\Delta y$. For the developed solver, the appropriate steps were taken to ensure that components and derivatives perpindicular to the boundary were made zero as required, otherwise, mirroring was used.

Rarely can outdoor ground surfaces be taken as being ideally rigid. Further, from a numerical standpoint it is difficult to represent something as being 'somewhat' absorbing or reflecting. For example, sound energy can propagate quite well and once it reaches the ground, depending on the type of surface, the energy is dissipated by viscosity and thermal conduction. With snow, much of the energy is absorbed because of the large pores that type of surface has. With a surface such as cement or asphalt, most of the energy is reflected. For a surface such as soil, the amount of reflection and absorption can vary.

To define sound propagation through porous media, equations have been developed by Morse and Ingard [45]. Of note is the use of the static flow resistivity factor. It essentially keeps the static pressure gradient from accelerating the fluid to infinite speed; instead, the velocity of the fluid will approach a finite constant value determined by the viscous drag of the porous material over which it passes [63].

$$\frac{\partial p}{\partial t} = -(\frac{\kappa_p}{\Omega})\nabla \cdot \mathbf{w} \tag{2.29}$$

$$\rho_p \frac{\partial \mathbf{w}}{\partial t} + \sigma \mathbf{w} = -\nabla p \tag{2.30}$$

In the equations above, κ_p is the bulk modulus of air in the pores of the permeable material, ρ_p is the density of the air in the pores, σ is the static flow resistivity, and Ω is the porosity. Briefly considering the bulk modulus of the air, two different values are used to bound its range, those for the isothermal and adiabatic cases, respectively, where γ is the heat capacity ratio.

$$\kappa_p = \frac{\rho c^2}{\gamma} \tag{2.31}$$

$$\kappa_p = \rho c^2 \tag{2.32}$$

For situations involving small pores and/or low frequencies, isothermal conditions typically exist. Further, at low frequencies [17],

$$\rho_p = \frac{s_c}{\Omega}. \tag{2.33}$$

The structure constant (effective density factor), s_c, is related to the tortuosity factor q by $1/\cos\theta$ (for a fixed angle). The tortuosity is used to describe the geometry of the pores. In idealized cases, it is possible to determine the structure constant through its relation to tortousity. For example, Attenborough [17] has shown that for cylindrical pores

$$s_c = \frac{4}{3}\frac{q^4}{\Omega}. \tag{2.34}$$

Nevertheless, the relationship between s_c and q depends strongly on the pore geometry. If one is doing analysis entailing isothermal conditions and using the lower end of the frequency spectrum, which results in the density of the air in the pores being constant and the bulk modulus being isothermal, the initial equations defined Morse and Ingard can be rewritten as shown below.

$$\frac{\partial p}{\partial t} = -\kappa_e \nabla \cdot \mathbf{w} \tag{2.35}$$

$$\rho_e \frac{\partial \mathbf{w}}{\partial t} + \sigma \mathbf{w} = -\nabla p \tag{2.36}$$

The effective bulk modulus and density in the pores are introduced above. They are defined as $\kappa_e = \frac{\rho c^2}{\gamma \Omega}$ and $\rho_e = \frac{s_c \rho}{\Omega}$. To implement the equations numerically, one can follow a similar procedure as was done for the equations of sound propagation earlier. To simplify the equations, $b_e = \frac{1}{\rho_e}$.

$$\frac{\partial p}{\partial t} = -\kappa_e \left(\frac{\partial w_x}{\partial x} + \frac{\partial w_y}{\partial y} + \frac{\partial w_z}{\partial z} \right) \tag{2.37}$$

$$\frac{\partial w_x}{\partial t} = -b_{ex} \left(\sigma_x w_x + \frac{\partial p_x}{\partial x} \right) \tag{2.38}$$

$$\frac{\partial w_y}{\partial t} = -b_{ey} \left(\sigma_y w_y + \frac{\partial p_y}{\partial y} \right) \tag{2.39}$$

$$\frac{\partial w_z}{\partial t} = -b_{ez} \left(\sigma_z w_z + \frac{\partial p_z}{\partial z} \right) \tag{2.40}$$

The finite-difference approximations to the spatial derivatives in the boundary condition equations are provided in Appendix A. Based on testing the developed method for this study, the size of a single absorbing layer should be set to a minimum of five times the source wavelength for optimal results. In addition to the developed equations for the porous boundary condition, shown below is a table of different ground conditions provided by Wilson and Liu (Table 2.1). The values have been arranged from least absorbing (asphalt), to most absorbing (forest/snow).

Material	**Flow Resistivity** $\left(\frac{Pa-s}{m^2}\right)$**,** σ	**Porosity,** Ω	**Tortuosity, q**
Asphalt	3 x 10^7	0.1	3.2
Sand	5 x 10^4	0.35	1.6
Grass	2 x 10^5	0.5	1.4
Forest	1 x 10^5	0.6	1.3
Snow	1 x 10^3	0.6	1.7

Table 2.1: *Ground Condition Parameters* [63]

2.1.3 Absorbing Boundary Conditions

Around the edges of the control volume, absorbing boundary conditions were put in place to avoid spurious numerical reflections. These absorbing boundaries however, were artificial. However, it is of interest to determine if a porous layer implementation (as developed above) can be used as an absorbing boundary. Hence, one can first consider the equation for the reflection coefficient of waves impacting a solid surface, as defined by Kinsler et al. [40]

$$R = \frac{Z_c \cos\theta_i - \rho c \cos\theta_t}{Z_c \cos\theta_i + \rho c \cos\theta_t}. \tag{2.41}$$

In the above, the angles of incidence and transmission, θ_i and θ_t, are measured with respect to a normal to the surface, while Z_c defines the characteristic impedance of the wave propagating through the absorbing layer. In addition, the transmission angle is normally complex and is associated with waves that exponentially decay from the boundary at which they form. Because one is to consider a truly absorbing boundary, the reflection coefficient is taken to be zero. Therefore, the above equation becomes

$$\frac{Z_c}{\rho c} = \frac{\cos\theta_t}{\cos\theta_i}. \tag{2.42}$$

By considering Snell's Law, one can eliminate $\cos\theta_t$ with

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{k_2}{k_1}. \tag{2.43}$$

This is done by first substituting the incident wave number k_1 with wave number k, the transmission wave number k_2 with complex wave number k_c, the incident angle θ_1 with θ_i, and transmission angle θ_2 with θ_t; the wave number being defined by, $k = \frac{\omega}{c}$, where ω is angular frequency. Rearranging in terms of transmission angle θ_t, one can obtain

$$\theta_t = \arcsin\left[\sin\theta_i \left(\frac{k}{k_c}\right)\right]. \tag{2.44}$$

The rearranged Snell's Law for θ_t, can then be substituted into equation 2.42 to yield

$$\left(\frac{Z_c}{\rho c}\right)^2 \cos^2\theta_i + \left(\frac{k}{k_c}\right)^2 \sin^2\theta_i = 1. \tag{2.45}$$

To replace the remaining coefficients of Z_c and k_c, one can obtain plane-wave solutions for equations 2.35 and 2.36 provided by Morse and Ingard. Hence, as was solved by Wilson and Liu [63], for a wave propagating in the positive x-direction

$$P = Ae^{i(k_c x - \omega t)}, \tag{2.46}$$

$$w_x = Be^{i(k_c x - \omega t)}, \tag{2.47}$$

$$w_y = w_z = 0. \tag{2.48}$$

Upon performing substitution, one can obtain a solution for coefficients $\frac{A}{B}$, from which useful quantities can be determined; the newly appearing term below is the complex density, ρ_c.

$$\frac{A}{B} = \frac{k_c \kappa_e}{\omega} = \frac{\omega \rho_c}{k_c} \tag{2.49}$$

$$\rho_c = \rho_e + \frac{i\sigma}{\omega} \tag{2.50}$$

$$k_c = \omega \sqrt{\frac{\rho_c}{\kappa_e}} \tag{2.51}$$

$$Z_c = \sqrt{\kappa_e \rho_c} \tag{2.52}$$

Substituting k_c and Z_c into equation 2.45, one obtains

$$\frac{\rho_c \kappa_e}{\rho^2 c^2} \cos^2 \theta_i + \frac{\kappa_e}{\rho_c c^2} \sin^2 \theta_i = 1. \tag{2.53}$$

One may now test whether the use of a porous layer will suffice as an absorbing boundary. To do so, it is necessary to determine the real and imaginary components of equation 2.53.

$$\frac{\left(\rho_e + \frac{i\sigma}{\omega}\right)}{\rho^2 c^2} \kappa_e \cos^2 \theta_i + \frac{\left(\rho_e - \frac{i\sigma}{\omega}\right)}{\left(\rho_e^2 + \frac{\sigma^2}{\omega^2}\right) c^2} \sin^2 \theta_i = 1 \tag{2.54}$$

Hence, one may set set the bulk modulus to $\kappa_e = \rho c^2$ and $\rho_e = \rho$, if $\frac{\sigma}{\omega}$ is small compared to ρ_e. This condition will be satisfied and can be applied for all of θ_i, if there is any kind of weakening of the incident wave. Therefore, it should be noted that if $\frac{\sigma}{\omega\rho} << 1$ over the desired range of analysis, a porous layer may be used as an absorbing boundary condition.

2.1.4 Implementation Considerations

Wilson and Liu [63], provide details of considerations to take into account when implementing a numerical method. To ensure that a numerical simulation is stable, the timestep and grid spacing are defined as

$$\Delta t = \frac{\Delta r}{\tilde{v}}, \tag{2.55}$$

$$\Delta r = \sqrt{\Delta x^2 + \Delta y^2 + \Delta z^2}. \tag{2.56}$$

As a result of defining stability with the timestep, and when solving problems in general, it is also beneficial to define the Courant number

$$C = \frac{\tilde{v} \Delta t}{\Delta r}. \tag{2.57}$$

For numerical stability, $C \ll 1$. One should note that although the above can be satisfied, it does not mean that numerical accuracy will be guaranteed. In the case of a uniform flow, it is also possible to

define grid spacing in terms of Mach number, M, and the number of grid nodes, N.

$$\Delta r = \frac{\lambda}{N}(1 - M) \tag{2.58}$$

Observing the above, one should keep in mind that Δr should be much smaller than the wavelength for accuracy. At minimum, the chosen grid spacing should contain five nodes per wavelength for a sinusoidal signal to resolve correctly. To determine the correct direction to determine the wavelength, one should consider that the wavelength in the downwind direction of propagation will be shorter. Hence, the downwind wavelength is taken as a reference ($\lambda = \frac{\tilde{c}-\tilde{v}}{f}$). Substituting this result into the previously defined timestep and using the downwind propagation speed, one obtains

$$\Delta t < \frac{1}{Nf}\frac{1 - M}{1 + M} \tag{2.59}$$

As can be seen in the above formula, as the flow Mach number increases, the timestep must decrease, and resulting in more computational time is required. Therefore, the timestep must be chosen correctly, as it also controls the convergence of the solution. As Δt is made larger, convergence tends to decrease [60]. Conversly, as Δt is decreased, one would expect the convergence rate to increase, but the amount of computing time needed to increase. One should also note that decreasing the timestep to a value below the minimum for convergence may not result in a suitable solution. For the purposes of this study, it was necessary to choose a time-marching method to develop a solution to the equations for sound propagation, as well as those for the boundaries developed earlier. Several different time-marching methods were considered by Wilson and Liu. They concluded that although different methods may have the same order of accuracy, they may not produce the same results. Based on the trials there were performed in their study, the unstaggered 'leap-frog', Aldrige, and fourth-order Runge-Kutta methods were determined to be best, as "they all provide good accuracy and stability for a similar level of computational effort" [63]. The actual time-marching of the solution for this study was performed with the assistance of the Portable Extensible Toolkit for Scientific Computation (PETSc) library, which can provides access to different implementations (Euler, CN, Runge-Kutta, etc).

2.2 Sound Generation Model

There are different types of sound generation source patterns which exist. These are the monopole, dipole, and quadrapole sources. Each of these sources is an elementary solution to the inhomogenous wave equation. As demonstrated by Blake [19], to develop the equation, one can begin with the continuity equation

$$\frac{\partial \tilde{\rho}}{\partial t} + \frac{\partial}{\partial x_i}(\tilde{\rho}\tilde{v_i}) = 0. \tag{2.60}$$

Taking the time derivative of the above yields

$$\frac{\partial^2 \tilde{\rho}}{\partial t^2} + \frac{\partial^2}{\partial t \partial x_i}(\tilde{\rho}\tilde{v_i}) = 0. \tag{2.61}$$

Proceeding further, one can then consider the momentum equation in the form

$$\frac{\partial}{\partial t}(\tilde{\rho}\tilde{v_i}) + \frac{\partial}{\partial x_j}(\tilde{\rho}\tilde{v_i}\tilde{v_j}) + \frac{\partial \tilde{P}}{\partial x_i} = 0. \tag{2.62}$$

Taking the divergence of the momentum equation produces

$$\frac{\partial^2}{\partial x_i \partial t}(\tilde{\rho}\tilde{v_i}) + \frac{\partial^2}{\partial x_i \partial x_j}(\tilde{\rho}\tilde{v_i}\tilde{v_j}) + \frac{\partial^2 \tilde{P}}{\partial x_i^2} = 0. \tag{2.63}$$

Now, by subtracting equation 2.63 from 2.61, one can eliminate the term $\tilde{\rho}\tilde{v_i}$, giving

$$\frac{\partial^2 \tilde{\rho}}{\partial t^2} - \frac{\partial^2}{\partial x_i \partial x_j}(\tilde{\rho}\tilde{v_i}\tilde{v_j}) - \frac{\partial^2 \tilde{P}}{\partial x_i^2} = 0. \tag{2.64}$$

Equation 2.64 could then be linearized, using perturbation values which represent acoustic quantities. The variables P and ρ show below, represent the pressure and density at ambient conditions (the velocity for the ambient atmopshere was taken as being zero). The variables p, η, and w, represent the pressure, density, and velocity perturbations. The equations for the linearization were defined as

$$\tilde{P} = P + p, \tag{2.65}$$

$$\tilde{\rho} = \rho + \eta, \tag{2.66}$$

$$\tilde{v_i} = v_i + w_i. \tag{2.67}$$

Substituting the equations for linearization into equation 2.64 and rearranging yields

$$\frac{\partial^2 \eta}{\partial t^2} - \frac{\partial^2 p}{\partial x_i^2} = 0. \tag{2.68}$$

If one considers the fluid in the domain to be isentropic, the pressure perturbation changes linearly with the density perturbation; Wagner et al. showed it can be represented by

$$p = c^2 \eta. \tag{2.69}$$

In the above, the density perturbation is scaled by the speed of sound, c. To complete the development of the wave equation, one can choose to formulate it with terms of density or pressure; for the purposes of this example, terms are formulated for pressure. Therefore, substituting equation 2.69 into 2.68, one can obtain the inhomogenous wave equation with pressure terms

$$\frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} - \frac{\partial^2 p}{\partial x_i^2} = 0. \tag{2.70}$$

A similar formulation can be obtained using terms of density for pressure through equation 2.69. Finally, by introducing a source $\psi(\vec{x}, t)$ into the domain, one can write the inhomogenous wave equation as

$$\frac{1}{c^2}\frac{\partial^2 p(\vec{x}, t)}{\partial t^2} - \frac{\partial^2 p(\vec{x}, t)}{\partial {x_i}^2} = \psi(\vec{x}, t). \tag{2.71}$$

In classifying and developing a noise source, there are several different components which need to be considered. Each of these components can be represented by different directivity patterns which are all elementary solutions to the above inhomogenous wave equation. The typical directivity pattern and solution for a wind turbine is given as a dipole shaped pattern [59].

2.2.1 Noise Model Classes

Lowson, through his work in aerodynamics, suggests the use of different classes to specify how detailed a model should be used to solve a particular noise generation problem [44]. These classes can be divided into three levels:

Class I - Basic methods which are meant to be used as a quick estimate based on a variety of basic parameters (for example, rotor size and output power, etc).
Class II - More advanced methods which use scaling factors based on both theoretical and empirical data. Each of the different causes of noise on a turbine are considered when formulating the model.
Class III - The most advanced methods which use detailed data of the wind turbine (such as its geometry and aerodynamic characteristics).

2.2.2 Solutions of Wave Equation

Based on the equations of sound propagation developed earlier, sources of sound are represented as either an in-flow rate density Q or a force **F**, for a monopole and dipole, respectively. In defining the sources to be used however, it is necessary to correctly scale the source term. The required source term can be calibrated using two different methods. The first involves specifying a Sound Power Level (SWL) value. The SWL is a function of the source characteristics and defines the power of the radiating sound level, measured in dB. The second method of scaling involves the SPL value. The SPL is a function of the receiver's position relative to the source and is also measured in dB. Therefore, the SPL value will change depending on what location it is taken at, while the SWL is characteristic of the source regardless of observer location. Either of these reference values are necessary for the proper scaling to be applied when providing source generation data to the sound propagation model for terms Q and **F**. Before proceeding further, it is also of benefit to describe the idealogy of a compact source. It is defined as being based on a dimension L, and said to be compact if L is small in comparison to the wavelength of the wave under study; that is, $\frac{L}{\lambda} \ll 1$. If this relation is satisfied, the different values of retarded time within the region the source is defined in can be ignored [59]; the retarded time will be defined in

the upcoming explanation of the different types of sources. Futher, the assumption of a compact source allows one to define the source at a point.

Monopole

A monopole source can be described as a source of mass generation. It is axisymmetric and produces the same value at any point on the surface of a sphere of radius r in a quiescent, uniform fluid. As shown by Wagner et al. [59], if the source is distributed throughout the domain and has an inflow-rate density of Q, one can rewrite the continuity equation as

$$\frac{\partial \tilde{\rho}}{\partial t} + \frac{\partial}{\partial x_i}(\tilde{\rho}\tilde{v_i}) = \rho Q(\vec{x}, t). \tag{2.72}$$

By taking the time derivative of the continuity equation, the divergence of the momentum equation, and then subtracting the former from the latter, a linearization can be performed (using the same linearization terms as was done for the inhomogenous wave equation) where one can obtain

$$\frac{\partial^2 \eta}{\partial t^2} - c^2 \frac{\partial^2 \eta}{\partial x_i^2} = \rho \dot{Q}(\vec{x}, t). \tag{2.73}$$

In the above, $\dot{Q}$ represents the mass introduction per unit volume (monopole strength per unit volume). Observing the above, one has obtained an inhomogenous partial differential equation, which cannot be solved easily. To continue the development of the solution, another operation can be performed by replacing the source term spanning the domain with a single point source. This can be done using a Dirac delta function, and by introducing a mass introduction rate (monopole strength), represented by δ and $\dot{q}$, respectively.

$$\frac{\partial^2 \eta}{\partial t^2} - c^2 \frac{\partial^2 \eta}{\partial x_i^2} = \rho \dot{q}(t) \cdot \delta(\vec{x} - \vec{y}) \tag{2.74}$$

To solve the above and determine a solution to the wave equation, one can use Green's functions to obtain

$$p(\vec{x}, t) = \rho \frac{\dot{q}(t - \frac{r}{c_0})}{4\pi r} = \rho \frac{\dot{q}(\tau)}{4\pi r} = \rho \frac{[\dot{q}]}{4\pi r}. \tag{2.75}$$

Within the equation, the r variable can be replaced by $|\vec{x} - \vec{y}|$, and represents the position of an observer $\vec{x}$, in relation to the source $\vec{y}$. In addition, the retarded time is represented by $\tau = t - \frac{r}{c}$, and is the difference between time t and the interval of time it takes a wave to travel from the source to the observer; essentially, it is the difference between the observer recieving a wave and it actually being emitted. The functions that are evaluated at the retarded time τ, are indicated in square brackets (ex. for the source, $[\dot{q}]$). Hence, the resulting solution is based on a point source in space which has a monopole directivity pattern (which is radially symmetric), and can be interpreted as a pulsating sphere. A real-world example of a point monopole is a buzzer or siren. One can develop an expression for the monopole strength; introducing a volumetric sinusoid function that will represent the source $\dot{q}$

$$q(t) = Acos(\omega t). \tag{2.76}$$

Taking the derivative, one can then obtain

$$\dot{q} = -A\omega sin(\omega\tau) = -A\omega sin(\omega t - k_0 r). \tag{2.77}$$

Taking the Root Mean Square (RMS) (over the entire period of the waveform) of $\dot{q}$, and substituting into the previously found solution of the wave equation $p(\vec{x}, t)$, the fully developed solution is

$$P_{RMS} = \frac{\sqrt{2}\rho A\omega}{8\pi r}. \tag{2.78}$$

To utilize the above within the equations of sound propagation, the above expression is rearranged in terms of the source amplitude A. Thus to produce a specific RMS pressure P_{RMS}, at a receiver located distance r from the source, within a uniform, quiescent fluid, requires an amplitude of

$$A = \frac{8\pi r P_{RMS}}{\sqrt{2}\rho\omega}. \tag{2.79}$$

Using an expression for sound pressure level (where P_{ref} is 20 μPa)

$$L_P = 10 \log \frac{P_{RMS}^2}{P_{ref}^2}, \tag{2.80}$$

and rearranging for RMS pressure, one can make a substitution into equation 2.79 to obtain

$$P_{RMS} = P_{ref}\sqrt{10^{(0.1L_P)}}. \tag{2.81}$$

Upon making the substitution, the amplitude required for a monopole of a known strength is given by

$$A = \frac{8\pi r(P_{ref}\sqrt{10^{0.05L_P}})}{\sqrt{2}\rho\omega}. \tag{2.82}$$

This amplitude value can be used in the coupled equations of sound propagation, to specifiy a magnitude for the mass term Q.

Dipole

As was done previously for the monopole, Wagner showed it is necessary to manipulate the continuity and momentum equations to obtain a solution of the wave equation for a dipole [59]. However, the key difference being that for a dipole one is no longer dealing with a source generating mass, but rather a source applying a force on the fluid medium. Hence, rewriting the momentum equation with a forcing term

$$\frac{\partial(\tilde{\rho}\tilde{v_i})}{\partial t} + \frac{\partial(\tilde{\rho}\tilde{v_i}\tilde{v_j})}{\partial x_j} + \frac{\partial\tilde{P}}{\partial x_i} = F_i(\vec{x}, t). \tag{2.83}$$

Completing the same steps as before (subtracting continuity from momentum, applying time derivative and divergence, and performing a linearization), one can obtain:

$$\frac{\partial^2 \eta}{\partial t^2} - c^2 \frac{\partial^2 \eta}{\partial x_i^2} = -\frac{\partial F_i(\vec{x}, t)}{\partial x_i} = \dot{F}_i(\vec{x}, t). \tag{2.84}$$

To represent a compact source it is necessary to rewrite the above as a point source. As was done previously with the monopole inflow rate density, replacing the above with

$$F_i = f_i(t)\delta(\vec{x} - \vec{y}). \tag{2.85}$$

After making the substitution, one can solve using Greens functions for the pressure to obtain

$$p(\vec{x}, t) = -\frac{\partial}{\partial x_i}\left(\frac{f_i(\tau)}{r}\right). \tag{2.86}$$

Taking the above and expanding further by applying the divergence operator, the solution is given by

$$p(\vec{x}, t) = \frac{1}{4\pi}\frac{\vec{x_i} - \vec{y_i}}{|\vec{r}|}\left(\frac{1}{rc}\frac{\partial f_i(\tau)}{\partial t} + \frac{1}{r^2} f_i(\tau)\right). \tag{2.87}$$

With the obtained solution, there are two different situations for which it can be used. The first of these situations deals with the near-field solution, represented by $\frac{1}{r^2}$ term, while the second involves the far-field solution, defined by $\frac{1}{r}$. Both of the terms are useful as they indicate that a dipole source does not radiate sound in the same way as does a monopole. That is, the dipole has a strong near-field region, which radiates much more noise away in the near-field in comparison to the monopole; this is in contrast to the far field, which dissipates according to $\frac{1}{r}$. To develop the forcing term, one can introduce an oscillating force term and take the derivative of it

$$f_i(\tau) = Ad_i cos(\omega t - k_0 r), \tag{2.88}$$

$$\frac{\partial f_i(\tau)}{\partial t} = -Ad_i \omega sin(\omega t - k_0 r). \tag{2.89}$$

The derivative of the forcing term can then be put into equation 2.87 above to yield

$$p(\vec{x}, t) = \frac{1}{4\pi}(o_i d_i)\frac{A}{r^2}(-k_0 r sin(\omega t k_0 r) + cos(\omega t - k_0 r)). \tag{2.90}$$

Because the RMS pressure is required for the sound propagation model, the same method is applied as for a monopole to find it. Taking the RMS value, one can obtain

$$P_{RMS} = \frac{\sqrt{2(1 + k_0^2 r^2)}}{8\pi}(\frac{\vec{x} - \vec{y}}{|\vec{r}|})\frac{Ad_i}{r^2}. \tag{2.91}$$

The variable d_i, represents the directional unit vector of the dipole. Rearranging to solve for the amplitude and substituting for RMS pressure

$$A = \frac{P_{ref}(10^{0.05L_P})(8\pi r^2)}{\sqrt{2(1+k_0^2r^2)(\frac{\vec{x}-\vec{y}}{|\vec{r}|})d_i}}. \tag{2.92}$$

As was done earlier, an equation has been developed for the scaled source amplitude of a specific sound pressure level at a receiver. Further, it is beneficial to mention that due to the nature of a dipole, there will be a region of maximum pressure along the dipole axis at 0 and π, while a region of minimum pressure will exist perpendicular to the dipole axis at $\frac{\pi}{2}$ and $\frac{3\pi}{2}$ (Fig. 2.3).

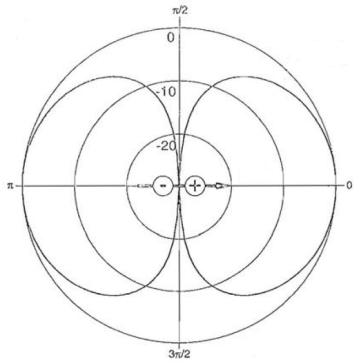


Figure 2.3: *Dipole Directivity Pattern* [59]

2.2.3 Noise Sources

According to Wagner et al. [59], there are various causes of noise being emitted from a turbine (Fig 2.4). In the broadest sense, the two major causes are mechanical noise and noise caused by the flow interacting with the turbine itself. Several different components of the wind turbine are responsible for mechanical noise; they include the gearbox, generator, cooling equipment, and hydraulics for blade pitch control. The actual emission of the noise however, can take place through either the air itself or through the structure of the turbine first and then be emitted through the air.

The greatest contributor of mechanical noise from the above components is the gearbox. The component itself consists of several smaller internal parts, of which the gears create most of the noise. The noise results from errors in their manufacturing, leading to issues in the gear meshing. The errors in the meshing then lead to vibration and improper loading. If the errors in the gear manufacturing were to be doubled, a noise increase of approximately 6dB would result. Insulation for the nacelle however, can reduce noise emissions by as much as 15 dB [52]. A further 1dB decrease per degree of angle can be achieved by using helical gears [59].

Aerodynamic noise can be divided into three separate categories; steady thickness and loading noise, inflow turbulence noise, and airfoil self-noise. Primarily, the noise results due to the interactions of the

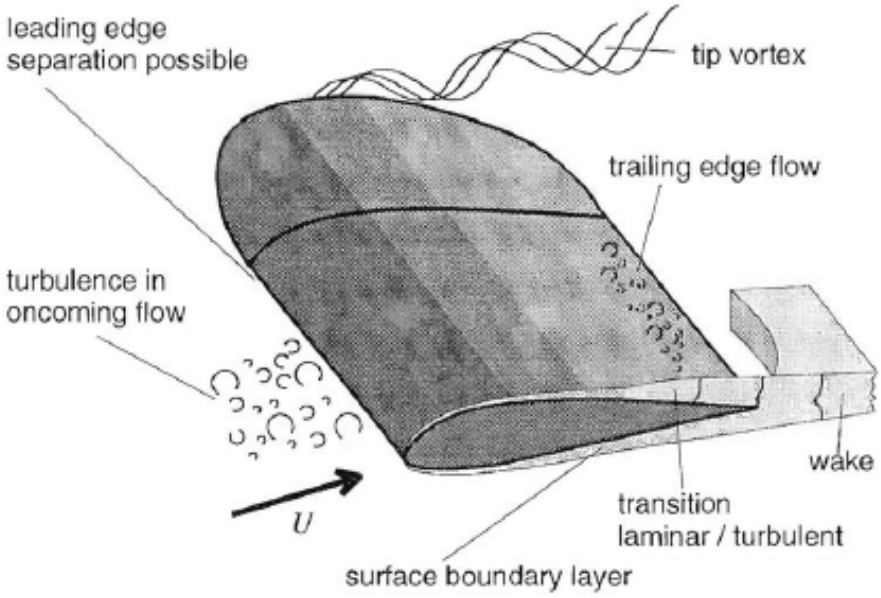


Figure 2.4: *Noise components on an airfoil* [19]

boundary layer with the surface of the airfoil.

Steady Thickness and Loading Noise

The mechanisms responsible for this type of noise, primarily cause low frequency noise when the blades ingest variations in an otherwise uniform flow, such as the wake from an obstacle upstream or a flow gradient. Essentially what takes place, is the disturbed flow enters the region occupied by the spinning turbines blades. When this happens, there is a change in the dynamic pressure and angle of attack resulting in a fluctuation of how much load is applied on the blade. This resulting load change then fluctuates based on the disturbed flow. The amount of sound generated by this specific noise type is typically not of significant issue with wind turbines that have the rotor plane upwind of an obstruction generating a disturbance to the flow (i.e. such as the support tower); however, if the tower is placed upwind of the rotor, noise can result from interaction with the blades. The sound generated by the blades encountering the disturbed flow is tonal and is typically in the range of 1-3Hz. When the rotor experiences an increased rotational speed, the frequency of the sound increases however [32]. The low frequency noise may not seem significant as it is well below the lower range of human hearing. However, it is more perceptible as vibration which can in fact affect humans internally, as well as structures and their components in the immediate vicinity (such as buildings with windows). This low frequency noise is known as infrasound [54].

Inflow-Turbulence Noise

Turbulence alone is known to cause broadband noise, and can be encountered throughout the atmosphere. There are two primary causes of it: that due to the interaction of the flow with any kind of surface (aerodynamic) and that due to rising air from heating by the sun (thermal) [44]. To quantify it, typically a turbulence intensity value is used, which compares the standard deviation and mean wind velocity.

The noise produced by turbulence can result in both low and high frequency noise. Whether low or high end noise will result however depends on the size of the accompanying turbulent eddy in relation to the turbine blade chord length. If the eddy size is large in comparison to the chord, a dipole noise pattern will result and the blade will experience a change in its loading. If the eddy size is similar or small in comparison to the chord, the noise pattern will depend on the blade geometry and only a local change in loading will result. Based on the wavelength

$$\lambda = \frac{\Lambda}{M}, \tag{2.93}$$

Where the turbulent eddy size is defined by Λ, and if the ratio of the speed of sound to wavelength is much less than one, the noise source resulting from the blade can be taken as being compact, allowing the use of a point source [19]. If the ratio mentioned above is not satisfied, the source cannot be taken as compact, and a significantly more complex problem results, whereby the noise pattern will depend on the physical characteristics of the blade. The actual noise resulting from inflow turbulence is normally taken as a swishing noise as the blades rotate. It is predominant in noise generation up to a maximum of 1000Hz [59]. It has been said that the shape of the airfoil and its leading edge also play a significant role in the amount of resulting noise [16],[23].

Airfoil Self-Noise

The boundary layer of an airfoil plays an important role in airfoil self-noise, regardless if there is a contribution from steady thickness and loading and inflow turbulence noise. The different components producing this noise are

Trailing Edge

This type of noise involves the boundary layer encompassing turbulent flow and interacting with the trailing edge. Essentially, the eddys formed in the turbulent layer are able to produce noise as they meet a corner or edge (in contrast to no edges or corners where the eddys would not produce substantial noise). As a result, the emission of noise can be reduced by a significant amount if one uses a swept blade [62] or a serrated shape [38], in effect 'cushioning' the flow off the blade. The wave number, distance of the eddy from the trailing edge, eddy convection speed, and turbulence composition, also play important roles in the amount of noise produced. The sound resulting from the trailing edge is normally perceived as being a swishing noise and exists in the range of 500 to 1500 Hz, depending on the turbine configuration [59].

Laminar-Boundary-Layer-Vortex-Shedding

The vortex shedding noise results due to certain Reynolds number flows (between 10^5 and 10^6), as the transition point from laminar to turbulent flow moves significantly downstream towards the trailing edge. As the transition point moves to the end of the blade, an interaction takes place between the trailing edge and turbulent boundary layer. The noise can be decreased or eliminated by causing transition from laminar to turbulent flow to occur much sooner using a vortex generator or by serrating the leading edge [35]. Nevertheless, because the Reynolds number of the flow over most modern large turbines is significantly larger than the conditions mentioned earlier, vortex shedding is not typically an issue. Smaller and older turbine blades however are more susceptible to producing this type of noise. The resulting noise from vortex shedding is tonal in nature [59].

Tip

The noise produced from the tip is caused by overflow from the pressure side to the suction side of the airfoil. The noise is a consequence of a vortex forming at the tip and interacting with it [37], as well as and the trailing edge [20],[21]; the amount produced depends on the location, size, and strength of the core, the vortex convection speed, the Reynolds number of the flow, and the loading situation of the blade [59],[21]. The production of noise because of the overflow is similar to how trailing-edge noise takes place, due to interaction with an edge. In addition, because the flow reaches a sharp edge, separation can occur which can further produce more noise, similar to vortex shedding. To reduce the amount of interaction with the tips of the blades, a designer may use a different tip shape similar to winglets used on aircraft. Tip noise is said to be broadband in nature [59].

Separated Flow

When the flow separates over an airfoil, it is said to be in a stalled condition. A similar scenario takes place when the flow over a turbine blade separates. The noise itself is only eliminated by preventing the blade from entering a stalled condition [59]. This separated flow interacts with the trailing edge shape of the blade and causes broadband noise which can be in excess of 10dB [25]. If a deep stall takes place, noise is produced from the entire blade and not only from the trailing edge [51].

Blunt Trailing-Edge

As the flow passes over the edge of the turbine blade, it may produce vortex shedding in a similar fashion as if it was passing by a cylindrical object. That is, as the flow moves downstream, it would separate from the cylinder. Hence, due to the shape of the trailing edge, vortex shedding can take place and produce a dipole type tonal noise [19]. This tonal noise is controlled by the shape of the trailing edge, Reynolds number, and trailing edge/boundary layer thickness ratio. As the size of the edge decreases,

the tone produced moves above the range of perceptible hearing [31].

Blade Imperfection

As with anything manufactured, there are always certain defects which can occur to the part, however minor they might be. With the case of an airfoil (or turbine blade), imperfections in the blade can have adverse consequences on noise production. Anything causing physical imperfections or surface contamination leading to an increase in the drag can cause a disturbance to the flow over the blade, potentially leading to a tonal noise issue [48].

Shown below is an example of the different noise source mechanisms and their contribution to the overall noise produced from a wind turbine (Fig. 2.5).

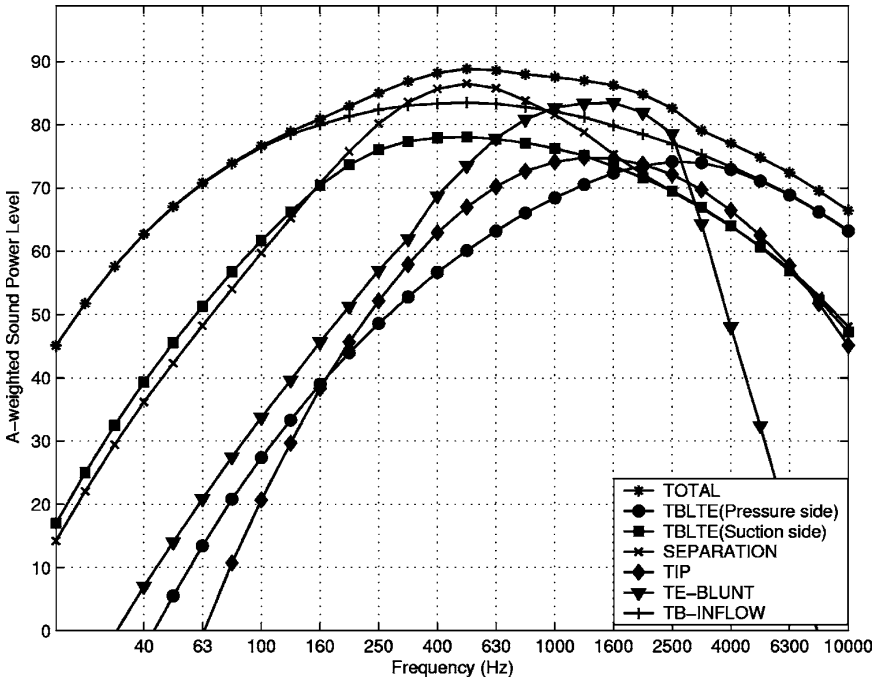


Figure 2.5: *Noise Sources of a NACA 632xx Turbine Blade (Bonus 300 kW)* [64]

2.2.4 Noise Emission Estimation

To estimate the noise emitted from a source turbine itself, a simple method developed by Hau [34] was used. It is defined as a 'Class I' method, being based on empirical data, and only requiring a rotor diameter input in metres. The Hau model was chosen over others by Lowson and Hagg, as it best modelled higher power turbines (approximately 2 MW) in comparison to measured data. With a

diameter, one can obtain an A-weighted SWL

$$L_{WA} = 22log_{10}D + 72. \tag{2.94}$$

Before the SWL is of use however, it is necessary to perform a conversion. First, the A-weighting of the SWL must be removed. To do this, one can use the following equation provided by Glenister, and presented in ANSI standards S1.4-1983 and S1.42-2001 [29]

$$A(f) = 2 + 20log_{10}\left[\frac{12200^2 f^4}{(f^2+20.6^2)\sqrt{(f^2+107.7^2)(f^2+737.9^2)}(f^2+12200^2)}\right]. \tag{2.95}$$

In the above, the frequency under study is input and a weighting value, A(f), is determined. Upon determing A(f), one can add this to an A-weighted SPL to determine the un-weighted value for SWL.

$$L_W = L_{WA} + A(f) \tag{2.96}$$

This value was used in the previously developed sound generation model to be able to determine the amplitude of the particular source type.

2.3 Workflow

Due to the nature of the mathematics involved, it was necessary to develop a computer program to solve the sound propagation equations, as well as those for the boundary conditions. The code was developed using the C programming language, utilizing the Portable, Extensible Toolkit for Scientific Computation PETSc library. The library contains specific tools for solving equations, as well as tools to allow the use of parallel processing to significantly improve the computing time. The source code developed for this study is included in Appendix C. However, due to the inherent nature of using an FDTD method, a significant amount of computing power was required. The processing facilities were provided by the Compute Canada High Performance Computing (HPC) consortium cluster of SHARCNET (ORCA), as well as the University of Toronto cluster SCINET; it should also be noted that minor testing initially was also carried out on a small local computing cluster. The use of the HPC clusters significantly reduced the amount of time required to run the code in comparison to a standard desktop computer.

To run the code on the clusters, it was necessary to create submission scripts for each job submitted, after the computer code was loaded onto the system. The scripts were created to pass arguments in regards to the specific parameters required, as well as the cluster specific instructions (such as the requested amount of resources - run time, CPU usage, and amount of memory required). Although creating the submission script was trivial, each cluster required a different format. Generic examples are provided in Appendix B for submitting a computing jobs on SHARCNET, SCINET, and the local cluster that was used.

During computation, one was able to observe the status of the solver. That is, as the equations were being solved, the time and percentage completed, as well as the time remaining, was presented.

Upon completing the computation, data was stored in output files. The output files present both text and graphical information. The text, gave the ground plane maximum and minimum SPL and Sound Intensity Level (SIL), as well as data pertaining to a probe location, if specified by the user. The graphical data, gave a visual representation of SPL, and was presented in the form of a 'slice' of the solution taken at the ground plane of the three dimensional domain. To visualize the information, an open-source scientific visualization software was used called ParaView, shown in the figure below (Fig. 2.6).

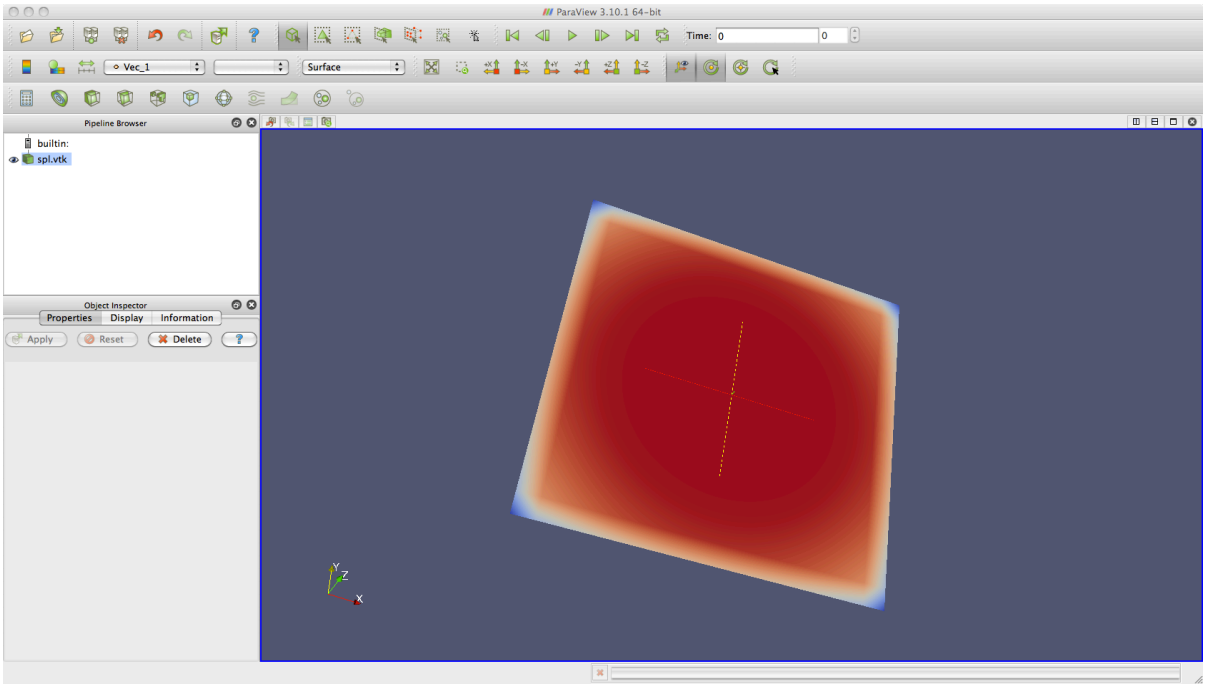


Figure 2.6: *ParaView Interface*

Chapter 3

Results

The following sections detail both the results of the validation performed as well as the final results obtained from running the solver. To determine that the developed model was working correctly, two separate validation steps were performed. The first dealt with testing against known data that was provided in the Wilson and Liu paper [63]; this was the two dimensional validation. The second case did not compare to any data, but was performed in the form of a grid convergence study; this was the three dimensional validation. For both validation cases, there was no flow.

3.1 Validation

For the first validation case, two separate grid resolutions were considered. A lower resolution case of 600x300 nodes, while the higher resolution case doubled this amount to 1200x600 nodes. The physical dimensions of the domain were setup to be 200 m horizontally and 100 m vertically, with the origin of the domain being at the centre. The boundary conditions for the domain were set such that a rigid ground layer was used, while the absorbing boundaries were set to be 20 m. For the source configuration, its origin was set at x=-60 m and y=-30 m. It was configured to emit as a point source monopole, at a frequency of 100Hz. Finally, the source was scaled such that it would produce a value of 1 dB at 1m away from its origin. This notion was validated later in the Grid Convergence Study (GCS).

To generate the results, each of the cases were run with a courant number of 0.8, using a run-time of 0.45 s. As can be seen in both the figures (Fig. 3.1, 3.2), an interference pattern of high and low pressure results due to the reflections from the ground due to the rigid ground boundary. Upon completing the two runs, a comparison could be made to that which was shown by Wilson and Liu, as shown in Figure 3.3.

Observing the results in Figures 3.1,3.2, and 3.3, the effects of the absorbing boundaries, as well as the rigid ground layer, are clear. Furthermore, the propagation patterns in both this study and that of Wilson and Liu are nearly identical.

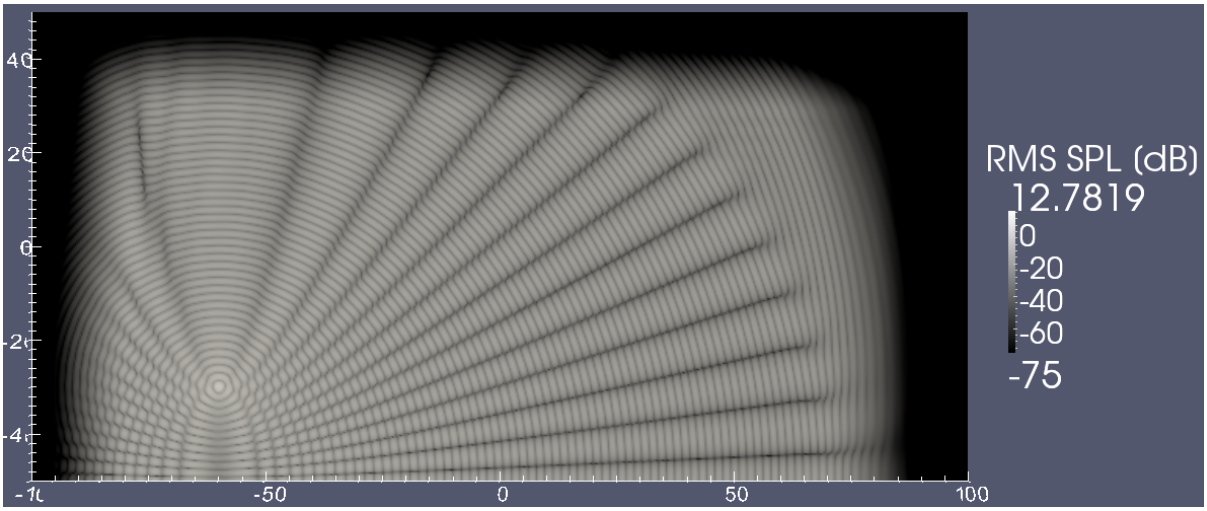


Figure 3.1: *Low-Resolution 2-D Validation*

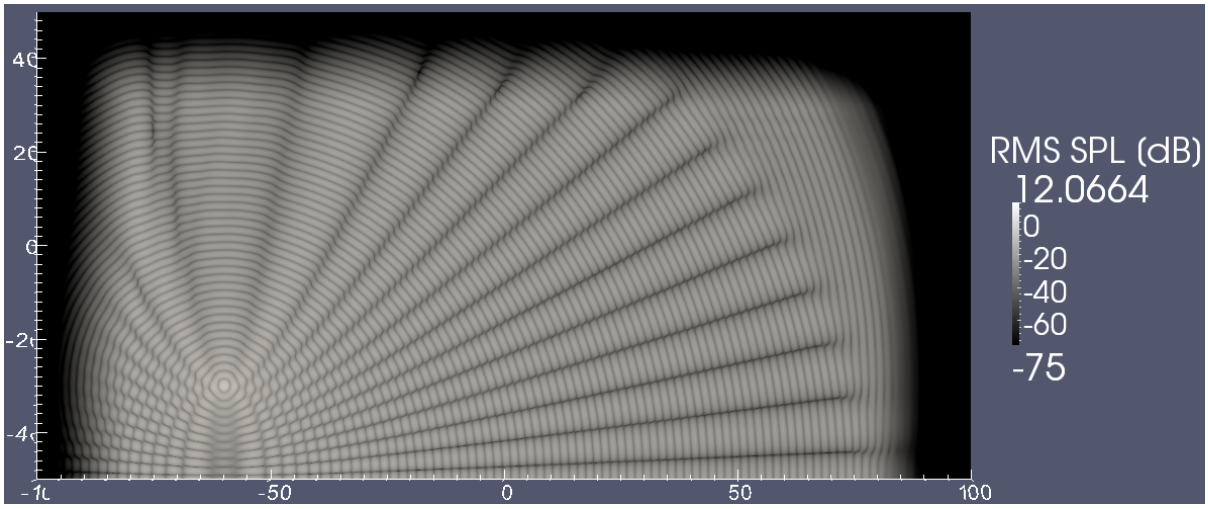


Figure 3.2: *High-Resolution 2-D Validation*

3.2 Grid Convergence Study

To validate the code and determine whether convergence could be achieved, a GCS was performed (for the purposes of this study, spatial convergence was examined). The method used Richardson Extrapolation, as well as three grids (fine, normal, coarse) to collect data. The general procedure and conditions used to conduct the study are described below (Table 3.1). Using a method shown by Slater [57] based on the work of Roache, a fine grid was first generated based on the ratio of nodes used and size of the domain. The number of nodes was also set to be equal in each cartesian coordinate direction to ensure

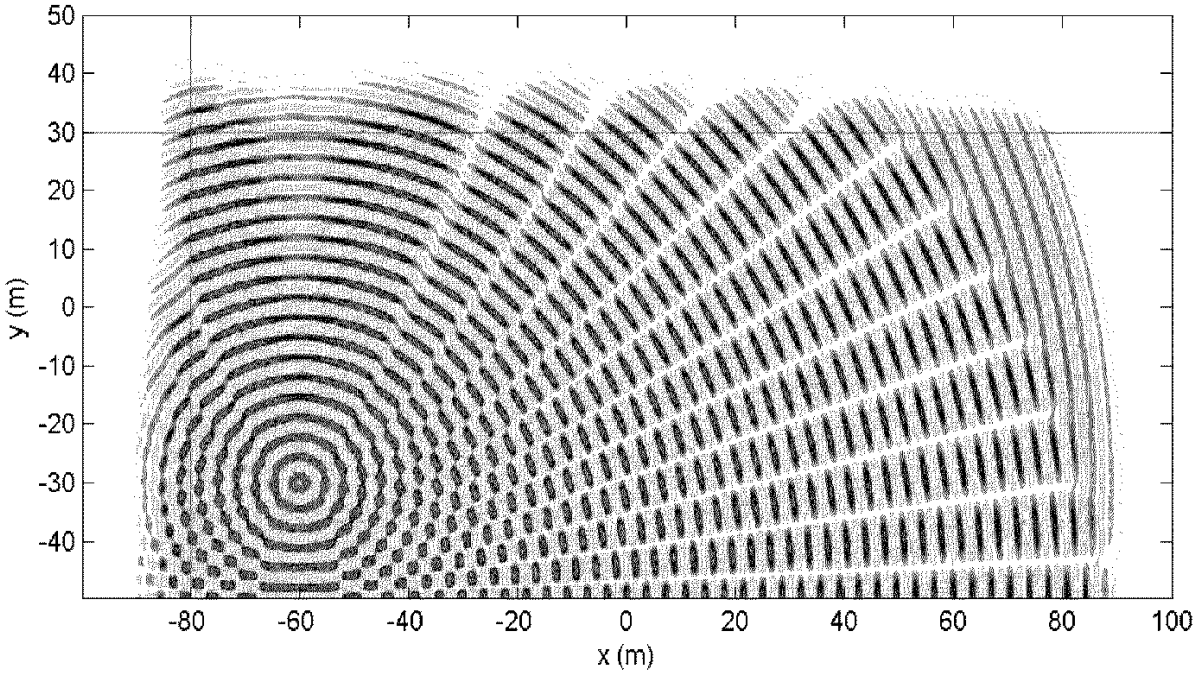


Figure 3.3: *Wilson and Liu Solution of Emitting Monopole Source* [63]

Variable	**Value**
Number of Nodes	651^3 (fine), 325^3 (normal), 163^3 (coarse)
Domain Size	60 m
Absorbing Boundary Condition	12 m
Time-Marching Method	Runge-Kutta (4th Order)
Time-Step	0.0001 s
RMS Average Period	0.033 s
Flow Type	Uniform
Sound Source Type	Mono (Calibrated Point Source - 100 dB at 1m)
Sound Source Frequency	300 Hz
Flow Mach Number	0.03

Table 3.1: *Grid Convergence Study Parameters*

a structured grid and because the time step is dependant on the shortest grid element. To select the domain size however, it was necessary to determine the wavelength of the sound source to ensure that the coarse grid would not be too widely spaced to encompass the sinusoid of the sound source.

$$\Delta x = \Delta y = \Delta z = \frac{x_{max} - (x_{min})}{N_x} \tag{3.1}$$

$$\lambda = \frac{c}{f} \tag{3.2}$$

Above, the x_{max} and x_{min} variables represent the domain dimension in the x-direction, while N_x is the number of nodes in the x-direction. Using the given equations above, an arbitrary number of nodes was selected (651^3 nodes). To check whether the coarse grid would still be suitable, the node selection was halved twice (once to obtain the normal grid, 325^3 nodes, and again to obtain the coarse grid, 163^3 nodes). It should be noted however, that one need not half the grids; a value such as 1.5 of the previous grid may be used. Nevertheless, the sizing should not be less than or equal to 1.1 [57]. Finally, the grid spacing was determined for each grid (Table 3.2).

Grid Type	**Grid Spacing (dx, dy, dz)**
Fine	0.09231 metres
Normal	0.1852 metres
Coarse	0.3704 metres

Table 3.2: *Grid Spacing*

To determine the wavelength, a frequency of 300 Hz was selected to reduce computational time (any frequency in the range of a wind turbines nominal output can be used for testing, but higher frequencies require finer grid spacing, requiring an increased compute time, as was noted earlier). To determine the speed of sound, the default parameters of the code were used ($\gamma = 1.4$, R = 287 J/kg-K, T = 293.15K). Using the chosen parameters, the speed of sound was determined to be approximately 343.202 m/s, yielding a wavelength value of 1.144 m. Comparing to the values of grid spacing, each value was smaller than the wavelength, indicating that the chosen values were satisfactory. Hence, upon determining the validity of the chosen sizes, scripts were written to submit to a computer cluster for processing.

Before proceeding further, it was necessary to choose a quantity for which convergence was to be achieved. The maximum sound pressure level at the ground was chosen as this value only appears once in the domain for each grid case. Hence, determining the values resulting the GCS, the first quantity was the order of convergence, which allowed one to determine how closely the exact value was to the numerical solution. Because a fourth order scheme was used, the order of convergence should indicate the same (which it did). However, because of inherent errors in the solution process, one would realistically expect the true order of convergence to be somewhat less. To calculate the order of convergence,

$$a = \frac{\ln(\frac{f_3 - f_2}{f_2 - f_1})}{\ln(m)}. \tag{3.3}$$

In equation 3.3, the f_3 variable represents the most coarse grid case, f_1 represents the finest, and m is the grid convergence ratio. One can use a Richardson Extrapolation to determine a continuum value to which the solution will appear to converge to with successive grid refinements (the value at zero grid

spacing). The continuum value can be found using:

$$f_{h=0} = f_1 + \frac{f_1 - f_2}{m^a - 1}. \tag{3.4}$$

To present the results correctly and to identify any errors in the study, the use of a Grid Convergence Index (GCI) was suggested. The index allows one to determine a percent value of how closely the value to which the successive grids approach compare to the calculated continuum value. The smaller the percentage, the closer the values are to each other, indicating a more desirable solution. To determine the GCI for other (coarser) grids, one can simply scale the value found above using m^a. The F_s variable below refers to the safety factor; because three grids were used, a value of 1.25 was adequate (if two grids were used, a recommended value of 3 was to be used) [57]. To calculate the grid convergence,

$$GCI = \frac{F_s|\frac{f_2 - f_1}{f_1}|}{m^a - 1} * 100\%. \tag{3.5}$$

The results which were collected from this particular study are shown below; they did appear to converge to the expected value of 100 dB. This is important, as it validates that the solver converged to the value predicted by an analytical solution to a point source monopole of the same strength. The error from the GCI was very small albeit indicates that some very minor improvements could be made to the acquired data by perhaps utilizing a different set of grid sizes or a different flow configuration. Nevertheless, the study did illustrate that the code is able to converge to a single value similar to what was expected, as shown in Table 3.3 and 3.4.

Grid Type	**Value [dB]**	**Normalized Grid Spacing**
Fine	100.017	1
Normal	100.924	2
Coarse	107.764	4

Table 3.3: *Sound Pressure Levels at Varied Grid Spacing*

Variable	**Value**
Order of Convergence	2.915
Continuum Value	99.878 dB
GCI	0.173%

Table 3.4: *Grid Spacing*

After obtaining the data, a plot was produced to display the values as well as to show their convergence to a single point. The plot below (Fig. 3.4) contains both the data values which were computed by the code, as well as the continuum value which was determined with the GCS. It is clear that the numerical data was converging to a value close to that expected as the number of nodes in each case was increased.

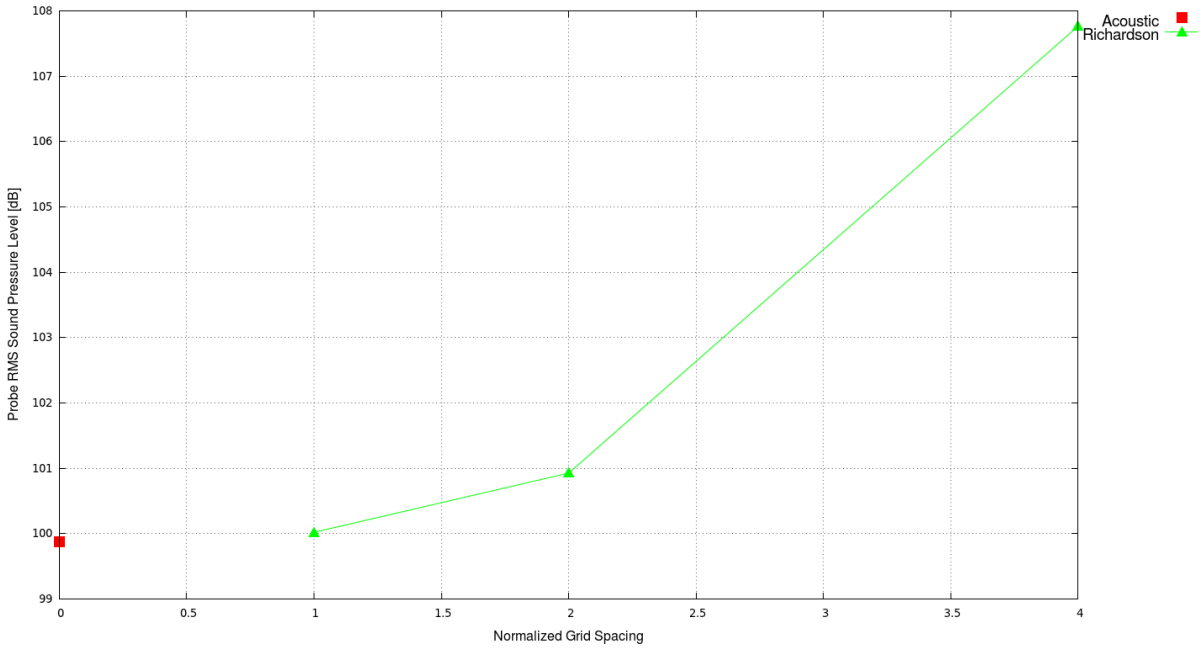


Figure 3.4: *Grid Convergence Study Plot*

3.3 Test Cases

To test the operation of the developed model and to demonstrate how sound propagates under varying conditions, several different flow situations were formulated. Using the GCS as a guide, the different cases were developed such that a suitable level of detail would be achieved in the mesh, while still achieving convergence. For the purposes of this study, 24 different situations were tested with varying ground parameters, source types, and flow velocity profiles; Table 3.5 below lists the values which were kept constant through the range of tests. These parameters were kept constant to ensure that each of the cases would have the same domain, run-time, and source emission characteristics. A frequency of 100 Hz was chosen as a test frequency in consideration of the total noise spectrum emitted from a turbine, but also to expedite compute times. The SWL value which was used corresponds to a wind turbine with a diameter of 101 m, producing 2.3 MW of energy (Siemens SWT-2.3-101) [55]. It was chosen to represent the most commonly installed turbine at wind farms in Ontario at the time of writing, based on data obtained for the various installation sites [10]. For the problem domain and grid spacing, a conservative spacing of 0.15 was used, in comparison to the grid convergence study. This allowed for a resolution of approximately 22 nodes per wavelength. The physical size of the domain was chosen to help illustrate the effects of propagation and absoption, while again attempting to minimize the amount of computing

resources used. Finally, to choose the time period of the simulation to capture a useful amount of information, the total simulation time was the sum of the time required for a wave to propagate to the nearest boundary plus an RMS integration interval equal to 10 periods of the source.

Parameter	**Value**
Number of Nodes	401^3
Domain Size (x,y,z)	60 m
Absorbing Boundary Size (x,y,z)	18 m
Total Run-time	0.19 s
RMS Period	0.1 s
Frequency	100 Hz
SWL	97 dB
Time-Marching Method	Adaptive 4th-Order Runge-Kutta
Timestep	Variable (approx. 0.0001 s)

Table 3.5: *Tested Flow Conditions - Constants*

The second table (Table 3.6), lists the values which were varied. The velocities chosen for the flow, were based on tests conducted by Leloudas et al. [42], and collected at a hub height of 80 m, for a turbine similar to the SWT-2.3-101 (93 m diameter SWT-2.3-93). The chosen values compared favourably to the average wind speeds observed in Ontario [28] at the same height. The ground conditions which were chosen, were based on those provided earlier in Table 2.1 - Ground Condition Parameters. The different conditions allowed comparison of a variety of situations which might exist at a potential turbine installation. In addition to the varying absorbing ground conditions, a reflecting (rigid) ground condition was also tested, resembling a water installation.

To compute the results, the two largers clusters (SHARCNET and SCINET) were used, requiring approximately 24 hours per case. In addition, each of these cases were run with 16 nodes, utilizing a total of 128 processor cores. From testing the code, it was determined that approximately 1.25 kilobytes of RAM per node would be required for storage; the memory allocated was therefore set at one gigabyte per process, at a minimum. To compare the different cases, a slice of the ground plane for each case was taken to obtain the RMS of the acoustic pressure (SPL), excluding the absorbing boundary regions. From these contour plots, one was able to take a further slice extending from the origin outwards in the positive x-direction (downwind). From the second slice, line plots were produced for the different ground conditions. Visible in both the line and contour plots were the effects of the different ground conditions on sound propagation. In the contour plots below, the maximum values are indicated in dark shading, while the minimum values are indicated in light shading; their magnitudes are based on the values shown in the line plots.

The first set of results illustrates sound propagation and absorption quite well. One can see that as the distance from the source increases, the SPL decreases slightly with each wavelength (Fig. 3.5). As well, the amount of absorption between the cases is quite evident. For example, the case which simulates asphalt, is aborbed much less than those simulating grass and a forest. This would be expected, as the porosity Ω of asphalt is less than that for grass or a forest. The same behaviour can be observed for the

Case Number	Ground Type	Source Type	Flow Type	Flow Velocity [m/s]
1	Rigid/Water	Monopole	Uniform	7.9
2	Asphalt	Monopole	Uniform	7.9
3	Sand	Monopole	Uniform	7.9
4	Grass	Monopole	Uniform	7.9
5	Forest	Monopole	Uniform	7.9
6	Snow	Monopole	Uniform	7.9
7	Rigid/Water	Dipole	Uniform	7.9
8	Asphalt	Dipole	Uniform	7.9
9	Sand	Dipole	Uniform	7.9
10	Grass	Dipole	Uniform	7.9
11	Forest	Dipole	Uniform	7.9
12	Snow	Dipole	Uniform	7.9
13	Rigid/Water	Monopole	Power Law	8 @ 80 m
14	Asphalt	Monopole	Power Law	8 @ 80 m
15	Sand	Monopole	Power Law	8 @ 80 m
16	Grass	Monopole	Power Law	8 @ 80 m
17	Forest	Monopole	Power Law	8 @ 80 m
18	Snow	Monopole	Power Law	8 @ 80 m
19	Rigid/Water	Dipole	Power Law	8 @ 80 m
20	Asphalt	Dipole	Power Law	8 @ 80 m
21	Sand	Dipole	Power Law	8 @ 80 m
22	Grass	Dipole	Power Law	8 @ 80 m
23	Forest	Dipole	Power Law	8 @ 80 m
24	Snow	Dipole	Power Law	8 @ 80 m

Table 3.6: *Tested Flow Conditions - Variables*

sand and snow cases, which also clearly show that they are absorbing more than the asphalt. However, the propagation and absorption of the reflecting case (water), appears to do quite the opposite, and falls well below the least absorbing case in terms of propagation; this was also evident in the latter cases for different variations of parameters. A possible explanation for this may be that due to the amount of reflection taking place for a fully reflecting case, the sound waves dissipate sooner in comparison to the other ground conditions; this appears to be evident in the contour plot for water (Fig. 3.6). It was also interesting to take note of the results for the grass and forest cases, where the SPL values were determined to be almost identical, varying slightly from one another. However, this was also expected as both conditions have very similar values of flow resistivity, porosity, and tortuosity. Finally, one would have expected the snow case to be absorbed quite less than it was based on its porosity. However, upon observing its tortuosity value, it is much closer to asphalt. Therefore, it is likely that the tortuosity decreased the amount it absorbed.

Considering the contour plots for each of the tested ground conditions (Fig. 3.6), one can confirm the results presented in the line plots; it is clear that for the different cases tested, the ground conditions are able to absorb the waves propagating from the source. The maximum and minimum pattern which

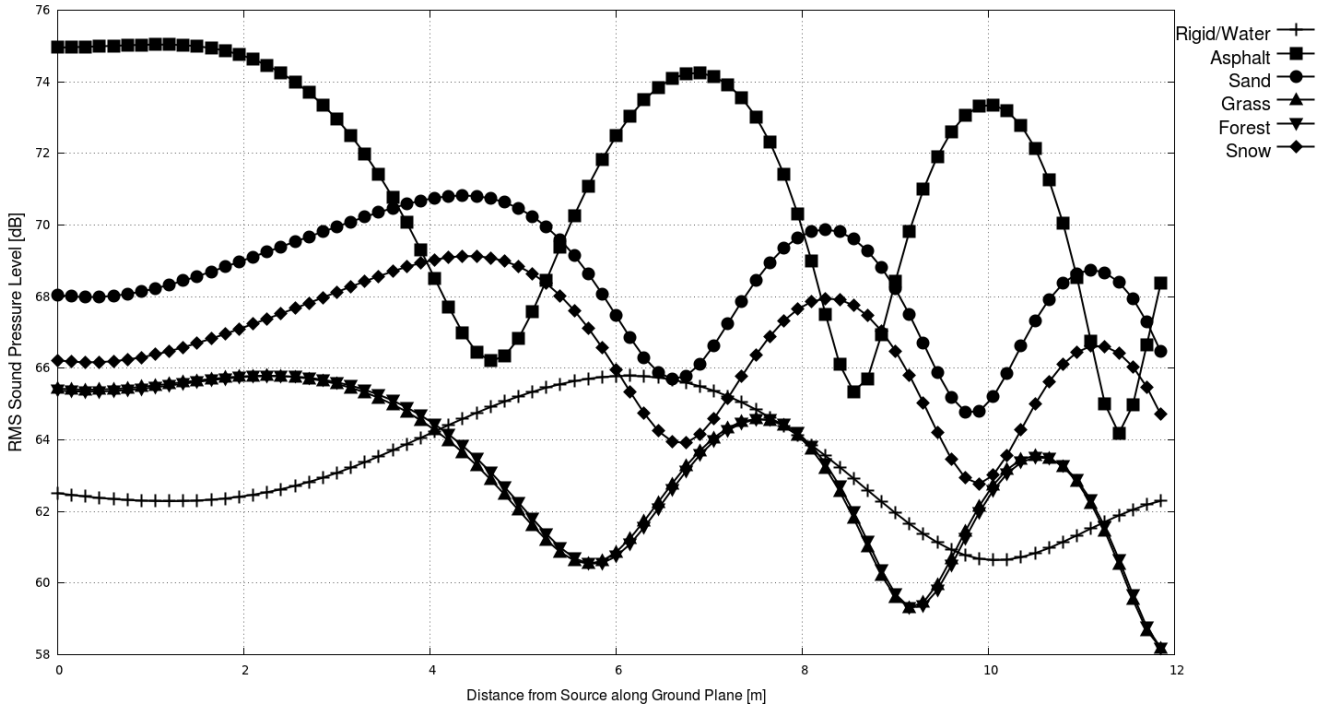
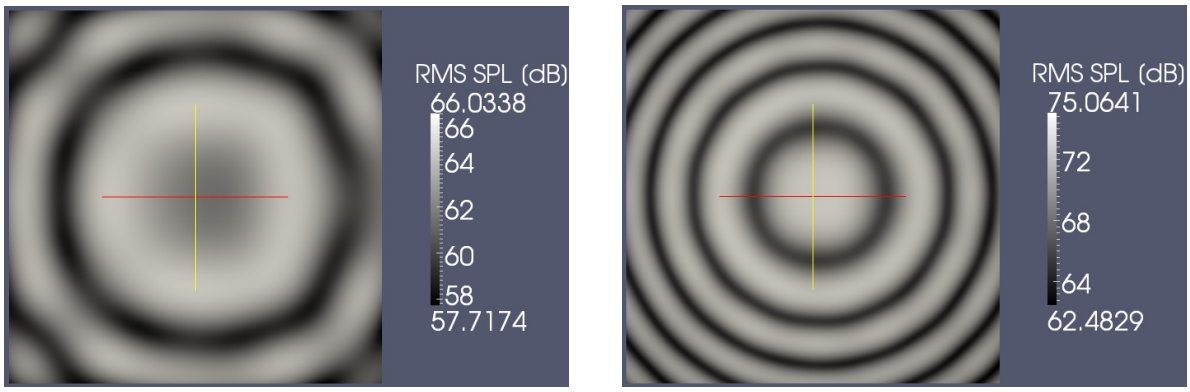


Figure 3.5: *Sound Pressure Level vs. Distance from Source - Monopole, Uniform Flow*

exists in each of the plots is a result of reflections from the waves not fully being absorbed by the ground layer. For surfaces which have higher rates of absorption, the plots tend to be lighter; for the surfaces which have lower rates of absorption, the plots are darker. The effects of the flow velocity on the sound propagation are also evident. Each of the sources are shifted slightly to the right, in the direction of propagation. Although at such a low flow velocity the effects on sound propagation are not drastic, the plots demonstrate the ability of the solver to account for a moving flow. Further, each of the contour plots clearly illustrates the proper directivity patterns of a monopole. As was mentioned earlier, the results for rigid/water surface did not produced favourable results (Fig. 3.6 (a)).

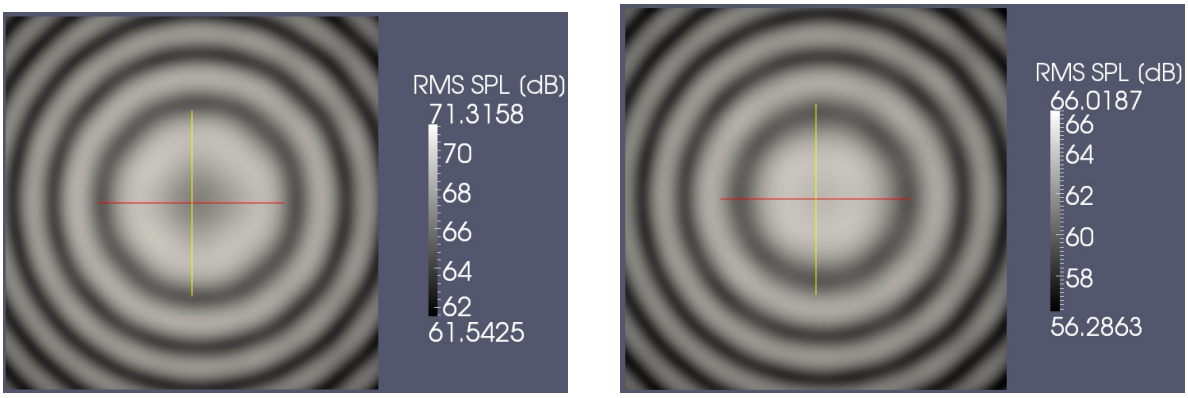
By using these results, one can also apply a linear trendline to the test cases on the line plots in Figure 3.5, to obtain an estimate of the propagation distance. Using this method, a table of propagation distances was formulated (Table 3.7). Although the method is crude, it does provide a very basic procedure for estimating propagation and absorption for the chosen domain size; a more accurate method would be to increase the domain size and observe the decrease in SPL until it reaches an in-audible level. One should also note that a dipole pattern is typically used for a wind turbine, hence the monopole source may be the cause of the under-prediction of values.

Observing the second set of results beginning with Fig. 3.7, one can immediately notice the difference in the amount of sound being radiated away from the source. As was mentioned earlier, this is due to the strong near-field region for dipoles. As a result, the amplitude of the sound being radiated diminishes



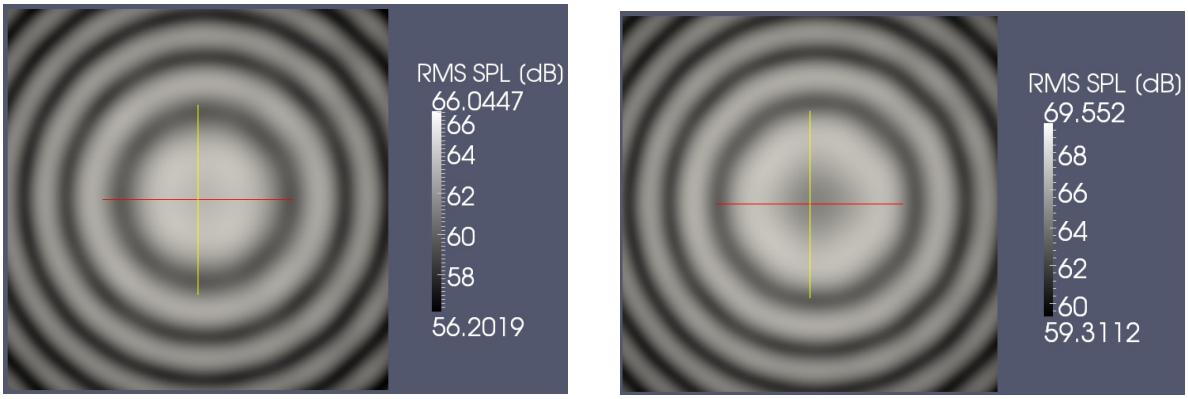
(a) (b)

Figure 3.6: Uniform Flow Monopole, Rigid/Water (left) and Asphalt (right)



(c) (d)

Figure 3.6: Uniform Flow Monopole, Sand (left) and Grass (right)



(e) (f)

Figure 3.6: Uniform Flow Monopole, Forest (left) and Snow (right)

Ground Type	Propagation Distance [m]
Rigid/Water	1240
Asphalt	298
Sand	660
Grass	298
Forest	298
Snow	660

Table 3.7: *Estimated Propagation Distances - Monopole, Uniform Flow*

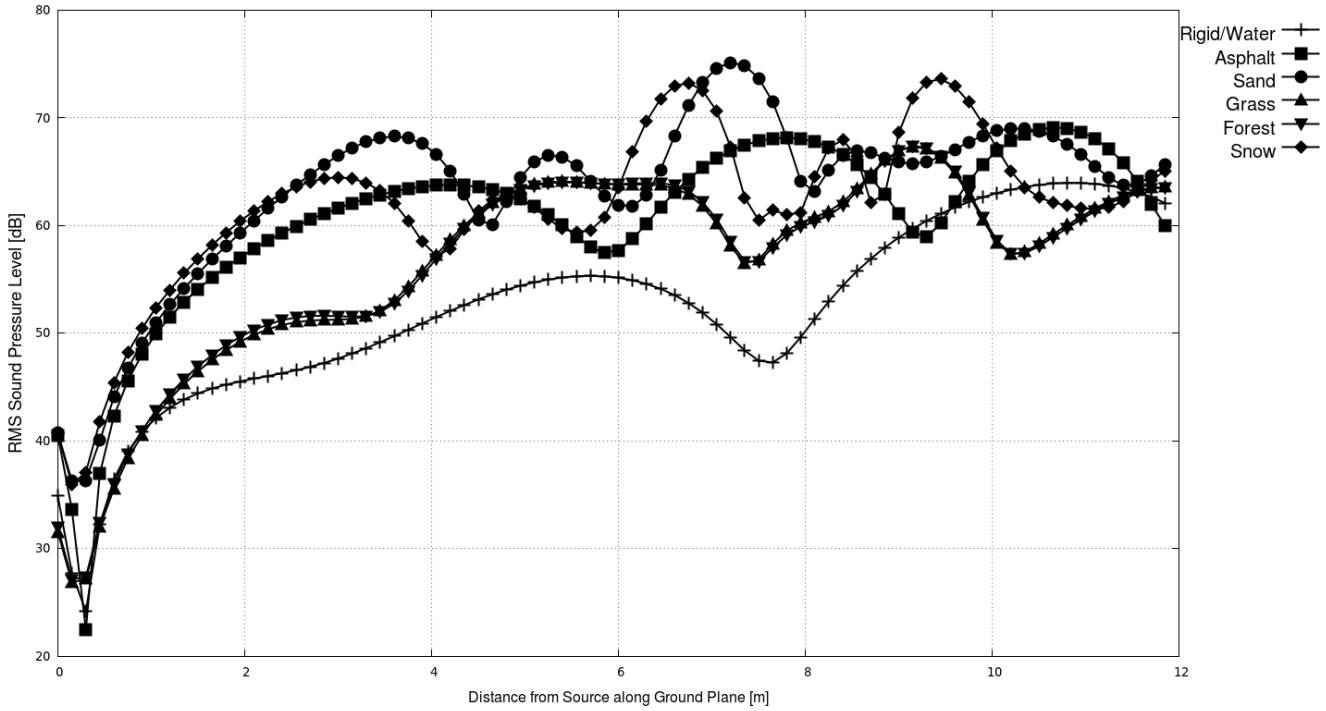
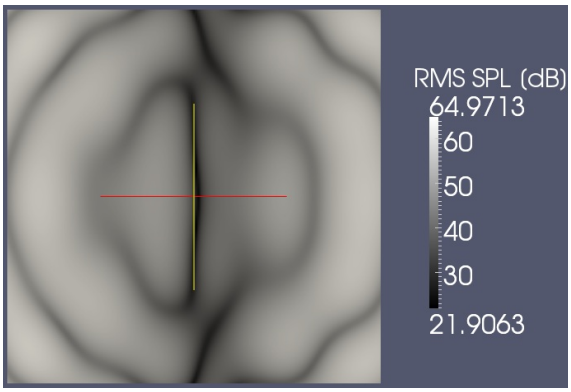
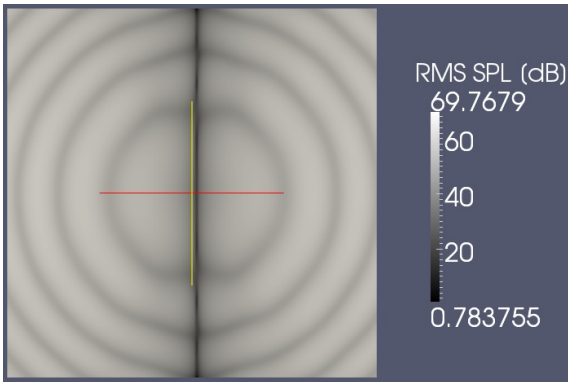


Figure 3.7: *Sound Pressure Level vs. Distance from Source - Dipole, Uniform Flow*

much slower than for a monopole source. Another aspect of the plot involves the initial profiles of each of the cases. In comparison to the monopole plots, the dipole initially starts at a much lower SPL value. However, this is due to the phase difference between the monopole and dipole. A monopole is modelled as a single point source in space, that emits sound only from this point. A dipole, works in a similar fashion acting as a point at a single location; however, the emission direction varies and is out of phase by 180 degrees. Therefore, a dipole will emit a wave in one direction (positive x-direction), and then emit in the other (negative x-direction), but not at the same time. Hence, as can be seen on the line plot, the implementation of the dipole does follow that trend in that there is a substantially decreased area of sound emission at the source, which then increases as the distance from the source increases.

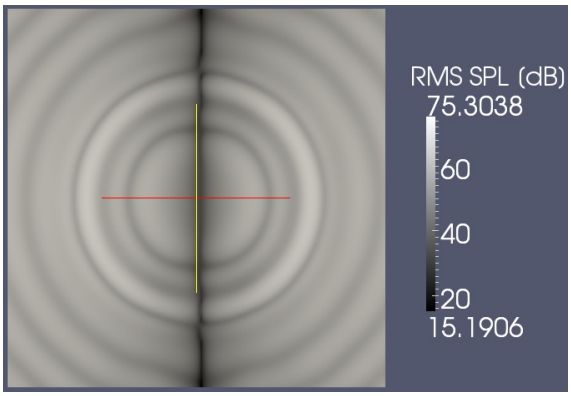


(a)

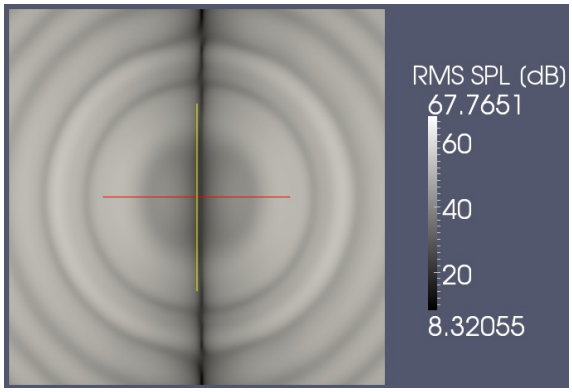


(b)

Figure 3.8: Uniform Flow Dipole, Rigid/Water (left) and Asphalt (right)

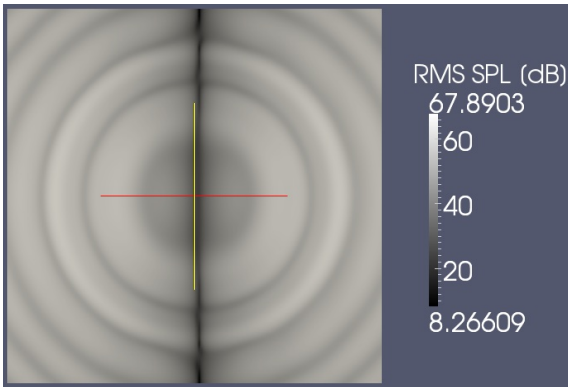


(c)

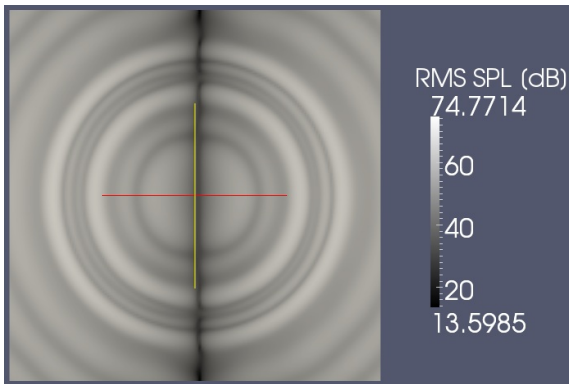


(d)

Figure 3.8: Uniform Flow Dipole, Sand (left) and Grass (right)



(e)



(f)

Figure 3.8: Uniform Flow Dipole, Forest (left) and Snow (right)

In comparison to the monopole cases, the effects of the porosity were not as pronounced. It was not immediately clear from both the line or contour plot data, which ground condition was absorbing the wave the most (Fig. 3.8). It was also interesting to note that the cases of 'grass' and 'forest' nearly mimicked what took place for the monopole cases, in that they both produced results that were nearly identical. However, as mentioned earlier, both cases have very similar ground characteristics. Finally, as mentioned for the monopole case, the results for the rigid/water surface did not produce satisfactory results, indicating quite the opposite of what was expected for both the line and contour plots (Fig. 3.8 (a)).

The contour plots for the dipole case were shown in Fig. 3.8; although not as clear as in the monopole cases they did give a rough indication of the effects of the ground conditions on propagation from the source, as indicated by the darker plots for the reflecting cases, and the lighter plots for absorbing cases. Further, as briefly mentioned above, for dipoles there is normally a 'dead' zone at $\frac{\pi}{2}$ and $\frac{3\pi}{2}$, where nearly no propagation exists. These regions are due to interference from the two lobes of the dipole. The region is visible in the contour plots, indicating the two monopole sources making up the dipole are out of phase, and indicating proper operation of the solver in that regard.

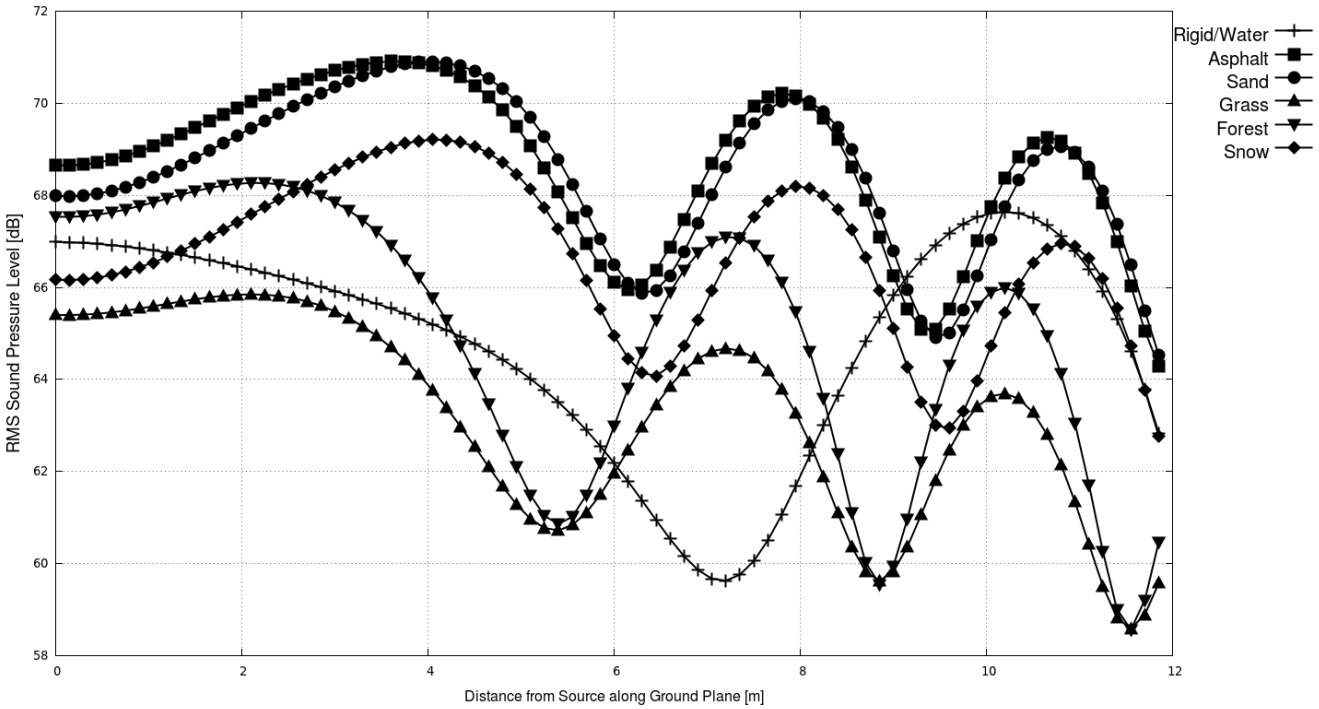
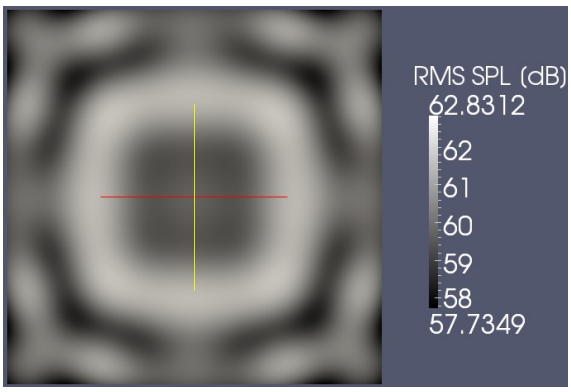
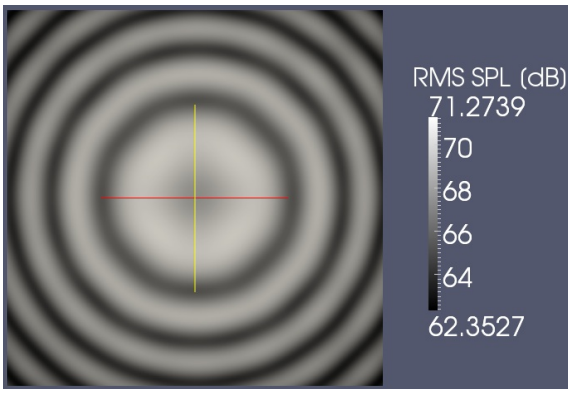


Figure 3.9: *Sound Pressure Level vs. Distance from Source - Monopole, Power Law Flow*

The third set of results shifted focus slightly in comparison to the first two sets. A power law profile resembling a case more indicative of a real-world flow scenario was used beginning with the third set of cases. Beginning with Fig. 3.9, in comparison to the first set of monopole cases with uniform flow, the

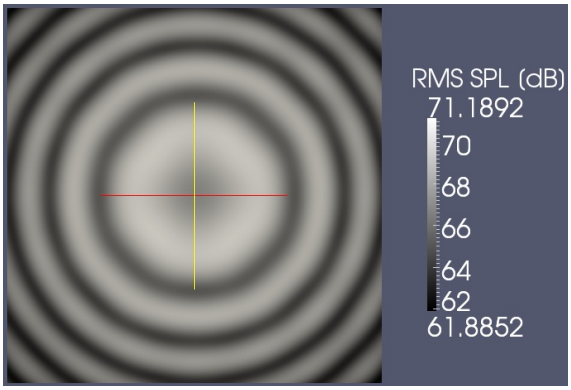


(a)



(b)

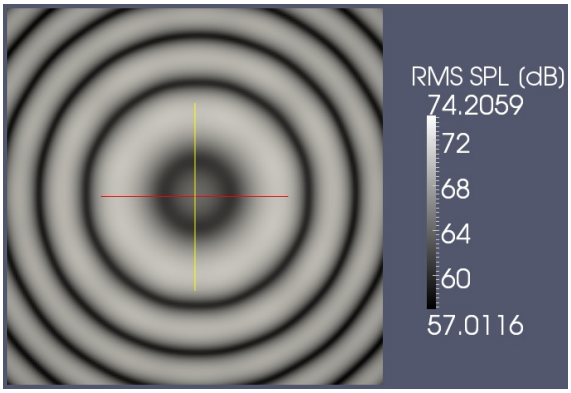
Figure 3.10: Power Law Flow Monopole, Rigid/Water (left) and Asphalt (right)



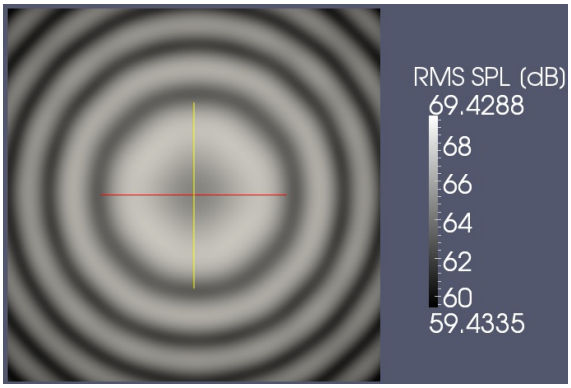
(c)

(d)

Figure 3.10: Power Law Flow Monopole, Sand (left) and Grass (right)



(e)



(f)

Figure 3.10: Power Law Flow Monopole, Forest (left) and Snow (right)

SPL values decreased slightly, resulting in some decreased amplitudes being produced. This is likely due to the change in flow profile between a uniform flow and power law flow, near the ground. Otherwise, as was observed earlier, the effects of propagation and absorption of the flow are evident. The asphalt case once again is absorbed the least, while the grass and forest cases absorb the most. Of note is the forest ground case, which saw an increase in the peak amplitude values during its propagation in comparison to the uniform flow case. Further, it is also interesting to note that the asphalt propagation is nearly the same as that for the sand; although the values are similar, this is in contrast to the monopole case for which the values were apart from each other. As mentioned earlier, the rigid/water case once again did not perform as expected. Nevertheless, the effects of absorption on propagation were evident.

The contour plots produced for the third set of cases were shown in Fig. 3.10. As was indicated in the monopole cases for uniform flow, the effects of sound propagation could be seen on the contour plots. However, for the power law flow, the changes between each of the ground conditions were not as pronounced. Nevertheless, the effects of the flow velocity were evident on the contour plots, as they were for the monopole uniform flow cases. Each of the propagation patterns were modelled correctly, but shifted slightly to the right, in the direction of the flow. Lastly, the rigid/water case again did not appear to generate the expected trend of reflecting the most.

Using the same methodology as for the uniform flow case, an estimate of propagation distances was produced (Table 3.8).

Ground Type	**Propagation Distance [m]**
Rigid/Water	1000
Asphalt	540
Sand	540
Grass	290
Forest	248
Snow	600

Table 3.8: *Estimated Propagation Distances - Monopole, Power Law Flow*

The final set of data produced was for a dipole source with a power law flow. Shown in the line plot (Fig. 3.11), and as was mentioned in the results for the monopole power flow above, it was expected that the SPL values and amplitudes would be lower at the ground plane. This was the case with the dipole source as well. Furthermore, due to the strong near-field region, the effects of absorption were much less pronounced in comparison to the monopole cases. However, the values generated did not produced results as expected in the monopole cases; the porosity of the ground layer did not have a discernible effect on the amount of absorption for each case. However, it was of interest to note that as observed with some of the ground situations in the other cases, the asphalt, sand, and snow ground conditions produced nearly the same propagation pattern, while not possessing the same ground parameters; in fact, asphalt and snow were the least absorbing and most absorbing conditions, respectively, with the exception of the rigid/water surface which again did not produce any discernible information in regards to propagation. Based on the results obtained, the grass scenario appeared to have the least amount of

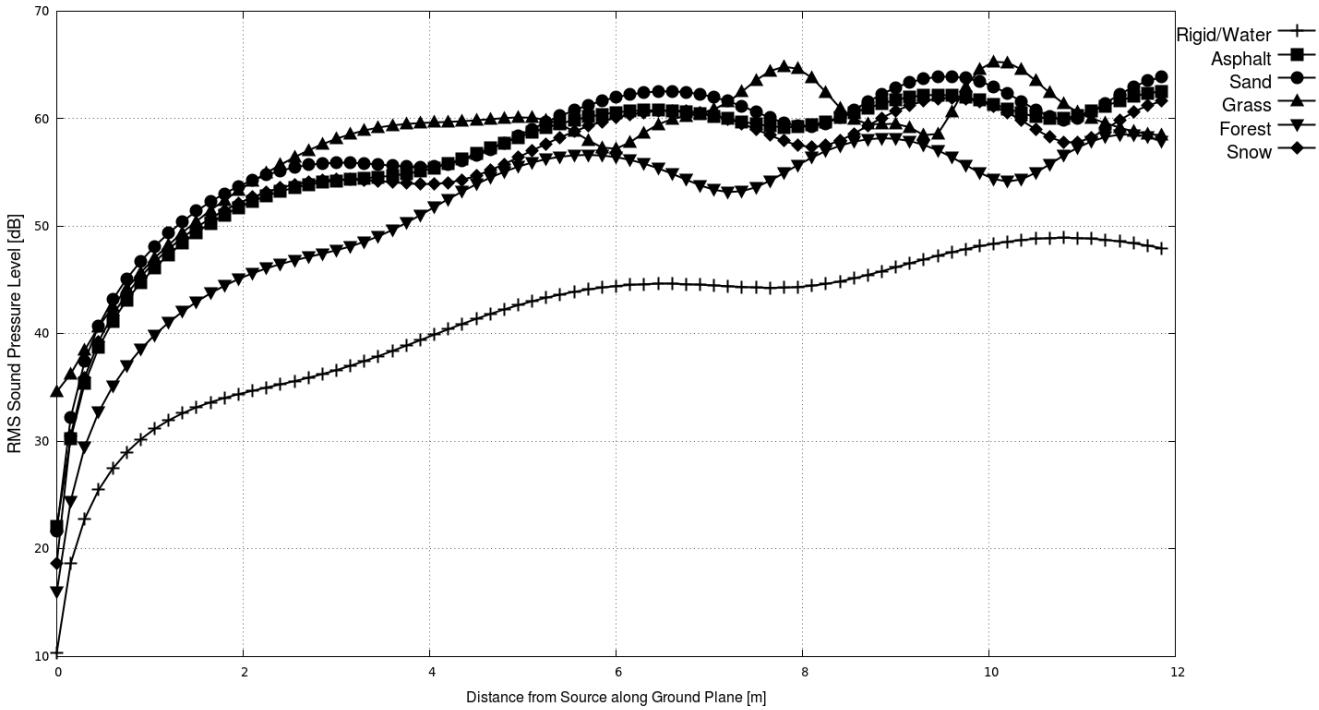
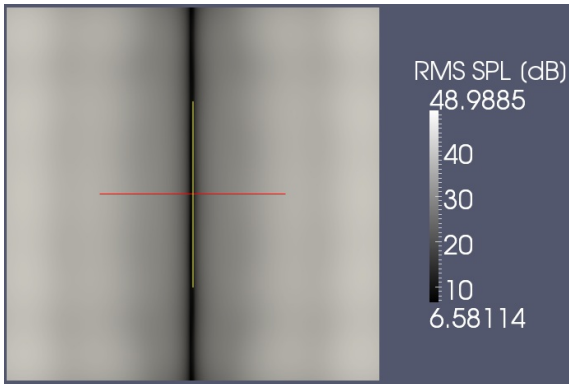


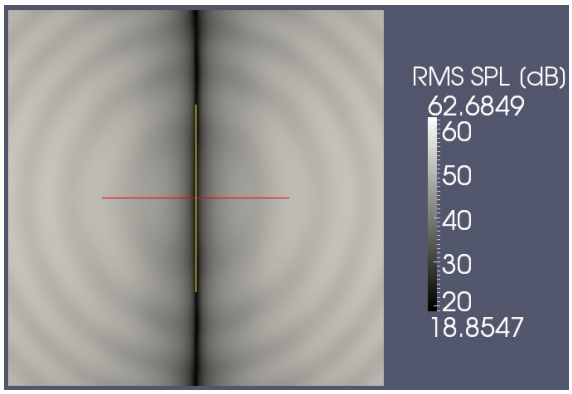
Figure 3.11: *Sound Pressure Level vs. Distance from Source - Dipole, Power Law Flow*

absorption, which was clearly not correct.

Finally, taking note of the contour plots for the final set of cases (Fig. 3.12), as was shown earlier with the uniform flow dipole, the modelling of the dipole indicated an area of minimum SPL at $\frac{\pi}{2}$ and $\frac{3\pi}{2}$. As well, the two monopole sources out of phase producing the dipole were clearly visible. However, as was seen in the line plots, the data that was obtained from the cases did not generate a significantly useful amount of information. It was best interpreted as a general case scenario simply indicating the basic propagation pattern resulting from a dipole source.

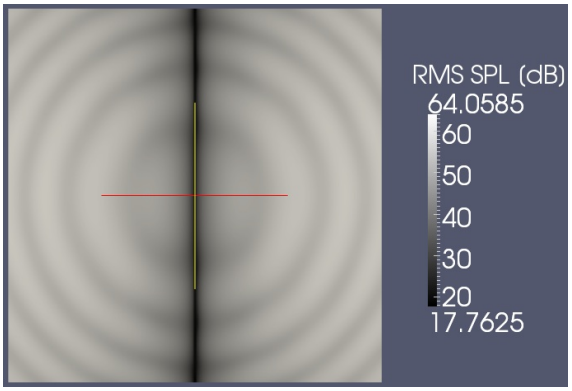


(a)

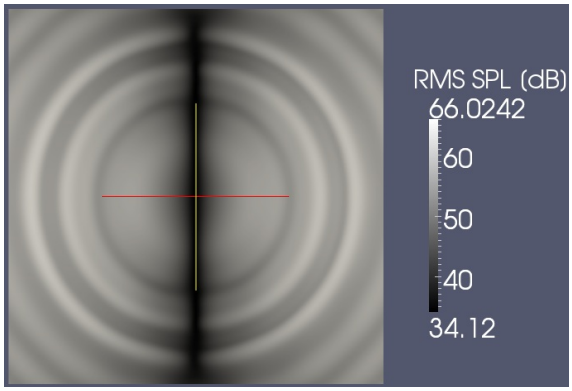


(b)

Figure 3.12: Power Law Flow Dipole, Rigid/Water (left) and Asphalt (right)

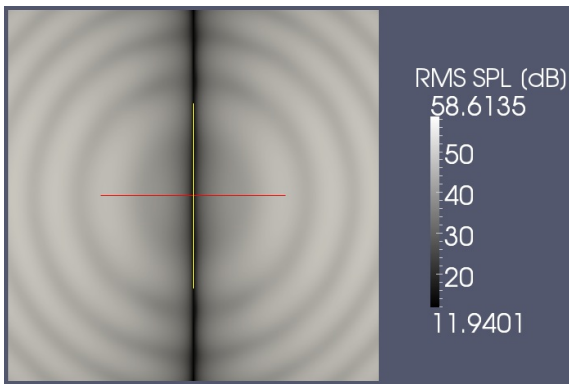


(c)

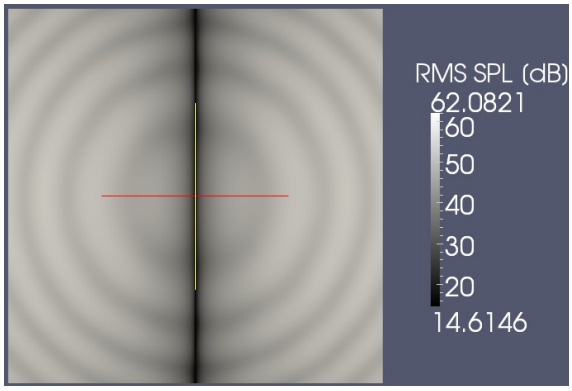


(d)

Figure 3.12: Power Law Flow Dipole, Sand (left) and Grass (right)



(e)



(f)

Figure 3.12: Uniform Flow Monopole, Forest (left) and Snow (right)

Chapter 4

Conclusion

The goal of this study was to produce a detailed sound propagation model, that would be used to help predict setback distances in the development of wind turbines. A detailed numerical model was produced, and results were generated to test its operation. It was found that the model was able to show sound propagation from both a monopole and dipole source. Furthermore, it was also able to show the effects of absorption due to varying ground conditions, as well as the effects on sound propagation from using different flow profiles. It should again be emphasized however, that this model is a starting point to a larger development; the framework has been developed for larger and more complex simulations to take place and be tested and is by no means a final build ready to be used for any real-world situation which one may encounter. There are several immediate aspects of this study which could be improved upon.

4.1 Recommendations

Perhaps the largest issue would be the use of larger simulation domains and finer grids to test sound propagation characteristics as the wave moves through a domain and dissipates near the boundaries. In comparison to the domains used for testing in this study, a larger domain would be a much better indicator as to whether the correct sound propagation characteristics are taking place further downstream of the turbine. Furthermore, with larger domain simulations, the area around which a turbine is installed could be used to collected empirical data which would then be used for validation to truly determine whether the current setback distances are adequate for use in Ontario. With the increased and more detailed domain, one may also consider implementing a different set of boundary conditions based on a Perfectly Matched Layer (PML). The PML boundary condition is much more effective at eliminating the reflected wave from the exterior boundary. However, the implementation of PML is not trivial, but could be of interest to implement in a FDTD setting.

Although a flat terrain was used for this study, and was because most of the terrain in southern Ontario is, it would be worthwhile to test the implementation of variable terrain. The changing terrain

could be obtained from GIS data, collected for a different location in the province, potentially being considered for wind turbine development. Furthermore, with the inclusion of a variable terrain, it would also be beneficial to test the effects of upstream and downstream obstructions on the sound propagation from wind turbines. The obstructions may result in refractions and reflections which may potentially alter the sound characteristics of the turbines due to changes in the incoming flow.

In terms of validation, it would be beneficial to obtain the noise profile for the particular turbine under study (SWT-2.3-101). This type of data would allow one to develop a more detailed Class II or III sound generation model. A comparison could then be made to a real-world installation. With a detailed noise model, the entire spectrum of specific noise mechanisms emitted from the turbine could also be captured (steady thickness and loading, inflow, airfoil-self). The noise model used for this study was based upon using a single frequency for the entire spectrum emitted. It should be noted however, that the functionality for testing the entire frequency range was being implemented into the solver at the time of writing, for both monopole and dipole noise sources.

Appendix 1

Finite-Differences Approximations

The approximations to the spatial derivatives are provided below (from left to right). Mesh indices are denoted by (i, j, k) while array indices are denoted by $[i, j, k]$.

1.1 Sound Propagation Equations

1.1.1 Equation 2.25

$$\rho(i, j, k) = \rho[i, j, k], \tag{1.1}$$

$$c(i, j, k) = c[i, j, k], \tag{1.2}$$

$$Q(i, j, k) = Q[i, j, k], \tag{1.3}$$

$$\frac{\partial w_x}{\partial x}(i, j, k) = \frac{w_x[i, j, k] - w_x[i-1, j, k]}{\Delta x}, \tag{1.4}$$

$$\frac{\partial w_y}{\partial y}(i, j, k) = \frac{w_y[i, j, k] - w_y[i, j-1, k]}{\Delta y}, \tag{1.5}$$

$$\frac{\partial w_z}{\partial z}(i, j, k) = \frac{w_z[i, j, k] - w_z[i-1, j, k-1]}{\Delta z}, \tag{1.6}$$

$$v_x(i, j, k) = \frac{v_x[i, j, k] + v_x[i-1, j, k]}{2}, \tag{1.7}$$

$$\frac{\partial P}{\partial x}(i, j, k) = \frac{P[i+1, j, k] - P[i-1, j, k]}{2\Delta x}, \tag{1.8}$$

$$v_y(i,j,k) = \frac{v_x[i,j,k] + v_x[i,j-1,k]}{2}, \tag{1.9}$$

$$\frac{\partial P}{\partial y}(i,j,k) = \frac{P[i,j+1,k] - P[i,j-1,k]}{2\Delta y}, \tag{1.10}$$

$$v_z(i,j,k) = \frac{v_x[i,j,k] + v_x[i,j,k-1]}{2}, \tag{1.11}$$

$$\frac{\partial P}{\partial z}(i,j,k) = \frac{P[i,j,k+1] - P[i,j,k-1]}{2\Delta z}. \tag{1.12}$$

1.1.2 Equation 2.26

$$F_x(i,j,k) = F_x[i,j,k], \tag{1.13}$$

$$b_x(i+\frac{1}{2},j,k) = \frac{b_x[i+1,j,k] + b_x[i,j,k]}{2}, \tag{1.14}$$

$$\frac{\partial P_x}{\partial x}(i+\frac{1}{2},j,k) = \frac{P[i+1,j,k] - P[i,j,k]}{\Delta x}, \tag{1.15}$$

$$v_x(i+\frac{1}{2},j,k) = v_x[i,j,k], \tag{1.16}$$

$$\frac{\partial w_x}{\partial x}(i+\frac{1}{2},j,k) = \frac{w_x[i+1,j,k] - w_x[i-1,j,k]}{2\Delta x}, \tag{1.17}$$

$$v_y(i+\frac{1}{2},j,k) = \frac{v_y[i,j,k] + v_y[i+1,j,k] + v_y[i,j-1,k] + v_y[i+1,j-1,k]}{4}, \tag{1.18}$$

$$\frac{\partial w_x}{\partial y}(i+\frac{1}{2},j,k) = \frac{w_x[i,j+1,k] - w_x[i,j-1,k]}{2\Delta y}, \tag{1.19}$$

$$v_z(i+\frac{1}{2},j,k) = \frac{v_z[i,j,k] + v_y[i+1,j,k] + v_y[i,j,k-1] + v_y[i+1,j,k-1]}{4}, \tag{1.20}$$

$$\frac{\partial w_z}{\partial z}(i+\frac{1}{2},j,k) = \frac{w_x[i,j,k+1] - w_x[i,j,k-1]}{2\Delta z}, \tag{1.21}$$

$$w_x(i+\frac{1}{2},j,k) = w_x[i,j,k], \tag{1.22}$$

$$\frac{\partial v_x}{\partial x}(i+\frac{1}{2},j,k) = v_x[i,j,k], \tag{1.23}$$

$$w_y(i+\frac{1}{2},j,k) = \frac{w_y[i,j,k] + w_y[i+1,j,k] + w_y[i,j-1,k] + w_y[i+1,j-1,k]}{4}, \tag{1.24}$$

$$\frac{\partial v_x}{\partial y}(i+\frac{1}{2},j,k) = v_x[i,j,k], \tag{1.25}$$

$$v_z(i+\frac{1}{2},j,k) = \frac{v_z[i,j,k] + v_z[i+1,j,k] + v_z[i,j,k-1] + v_z[i+1,j,k-1]}{4}, \tag{1.26}$$

$$\frac{\partial v_x}{\partial z}(i+\frac{1}{2},j,k) = v_x[i,j,k]. \tag{1.27}$$

1.1.3 Equation 2.27

$$F_y(i,j,k) = F_y[i,j,k], \tag{1.28}$$

$$b_y(i,j+\frac{1}{2},k) = \frac{b_y[i,j+1,k] + b_y[i,j,k]}{2}, \tag{1.29}$$

$$\frac{\partial P_y}{\partial y}(i,j+\frac{1}{2},k) = \frac{P[i,j+1,k] - P[i,j,k]}{2\Delta y}, \tag{1.30}$$

$$v_x(i,j+\frac{1}{2},k) = \frac{v_x[i,j,k] + v_x[i,j+1,k] + v_x[i-1,j+1,k] + v_x[i-1,j,k]}{4}, \tag{1.31}$$

$$\frac{\partial w_y}{\partial x}(i,j+\frac{1}{2},k) = \frac{w_y[i+1,j,k] - w_x[i-1,j,k]}{2\Delta x}, \tag{1.32}$$

$$v_y(i,j+\frac{1}{2},k) = v_y[i,j,k], \tag{1.33}$$

$$\frac{\partial w_y}{\partial y}(i,j+\frac{1}{2},k) = \frac{w_y[i,j+1,k] - w_x[i,j-1,k]}{2\Delta y}, \tag{1.34}$$

$$v_z(i,j+\frac{1}{2},k) = \frac{v_z[i,j,k] + v_z[i,j+1,k] + v_z[i,j+1,k] + v_z[i,j,k-1]}{4}, \tag{1.35}$$

$$\frac{\partial w_y}{\partial z}(i,j+\frac{1}{2},k) = \frac{w_y[i,j,k+1] - w_x[i,j,k-1]}{2\Delta z}, \tag{1.36}$$

$$w_x(i,j+\frac{1}{2},k) = \frac{w_x[i,j,k] + w_x[i,j+1,k] + w_x[i-1,j+1,k] + w_x[i-1,j,k]}{4}, \tag{1.37}$$

$$\frac{\partial v_y}{\partial x}(i,j+\frac{1}{2},k) = v_y[i,j,k], \tag{1.38}$$

$$w_y(i, j + \frac{1}{2}, k) = w_y[i, j, k], \tag{1.39}$$

$$\frac{\partial v_y}{\partial y}(i, j + \frac{1}{2}, k) = v_y[i, j, k], \tag{1.40}$$

$$w_z(i, j + \frac{1}{2}, k) = \frac{w_z[i, j, k] + w_z[i, j + 1, k] + w_z[i, j + 1, k - 1] + w_z[i, j, k - 1]}{4}, \tag{1.41}$$

$$\frac{\partial v_y}{\partial z}(i, j + \frac{1}{2}, k) = v_y[i, j, k]. \tag{1.42}$$

1.1.4 Equation 2.28

$$F_z(i, j, k) = F_z[i, j, k], \tag{1.43}$$

$$b_z(i, j, k + \frac{1}{2}) = \frac{b_z[i, j, k + 1] + b_z[i, j, k]}{2}, \tag{1.44}$$

$$\frac{\partial P_z}{\partial z}(i, j, k + \frac{1}{2}) = \frac{P[i, j, k + 1] - P[i, j, k]}{2\Delta z}, \tag{1.45}$$

$$v_x(i, j, k + \frac{1}{2}) = \frac{v_x[i, j, k] + v_x[i - 1, j, k] + v_x[i, j, k + 1] + v_x[i - 1, j, k + 1]}{4}, \tag{1.46}$$

$$\frac{\partial w_z}{\partial x}(i, j, k + \frac{1}{2}) = \frac{w_z[i + 1, j, k] - w_z[i - 1, j, k]}{2\Delta x} \tag{1.47}$$

$$v_y(i, j, k + \frac{1}{2}) = \frac{v_y[i, j, k] + v_y[i, j - 1, k] + v_y[i, j, k + 1] + v_y[i - 1, j, k + 1]}{4}, \tag{1.48}$$

$$\frac{\partial w_z}{\partial y}(i, j, k + \frac{1}{2}) = \frac{w_z[i, j + 1, k] - w_z[i, j - 1, k]}{2\Delta y}, \tag{1.49}$$

$$v_z(i, j, k + \frac{1}{2}) = v_z[i, j, k], \tag{1.50}$$

$$\frac{\partial w_z}{\partial z}(i, j, k + \frac{1}{2}) = \frac{w_z[i, j, k + 1] - w_z[i, j, k - 1]}{2\Delta z}, \tag{1.51}$$

$$w_x(i, j, k + \frac{1}{2}) = \frac{w_x[i, j, k] + w_x[i - 1, j, k] + w_x[i, j, k + 1] + w_x[i - 1, j, k + 1]}{4}, \tag{1.52}$$

$$\frac{\partial v_z}{\partial x}(i, j, k + \frac{1}{2}) = v_z[i, j, k], \tag{1.53}$$

$$w_y(i,j,k+\frac{1}{2}) = \frac{w_y[i,j,k] + w_y[i,j-1,k] + w_y[i,j,k+1] + w_y[i,j-1,k+1]}{4}, \tag{1.54}$$

$$\frac{\partial v_z}{\partial y}(i,j,k+\frac{1}{2}) = v_z[i,j,k], \tag{1.55}$$

$$w_z(i,j,k+\frac{1}{2}) = w_z[i,j,k], \tag{1.56}$$

$$\frac{\partial v_z}{\partial z}(i,j,k+\frac{1}{2}) = v_z[i,j,k]. \tag{1.57}$$

1.2 Absorbing Boundary Conditions

1.2.1 Equation 2.37

$$\kappa_e(i,j,k) = \kappa_e[i,j,k] \tag{1.58}$$

$$\frac{\partial w_x}{\partial x}(i,j,k) = \frac{w_x[i,j,k] - w_x[i-1,j,k]}{\Delta x}, \tag{1.59}$$

$$\frac{\partial w_y}{\partial y}(i,j,k) = \frac{w_y[i,j,k] - w_y[i,j-1,k]}{\Delta y}, \tag{1.60}$$

$$\frac{\partial w_z}{\partial z}(i,j,k) = \frac{w_z[i,j,k] - w_z[i-1,j,k-1]}{\Delta z}, \tag{1.61}$$

1.2.2 Equation 2.38

$$b_e(i+\frac{1}{2},j,k) = \frac{b_e[i+1,j,k] + b_e[i,j,k]}{2} \tag{1.62}$$

$$\sigma_x(i,j,k) = \sigma_x[i,j,k] \tag{1.63}$$

$$w_x(i+\frac{1}{2},j,k) = w_x[i,j,k] \tag{1.64}$$

$$\frac{\partial p_x}{\partial x}(i+\frac{1}{2},j,k) = \frac{p_x[i+1,j,k] - p_x[i,j,k]}{\Delta x}, \tag{1.65}$$

1.2.3 Equation 2.39

$$b_e(i,j+\frac{1}{2},k) = \frac{b_e[i,j+1,k] + b_e[i,j,k]}{2} \tag{1.66}$$

$$\sigma_y(i,j,k) = \sigma_y[i,j,k] \tag{1.67}$$

$$w_y(i, j + \frac{1}{2}, k) = \frac{w_y[i, j, k] + w_y[i + 1, j, k] + w_y[i, j - 1, k] + w_y[i + 1, j - 1, k]}{4} \tag{1.68}$$

$$\frac{\partial p_y}{\partial x}(i, j + \frac{1}{2}, k) = \frac{p_y[i, j + 1, k] - p_y[i, j, k]}{\Delta y}, \tag{1.69}$$

1.2.4 Equation 2.40

$$b_e(i, j, k + \frac{1}{2}) = \frac{b_e[i, j, k + 1] + b_e[i, j, k]}{2} \tag{1.70}$$

$$\sigma_z(i, j, k) = \sigma_z[i, j, k] \tag{1.71}$$

$$w_z(i, j, k + \frac{1}{2}) = \frac{w_z[i, j, k] + w_z[i + 1, j, k] + w_z[i, j, k - 1] + w_z[i + 1, j, k - 1]}{4} \tag{1.72}$$

$$\frac{\partial p_z}{\partial z}(i, j, k + \frac{1}{2}, k) = \frac{p_y[i, j, k + 1] - p_y[i, j, k]}{\Delta z}, \tag{1.73}$$

Appendix 2

Sample Submission Scripts

2.1 SHARCNET

```
#!/bin/sh

. ~/bin/common.sh

load_petsc
set_work_path

sqsub -r 24h --mpp=1.5G -n 128 -m -q mpi -o jobgcs -j jobgcs \
acoustic \
-N_x 651 -N_y 651 -N_z 651 \
-x_max 150 -x_min -150 \
-y_max 150 -y_min -150 \
-z_max 150 -z_min -150 \
-x_layer 18 -y_layer 18 -z_layer 18 \
-t_max 0.47 \
-T 0.03333 \
-ts_type RK4 \
-dt 1E-4 \
-mean mean_uniform \
-source monopole \
-mono_f 300 \
-measured_swl 97 \
-mono_origin_x 0 \
-mono_origin_y 0 \
-mono_origin_z 0 \
-mean_uniform_M 0.023
-ground_flow_resistivity 3E+7 \
-ground_porosity 0.1 \
-ground_tortuosity 3.2 \
-rms_probe 10 \
-log_summary
```

2.2 SCINET

```
#!/bin/bash
#
#PBS -N jobgcs
#PBS -l nodes=16:ppn=8,walltime=24:00:00
#PBS -j oe

. ~/.bashrc

echo_hosts_file

load_petsc

cd $PBS_O_WORKDIR

# Some parameter settings
Nx=651
Ny=651
Nz=651
tmax=0.47
T=0.03333

mpirun -np ${NSLOTS} -hostfile ${PBS_NODEFILE} acoustic \
-N_x ${Nx} -N_y ${Ny} -N_z ${Nz} \
-x_max 150 -x_min -150 -y_max 150 -y_min -150 -z_max 150 -z_min -150 \
-x_layer 18 -y_layer 18 -z_layer 18 \
-t_max ${tmax} \
-T ${T} \
-ts_type RK4 \
-dt 1E-4 \
-mean mean_uniform \
-source monopole \
-mono_f 300 \
-measured_swl 97 \
-mono_origin_x 0 -mono_origin_y 0 -mono_origin_z 0 \
-mean_uniform_M 0.023 \
-ground_flow_resistivity 3E+7 \
-ground_porosity 0.1 \
-ground_tortuosity 3.2 \
-rms_probe 10 \
-log_summary
```

2.3 Local Cluster

```
#/bin/bash
#
#$ -N jobgcs
#$ -q parallel
```

```
5 #$ -pe mpi 28
6 #$ -l h_vmem=1.4G
7 #
8
9 . $TMPDIR/env.sh
10
11 cd ${SGE_O_WORKDIR}
12
13 mpirun -np ${NSLOTS} acoustic \
14 -N_x 651 -N_y 651 -N_z 651 \
15 -x_max 150 -x_min -150 \
16 -y_max 150 -y_min -150 \
17 -z_max 150 -z_min -150 \
18 -x_layer 18 -y_layer 18 -z_layer 18 \
19 -t_max 0.47 \
20 -T 0.03333 \
21 -ts_type RK4 \
22 -dt 1E-4 \
23 -mean mean_uniform \
24 -source monopole \
25 -mono_f 300 \
26 -measured_swl 97 \
27 -mono_origin_x 0 \
28 -mono_origin_y 0 \
29 -mono_origin_z 0 \
30 -mean_uniform_M 0.023
31 -ground_flow_resistivity 3E+7 \
32 -ground_porosity 0.1 \
33 -ground_tortuosity 3.2 \
34 -rms_probe 10 \
35 -log_summary
```

Appendix 3

Source Code

3.1 main.c

```
1 #include<stdlib.h>
2 #include<assert.h>
3
4 #include<petsc.h>
5
6 #include"param.h"
7 #include"setup.h"
8 #include"timemarch.h"
9 #include"io.h"
10 #include"mean.h"
11 #include"source.h"
12
13 /*! \mainpage FDTD Solver
14     \section intro_sec Introduction
15     This is a solver for acoustic pressure and velocity using the finite-difference time
           -domain FDTD method. Various acoustic source models are available. This code is
           experimental.
16
17     \section install_sec Installation
18     This solver uses the GNU autotools package. If you have a copy of the source code,
           simply run: ./configure; make; make install.
19
20
21 */
22
23 /** Global variable to control execution of the time marching method. */
24 char RunSolver=1;
25
26 /**
27  * A function to print usage flags, excluding those parameters stored in a PetscBag
       structure.
```

```
28  */
29 void PrintUsage()
30 {
31   PetscPrintf(PETSC_COMM_WORLD," -auto_layer N: automatically size BC layer based upon
       wavelength.\n");
32   PetscPrintf(PETSC_COMM_WORLD," -auto_span N: automatically size domain based upon
       nodes per wavelength.\n");
33   PetscPrintf(PETSC_COMM_WORLD," -restart: load parameters and solution from previous
       incomplete run.\n");
34   PetscPrintf(PETSC_COMM_WORLD," -load_solution: load solution from previous run.\n");
35   PetscPrintf(PETSC_COMM_WORLD," -save_max_abs_pressure: compute transmission loss.\n")
       ;
36   PetscPrintf(PETSC_COMM_WORLD," -save_vtk: load saved solution, and export solution as
        p.vtk and w.vtk.\n");
37   PetscPrintf(PETSC_COMM_WORLD," -save_transient: export pressure for ground plane as
       p_tsXXXXXXX.vtk.\n");
38   PetscPrintf(PETSC_COMM_WORLD," -save_i_slice N: export solution for slice
       perpendicular to i plane as slice_i_N.vtk.\n");
39   PetscPrintf(PETSC_COMM_WORLD," -save_j_slice N: export solution for slice
       perpendicular to j plane as slice_j_N.vtk.\n");
40   PetscPrintf(PETSC_COMM_WORLD," -save_k_slice N: export solution for slice
       perpendicular to k plane as slice_k_N.vtk.\n");
41 }
42
43 /**
44  *  The main function which processes all command line flags.
45  *  Based upon the command line flags, the solver may:
46  *       - Load a previously saved solution
47  *       - Initialize the flow field
48  *       - Time march the initial value problem
49  *       - Save the entire solution or a slice of the flow domain.
50  *
51  */
52 int main(int argc, char *argv[])
53 {
54   const int MaxMeanFlowTypeStr=256;     /* The maximum length of mean flow type name.*/
55   const int MaxTSTypeStr=256;          /* The maximum length of a time march method name.
       */
56   PetscErrorCode ierr;                 /* PETSc error code */
57   Vec solution;                        /* PETSc vector to store current solution */
58   parameters *params;                  /* Run-time parameter storage */
59   fluid_properties *properties;        /* Global fluid properties */
60   void *sourceparams,*meanparams,*bcparams; /* Ptrs to various parameter groups */
61   app_ctx context;                     /* Run-time context for passing to various PETSc
       functions */
62   PetscInt i,NWays=1;
63   PetscTruth TSMonitor=PETSC_FALSE, HelpRequested=PETSC_FALSE, FlagExists=PETSC_FALSE,
       DoLoadSolution=PETSC_FALSE, DoSaveVTK=PETSC_FALSE, DoLoadParameters=PETSC_FALSE,
       AutoLayer=PETSC_FALSE, DoSaveISlice=PETSC_FALSE, DoSaveJSlice=PETSC_FALSE,
       DoSaveKSlice=PETSC_FALSE;
```

```
64   PetscInt i_slice=0,j_slice=0,k_slice=0;
65   char meanflowtypestr[MaxMeanFlowTypeStr];
66   char tstypestr[MaxTSTypeStr];
67   PetscLogStage ProfileSolve,ProfileSave;
68   int size;
69
70   /* Initialize PETSc, exit if fail. */
71   if (ierr=PetscInitialize(&argc,&argv,PETSC_NULL,PETSC_NULL))
72     {
73       fprintf(stderr,"Unable to initialize PETSc environment.\n");
74       return ierr;
75     }
76
77   MPI_Comm_size(PETSC_COMM_WORLD,&size);
78   if (size>1) PetscPrintf(PETSC_COMM_WORLD,"Running %d-way\n",size);
79
80   /* Look for extra command-line flags. */
81   DoLoadParameters=PETSC_FALSE;
82   ierr=PetscOptionsGetTruth(PETSC_NULL,"-restart",&DoLoadParameters,&FlagExists);
83   if (ierr) RunSolver=0;
84
85   if (FlagExists && DoLoadParameters) DoLoadSolution=PETSC_TRUE;
86
87   ierr=PetscOptionsGetTruth(PETSC_NULL,"-save_vtk",&DoSaveVTK,PETSC_NULL);
88   if (ierr) RunSolver=0;
89
90   ierr=PetscOptionsGetInt(PETSC_NULL,"-save_i_slice",&i_slice,&DoSaveISlice);
91   if (ierr) RunSolver=0;
92
93   ierr=PetscOptionsGetInt(PETSC_NULL,"-save_j_slice",&j_slice,&DoSaveJSlice);
94   if (ierr) RunSolver=0;
95
96   ierr=PetscOptionsGetInt(PETSC_NULL,"-save_k_slice",&k_slice,&DoSaveKSlice);
97   if (ierr) RunSolver=0;
98
99   context.rms_slice=0;
100  ierr=PetscOptionsGetInt(PETSC_NULL,"-rms_slice",&(context.rms_slice),&FlagExists);
101  if (ierr) RunSolver=0;
102
103  context.DoProbe=PETSC_FALSE;
104  context.RMSProbeDistance=0.0;
105  ierr=PetscOptionsGetReal(PETSC_NULL,"-rms_probe",&context.RMSProbeDistance,&context.
         DoProbe);
106  if (ierr) RunSolver=0;
107  if (!context.DoProbe)
108    {
109      ierr=PetscOptionsGetReal(PETSC_NULL,"-probe",&context.RMSProbeDistance,&context.
             DoProbe);
110      if (ierr) RunSolver=0;
111    }
```

```
112
113   context.SaveTransientPressure=PETSC_FALSE;
114   ierr=PetscOptionsGetTruth(PETSC_NULL,"-save_transient",&context.SaveTransientPressure,
          PETSC_NULL);
115   if (ierr) RunSolver=0;
116
117   context.SaveTransient=PETSC_FALSE;
118   ierr=PetscOptionsGetTruth(PETSC_NULL,"-save_full_transient",&context.SaveTransient,
          PETSC_NULL);
119   if (ierr) RunSolver=0;
120
121   ierr=PetscOptionsGetTruth(PETSC_NULL,"-save_max_abs_pressure",&context.MaxAbsPressure,
          PETSC_NULL);
122   if (ierr) RunSolver=0;
123
124   ierr=PetscOptionsGetTruth(PETSC_NULL,"-load_solution",&DoLoadSolution,PETSC_NULL);
125   if (ierr) RunSolver=0;
126
127   /* Check for -ts_type */
128   ierr=PetscOptionsGetString(PETSC_NULL,"-ts_type",tstypestr,MaxTSTypeStr,&FlagExists);
129   if (ierr) RunSolver=0;
130   if (FlagExists)
131     {
132       if ((strcasecmp(tstypestr,"beuler")==0) ||
133     (strcasecmp(tstypestr,"crank-nicholson")==0))
134   {
135     PetscPrintf(PETSC_COMM_WORLD,"Running implicit codes requires unreasonable amounts
            of memory! Do not use.\n");
136     RunSolver=1;
137   }
138     }
139
140   if (DoSaveVTK || DoSaveISlice || DoSaveJSlice || DoSaveKSlice) DoLoadParameters=
          PETSC_TRUE;
141
142   /* Register all runtime parameters and command line flags */
143   /* General parameters */
144   ierr=RegisterParameterBag(&context.bag,&params,DoLoadParameters);
145   if (ierr) RunSolver=0;
146   context.params=params;
147
148   /* Fluid properties */
149   ierr=RegisterPropertiesBag(&context.pbag,&properties,DoLoadParameters);
150   if (ierr) RunSolver=0;
151   context.properties=properties;
152
153   /* Source properties */
154   switch (params->source)
155     {
156     case SOURCE_MONO_PING:
```

```
157         ierr=RegisterMonopolePingBag(&context.sourcebag,&sourceparams,DoLoadParameters);
158         context.psource=sourceparams;
159         context.InitSource=InitializeMonopolePingSource;
160         context.CalculateSource=MonopolePing;
161         context.GetSourceLowestFrequency=GetMonopolePingFrequency;
162         context.GetSourceHighestFrequency=GetMonopolePingFrequency;
163         context.GetOrigin=GetMonopolePingOrigin;
164         context.DestroySource=NULL;
165         break;
166       case SOURCE_MONOPOLE:
167         ierr=RegisterMonopoleBag(&context.sourcebag,&sourceparams,DoLoadParameters);
168         context.psource=sourceparams;
169         context.InitSource=InitializeMonopoleSource;
170         context.CalculateSource=Monopole;
171         context.GetSourceLowestFrequency=GetMonopoleFrequency;
172         context.GetSourceHighestFrequency=GetMonopoleFrequency;
173         context.GetOrigin=GetMonopoleOrigin;
174         context.DestroySource=NULL;
175         break;
176       case SOURCE_DIPOLE:
177         ierr=RegisterDipoleBag(&context.sourcebag,&sourceparams,DoLoadParameters);
178         context.psource=sourceparams;
179         context.InitSource=InitializeDipoleSource;
180         context.CalculateSource=Dipole;
181         context.GetSourceLowestFrequency=GetDipoleFrequency;
182         context.GetSourceHighestFrequency=GetDipoleFrequency;
183         context.GetOrigin=GetDipoleOrigin;
184         context.DestroySource=NULL;
185         break;
186       case SOURCE_MONOPOLE_SERIES:
187         ierr=RegisterMonopoleSeriesBag(&context.sourcebag,&sourceparams,DoLoadParameters);
188         context.psource=sourceparams;
189         context.InitSource=InitializeMonopoleSeriesSource;
190         context.CalculateSource=MonopoleSeries;
191         context.GetSourceLowestFrequency=GetMonopoleSeriesLowestFrequency;
192         context.GetSourceHighestFrequency=GetMonopoleSeriesHighestFrequency;
193         context.GetOrigin=GetMonopoleSeriesOrigin;
194         context.DestroySource=DestroyMonopoleSeriesSource;
195         break;
196       default:
197         context.psource=NULL;
198         context.InitSource=NULL;
199         context.CalculateSource=NULL;
200         context.GetSourceLowestFrequency=NULL;
201         context.GetSourceHighestFrequency=NULL;
202         context.GetOrigin=NULL;
203         context.DestroySource=NULL;
204       };
205     if (ierr) RunSolver=0;
206
```

```
207   /* Mean flow parameters */
208   switch (params->mean)
209     {
210     case POWER_PROFILE:
211       ierr=RegisterPowerProfileMeanBag(&context.meanbag,&meanparams,DoLoadParameters);
212       context.pmean=meanparams;
213       context.InitMeanFlow=InitializePowerProfileMean;
214       break;
215     case MEAN_UNIFORM:
216     default:
217       ierr=RegisterUniformMeanBag(&context.meanbag,&meanparams,DoLoadParameters);
218       context.pmean=meanparams;
219       context.InitMeanFlow=InitializeUniformMean;
220     };
221   if (ierr) RunSolver=0;
222
223   /* BC parameters */
224   ierr=RegisterBCBag(&context.bcbag,&bcparams,DoLoadParameters);
225   if (ierr) RunSolver=0;
226   context.pbc=bcparams;
227   context.InitBC=InitializeBC;
228
229   PetscInt NWavesInLayer=5;
230   ierr=PetscOptionsGetInt(PETSC_NULL,"-auto_layer",&context.params->mesh.NumWavesInLayer
        ,&FlagExists);
231   if (ierr) RunSolver=0;
232
233   PetscInt NodesPerWave=10;
234   ierr=PetscOptionsGetInt(PETSC_NULL,"-auto_span",&context.params->mesh.NodesPerWave,&
        FlagExists);
235   if (ierr) RunSolver=0;
236
237   /* Check for -help */
238   ierr=PetscOptionsGetTruth(PETSC_NULL,"-help",&HelpRequested,&FlagExists);
239   if (ierr) RunSolver=0;
240   if (FlagExists && HelpRequested)
241     {
242       PrintUsage();
243       RunSolver=0;
244     }
245
246   if (ParametersAreIncorrect(params)) RunSolver=0;
247
248   /* Register profiling events */
249   PetscLogEventRegister("Problem Setup",0,&context.SetupEvent);
250   PetscLogEventRegister("Form ODE",0,&context.FormODEEvent);
251   PetscLogEventRegister("Calc Source",0,&context.CalcSourceEvent);
252   PetscLogStageRegister("Solve Stage",&ProfileSolve);
253   PetscLogStageRegister("Save Stage",&ProfileSave);
254
```

```
255   if (RunSolver)  /* If OK to run the solver... */
256     {
257
258       /* Initialize the mean flow, if necessary */
259       if (!ierr && context.InitMeanFlow)
260   ierr=context.InitMeanFlow(&context);
261
262       /* Initialize the source terms, if necessary */
263       if (!ierr && context.InitSource)
264   ierr=context.InitSource(&context);
265
266       /* Initialize the boundary conditions, if necessary */
267       if (!ierr && context.InitBC)
268   ierr=context.InitBC(&context);
269
270       /* Allocate storage for vectors and other data structures. */
271       ierr=CreateStorage(&solution,&context);
272
273       if (!ierr)
274   {
275     if (DoLoadSolution || DoSaveVTK || DoSaveISlice || DoSaveJSlice || DoSaveKSlice) /*
          If we are saving a solution or slice... */
276       LoadSolution(solution,&context);
277     else /* otherwise initialize the solution */
278       InitializeSolution(solution,&context);
279     if (!DoSaveVTK && !DoSaveISlice && !DoSaveJSlice && !DoSaveKSlice)
280       {
281         /* Not saving anything, so solve the initial value problem */
282         PetscPrintf(PETSC_COMM_WORLD,"Solving the initial value problem.\n");
283         PetscLogStagePush(ProfileSolve);
284         ierr=Solve(&params->tm,&context,solution);
285         PetscLogStagePop();
286       }
287     else
288       {
289         /* Report that we will be saving a solution/slice */
290         PetscPrintf(PETSC_COMM_WORLD,"Exporting solution.\n");
291       }
292   }
293
294       if (!ierr)
295   {
296     PetscLogStagePush(ProfileSave);
297     if (!DoSaveVTK && !DoSaveISlice && !DoSaveJSlice && !DoSaveKSlice)
298       {
299         SaveCheckpoint(solution,&context);
300         SaveGroundVTK(&context);
301         ReportSoundPowerLevel(&context);
302       }
303     if (DoSaveVTK)
```

```
304       {
305         SaveVTK(solution,&context);
306       }
307     if (DoSaveISlice)
308       {
309         ierr=SaveSlice(i_slice,X_SLICE,solution,&context);
310       }
311     if (DoSaveJSlice)
312       {
313         ierr=SaveSlice(j_slice,Y_SLICE,solution,&context);
314       }
315     if (DoSaveKSlice)
316       {
317         ierr=SaveSlice(k_slice,Z_SLICE,solution,&context);
318       }
319     PetscLogStagePop();
320
321     /* Destroy the source terms, if necessary */
322     if (!ierr && context.DestroySource)
323       ierr=context.DestroySource(&context);
324
325     /* Destroy all allocated data structures related to the solution */
326     DestroyStorage(solution,&context);
327   }
328     }
329   else
330     {
331       PetscPrintf(PETSC_COMM_WORLD,"Not running the solver.\n");
332     }
333
334   /* Destroy all parameter storage */
335   PetscBagDestroy(context.bag);
336   if (context.psource) PetscBagDestroy(context.sourcebag);
337   if (context.pmean) PetscBagDestroy(context.meanbag);
338   if (context.pbc) PetscBagDestroy(context.bcbag);
339
340   /* Exit PETSc environment */
341   if (ierr=PetscFinalize())
342     {
343       fprintf(stderr,"Unable to initialize PETSc environment.\n");
344       return 1;
345     }
346
347   if (!RunSolver) return 1;
348
349   return 0;
350
351 }
```

3.2 bc.c

```
1 #include<stdlib.h>
2 #include<assert.h>
3 #include<petsc.h>
4
5 #include"bc.h"
6 #include"ode.h"
7
8 const char BCFile[]="bcparameters.dat";
9
10 void SaveBCBag(PetscBag bag)
11 {
12   PetscViewer bagviewer;
13   /* Save the runtime parameters to a binary file */
14   PetscViewerBinaryOpen(PETSC_COMM_WORLD,BCFile,FILE_MODE_WRITE,&bagviewer);
15   PetscBagView(bag,bagviewer);
16   PetscViewerDestroy(bagviewer);
17 }
18
19 PetscErrorCode RegisterBCBag(PetscBag *bag,void **pparams,PetscTruth LoadFromFile)
20 {
21   PetscErrorCode ierr;
22   assert(bag);
23   assert(pparams);
24
25   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(bc_parameters),bag); CHKERRQ(ierr);
26   ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
27   ierr=PetscBagSetName(*bag,"Boundary Condition Parameters","runtime parameters for a
       boundary conditions"); CHKERRQ(ierr);
28
29   bc_parameters *pbc=(bc_parameters*)(*pparams);
30   /* Porous ground BC parameters */
31   ierr=PetscBagRegisterReal(*bag,&pbc->ground.sigma,1E3,"ground_flow_resistivity","
       ground resistivity, sigma (Pa-s m^-2)"); CHKERRQ(ierr);
32   ierr=PetscBagRegisterReal(*bag,&pbc->ground.Omega,0.6,"ground_porosity","ground
       porosity, Omega"); CHKERRQ(ierr);
33   ierr=PetscBagRegisterReal(*bag,&pbc->ground.q,1.7,"ground_tortuosity","ground
       tortuosity, q"); CHKERRQ(ierr);
34   ierr=PetscBagRegisterTruth(*bag,&pbc->ReflectiveGround,PETSC_FALSE,"ground_reflection"
       ,"apply ground reflection");
35   ierr=PetscBagRegisterTruth(*bag,&pbc->NoGround,PETSC_FALSE,"no_ground","no ground
       layer (for testing)");
36
37   /* Far-field ABC parameters */
38   ierr=PetscBagRegisterReal(*bag,&pbc->farfield.sigma_hi,1E3,"
       far_field_flow_resistivity_max","far-field resistivity farthest from source, sigma
        (Pa-s m^-2)"); CHKERRQ(ierr);
39   ierr=PetscBagRegisterReal(*bag,&pbc->farfield.sigma_lo,1E2,"
       far_field_flow_resistivity_min","far-field resistivity closest to source, sigma (
```

```
      Pa-s m^-2)"); CHKERRQ(ierr);
40  ierr=PetscBagRegisterReal(*bag,&pbc->farfield.sc,1,"far_field_structure_constant","far
      -field structure constant"); CHKERRQ(ierr);
41  ierr=PetscBagRegisterReal(*bag,&pbc->farfield.Omega,1,"far_field_porosity","far-field
      porosity, Omega"); CHKERRQ(ierr);
42  ierr=PetscBagRegisterReal(*bag,&pbc->farfield.gamma,1,"far_field_gamma","far-field
      ratio of specific heats, gamma"); CHKERRQ(ierr);
43
44  if (LoadFromFile)
45    {
46      PetscViewer bagviewer;
47      if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,BCFile,FILE_MODE_READ,&bagviewer))
48  {
49    PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",BCFile);
50    CHKERRQ(ierr);
51  }
52      ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
53      ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
54      ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
55      PetscPrintf(PETSC_COMM_WORLD,"Loaded boundary condition parameters:\n");
56      ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
57    }
58  return 0;
59 }
60
61 PetscReal ABC_kappae(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
62 {
63  app_ctx *pctx=(app_ctx*)pvoid;
64  bc_parameters *pbc=(bc_parameters*)(pctx->pbc);
65  assert(pvoid);
66  assert(pctx);
67  assert(pbc);
68
69  /* Ground region at lower j end */
70  if (!pbc->NoGround && j<=pctx->j_lo && j>=0)
71    return pbc->ground.kappae;
72  return pbc->farfield.kappae;
73 }
74
75 PetscReal ABC_be(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
76 {
77  app_ctx *pctx=(app_ctx*)pvoid;
78  bc_parameters *pbc=(bc_parameters*)(pctx->pbc);
79  assert(pvoid);
80  assert(pctx);
81  assert(pbc);
82
83  /* Ground region at lower j end */
84  if (!pbc->NoGround && j<=pctx->j_lo && j>=0)
85    return pbc->ground.be;
```

```
86   return pbc->farfield.be;
87 }
88
89 PetscReal ABC_se(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
90 {
91   app_ctx *pctx=(app_ctx*)pvoid;
92   bc_parameters *pbc=(bc_parameters*)(pctx->pbc);
93   assert(pvoid);
94   assert(pctx);
95   assert(pbc);
96
97   /* Ground region at lower j end. Select this over far-field. */
98   if (!pbc->NoGround && j<=pctx->j_lo && j>=0)
99     return pbc->ground.se;
100  /* Within a far-field region */
101  PetscReal ri=1,rj=1,rk=1;
102  PetscReal se_lo=pbc->farfield.se_lo;
103  if (i<=pctx->i_lo && i>=0)
104    ri=((PetscReal)(i-0))/((PetscReal)(pctx->i_lo-0));
105  else if (i>=pctx->i_hi && i<=pctx->params->mesh.Nx-1)
106    ri=((PetscReal)(i-pctx->params->mesh.Nx-1))/((PetscReal)(pctx->i_hi-pctx->params->
         mesh.Nx-1));
107  if (j<=pctx->j_lo && j>=0)
108    rj=((PetscReal)(j-0))/((PetscReal)(pctx->j_lo-0));
109  else if (j>=pctx->j_hi && j<=pctx->params->mesh.Ny-1)
110    rj=((PetscReal)(j-pctx->params->mesh.Ny-1))/((PetscReal)(pctx->j_hi-pctx->params->
         mesh.Ny-1));
111  if (!TwoDimensional(pctx))
112    {
113      if (k<=pctx->k_lo && k>=0)
114  rk=((PetscReal)(k-0))/((PetscReal)(pctx->k_lo-0));
115      else if (k>=pctx->k_hi && k<=pctx->params->mesh.Nz-1)
116  rk=((PetscReal)(k-pctx->params->mesh.Nz-1))/((PetscReal)(pctx->k_hi-pctx->params->mesh
       .Nz-1));
117    }
118  return (1-ri*rj*rk)*pbc->farfield.se_hi+ri*rj*rk*pbc->farfield.se_lo;
119 }
120
121 PetscErrorCode InitializeBC(void *vpctx)
122 {
123   app_ctx *pctx=(app_ctx*)vpctx;
124   PetscInt d;
125   PetscReal c,rho,gamma;
126   PetscReal s_c,rho_e;
127   mean_uniform_parameters *pmean=(mean_uniform_parameters*)pctx->pmean;
128   bc_parameters *pbc=(bc_parameters*)pctx->pbc;
129
130   assert(pctx);
131   assert(pmean);
132   assert(pbc);
```

```
133
134   c=GetSpeedOfSound(pctx->properties);
135   rho=GetDensity(pctx->properties);
136   gamma=GetRatioOfSpecificHeats(pctx->properties);
137
138   pctx->calc_kappae=ABC_kappae;
139   pctx->calc_be=ABC_be;
140   pctx->calc_se=ABC_se;
141   PetscPrintf(PETSC_COMM_WORLD,"Initializing boundary region:\n");
142
143
144   /* Pre-computed ground parameters */
145   /* Assuming circular, cylindrical pores */
146   s_c=(4/3)*pbc->ground.q*pbc->ground.q/pbc->ground.Omega;
147   rho_e=s_c*rho/pbc->ground.Omega;
148   pbc->ground.kappae=rho*c*c/gamma/pbc->ground.Omega;
149   pbc->ground.be=1.0/rho_e;
150   pbc->ground.se=pbc->ground.sigma/rho_e;
151   if (pbc->NoGround)
152     {
153       PetscPrintf(PETSC_COMM_WORLD,"No ground surface.\n");
154     }
155   else if (pbc->ReflectiveGround)
156     {
157       PetscPrintf(PETSC_COMM_WORLD,"Using reflective ground surface.\n");
158     }
159   else
160     {
161       PetscPrintf(PETSC_COMM_WORLD," ground: s_c=%g rho_e=%g.\n",s_c,rho_e);
162       PetscPrintf(PETSC_COMM_WORLD," ground: kappa_e=%g b_e=%g s_e=%g.\n",pbc->ground.
              kappae,pbc->ground.be,pbc->ground.se);
163     }
164   /* Pre-computed absorbing bc region parameters */
165   rho_e=pbc->farfield.sc*rho/pbc->farfield.Omega;
166   pbc->farfield.kappae=rho*c*c/pbc->farfield.gamma/pbc->farfield.Omega;
167   pbc->farfield.be=1.0/rho_e;
168   pbc->farfield.se_lo=pbc->farfield.sigma_lo/rho_e;
169   pbc->farfield.se_hi=pbc->farfield.sigma_hi/rho_e;
170   PetscPrintf(PETSC_COMM_WORLD," far-field: s_c=%g rho_e=%g.\n",pbc->farfield.sc,rho_e);
171   PetscPrintf(PETSC_COMM_WORLD," far-field: kappa_e=%g b_e=%g s_e=%g to %g.\n",pbc->
          farfield.kappae,pbc->farfield.be,pbc->farfield.se_lo,pbc->farfield.se_hi);
172
173   return 0;
174 }
```

3.3 bc.h

```
1 #ifndef _BC_H_
2 #define _BC_H_
```

```
3 #include<petsc.h>
4 #include<petscbag.h>
5
6 /*! \file bc.h
7     \brief Boundary conditions parameters.
8
9 This header file defines the parameters and functions for applying boundary conditions.
10 */
11
12 /*! \brief ABC parameters
13
14   Absorbing boundary condition (ABC) region parameters. */
15 typedef struct {
16   PetscReal sc;        /*!< far field structure constant */
17   PetscReal Omega;     /*!< far field porosity */
18   PetscReal gamma;     /*!< far field ratio of specific heats */
19   PetscReal sigma_lo; /*!< far field flow resistivity lower limit */
20   PetscReal sigma_hi; /*!< far field flow resistivity higher limit */
21   PetscReal kappae;     /*!< \f$\frac{\rho c^2}{\gamma\Omega}\f$ */
22   PetscReal be;         /*!< \f$\frac{1}{\rho_e}\f$ */
23   PetscReal se_lo;      /*!< \f$\frac{\sigma_{low}}{\rho_e}\f$ */
24   PetscReal se_hi;      /*!< \f$\frac{\sigma_{high}}{\rho_e}\f$ */
25 } absorbing_bc_parameters;
26
27 /*! \brief Ground parameters
28
29   Porous ground region (similar to ABC) parameters. */
30 typedef struct {
31   PetscReal sigma;      /*!< ground flow resistivity */
32   PetscReal Omega;      /*!< ground porosity */
33   PetscReal q;          /*!< ground tortuosity */
34   PetscReal kappae;     /*!< \f$\frac{\rho c^2}{\gamma\Omega}\f$ */
35   PetscReal be;         /*!< \f$\frac{1}{\rho_e}\f$ */
36   PetscReal se;         /*!< \f$\frac{\sigma}{\rho_e}\f$ */
37 } porous_ground_bc_parameters;
38
39 /*! \brief All BC parameters
40
41   Collection of BC parameters. */
42 typedef struct {
43   absorbing_bc_parameters farfield;   /*!< Far field BC parameters */
44   porous_ground_bc_parameters ground; /*!< Ground BC parameters */
45   PetscTruth ReflectiveGround;        /*!< Reflective ground flag */
46   PetscTruth NoGround;                /*!< No ground flag */
47 } bc_parameters;
48
49 /*! Save BC parameter values to file. */
50 void SaveBCBag(PetscBag bag);
51
52 /*! Register BC parameters, and optionally load values from file. */
```

```
53 PetscErrorCode RegisterBCBag(PetscBag *bag,void **params,PetscTruth LoadFromFile);
54
55 /*! Initialize BC parameter values. */
56 PetscErrorCode InitializeBC(void *vpctx);
57
58 #endif
```

3.4 io.h

```
1 #ifndef _IO_H_
2 #define _IO_H_
3
4 #include<petsc.h>
5
6 /*! \file io.h
7     \brief Input and output
8
9 Declares functions for loading and saving the solution, as well as saving slices of the
      domain.
10 */
11
12 /*! Load the solution from files solution.[dat,info] */
13 PetscErrorCode LoadSolution(Vec solution,
14           app_ctx *pctx);
15
16 /*! Save the solution to files solution.[dat,info] */
17 PetscErrorCode SaveSolution(Vec solution,
18           app_ctx *pctx);
19
20 /*! Save all parameters stored in PetscBag */
21 PetscErrorCode SaveParameters(app_ctx *pctx);
22
23 /*! Save a snapshot of the current solution to a VTK file */
24 PetscErrorCode SaveCheckpoint(Vec solution,
25             app_ctx *pctx);
26
27 /*! Save the solution to p.vtk and w.vtk */
28 PetscErrorCode SaveVTK(Vec solution,
29           app_ctx *pctx);
30
31 /*! Save the ground plane slice of sound pressure level */
32 PetscErrorCode SaveGroundVTK(app_ctx *pctx);
33
34 /*! Report the total sound power level */
35 PetscErrorCode ReportSoundPowerLevel(app_ctx *pctx);
36
37 /*! Save the solution at this iteration */
38 PetscErrorCode SaveVTKByIter(PetscInt iter,
39           PetscReal t,
```

```
40            Vec solution,
41            app_ctx *pctx);
42
43 /*! Save the full solution in one file at this iteration */
44 PetscErrorCode SaveFullVTKByIter(PetscInt iter,
45          PetscReal t,
46          Vec solution,
47          app_ctx *pctx);
48
49 typedef enum {
50   X_SLICE=0,  /*!< A slice perpendicular to x direction */
51   Y_SLICE=1,  /*!< A slice perpendicular to y direction */
52   Z_SLICE=2   /*!< A slice perpendicular to z direction */
53 } slice_plane;
54
55 /*! Save a slice of the solution perpendicular to the x, y, or z direction,
56  using a specific file name. */
57 PetscErrorCode SaveSliceWithName(PetscInt slice,
58          slice_plane plane,
59          PetscInt d0,
60          PetscInt dN,
61          Vec solution,
62          app_ctx *pctx,
63          char *FileName);
64
65 /*! Save a generic slice of the solution */
66 PetscErrorCode SaveSlice(PetscInt slice,
67        slice_plane plane,
68        Vec solution,
69        app_ctx *pctx);
70
71 /*! Create a scatter/gather context for the requested solution slice */
72 PetscErrorCode CreateSliceScatter(app_ctx *pctx,
73           Vec vec_from,
74           PetscInt ilo,
75           PetscInt ihi,
76           PetscInt jlo,
77           PetscInt jhi,
78           PetscInt klo,
79           PetscInt khi,
80           PetscInt dlo,
81           PetscInt dhi,
82           DA *da_to,
83           Vec *vec_to,
84           VecScatter *pscatter);
85 void ComputeFFT(PetscInt NHistory,
86     PetscReal *tHistory,
87     PetscReal *vHistory,
88     PetscInt *NFrequency,
89     PetscReal **f,
```

```
90      PetscReal **vmag,
91      PetscReal **vphase,
92      PetscInt *NBands,
93      PetscReal **fband,
94      PetscReal **vband);
95 #endif
```

3.5 io.c

```
1  #include<stdlib.h>
2  #include<assert.h>
3
4  #include<petsc.h>
5
6  #include<ode.h>
7  #include<float.h>
8  #include<math.h>
9
10 #include"io.h"
11
12 #include"fftw3.h"
13
14 PetscErrorCode LoadSolution(Vec solution,
15           app_ctx *pctx)
16 {
17   PetscViewer viewer;
18   PetscPrintf(PETSC_COMM_WORLD,"Loading solution.\n");
19   PetscViewerBinaryOpen(PETSC_COMM_WORLD,"solution.dat",FILE_MODE_READ,&viewer);
20   VecLoadIntoVector(viewer,solution);
21   PetscViewerDestroy(viewer);
22   return 0;
23 }
24
25
26 PetscErrorCode SaveSolution(Vec solution,
27           app_ctx *pctx)
28 {
29   PetscViewer viewer;
30   PetscPrintf(PETSC_COMM_WORLD,"Saving solution.\n");
31   PetscViewerBinaryOpen(PETSC_COMM_WORLD,"solution.dat",FILE_MODE_WRITE,&viewer);
32   VecView(solution,viewer);
33   PetscViewerDestroy(viewer);
34   return 0;
35 }
36
37 PetscErrorCode SaveParameters(app_ctx *pctx)
38 {
39   assert(pctx);
40   PetscPrintf(PETSC_COMM_WORLD,"Saving parameters.\n");
```

```
41   SaveParameterBag(pctx->bag);
42   if (pctx->psource) SaveSourceBag(pctx->sourcebag);
43   if (pctx->properties) SaveFluidPropertiesBag(pctx->pbag);
44   if (pctx->pmean) SaveMeanBag(pctx->meanbag);
45   if (pctx->pbc) SaveBCBag(pctx->bcbag);
46   return 0;
47 }
48
49 PetscErrorCode SaveCheckpoint(Vec solution,
50             app_ctx *pctx)
51 {
52   SaveSolution(solution,pctx);
53   SaveParameters(pctx);
54   return 0;
55 }
56
57 PetscErrorCode SaveVTK(Vec solution,
58           app_ctx *pctx)
59 {
60   if (TwoDimensional(pctx))
61     {
62       PetscErrorCode ierr;
63       ierr=SaveSliceWithName(0,Z_SLICE,0,0,solution,pctx,"p.vtk"); CHKERRQ(ierr);
64       ierr=SaveSliceWithName(0,Z_SLICE,1,2,solution,pctx,"w.vtk"); CHKERRQ(ierr);
65       return 0;
66     }
67
68   PetscViewer viewer;
69   PetscInt i,j,k,d,sx,sy,sz,mx,my,mz;
70   DAPeriodicType wrap;
71   DAStencilType stenciltype;
72   PetscInt stencilwidth,M,N,P,m,n,p;
73   PetscScalar ****s,***pp,****pw;
74   DA da1,da3;
75   Vec vp,vw;
76   parameters *params=pctx->params;
77
78   assert(pctx);
79   assert(params);
80
81   /* Get the dimensions of master DA to use for other DAs */
82   DAGetInfo(pctx->da,PETSC_IGNORE,
83       &M,&N,&P,
84       &m,&n,&p,
85       PETSC_IGNORE,
86       &stencilwidth,
87       &wrap,&stenciltype);
88
89   /* Mesh distributed array for dof=1 */
90   DACreate3d(PETSC_COMM_WORLD,wrap,stenciltype,
```

```
91          M,N,P,
92          m,n,p,
93          1,
94          stencilwidth,
95          PETSC_NULL,PETSC_NULL,PETSC_NULL,
96          &da1);
97
98    DASetUniformCoordinates(da1,
99            params->mesh.xmin, params->mesh.xmax,
100           params->mesh.ymin, params->mesh.ymax,
101           params->mesh.zmin, params->mesh.zmax);
102
103   /* Mesh distributed array for dof=3 */
104   DACreate3d(PETSC_COMM_WORLD,wrap,stenciltype,
105         M,N,P,
106         m,n,p,
107         3,
108         stencilwidth,
109         PETSC_NULL,PETSC_NULL,PETSC_NULL,
110         &da3);
111
112   DASetUniformCoordinates(da3,
113           params->mesh.xmin, params->mesh.xmax,
114           params->mesh.ymin, params->mesh.ymax,
115           params->mesh.zmin, params->mesh.zmax);
116
117   DACreateGlobalVector(da1,&vp);
118   DACreateGlobalVector(da3,&vw);
119
120   /* Get local grid boundaries, which are independent of dof */
121   DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
122
123   DAVecGetArrayDOF(pctx->da,solution,&s);
124   DAVecGetArrayDOF(da3,vw,&pw);
125   DAVecGetArray(da1,vp,&pp);
126
127   for(i=sx;i<sx+mx;i++)
128      for(j=sy;j<sy+my;j++)
129         for(k=sz;k<sz+mz;k++)
130   {
131      pp[k][j][i]=s[k][j][i][0];
132      for(d=0;d<3;d++)
133         {
134            pw[k][j][i][d]=s[k][j][i][1+d];
135         }
136   }
137
138   DAVecRestoreArray(da1,vp,&pp);
139   DAVecRestoreArrayDOF(da3,vw,&pw);
140
```

```
141   DAVecRestoreArrayDOF(pctx->da,solution,&s);
142
143   /* Save p as VTK file */
144   PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
145   PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
146   PetscViewerFileSetName(viewer,"p.vtk");
147   PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
148   DAView(da1,viewer);
149   VecView(vp,viewer);
150   PetscViewerDestroy(viewer);
151
152   /* Save w as VTK file */
153   PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
154   PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
155   PetscViewerFileSetName(viewer,"w.vtk");
156   PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
157   DAView(da3,viewer);
158   VecView(vw,viewer);
159   PetscViewerDestroy(viewer);
160
161   //   DARestoreLocalVector(da1,&vp);
162   //   DARestoreLocalVector(da3,&vw);
163   VecDestroy(vp);
164   VecDestroy(vw);
165
166   DADestroy(da1);
167   DADestroy(da3);
168
169   return 0;
170 }
171
172 PetscErrorCode SaveGroundVTK(app_ctx *pctx)
173 {
174   const PetscReal pref=20E-6; // Pa
175   const PetscReal Iref=1E-12; // W/m^2
176   const PetscReal rms_smallest=-10000;
177   const PetscReal I_smallest=-10000;
178
179   PetscViewer viewer;
180   PetscReal max,min;
181   PetscInt imax,imin;
182   assert(pctx);
183
184   bc_parameters *bc=(bc_parameters*)pctx->pbc;
185   assert(bc);
186
187   if (pctx->MaxAbsPressure)
188     {
189       PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
190       PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
```

```
191         PetscViewerFileSetName(viewer,"pmaxabs.vtk");
192         PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
193         DAView(pctx->dagp,viewer);
194         VecView(pctx->pmaxabs,viewer);
195         PetscViewerDestroy(viewer);
196       }
197     /* Save ground-level sound pressure level (SPL) as VTK file */
198
199     /* We need to complete the RMS average by dividing by the intergration
200        time interval T */
201     if (pctx->j_lo==0 && !TwoDimensional(pctx) && !bc->ReflectiveGround)
202       {
203         PetscPrintf(PETSC_COMM_WORLD,"Ground layer is at bottom of mesh. Not saving spl.
               vtk!\n");
204         return 1;
205       }
206
207     if (pctx->tlast-pctx->tstart<=0)
208       {
209         PetscPrintf(PETSC_COMM_WORLD,"RMS averaging incomplete (time interval is zero).\n"
               );
210         return 1;
211       }
212
213     if (!pctx->IntegrationStarted && pctx->j_lo>0)
214       {
215         PetscPrintf(PETSC_COMM_WORLD,"RMS averaging was not done. Restart with t_max>=t_0+
               T.\n");
216         return 1;
217       }
218
219     /* Note: Time average needs to be divided by T to complete calculation */
220     PetscScalar invT=1.0/(pctx->tlast-pctx->tstart);
221
222     VecScale(pctx->prms,invT);
223     VecScale(pctx->I,invT);
224
225     /* pctx->prms now contains RMS of pressure */
226
227     /* Calculate SPL using 10*log_10 (prms^2/pref^2) */
228     /* Avoid -Inf values when calculating log(0) which can occur if no wave reaches ground
          plane
229        during RMS average*/
230     PetscScalar *v;
231     PetscScalar pscale=1.0/(pref*pref);
232     PetscInt i,n;
233     VecGetLocalSize(pctx->prms,&n);
234     VecGetArray(pctx->prms,&v);
235     for(i=0;i<n;i++)
236       {
```

```
237         v[i] *= pscale;
238         if (v[i]!=0.0)
239       v[i] = 10*log10(v[i]);
240         else
241       v[i] = rms_smallest;
242       }
243       VecRestoreArray(pctx->prms,&v);
244
245       PetscPrintf(PETSC_COMM_WORLD,"Ground surface averaged over interval %g s\n",pctx->
              tlast-pctx->tstart);
246       VecMin(pctx->prms,&imin,&min);
247       VecMax(pctx->prms,&imax,&max);
248       PetscPrintf(PETSC_COMM_WORLD," Max sound pressure level %g dB\n",max);
249       PetscPrintf(PETSC_COMM_WORLD," Min sound pressure level %g dB\n",min);
250
251       /* pctx->prms now contains SPL */
252
253       PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
254       PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
255       PetscViewerFileSetName(viewer,"spl.vtk");
256       PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
257       DAView(pctx->dagp,viewer);
258       VecView(pctx->prms,viewer);
259       PetscViewerDestroy(viewer);
260
261       if (pctx->DoProbe)
262         {
263           PetscScalar ***prms;
264           PetscInt sx,sy,sz,mx,my,mz;
265           /* Get local grid boundaries, which are independent of dof */
266           DAGetCorners(pctx->dagp,&sx,&sy,&sz,&mx,&my,&mz);
267
268           DAVecGetArrayDOF(pctx->dagp,pctx->prms,&prms);
269
270           PetscReal probe_value=0.0;
271           PetscInt i,j;
272           for(j=0;j<2;j++)
273       for(i=0;i<2;i++)
274         {
275           if ((pctx->probe_li+i>=sx && pctx->probe_li+i<sx+mx) &&
276         (pctx->probe_lj+j>=sy && pctx->probe_lj+j<sy+my))
277             {
278         probe_value += pctx->probe_w[i][j]*prms[pctx->probe_lj+j][pctx->probe_li+i][0];
279             }
280         }
281
282           DAVecRestoreArrayDOF(pctx->dagp,pctx->prms,&prms);
283           double val_out=probe_value,val_in=0;
284           MPI_Reduce(&val_out,&val_in,1,MPI_DOUBLE,MPI_SUM,0,PETSC_COMM_WORLD);
285
```

```
286       PetscReal probe_SPL=val_in;
287       PetscPrintf(PETSC_COMM_WORLD,"SPL @ probe: %g dB\n",probe_SPL);
288       PetscReal probe_rms=pref*pow(10.0,0.05*probe_SPL);
289       PetscPrintf(PETSC_COMM_WORLD,"RMS of pressure @ probe: %g Pa\n",probe_rms);
290
291       int rank; MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
292
293       /* Compute FFT of pHistory */
294       if (!rank)
295   {
296     assert(pctx->tHistory);
297     assert(pctx->pHistory);
298
299     PetscInt i;
300     PetscInt N,NBand;
301     PetscReal *f=NULL,*pmag=NULL,*pphase=NULL,*fband=NULL,*pband=NULL;
302     FILE *fp;
303
304     fp=fopen("p.dat","w");
305     for(i=0;i<pctx->NHistory;i++)
306       fprintf(fp,"%g %g\n",pctx->tHistory[i],pctx->pHistory[i]);
307     fclose(fp);
308
309     ComputeFFT(pctx->NHistory,pctx->tHistory,pctx->pHistory,
310           &N,&f,&pmag,&pphase,
311           &NBand,&fband,&pband);
312     assert(f);
313     assert(pmag);
314     assert(pphase);
315
316     fp=fopen("spl.dat","w");
317     if (fp)
318       {
319         fprintf(fp,"# f (Hz) SPL (dBA) magnitude phase\n");
320         for(i=1;i<N;i++)
321     {
322       PetscReal SPL=(pmag[i]==0?-1000:20.0*log10(pmag[i]/pref)+GetAWeighting(f[i]));
323       fprintf(fp,"%g %g %g %g\n",f[i],SPL,pmag[i],pphase[i]);
324     }
325       }
326     else
327       {
328         PetscSynchronizedPrintf(PETSC_COMM_WORLD,"Unable to open spl.dat\n");
329       }
330     fclose(fp);
331
332
333     fp=fopen("sploctave.dat","w");
334     if (fp)
335       {
```

```
336           fprintf(fp,"# f (Hz) SPL (dBA)\n");
337           for(i=1;i<NBand;i++)
338       {
339         PetscReal SPL=(pband[i]==0?-1000:20.0*log10(pband[i]/pref)+GetAWeighting(fband[i])
                ;
340         fprintf(fp,"%g %g\n",fband[i],SPL);
341       }
342         }
343       else
344         {
345           PetscSynchronizedPrintf(PETSC_COMM_WORLD,"Unable to open splocatve.dat\n");
346         }
347       fclose(fp);
348
349       PetscFree(f);
350       PetscFree(pmag);
351       PetscFree(pphase);
352
353       PetscSynchronizedFlush(PETSC_COMM_WORLD);
354     }
355         else
356     {
357       PetscSynchronizedFlush(PETSC_COMM_WORLD);
358     }
359       }
360
361     /* Recycle prms for calculating sound intensity level */
362
363     /* Calculate SIL using 10*log_10 (I/Iref) */
364     /* Avoid -Inf values when calculating log(0) which can occur if no wave reaches ground
            plane
365        during RMS average*/
366     PetscScalar *I;
367     PetscScalar Iscale=1.0/Iref;
368     PetscInt nI;
369
370     VecGetLocalSize(pctx->prms,&n);
371     VecGetLocalSize(pctx->I,&nI);
372     assert(3*n==nI);
373
374     VecGetArray(pctx->prms,&v);
375     VecGetArray(pctx->I,&I);
376     for(i=0;i<n;i++)
377       {
378         v[i] = Iscale*sqrt(I[i*3+0]*I[i*3+0]+
379          I[i*3+1]*I[i*3+1]+
380          I[i*3+2]*I[i*3+2]);
381         if (v[i]!=0.0)
382     v[i] = 10*log10(v[i]);
383         else
```

```
384   v[i] = rms_smallest;
385     }
386   VecRestoreArray(pctx->I,&I);
387   VecRestoreArray(pctx->prms,&v);
388
389   /* pctx->prms now contains SIL */
390
391   PetscPrintf(PETSC_COMM_WORLD,"Ground surface averaged over interval %g s\n",pctx->
          tlast-pctx->tstart);
392   VecMin(pctx->prms,&imin,&min);
393   VecMax(pctx->prms,&imax,&max);
394   PetscPrintf(PETSC_COMM_WORLD," Max sound intensity level %g dB\n",max);
395   PetscPrintf(PETSC_COMM_WORLD," Min sound intensity level %g dB\n",min);
396
397   if (pctx->DoProbe)
398     {
399       PetscScalar ***prms;
400       PetscInt sx,sy,sz,mx,my,mz;
401       /* Get local grid boundaries, which are independent of dof */
402       DAGetCorners(pctx->dagp,&sx,&sy,&sz,&mx,&my,&mz);
403
404       DAVecGetArrayDOF(pctx->dagp,pctx->prms,&prms);
405
406       PetscReal probe_value=0.0;
407       PetscInt i,j;
408       for(j=0;j<2;j++)
409   for(i=0;i<2;i++)
410     {
411       if ((pctx->probe_li+i>=sx && pctx->probe_li+i<sx+mx) &&
412     (pctx->probe_lj+j>=sy && pctx->probe_lj+j<sy+my))
413         {
414     probe_value += pctx->probe_w[i][j]*prms[pctx->probe_lj+j][pctx->probe_li+i][0];
415         }
416     }
417
418       DAVecRestoreArrayDOF(pctx->dagp,pctx->prms,&prms);
419       double val_out=probe_value,val_in=0;
420       MPI_Reduce(&val_out,&val_in,1,MPI_DOUBLE,MPI_SUM,0,PETSC_COMM_WORLD);
421
422       PetscReal probe_SIL=val_in;
423       PetscPrintf(PETSC_COMM_WORLD,"SIL @ probe: %g dB\n",probe_SIL);
424     }
425
426   PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
427   PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
428   PetscViewerFileSetName(viewer,"sil.vtk");
429   PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
430   DAView(pctx->dagp,viewer);
431   VecView(pctx->prms,viewer);
432   PetscViewerDestroy(viewer);
```

```
433
434
435   return 0;
436 }
437
438 PetscErrorCode ReportSoundPowerLevel(app_ctx *pctx)
439 {
440   const PetscReal Wref=1E-12; // W
441
442   if (pctx->tlast-pctx->tstart<=0)
443     {
444       PetscPrintf(PETSC_COMM_WORLD,"Power integration incomplete (time interval is zero)
              .\n");
445       return 1;
446     }
447
448   /* Note: Time average needs to be divided by T to complete calculation */
449   PetscScalar invT=1.0/(pctx->tlast-pctx->tstart);
450
451   pctx->SWL *= invT;
452
453   PetscPrintf(PETSC_COMM_WORLD,"Power: %g W.\n",pctx->SWL);
454
455   pctx->SWL = 10*log10(pctx->SWL/Wref);
456
457   PetscPrintf(PETSC_COMM_WORLD,"Sound power level: %g dB SWL.\n",pctx->SWL);
458   PetscPrintf(PETSC_COMM_WORLD,"Integrated from %g to %g s.\n",pctx->tstart,pctx->tlast)
          ;
459
460   int rank=0; MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
461
462
463   /* Compute FFT of pHistory */
464   if (!rank)
465     {
466       assert(pctx->tHistory);
467       assert(pctx->PowerHistory);
468
469       PetscInt i;
470       PetscInt N,NBand;
471       PetscReal *f=NULL,*Wmag=NULL,*Wphase=NULL,*fband=NULL,*Wband=NULL;
472       FILE *fp;
473
474       fp=fopen("power.dat","w");
475       for(i=0;i<pctx->NHistory;i++)
476   fprintf(fp,"%g %g\n",pctx->tHistory[i],pctx->PowerHistory[i]);
477       fclose(fp);
478
479       ComputeFFT(pctx->NHistory,pctx->tHistory,pctx->PowerHistory,
480      &N,&f,&Wmag,&Wphase,
```

```
481       &NBand,&fband,&Wband);
482         assert(f);
483         assert(Wmag);
484         assert(Wphase);
485
486         fp=fopen("swl.dat","w");
487         if (fp)
488     {
489       fprintf(fp,"# f (Hz) SPL (dBA) magnitude phase\n");
490       for(i=1;i<N;i++)
491         {
492           double SWL=(Wmag[i]==0?-1000:10.0*log10(Wmag[i]/Wref)+GetAWeighting(f[i]));
493           fprintf(fp,"%g %g %g %g\n",f[i],SWL,Wmag[i],Wphase[i]);
494         }
495     }
496         else
497     {
498       PetscSynchronizedPrintf(PETSC_COMM_WORLD,"Unable to open spl.dat\n");
499     }
500         fclose(fp);
501
502         fp=fopen("swloctave.dat","w");
503         if (fp)
504     {
505       fprintf(fp,"# f (Hz) SWL (dBA)\n");
506       for(i=1;i<NBand;i++)
507         {
508           PetscReal SWL=(Wband[i]==0?-1000:20.0*log10(Wband[i]/Wref)+GetAWeighting(fband[i
                  ]));
509           fprintf(fp,"%g %g\n",fband[i],SWL);
510         }
511     }
512         else
513     {
514       PetscSynchronizedPrintf(PETSC_COMM_WORLD,"Unable to open splocatve.dat\n");
515     }
516         fclose(fp);
517
518         PetscFree(f);
519         PetscFree(Wmag);
520         PetscFree(Wphase);
521
522         PetscSynchronizedFlush(PETSC_COMM_WORLD);
523       }
524     else
525       {
526         PetscSynchronizedFlush(PETSC_COMM_WORLD);
527       }
528
529     return 0;
```

```
530 }
531
532 PetscErrorCode SaveVTKByIter(PetscInt steps,
533             PetscReal t,
534             Vec solution,
535             app_ctx *pctx)
536 {
537   char FileName[]="p_ts000000.vtk";
538   PetscViewer viewer;
539
540   assert(pctx);
541   assert(pctx->params);
542
543   sprintf(FileName,"p_ts%06d.vtk",steps);
544
545   if (TwoDimensional(pctx))
546     return SaveSliceWithName(0,Z_SLICE,0,0,solution,pctx,FileName);
547
548   PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
549   PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
550   PetscViewerFileSetName(viewer,FileName);
551   PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
552   DAView(pctx->dagp,viewer);
553
554   VecScatterBegin(pctx->groundp,solution,pctx->currp,INSERT_VALUES,SCATTER_FORWARD);
555   VecScatterEnd(pctx->groundp,solution,pctx->currp,INSERT_VALUES,SCATTER_FORWARD);
556   VecView(pctx->currp,viewer);
557   PetscViewerDestroy(viewer);
558   return 0;
559 }
560
561
562 PetscErrorCode SaveFullVTKByIter(PetscInt steps,
563           PetscReal t,
564           Vec solution,
565           app_ctx *pctx)
566 {
567   char FileName[]="sol_ts000000.vtk";
568   PetscViewer viewer;
569
570   assert(pctx);
571   assert(pctx->params);
572
573   sprintf(FileName,"sol_ts%06d.vtk",steps);
574
575   if (TwoDimensional(pctx))
576     return SaveSliceWithName(0,Z_SLICE,0,2,solution,pctx,FileName);
577
578   PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
579   PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
```

```
580   PetscViewerFileSetName(viewer,FileName);
581   PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
582   DAView(pctx->da,viewer);
583
584   VecView(solution,viewer);
585   PetscViewerDestroy(viewer);
586   return 0;
587 }
588
589 PetscErrorCode SaveSlice(PetscInt slice,
590         slice_plane plane,
591         Vec solution,
592         app_ctx *pctx)
593 {
594   return SaveSliceWithName(slice,plane,0,3,solution,pctx,NULL);
595 }
596
597 PetscErrorCode SaveSliceWithName(PetscInt slice,
598           slice_plane plane,
599           PetscInt dlo,
600           PetscInt dhi,
601           Vec solution,
602           app_ctx *pctx,
603           char *RequestedFileName)
604 {
605   PetscErrorCode ierr;
606
607   PetscTruth SliceWithinMesh=PETSC_FALSE;
608   switch (plane)
609     {
610     case X_SLICE:
611       if (slice>=0 && slice<=pctx->params->mesh.Nx-1) SliceWithinMesh=PETSC_TRUE;
612       break;
613     case Y_SLICE:
614       if (slice>=0 && slice<=pctx->params->mesh.Ny-1) SliceWithinMesh=PETSC_TRUE;
615       break;
616     case Z_SLICE:
617       if (slice>=0 && slice<=pctx->params->mesh.Nz-1) SliceWithinMesh=PETSC_TRUE;
618       break;
619     };
620
621   if (SliceWithinMesh)
622     {
623       if (RequestedFileName)
624   PetscPrintf(PETSC_COMM_WORLD,"Saving %s\n",RequestedFileName);
625       else
626   {
627     PetscPrintf(PETSC_COMM_WORLD,"Saving slice at ");
628     switch (plane)
629       {
```

```
630       case X_SLICE:
631         PetscPrintf(PETSC_COMM_WORLD,"i");     break;
632       case Y_SLICE:
633         PetscPrintf(PETSC_COMM_WORLD,"j");     break;
634       case Z_SLICE:
635         PetscPrintf(PETSC_COMM_WORLD,"k");     break;
636       };
637     PetscPrintf(PETSC_COMM_WORLD,"=%d\n",slice);
638   }
639
640       PetscInt M,N,P,dof;
641       /* Get the global and local dimensions of solution DA */
642       DAGetInfo(pctx->da,PETSC_IGNORE,
643     &M,&N,&P,
644     PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
645     &dof,
646     PETSC_IGNORE,
647     PETSC_IGNORE,PETSC_IGNORE);
648
649       assert(dlo>=0);
650       assert(dhi<=dof);
651
652       PetscInt ilo=(plane==X_SLICE?slice:0);
653       PetscInt ihi=(plane==X_SLICE?slice:M-1);
654       PetscInt jlo=(plane==Y_SLICE?slice:0);
655       PetscInt jhi=(plane==Y_SLICE?slice:N-1);
656       PetscInt klo=(plane==Z_SLICE?slice:0);
657       PetscInt khi=(plane==Z_SLICE?slice:P-1);
658
659       DA da2D;
660       Vec sol2D;
661       VecScatter slice_scatter;
662       CreateSliceScatter(pctx,solution,ilo,ihi,jlo,jhi,klo,khi,dlo,dhi,&da2D,&sol2D,&
              slice_scatter);
663
664       PetscViewer viewer;
665       PetscViewerCreate(PETSC_COMM_WORLD,&viewer);
666       PetscViewerSetType(viewer,PETSC_VIEWER_ASCII);
667
668       if (RequestedFileName)
669   {
670     PetscViewerFileSetName(viewer,RequestedFileName);
671   }
672       else
673   {
674     char FileName[]="slice_i_000000.vtk";
675     switch (plane)
676       {
677       case X_SLICE:
678         sprintf(FileName,"slice_%c_%06d.vtk",'i',slice); break;
```

```
679       case Y_SLICE:
680         sprintf(FileName,"slice_%c_%06d.vtk",'j',slice); break;
681       case Z_SLICE:
682         sprintf(FileName,"slice_%c_%06d.vtk",'k',slice); break;
683       };
684     PetscViewerFileSetName(viewer,FileName);
685   }
686       PetscViewerSetFormat(viewer,PETSC_VIEWER_ASCII_VTK);
687
688       DAView(da2D,viewer);
689
690       ierr=VecScatterBegin(slice_scatter,solution,sol2D,INSERT_VALUES,SCATTER_FORWARD);
              CHKERRQ(ierr);
691       ierr=VecScatterEnd(slice_scatter,solution,sol2D,INSERT_VALUES,SCATTER_FORWARD);
              CHKERRQ(ierr);
692       ierr=VecView(sol2D,viewer); CHKERRQ(ierr);
693
694       ierr=VecScatterDestroy(slice_scatter);
695       ierr=PetscViewerDestroy(viewer); CHKERRQ(ierr);
696       ierr=VecDestroy(sol2D); CHKERRQ(ierr);
697       ierr=DADestroy(da2D); CHKERRQ(ierr);
698       return 0;
699     }
700   return 1;
701 }
702
703 PetscErrorCode CreateSliceScatter(app_ctx *pctx,
704           Vec vec_from,
705           PetscInt ilo,
706           PetscInt ihi,
707           PetscInt jlo,
708           PetscInt jhi,
709           PetscInt klo,
710           PetscInt khi,
711           PetscInt dlo,
712           PetscInt dhi,
713           DA *da_to,
714           Vec *vec_to,
715           VecScatter *pscatter)
716 {
717   PetscErrorCode ierr;
718   int rank,size;
719
720   MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
721   MPI_Comm_size(PETSC_COMM_WORLD,&size);
722
723   PetscInt M,N,P,m,n,p,dof;
724   /* Get the global and local dimensions of solution DA */
725   DAGetInfo(pctx->da,PETSC_IGNORE,
726       &M,&N,&P,
```

```
727         &m,&n,&p,
728         &dof,
729         PETSC_IGNORE,
730         PETSC_IGNORE,PETSC_IGNORE);
731
732     PetscInt sx,sy,sz,mx,my,mz;
733     /* Get local grid boundaries, which are independent of dof */
734     DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
735
736     PetscInt M2D,N2D;
737     PetscReal xgmin,ygmin,xgmax,ygmax;
738     if (ilo==ihi) /* X_SLICE */
739       {
740         M2D=khi-klo+1,N2D=jhi-jlo+1;
741         xgmin=pctx->params->mesh.zmin+(klo-0)/pctx->invdz;
742         xgmax=pctx->params->mesh.zmax-(P-1-khi)/pctx->invdz;
743         ygmin=pctx->params->mesh.ymin+(jlo-0)/pctx->invdy;
744         ygmax=pctx->params->mesh.ymax-(N-1-jhi)/pctx->invdy;
745       }
746     else if (jlo==jhi) /* Y_SLICE */
747       {
748         M2D=ihi-ilo+1,N2D=khi-klo+1;
749         xgmin=pctx->params->mesh.xmin+(ilo-0)/pctx->invdx;
750         xgmax=pctx->params->mesh.xmax-(M-1-ihi)/pctx->invdx;
751         ygmin=pctx->params->mesh.zmin+(klo-0)/pctx->invdz;
752         ygmax=pctx->params->mesh.zmax-(P-1-khi)/pctx->invdz;
753       }
754     else if (klo==khi) /* Z_SLICE */
755       {
756         M2D=ihi-ilo+1,N2D=jhi-jlo+1;
757         xgmin=pctx->params->mesh.xmin+(ilo-0)/pctx->invdx;
758         xgmax=pctx->params->mesh.xmax-(M-1-ihi)/pctx->invdx;
759         ygmin=pctx->params->mesh.ymin+(jlo-0)/pctx->invdy;
760         ygmax=pctx->params->mesh.ymax-(N-1-jhi)/pctx->invdy;
761       }
762     else
763       {
764         PetscPrintf(PETSC_COMM_WORLD,"Error in slice specification!\n");
765         return 1;
766       }
767
768     PetscInt dofg=(dhi-dlo+1);
769
770     ierr=DACreate2d(PETSC_COMM_WORLD,DA_NONPERIODIC,DA_STENCIL_STAR,
771         M2D,N2D,
772         PETSC_DECIDE,PETSC_DECIDE,
773         dofg,
774         0,
775         PETSC_NULL,PETSC_NULL,
776         da_to);
```

```
777   CHKERRQ(ierr);
778
779   ierr=DASetUniformCoordinates(*da_to,
780                xgmin,xgmax,ygmin,ygmax,
781                0,0);
782   CHKERRQ(ierr);
783
784   /* A vector to store ground plane RMS of pressure */
785   ierr=DACreateGlobalVector(*da_to,vec_to);    CHKERRQ(ierr);
786
787   PetscInt n_idx,*idx_to=NULL,*idx_from=NULL;
788   IS is_to,is_from;
789
790   /* Create index set for scatter from 3D solution to 2D p vector */
791   if (sx+mx-1>=ilo && sx<=ihi &&
792       sy+my-1>=jlo && sy<=jhi &&
793       sz+mz-1>=klo && sz<=khi)
794     {
795       PetscInt i0=(sx>ilo?sx:ilo);
796       PetscInt iN=(sx+mx-1<ihi?sx+mx-1:ihi);
797       PetscInt j0=(sy>jlo?sy:jlo);
798       PetscInt jN=(sy+my-1<jhi?sy+my-1:jhi);
799       PetscInt k0=(sz>klo?sz:klo);
800       PetscInt kN=(sz+mz-1<khi?sz+mz-1:khi);
801
802       n_idx=(kN-k0+1)*(jN-j0+1)*(iN-i0+1)*dofg;
803       assert(n_idx>0);
804
805       PetscMalloc(sizeof(PetscInt)*n_idx,&idx_to);
806       PetscMalloc(sizeof(PetscInt)*n_idx,&idx_from);
807
808       const PetscInt *lx,*ly,*lz;
809       DAGetOwnershipRanges(pctx->da,&lx,&ly,&lz);
810
811       PetscInt Mg,Ng,Pg,mg,ng,pg;
812       /* Get the global and local dimensions of slice DA */
813       DAGetInfo(*da_to,PETSC_IGNORE,
814     &Mg,&Ng,PETSC_IGNORE,
815     &mg,&ng,PETSC_IGNORE,
816     PETSC_IGNORE,
817     PETSC_IGNORE,
818     PETSC_IGNORE,PETSC_IGNORE);
819
820       /* Get the ownership dimensions for the local 2D slice */
821       const PetscInt *lxg,*lyg;
822       DAGetOwnershipRanges(*da_to,&lxg,&lyg,PETSC_NULL);
823
824       /* i,j,k are the global node indices in 3D mesh. d is the index within dof*/
825       PetscInt i,j,k,d;
826       /* ig,jg are the global node indices in the 2D mesh */
```

```
827         PetscInt ig,jg;
828
829         /* bi,bj,bk index blocks of 3D mesh owned by processor */
830         PetscInt bi,bj,bk,sumi=0,sumj=0,sumk=0;
831         PetscInt big,bjg,sumig=0,sumjg=0;
832
833         /* Locate the offsets of all process blocks for mesh slice*/
834         PetscInt offsetg[mg][ng];
835         /* Zero out array */
836         for(bj=0;bj<ng;bj++)
837   for(bi=0;bi<mg;bi++)
838     offsetg[bi][bj]=0;
839         PetscInt offset=0;
840         for(bjg=0;bjg<ng;bjg++)
841   for(big=0;big<mg;big++)
842     {
843       offsetg[big][bjg] = offset;
844       offset += lxg[big]*lyg[bjg]*dofg;
845     }
846
847         /* Locate the offset of this process block of the mesh */
848         offset=0;
849         PetscInt proc=0;
850         for(bk=0;bk<p;bk++)
851   for(bj=0;bj<n;bj++)
852     for(bi=0;bi<m;bi++)
853       {
854         if (proc==rank) break;
855         offset += lx[bi]*ly[bj]*lz[bk]*dof;
856         proc++;
857       }
858
859         /* ii counts within idx_from and idx_to */
860         PetscInt ii=0;
861
862         /* Consume up to i0,j0,k0 to get the bi,bj,bk processor block */
863         sumk=0;
864         for(bk=0;bk<p;bk++)
865   {
866     if (sumk+lz[bk]>k0) break;
867     sumk += lz[bk];
868   }
869         assert(bk<p);
870
871         sumj=0;
872         for(bj=0;bj<n;bj++)
873   {
874     if (sumj+ly[bj]>j0) break;
875     sumj += ly[bj];
876   }
```

```
877         assert(bj<n);
878
879         sumi=0;
880         for(bi=0;bi<m;bi++)
881     {
882       if (sumi+lx[bi]>i0) break;
883       sumi += lx[bi];
884     }
885         assert(bi<m);
886
887         /* bi,bj,bk should point to this processor's block of the mesh.
888      sumi,sumj,sumk should be the number of nodes up to the lowest corner
889      of this process block.
890      We should never leave this processor's block, so bi,bj,bk and sumi,sumj,sumk
891      should not change past this point.
892         */
893
894         /* Need to keep track of big, bjg which can change due to the fact that
895      2D mesh slice is partitioned amongst all processors.
896      Further complication: concept of i and j directions on 2D mesh are
897      different depending on value of 'plane'.
898         */
899
900         for(k=k0;k<=kN;k++)
901     {
902       if (ilo==ihi || jlo==jhi) /* X_SLICE or Y_SLICE */
903         jg=k-klo;
904       for(j=j0;j<=jN;j++)
905         {
906           if (klo==khi) /* Z_SLICE */
907       jg=j-jlo;
908           else if (ilo==ihi) /* X_SLICE */
909       ig=j-jlo;
910           for(i=i0;i<=iN;i++)
911       {
912         if (jlo==jhi || klo==khi) /* Y_SLICE or Z_SLICE */
913           ig=i-ilo;
914
915         sumjg=0;
916         for(bjg=0;bjg<ng;bjg++)
917           {
918             if (sumjg+lyg[bjg]>jg) break;
919             sumjg += lyg[bjg];
920           }
921         assert(bjg<ng);
922
923         sumig=0;
924         for(big=0;big<mg;big++)
925           {
926             if (sumig+lxg[big]>ig) break;
```

```
927             sumig += lxg[big];
928           }
929         assert(big<mg);
930 
931         for(d=dlo;d<=dhi;d++)
932           {
933             assert(ii<n_idx);
934             idx_from[ii]=offset+(((k-sumk)*ly[bj]+(j-sumj))*lx[bi]+(i-sumi))*dof+d;
935             assert(idx_from[ii]<M*N*P*dof);
936 
937             idx_to[ii]=offsetg[big][bjg]+((jg-sumjg)*lxg[big]+(ig-sumig))*dofg+(d-dlo);
938             assert(idx_to[ii]<Mg*Ng*dofg);
939             ii++;
940           }
941       }
942         }
943     }
944       }
945     else
946       {
947         n_idx=0;
948       }
949 
950     ierr=ISCreateGeneral(PETSC_COMM_WORLD,n_idx,idx_from,&is_from);  CHKERRQ(ierr);
951     ierr=ISCreateGeneral(PETSC_COMM_WORLD,n_idx,idx_to,&is_to); CHKERRQ(ierr);
952 
953     if (idx_to) PetscFree(idx_to);
954     if (idx_from) PetscFree(idx_from);
955 
956     VecScatter slice_scatter;
957     ierr=VecScatterCreate(vec_from,is_from,*vec_to,is_to,pscatter); CHKERRQ(ierr);
958 
959     ISDestroy(is_from);
960     ISDestroy(is_to);
961 
962     return 0;
963 }
964 
965 void ComputeFFT(PetscInt NHistory,
966       PetscReal *tHistory,
967       PetscReal *vHistory,
968       PetscInt *N,
969       PetscReal **f,
970       PetscReal **vmag,
971       PetscReal **vphase,
972       PetscInt *Nband,
973       PetscReal **fband,
974       PetscReal **vband)
975 {
976     const PetscInt ANSIBandNumber[35]={10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
```

```
977             22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
978             34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44};
979   const PetscReal ANSINominalCenterFreq[35]={10.0, 12.5, 16, 20, 25, 31, 40, 50, 63,
980               80, 100, 125, 160, 200, 250, 315, 400,
981               500, 630, 800, 1000, 1250, 1600, 2000,
982               2500, 3150, 4000, 5000, 6300, 8000,
983               10000, 12500, 16000, 20000, 25000};
984   const PetscReal ANSIActualCenterFreq[35]={9.843, 12.401, 15.625, 19.686, 24.803,
985             31.250, 39.373, 49.606, 62.500, 78.745,
986             99.213, 125.000, 154.490, 198.425,
987             250.000, 314.980, 396.850, 500.000,
988             629.961, 793.701, 1000.000, 1259.921,
989             1587.401, 2000.000, 2519.842, 3174.802,
990             4000.000, 5039.684, 6349.604, 8000.00,
991             10079.368, 12699.208, 16000.000,
992             20158.737, 25398.417};
993   const PetscReal ANSIBandLowFreq[35]={8.769, 11.049, 13.920, 17.538, 22.098, 27.840,
994               35.077, 44.194, 55.681, 70.154, 88.388,
995               111.362, 140.308, 176.777, 222.725, 280.616,
996               353.553, 445.449, 561.123, 707.107, 890.899,
997               1122.462, 1414.214, 1781.797, 2244.924,
998               2828.427, 3563.595, 4489.848, 5656.854,
999               7127.190, 8979.696, 11313.708, 14253.379,
1000              17959.393, 22627.417};
1001  const PetscReal ANSIBandUpperFreq[35]={11.049, 13.920, 17.538, 22.098, 27.840,
1002          35.077, 44.194, 55.681, 70.154, 88.388,
1003          111.362, 140.308, 176.777, 222.725, 280.616,
1004          353.553, 445.449, 561.123, 707.107, 890.899,
1005          1122.462, 1414.214, 1781.797, 2244.924,
1006          2828.427, 3563.595, 4489.848, 5656.854,
1007          7127.190, 8979.696, 11313.708, 14253.379,
1008          17959.393, 22627.417, 28508.759};
1009  const PetscInt ANSIBandNumBins[35]={3, 4, 5, 6, 8, 10, 12, 16, 20, 24, 32, 40, 48,
1010              64, 80, 96, 128, 160, 192, 256, 320, 384, 512,
1011              640, 768, 1024, 1280, 1536, 2048, 2560, 3072,
1012              4096, 5120, 6144, 8192};
1013  const PetscInt ANSIBandBinUpper[35]={15, 19, 24, 30, 38, 48, 60, 76, 96, 120, 152,
1014               192, 240, 304, 384, 480, 608, 768, 960, 1216,
1015               1536, 1920, 2432, 3072, 3840, 4862, 6144,
1016               7680, 9728, 12288, 15360, 19456, 24576,
1017               30720, 38912};
1018
1019  int rank; MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
1020
1021  if (rank) return;
1022
1023  assert(tHistory);
1024  assert(vHistory);
1025
1026  /* vHistory may not be uniformly sampled, depending upon the time march method used.
```

```
1027      Resample for N samples */
1028
1029    const unsigned int DesiredSampleRate=48000;
1030    const unsigned int DesiredFFTBlocks=65516;
1031    const unsigned int bandlo=0;
1032    const unsigned int bandhi=43-ANSIBandNumber[0];
1033
1034    PetscReal T=tHistory[NHistory-1]-tHistory[0];
1035    unsigned int L=(unsigned int)(((double)DesiredSampleRate)*T);
1036    unsigned int NFFT=DesiredFFTBlocks;
1037
1038    if (L>NFFT) { L=NFFT; }
1039
1040    PetscReal Fs=L/T;    /* Sample frequency */
1041
1042    PetscPrintf(PETSC_COMM_WORLD,"Using sample length %d and FFT block size %d (Fs=%g)\n",
          L,NFFT,Fs);
1043
1044    double *in;
1045    fftw_complex *out;
1046    fftw_plan p;
1047
1048    in = (double*) fftw_malloc(sizeof(double) * NFFT);
1049    out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * (NFFT/2+1));
1050
1051    memset(in,0,sizeof(double)*NFFT); /* Zero out in vector, so it is zero padded at the
          end */
1052
1053    PetscReal dt=T/((PetscReal)(L-1));
1054
1055    /* Interpolate value history to in vector */
1056    int i,j;
1057    for(i=0,j=0;i<L;i++)
1058      {
1059        PetscReal t=i*dt+tHistory[0];
1060        while(j<NHistory && tHistory[j]<=t) j++;
1061        assert(j>0);
1062        if (j==NHistory) j=NHistory-1;
1063        in[i]=(vHistory[j]-vHistory[j-1])/(tHistory[j]-tHistory[j-1])*(t-tHistory[j-1])+
              vHistory[j-1];
1064        //        PetscPrintf(PETSC_COMM_WORLD,"%d %d %g %g (%g %g %g %g) \n",i,j,t,in[i],
              tHistory[j],tHistory[j-1],vHistory[j],vHistory[j-1]);
1065      }
1066
1067    p = fftw_plan_dft_r2c_1d(NFFT, in, out, FFTW_ESTIMATE);
1068
1069    fftw_execute(p);
1070
1071    (*N)=NFFT/2+1;
1072    PetscMalloc(sizeof(PetscReal)*(*N),f);
```

```
1073    PetscMalloc(sizeof(PetscReal)*(*N),vmag);
1074    PetscMalloc(sizeof(PetscReal)*(*N),vphase);
1075
1076    double deltaf=(Fs/2)/((double)((*N)-1));
1077    for(i=0;i<(*N);i++)
1078      {
1079        (*f)[i]=i*deltaf;
1080        (*vmag)[i]=sqrt(out[i][0]*out[i][0]+out[i][1]*out[i][1])/((double)NFFT);
1081        (*vphase)[i]=atan2(out[i][1],out[i][0]);
1082      }
1083
1084    (*Nband)=bandhi-bandlo;
1085    PetscMalloc(sizeof(PetscReal)*(*Nband),fband);
1086    PetscMalloc(sizeof(PetscReal)*(*Nband),vband);
1087    for(i=bandlo;i<bandhi;i++)
1088      {
1089        (*fband)[i-bandlo]=ANSINominalCenterFreq[i];
1090        (*vband)[i-bandlo]=0.0;
1091        unsigned int jlo=ANSIBandBinUpper[i]-ANSIBandNumBins[i];
1092        unsigned int jhi=ANSIBandBinUpper[i];
1093        assert(jlo<NFFT/2+1);
1094        assert(jhi<NFFT/2+1);
1095        for(j=jlo+1;j<=jhi;j++)
1096    {
1097      (*vband)[i-bandlo] += (*vmag)[j];
1098    }
1099      }
1100
1101    fftw_destroy_plan(p);
1102
1103    fftw_free(in);
1104    fftw_free(out);
1105  }
```

3.6 mean.h

```
1  #ifndef _MEAN_H_
2  #define _MEAN_H_
3  #include<petsc.h>
4  #include<petscbag.h>
5
6  /*! \file mean.h
7      \brief Mean flow parameters
8
9  Definitions and declared functions for describing the mean flow behaviour.
10 */
11
12 /*! \brief Ambient air properties
13
```

```
14   Free stream fluid properties of air.
15 */
16 typedef struct {
17   PetscReal rho;       /*!< fluid density (kg/m^3) */
18   PetscReal T;         /*!< fluid temperature (deg C) */
19   PetscReal R;         /*!< ideal gas constant (J/kg.K) */
20   PetscReal gamma;     /*!< ratio of specific heats */
21 } fluid_properties;
22
23 /*! \brief Mean uniform flow properties
24
25   Fluid properties for mean uniform flow, including quiescent fluid.  */
26 typedef struct {
27   PetscReal kappa,b,kappae,be,se;
28   PetscReal vx;
29   PetscReal vy;
30   PetscReal vz;
31   PetscReal M;         /*!< flow Mach number */
32   PetscReal dirx;      /*!< flow direction in x */
33   PetscReal diry;      /*!< flow direction in y (should always be zero) */
34   PetscReal dirz;      /*!< flow direction in z */
35 } mean_uniform_parameters;
36
37 /*! \brief Power law profile properties
38
39   Fluid properties for power law profile flow. */
40 typedef struct {
41   PetscReal kappa,b,kappae,be,se;
42   PetscReal dirx;       /*!< direction of flow in x */
43   PetscReal dirz;       /*!< direction of flow in z */
44   PetscReal a;          /*!< Power exponent, usually 1/7 */
45   PetscReal ur;         /*!< reference speed (m/s) */
46   PetscReal zr;         /*!< reference height (m) */
47 } power_profile_parameters;
48
49 /*! Available mean flow models */
50 typedef enum {
51   MEAN_UNIFORM=0,     /*!< uniform mean flow, including quiescent */
52   POWER_PROFILE=1     /*!< power law profile mean flow, including quiescent */
53 } mean_flow_type;
54
55 /*! Register fluid properties, and optionally load values from file. */
56 PetscErrorCode RegisterPropertiesBag(PetscBag *bag,fluid_properties **p,PetscTruth
       LoadFromFile);
57
58 /*! Get the ambient speed of sound (m/s) */
59 PetscReal GetSpeedOfSound(fluid_properties *pprop);
60 /*! Get the ambient pressure (Pa) */
61 PetscReal GetPressure(fluid_properties *pprop);
62 /*! Get the ambient density (kg/m^3) */
```

```
63 PetscReal GetDensity(fluid_properties *prop);
64 /*! Get the ratio of specific heats */
65 PetscReal GetRatioOfSpecificHeats(fluid_properties *prop);
66 /*! Get the ambient temperature (K) */
67 PetscReal GetTemperature(fluid_properties *prop);
68 /*! Get the ideal gas constant (J/kg.K) */
69 PetscReal GetGasConstant(fluid_properties *prop);
70
71 /*! Save mean flow parameter values to file. */
72 void SaveMeanBag(PetscBag bag);
73 /*! Save fluid properties to file. */
74 void SaveFluidPropertiesBag(PetscBag bag);
75
76 /*! Register uniform mean flow parameters, and optionally load values from file. */
77 PetscErrorCode RegisterUniformMeanBag(PetscBag *bag,void **params,PetscTruth
       LoadFromFile);
78
79 /*! Initialize uniform mean flow parameter values */
80 PetscErrorCode InitializeUniformMean(void *vpctx);
81
82 /*! Register power law profile flow parameters, and optionally load values from file. */
83 PetscErrorCode RegisterPowerProfileMeanBag(PetscBag *bag,void **params,PetscTruth
       LoadFromFile);
84
85 /*! Initialize power law profile mean flow parameter values */
86 PetscErrorCode InitializePowerProfileMean(void *vpctx);
87
88
89 #endif
```

3.7 mean.c

```
1 #include<stdlib.h>
2 #include<assert.h>
3 #include<petsc.h>
4
5 #include"mean.h"
6 #include"ode.h"
7
8 const char FluidPropertiesFile[]="properties.dat";
9 const char MeanParametersFile[]="mparameters.dat";
10
11 PetscErrorCode RegisterPropertiesBag(PetscBag *bag,fluid_properties **pparams,PetscTruth
       LoadFromFile)
12 {
13   PetscErrorCode ierr;
14
15   assert(bag);
16   assert(pparams);
```

```
17
18   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(fluid_properties),bag); CHKERRQ(ierr);
19   ierr=PetscBagGetData(*bag,(void**)pparams); CHKERRQ(ierr);
20
21   PetscBagSetName(*bag,"Fluid Properties","runtime parameters for fluid properties");
22
23   fluid_properties *properties=*pparams;
24   assert(properties);
25   ierr=PetscBagRegisterReal(*bag,&properties->rho,1.25,"fluid_rho","Fluid density (kg/m
       ^3)"); CHKERRQ(ierr);
26   ierr=PetscBagRegisterReal(*bag,&properties->T,20,"fluid_T","Fluid temperature (deg C)"
       ); CHKERRQ(ierr);
27   ierr=PetscBagRegisterReal(*bag,&properties->gamma,1.4,"fluid_gamma","Fluid ratio of
       specific heats"); CHKERRQ(ierr);
28   ierr=PetscBagRegisterReal(*bag,&properties->R,287,"fluid_R","Fluid gas constant (J/kg.
       K)"); CHKERRQ(ierr);
29
30   if (LoadFromFile)
31     {
32       PetscViewer bagviewer;
33       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,FluidPropertiesFile,FILE_MODE_READ
           ,&bagviewer))
34   {
35     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",FluidPropertiesFile);
36     CHKERRQ(ierr);
37   }
38       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
39       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
40       ierr=PetscBagGetData(*bag,(void**)pparams); CHKERRQ(ierr);
41       PetscPrintf(PETSC_COMM_WORLD,"Loaded fluid properties:\n");
42       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
43     }
44   return 0;
45 }
46
47 PetscReal GetSpeedOfSound(fluid_properties *pprop)
48 {
49   assert(pprop);
50   return sqrt(pprop->gamma*pprop->R*(pprop->T+273.15));
51 }
52
53 PetscReal GetPressure(fluid_properties *pprop)
54 {
55   assert(pprop);
56   return pprop->rho*pprop->R*(pprop->T+273.15);
57 }
58
59 PetscReal GetDensity(fluid_properties *prop)
60 {
61   assert(prop);
```

```
62   return prop->rho;
63 }
64
65 PetscReal GetRatioOfSpecificHeats(fluid_properties *prop)
66 {
67   assert(prop);
68   return prop->gamma;
69 }
70
71 PetscReal GetTemperature(fluid_properties *prop)
72 {
73   assert(prop);
74   return prop->T+273.15;
75 }
76
77 PetscReal GetGasConstant(fluid_properties *prop)
78 {
79   assert(prop);
80   return prop->R;
81 }
82
83 void SaveMeanBag(PetscBag bag)
84 {
85   PetscViewer bagviewer;
86   /* Save the runtime parameters to a binary file */
87   PetscViewerBinaryOpen(PETSC_COMM_WORLD, MeanParametersFile, FILE_MODE_WRITE,&bagviewer);
88   PetscBagView(bag, bagviewer);
89   PetscViewerDestroy(bagviewer);
90 }
91
92 void SaveFluidPropertiesBag(PetscBag bag)
93 {
94   PetscViewer bagviewer;
95   /* Save the runtime parameters to a binary file */
96   PetscViewerBinaryOpen(PETSC_COMM_WORLD, FluidPropertiesFile, FILE_MODE_WRITE,&bagviewer)
       ;
97   PetscBagView(bag, bagviewer);
98   PetscViewerDestroy(bagviewer);
99 }
100
101 PetscErrorCode RegisterUniformMeanBag(PetscBag *bag, void **pparams, PetscTruth
      LoadFromFile)
102 {
103   PetscErrorCode ierr;
104   assert(bag);
105   assert(pparams);
106
107   ierr=PetscBagCreate(PETSC_COMM_WORLD, sizeof(mean_uniform_parameters), bag); CHKERRQ(
       ierr);
108   ierr=PetscBagGetData(*bag, pparams); CHKERRQ(ierr);
```

```
109   ierr=PetscBagSetName(*bag,"Uniform Mean Flow Parameters","runtime parameters for a
          uniform mean flow"); CHKERRQ(ierr);
110
111   mean_uniform_parameters *pmean=(mean_uniform_parameters*)(*pparams);
112   /* Uniform mean flow parameters */
113   ierr=PetscBagRegisterReal(*bag,&pmean->M,0,"mean_uniform_M","Mean flow mach number");
          CHKERRQ(ierr);
114   ierr=PetscBagRegisterReal(*bag,&pmean->dirx,1,"mean_uniform_direction_x","Mean flow
          direction in x-direction"); CHKERRQ(ierr);
115   ierr=PetscBagRegisterReal(*bag,&pmean->diry,0,"mean_uniform_direction_y","Mean flow
          direction in y-direction"); CHKERRQ(ierr);
116   ierr=PetscBagRegisterReal(*bag,&pmean->dirz,0,"mean_uniform_direction_z","Mean flow
          direction in z-direction"); CHKERRQ(ierr);
117
118   if (LoadFromFile)
119     {
120       PetscViewer bagviewer;
121       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,MeanParametersFile,FILE_MODE_READ
              ,&bagviewer))
122   {
123     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",MeanParametersFile);
124     CHKERRQ(ierr);
125   }
126       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
127       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
128       ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
129       PetscPrintf(PETSC_COMM_WORLD,"Loaded uniform mean flow parameters:\n");
130       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
131     }
132   return 0;
133 }
134
135 PetscReal Uniform_kappa(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
136 {
137   app_ctx *pctx=(app_ctx*)pvoid;
138   mean_uniform_parameters *pmean=(mean_uniform_parameters*)(pctx->pmean);
139   assert(pvoid);
140   assert(pctx);
141   assert(pmean);
142
143   return pmean->kappa;
144 }
145
146
147 PetscReal Uniform_b(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
148 {
149   app_ctx *pctx=(app_ctx*)pvoid;
150   mean_uniform_parameters *pmean=(mean_uniform_parameters*)(pctx->pmean);
151   assert(pvoid);
152   assert(pctx);
```

```
153   assert(pmean);
154
155   return pmean->b;
156
157 }
158
159
160 PetscReal Uniform_v(PetscInt i,PetscInt j,PetscInt k,PetscInt d,void *pvoid)
161 {
162   app_ctx *pctx=(app_ctx*)pvoid;
163   mean_uniform_parameters *pmean=(mean_uniform_parameters*)(pctx->pmean);
164   assert(pvoid);
165   assert(pctx);
166   assert(pmean);
167   assert(d>=0 && d<3);
168   switch (d)
169     {
170     case 0:
171       return pmean->vx;
172       break;
173     case 1:
174       return pmean->vy;
175       break;
176     case 2:
177       return pmean->vz;
178       break;
179     };
180   return 0.0;
181 }
182
183
184 PetscReal Uniform_dv(PetscInt i,PetscInt j,PetscInt k,PetscInt d,PetscInt dd,void *pvoid
      )
185 {
186   return 0.0;
187 }
188
189 PetscErrorCode InitializeUniformMean(void *vpctx)
190 {
191   app_ctx *pctx=(app_ctx*)vpctx;
192   PetscInt d;
193   PetscReal c;
194   PetscReal rho;
195   PetscReal invlen;
196   mean_uniform_parameters *pmean=(mean_uniform_parameters*)pctx->pmean;
197
198   assert(pctx);
199   assert(pmean);
200   assert(pctx->properties);
201
```

```
202   c=GetSpeedOfSound(pctx->properties);
203   rho=GetDensity(pctx->properties);
204
205   pctx->calc_kappa=Uniform_kappa;
206   pctx->calc_b=Uniform_b;
207   pctx->calc_v=Uniform_v;
208   pctx->calc_dvdx=Uniform_dv;
209   pctx->calc_dvdy=Uniform_dv;
210   pctx->calc_dvdz=Uniform_dv;
211   pmean->b=1.0/rho;
212   pmean->kappa=rho*c*c;
213   invlen=1.0/sqrt(pmean->dirx*pmean->dirx+
214       pmean->diry*pmean->diry+
215       pmean->dirz*pmean->dirz);
216   pmean->vx=pmean->M*c*pmean->dirx*invlen;
217   pmean->vy=pmean->M*c*pmean->diry*invlen;
218   pmean->vz=pmean->M*c*pmean->dirz*invlen;
219
220   PetscPrintf(PETSC_COMM_WORLD,"Initializing uniform mean flow:\n");
221   PetscPrintf(PETSC_COMM_WORLD," c=%g m/s M=%g in %g,%g,%g direction.\n",c,pmean->M,
          pmean->dirx,pmean->diry,pmean->dirz);
222   PetscPrintf(PETSC_COMM_WORLD," kappa=%g b=%g.\n",pmean->kappa,pmean->b);
223   PetscPrintf(PETSC_COMM_WORLD," v_x=%g v_y=%g v_z=%g.\n",pmean->vx,pmean->vy,pmean->vz)
          ;
224
225   return 0;
226 }
227
228 PetscErrorCode RegisterPowerProfileMeanBag(PetscBag *bag,void **pparams,PetscTruth
      LoadFromFile)
229 {
230   PetscErrorCode ierr;
231   assert(bag);
232   assert(pparams);
233
234   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(power_profile_parameters),bag); CHKERRQ(
          ierr);
235   ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
236   ierr=PetscBagSetName(*bag,"Power Profile Mean Flow Parameters","runtime parameters for
          a power law profile mean flow"); CHKERRQ(ierr);
237
238   power_profile_parameters *pmean=(power_profile_parameters*)(*pparams);
239   /* Power law profile mean flow parameters */
240   ierr=PetscBagRegisterReal(*bag,&pmean->ur,7.5,"power_law_u_ref","Mean flow reference
          speed (m/s)"); CHKERRQ(ierr);
241   ierr=PetscBagRegisterReal(*bag,&pmean->zr,50,"power_law_z_ref","Mean flow reference
          speed height (m)"); CHKERRQ(ierr);
242   ierr=PetscBagRegisterReal(*bag,&pmean->a,1.0/7.0,"power_law_a","Mean flow reference
          power exponent"); CHKERRQ(ierr);
```

```
243   ierr=PetscBagRegisterReal(*bag,&pmean->dirx,1,"power_law_direction_x","Mean flow
          direction in x-direction"); CHKERRQ(ierr);
244   ierr=PetscBagRegisterReal(*bag,&pmean->dirz,0,"power_law_direction_z","Mean flow
          direction in z-direction"); CHKERRQ(ierr);
245
246   if (LoadFromFile)
247     {
248       PetscViewer bagviewer;
249       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,MeanParametersFile,FILE_MODE_READ
              ,&bagviewer))
250   {
251     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",MeanParametersFile);
252     CHKERRQ(ierr);
253   }
254       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
255       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
256       ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
257       PetscPrintf(PETSC_COMM_WORLD,"Loaded power law profile mean flow parameters:\n");
258       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
259     }
260   return 0;
261 }
262
263 PetscReal PowerProfile_kappa(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
264 {
265   app_ctx *pctx=(app_ctx*)pvoid;
266   power_profile_parameters *pmean=(power_profile_parameters*)(pctx->pmean);
267   assert(pvoid);
268   assert(pctx);
269   assert(pmean);
270
271   return pmean->kappa;
272 }
273
274
275 PetscReal PowerProfile_b(PetscInt i,PetscInt j,PetscInt k,void *pvoid)
276 {
277   app_ctx *pctx=(app_ctx*)pvoid;
278   power_profile_parameters *pmean=(power_profile_parameters*)(pctx->pmean);
279   assert(pvoid);
280   assert(pctx);
281   assert(pmean);
282
283   return pmean->b;
284
285 }
286
287
288 PetscReal PowerProfile_v(PetscInt i,PetscInt j,PetscInt k,PetscInt d,void *pvoid)
289 {
```

```
290   app_ctx *pctx=(app_ctx*)pvoid;
291   power_profile_parameters *pmean=(power_profile_parameters*)(pctx->pmean);
292   PetscReal y=0,u=0;
293
294   assert(pvoid);
295   assert(pctx);
296   assert(pmean);
297   assert(d>=0 && d<3);
298
299   if (y>0 && pmean->zr>0 && (d==0 || d==2))
300     {
301       if (j>pctx->j_lo)
302   {
303     assert(pctx->params);
304     assert(pctx->invdy);
305     y=((PetscReal)j-pctx->j_lo)*pctx->dy;
306     if (d==1) y += 0.5*pctx->dy;
307   }
308       u=pmean->ur*pow(y/pmean->zr,pmean->a);
309       switch (d)
310   {
311   case 0:
312     return u*pmean->dirx;
313     break;
314   case 2:
315     return u*pmean->dirz;
316     break;
317   };
318     }
319   return 0.0;
320 }
321
322
323 PetscReal PowerProfile_dv(PetscInt i,PetscInt j,PetscInt k,PetscInt d,PetscInt dd,void *
      pvoid)
324 {
325   app_ctx *pctx=(app_ctx*)pvoid;
326   power_profile_parameters *pmean=(power_profile_parameters*)(pctx->pmean);
327   PetscReal y=0,u=0,du=0;
328
329   assert(pvoid);
330   assert(pctx);
331   assert(pmean);
332   assert(d>=0 && d<3);
333   assert(dd>=0 && dd<3);
334
335   if (y>0 && pmean->zr>0 && (d==0 || d==2) && dd==1)
336     {
337       if (j>pctx->j_lo)
338   {
```

```
339       assert(pctx->params);
340       assert(pctx->invdy);
341       y=((PetscReal)j-pctx->j_lo)*pctx->dy;
342       if (d==1) y += 0.5*pctx->dy;
343     }
344         du=pmean->ur*pmean->a*pow(y/pmean->zr,pmean->a-1)/pmean->zr;
345         switch (d)
346     {
347     case 0:
348       return du*pmean->dirx;
349       break;
350     case 2:
351       return du*pmean->dirz;
352       break;
353     };
354       }
355     return 0.0;
356
357 }
358
359 PetscErrorCode InitializePowerProfileMean(void *vpctx)
360 {
361     app_ctx *pctx=(app_ctx*)vpctx;
362     PetscInt d;
363     PetscReal rho,c;
364     PetscReal invlen;
365     power_profile_parameters *pmean=(power_profile_parameters*)pctx->pmean;
366
367     assert(pctx);
368     assert(pmean);
369
370     c=GetSpeedOfSound(pctx->properties);
371     rho=GetDensity(pctx->properties);
372
373     pctx->calc_kappa=PowerProfile_kappa;
374     pctx->calc_b=PowerProfile_b;
375     pctx->calc_v=PowerProfile_v;
376     pctx->calc_dvdx=PowerProfile_dv;
377     pctx->calc_dvdy=PowerProfile_dv;
378     pctx->calc_dvdz=PowerProfile_dv;
379     pmean->b=1.0/rho;
380     pmean->kappa=rho*c*c;
381     invlen=1.0/sqrt(pmean->dirx*pmean->dirx+
382         pmean->dirz*pmean->dirz);
383
384     PetscPrintf(PETSC_COMM_WORLD,"Initializing power law profile mean flow:\n");
385     PetscPrintf(PETSC_COMM_WORLD," u=%g m/s at height %g m in %g,%g direction.\n",pmean->
          ur,pmean->zr,pmean->dirx,pmean->dirz);
386     PetscPrintf(PETSC_COMM_WORLD," kappa=%g b=%g.\n",pmean->kappa,pmean->b);
387
```

```
388   return 0;
389 }
```

3.8 ode.h

```
 1 #ifndef _ODE_H_
 2 #define _ODE_H_
 3 #include"petsc.h"
 4 #include"petscda.h"
 5
 6 /*! \file ode.h
 7     \brief ODE definition and application context.
 8
 9 This header file defines the application context (for passing parameters to various
       PETSc routines) and declares the functions for forming the RHS of the initial value
       problem.
10 */
11
12 #include"param.h"
13
14 /*! \brief The application context.
15
16   This structure is used to pass parameters and functions to PETSc TS functions. Note
       that functions are passed as pointer to functions.
17  */
18 typedef struct {
19   /* Petsc log events */
20   int SetupEvent,FormODEEvent,CalcSourceEvent;
21   //  int FormJacobianEvent;
22   /* Runtime parameter support. Stored here for checkpointing/signal handling */
23   PetscBag bag;                                        /*!< Parameter storage */
24
25   /* Source term calculator */
26   PetscErrorCode (*InitSource)(void *);   /*!< Source term initialization fxn. */
27   void (*CalculateSource)(PetscReal t,Vec v,void *); /*!< Calculate sources at time t,
       store in v. */
28   PetscReal (*GetSourceLowestFrequency)(void *); /*!< Get lowest frequency of source.*/
29   PetscReal (*GetSourceHighestFrequency)(void *); /*!< Get lowest frequency of source.*/
30   PetscReal (*GetOrigin)(void *,PetscInt i);  /*!< Get source origin in domain. */
31   PetscErrorCode (*DestroySource)(void *);   /*!< Source term deallocation fxn. */
32   PetscBag sourcebag;                       /*!< Source term parameter storage. */
33   void *psource;                            /*!< Ptr to source struct. */
34
35   /* Common fluid properties */
36   PetscBag pbag;                            /*!< Fluid properties storage */
37   fluid_properties *properties;             /*!< Ptr to fluid properties struct*/
38
39   /* Mean flow calculators (on-demand) */
40   PetscErrorCode (*InitMeanFlow)(void *);     /*!< Initialize mean flow */
```

```
41   PetscBag meanbag;                                  /*!< Mean flow parameter storage */
42   void *pmean;                                       /*!< Ptr to mean flow struct */
43   /*! Get \f$\kappa\f$ for mesh location \f$(i,j,k)\f$, corresponding to location \f$(x,
         y,z)\f$. */
44   PetscReal (*calc_kappa)(PetscInt i,PetscInt j,PetscInt k,void *pctx);
45   /*! Get \f$b\f$ for mesh location \f$(i,j,k)\f$, corresponding to location \f$(x,y,z)\
         f$. */
46   PetscReal (*calc_b)(PetscInt i,PetscInt j,PetscInt k,void *pctx);
47   /*! Get mean flow velocity component \f$v_d\f$ for mesh location \f$(i,j,k)\f$, offest
          accordingly. For example, \f$v_x\f$ is located at \f$(x+dx/2,y,z)\f$. */
48   PetscReal (*calc_v)(PetscInt i,PetscInt j,PetscInt k,PetscInt d,void *pctx);
49   /*! Get \f$\frac{\partial v_d}{\partial x}\f$ for mesh location \f$(i,j,k)\f$, offset
         accordingly. For example, \f$v_x\f$ is located at \f$(x+dx/2,y,z)\f$. */
50   PetscReal (*calc_dvdx)(PetscInt i,PetscInt j,PetscInt k,PetscInt d,PetscInt dd,void *
         pctx);
51   /*! Get \f$\frac{\partial v_d}{\partial y}\f$ for mesh location \f$(i,j,k)\f$, offset
         accordingly. For example, \f$v_x\f$ is located at \f$(x+dx/2,y,z)\f$. */
52   PetscReal (*calc_dvdy)(PetscInt i,PetscInt j,PetscInt k,PetscInt d,PetscInt dd,void *
         pctx);
53   /*! Get \f$\frac{\partial v_d}{\partial z}\f$ for mesh location \f$(i,j,k)\f$, offset
         accordingly. For example, \f$v_x\f$ is located at \f$(x+dx/2,y,z)\f$. */
54   PetscReal (*calc_dvdz)(PetscInt i,PetscInt j,PetscInt k,PetscInt d,PetscInt dd,void *
         pctx);
55
56   /* Absorbing boundary condition region (on-demand) */
57   PetscErrorCode (*InitBC)(void*);    /*!< Initialize boundary conditions */
58   PetscBag bcbag;                     /*!< BC parameter storage */
59   void *pbc;                          /*!< Ptr to BC struct */
60   /*! Get \f$\kappa_e\f$ in ABC region for mesh location \f$(i,j,k)\f$ , corresponding
         to location \f$(x,y,z)\f$. */
61   PetscReal (*calc_kappae)(PetscInt i,PetscInt j,PetscInt k,void *pctx);
62   /*! Get \f$b_e\f$ in ABC region for mesh location \f$(i,j,k)\f$ , corresponding to
         location \f$(x,y,z)\f$. */
63   PetscReal (*calc_be)(PetscInt i,PetscInt j,PetscInt k,void *pctx);
64   /*! Get \f$s_e\f$ in ABC region for mesh location \f$(i,j,k)\f$ , corresponding to
         location \f$(x,y,z)\f$. */
65   PetscReal (*calc_se)(PetscInt i,PetscInt j,PetscInt k,void *pctx);
66
67   /*! Global runtime parameters */
68   parameters *params;
69
70   /* Dimensions of the domain */
71   /*! Mesh spacing in \f$x\f$ */
72   PetscReal dx;
73   /*! Mesh spacing in \f$y\f$ */
74   PetscReal dy;
75   /*! Mesh spacing in \f$z\f$ */
76   PetscReal dz;
77   /*! Inverse of mesh spac
78 ing */
```

```
79    PetscReal invdx,invdy,invdz;
80    /*! Local lower and upper limits of the mesh */
81    PetscInt i_lo,i_hi,j_lo,j_hi,k_lo,k_hi;
82
83    /*! PETSc distributed array */
84    DA da;
85
86    /* Ground-plane sound pressure levels, sound power levels */
87    PetscTruth IntegrationStarted,SaveTransientPressure,SaveTransient,DoProbe,
          IntensityStarted,MaxAbsPressure;
88    DA dagp,dagw;
89    VecScatter groundp,groundw;
90    Vec prms,I,currp,currw,pmaxabs;
91    PetscScalar tstart,tlast,dtlast;
92    PetscInt rms_slice;
93    PetscReal RMSProbeDistance;
94    PetscInt probe_li,probe_lj;
95    PetscReal probe_w[2][2];
96
97    /* Integration of power along exterior of interior domain,
98       tracking of power history,
99       tracking of probe pressure history
100    */
101   PetscInt NHistory,HistoryAllocLen;
102   PetscReal SWL,*tHistory,*PowerHistory,*pHistory;
103
104
105   /* Checkpoint/Terminate support */
106   PetscTruth checkpoint,terminate;
107
108   /* Monitor support */
109   PetscLogDouble AvgSecondsPerIter;
110 } app_ctx;
111
112
113 /*! Initialize the solution vector for \f$t=0\f$. */
114 void InitializeSolution(Vec solution,
115       app_ctx *pctx);
116
117 /*! Setup the ODE problem, if necessary */
118 void SetupODE(app_ctx *pctx);
119
120 /*! \brief Form the RHS of the ODE.
121
122   Form the RHS of the ODE at time \f$t\f$, using the current solution vector \f$u\f$,
          storing the results in the residual vector \f$R\f$.
123
124   Note that the solution and residual vector are block vectors, where each block
          represents mesh location \f$(i,j,k)\f$. Each block of the solution vector
          represents acoustic pressure and velocity components \f$[p,w_x,w_y,w_z]\f$. Note
```

```
        that the mesh is staggard in space. The pressure value at mesh location \f$(i,j,k)
        \f$ represents location \f$(x,y,z)\f$, while \f$w_x\f$ represents the acoustic
        speed in the \f$x\f$ direction at location \f$(x+dx/2,y,z)\f$. Similarily, \f$w_y\
        f$ represents the acoustic speed in the \f$y\f$ direction at location \f$(x,y+dy
        /2,z)\f$, while \f$w_z\f$ represents the acoustic speed in the \f$z\f$ direction
        at \f$(x,y,z+dz/2)\f$.
125
126     The residual vector (RHS) is assembled as follows for interior points in the mesh.
127     \f{eqnarray*}{
128     R_{i,j,k}[0] &=&\frac{\partial p}{\partial t}= - \left(v_x\frac{\partial}{\partial x}+
            v_y\frac{\partial}{\partial y}+v_z\frac{\partial}{\partial z}\right) p - \kappa \
            left(\frac{\partial w_x}{\partial x}+ \frac{\partial w_y}{\partial y}+\frac{\
            partial w_z}{\partial z}\right) + \kappa Q\\
129     R_{i,j,k}[1] &=&\frac{\partial w_x}{\partial t}= - \left(w_x\frac{\partial}{\partial x
            }+w_y\frac{\partial}{\partial y}+w_z\frac{\partial}{\partial z}\right) v_x -\left(
            v_x\frac{\partial}{\partial x}+v_y\frac{\partial}{\partial y}+v_z\frac{\partial}{\
            partial z}\right) w_x -b \frac{\partial p}{\partial x} + b F_x\\
130     R_{i,j,k}[2] &=&\frac{\partial w_y}{\partial t}= - \left(w_x\frac{\partial}{\partial x
            }+w_y\frac{\partial}{\partial y}+w_z\frac{\partial}{\partial z}\right) v_y -\left(
            v_x\frac{\partial}{\partial x}+v_y\frac{\partial}{\partial y}+v_z\frac{\partial}{\
            partial z}\right) w_y -b \frac{\partial p}{\partial y} + b F_y\\
131     R_{i,j,k}[3] &=&\frac{\partial w_z}{\partial t}= - \left(w_x\frac{\partial}{\partial x
            }+w_y\frac{\partial}{\partial y}+w_z\frac{\partial}{\partial z}\right) v_z -\left(
            v_x\frac{\partial}{\partial x}+v_y\frac{\partial}{\partial y}+v_z\frac{\partial}{\
            partial z}\right) w_z -b \frac{\partial p}{\partial z} + b F_z\\
132     \kappa &=& \rho c^2\\
133     b &=& \frac{1}{\rho}\\
134     \f}
135
136     For absorbing boundary condition (ABC) regions of the mesh, the residual vector is
            assembled as follows.
137     \f{eqnarray*}{
138     R_{i,j,k}[0] &=&\frac{\partial p}{\partial t}= - \kappa_e \left(\frac{\partial w_x}{\
            partial x}+\frac{\partial w_y}{\partial y}+\frac{\partial w_z}{\partial z}\right)
            \\
139     R_{i,j,k}[1] &=&\frac{\partial w_x}{\partial t}= -s_e w_x -b_e \frac{\partial p}{\
            partial x}\\
140     R_{i,j,k}[2] &=&\frac{\partial w_y}{\partial t}= -s_e w_y -b_e \frac{\partial p}{\
            partial y}\\
141     R_{i,j,k}[3] &=&\frac{\partial w_z}{\partial t}= -s_e w_z -b_e \frac{\partial p}{\
            partial z}\\
142     \kappa_e &=& \frac{\rho c^2}{\gamma \Omega}\\
143     b_e &=& \frac{1}{\rho_e}\\
144     s_e &=& \frac{\sigma}{\rho_e}\\
145     \f}
146     See bc.h and source.h for more information.
147
148     For reflecting ground, the spatial derivatives and solution values across the
            reflecting plane are adjusted to preserve reflection properties.
149     a
```

```
150   Note that the above spatial derivatives are currently approximated using second-order
      centered differences, except at the boundaries of the mesh where \f$p=w_x=w_y=w_z
      =0\f$ is assumed outside the domain boundary.
151 */
152 PetscErrorCode FormODE(TS ts,            /*!< [in] PETSc Timestepper */
153            PetscReal t,      /*!< [in] time (s) */
154            Vec solution,     /*!< [in] current solution */
155            Vec residual,     /*!< [out] residual vector */
156            void *vpctx       /*!< [in] application context*/
157            );
158
159 /*! Determine if mesh location is within interior region. */
160 PetscTruth InteriorRegion(app_ctx *pctx,
161         PetscInt i,
162         PetscInt j,
163         PetscInt k);
164
165 /*! Determine if mesh location is within reflection region. */
166 PetscTruth ReflectionRegion(app_ctx *pctx,
167           PetscInt i,
168           PetscInt j,
169           PetscInt k);
170
171 /*! Determine if mesh location is within ABC region. */
172 PetscTruth AbsorbingBCRegion(app_ctx *pctx,
173            PetscInt i,
174            PetscInt j,
175            PetscInt k);
176
177 /*! Determine if mesh is two-dimensional */
178 PetscTruth TwoDimensional(app_ctx *pctx);
179
180 /*! Determine if at lowest i index of mesh. */
181 PetscTruth ILowFace(app_ctx *pctx,
182         PetscInt i,
183         PetscInt j,
184         PetscInt k);
185
186 /*! Determine if at lowest j index of mesh. */
187 PetscTruth JLowFace(app_ctx *pctx,
188         PetscInt i,
189         PetscInt j,
190         PetscInt k);
191
192 /*! Determine if at lowest k index of mesh. */
193 PetscTruth KLowFace(app_ctx *pctx,
194         PetscInt i,
195         PetscInt j,
196         PetscInt k);
197
```

```
198 /*! Determine if at highest i index of mesh. */
199 PetscTruth IHighFace(app_ctx *pctx,
200          PetscInt i,
201          PetscInt j,
202          PetscInt k);
203
204 /*! Determine if at highest j index of mesh. */
205 PetscTruth JHighFace(app_ctx *pctx,
206          PetscInt i,
207          PetscInt j,
208          PetscInt k);
209
210 /*! Determine if at highest k index of mesh. */
211 PetscTruth KHighFace(app_ctx *pctx,
212          PetscInt i,
213          PetscInt j,
214          PetscInt k);
215
216 #endif
```

3.9 ode.c

```
 1 #include<math.h>
 2 #include<assert.h>
 3 #include<stdlib.h>
 4 #include<petsc.h>
 5 #include<ode.h>
 6
 7 void InitializeSolution(Vec solution,
 8       app_ctx *pctx)
 9 {
10   PetscInt i,j;
11
12   /* Initial conditions are zero everywhere */
13   VecSet(solution,0.0);
14 }
15
16 void SetupODE(app_ctx *pctx)
17 {
18 }
19
20 PetscErrorCode FormODE(TS ts,
21            PetscReal t,
22            Vec solution,
23            Vec residual,
24            void *vpctx)
25 {
26   app_ctx *pctx=(app_ctx*)(vpctx);
27   PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz;
```

```
28  Vec local;
29  PetscScalar ****s;
30  PetscScalar ****r;
31
32  assert(pctx);
33
34  PetscLogEventBegin(pctx->FormODEEvent,0,0,0,0);
35
36  /* Assemble the residual vector,
37     which is already a local vector */
38
39  /* Zero residual vector */
40  VecSet(residual,0.0);
41
42  /* Borrow local vector representation from DA */
43  DAGetLocalVector(pctx->da,&local);
44
45  /* Determine global mesh nodal extents */
46  DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
47       PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
48       &dof,PETSC_IGNORE,
49       PETSC_IGNORE,PETSC_IGNORE);
50
51  /* Gather from global solution to local, ensure ghosts are up-to-date */
52  DAGlobalToLocalBegin(pctx->da,solution,INSERT_VALUES,local);
53  DAGlobalToLocalEnd(pctx->da,solution,INSERT_VALUES,local);
54
55  /* Grab the array representation of the local solution and
56     residual vectors */
57  DAVecGetArrayDOF(pctx->da,local,&s);
58  DAVecGetArrayDOF(pctx->da,residual,&r);
59
60  /* Get local grid boundaries, which are independent of dof */
61  DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
62
63  /* Fill in residual vector with source terms Q,F to save memory. */
64  if (pctx->CalculateSource) (*pctx->CalculateSource)(t,residual,vpctx);
65
66  /* To avoid a numerical problem referred to as odd-even decoupling, the spatial
       discretization of the solution is stored using a staggered mesh.
67
68     The solution is stored within one block array but represents a staggered mesh. p, b
         , kappa, Q are stored at whole mesh nodes, which represent locations (x=i*dx,y=
         j*dy,z=k*dz). vx, vy, vz, wx, wy, wz, Fx, Fy, Fz are stored at half nodes which
          represent the soluton at locations (x=(i+1/2)*dx,y=(j+1/2)*dy,z=(k+1/2)*dz).
69
70     Thus solution is a block vector which can be indexed using DAVecGetArray, which can
          be indexed as s[k][j][i][d] for the solution at node (i,j,k), equation d. As
         wx, vx, Fx, etc are stored in the same array, the (i,j,k)th block entry of the
         vector represents the solution p(i*dx,j*dy,k*dz), wx((i+1/2)*dx,j*dy,k*dz), wy(
```

```
          i*dx,(j+1/2)*dy,k*dz), wz(i*dx,j*dy,(k+1/2)*dz).
71
72        This means there is a band of nodes along i=0, j=0 and k=0 where no wx, wy, and wz
              are known.
73    */
74
75    /* Macros to make code more readable */
76    /* b at half node locations */
77 #define bi2(i,j,k) (0.5*(pctx->calc_b(i+1,j,k,vpctx)+pctx->calc_b(i,j,k,vpctx)))
78 #define bj2(i,j,k) (0.5*(pctx->calc_b(i,j+1,k,vpctx)+pctx->calc_b(i,j,k,vpctx)))
79 #define bk2(i,j,k) (0.5*(pctx->calc_b(i,j,k+1,vpctx)+pctx->calc_b(i,j,k,vpctx)))
80 #define BI2FLOPS 2
81 #define BJ2FLOPS 2
82 #define BK2FLOPS 2
83
84    /* be at half node locations, stored in b */
85 #define bei2(i,j,k) (0.5*(pctx->calc_be(i+1,j,k,vpctx)+pctx->calc_be(i,j,k,vpctx)))
86 #define bej2(i,j,k) (0.5*(pctx->calc_be(i,j+1,k,vpctx)+pctx->calc_be(i,j,k,vpctx)))
87 #define bek2(i,j,k) (0.5*(pctx->calc_be(i,j,k+1,vpctx)+pctx->calc_be(i,j,k,vpctx)))
88
89    /* se at half node locations, stored in se */
90 #define sei2(i,j,k) (0.5*(pctx->calc_se(i+1,j,k,vpctx)+pctx->calc_se(i,j,k,vpctx)))
91 #define sej2(i,j,k) (0.5*(pctx->calc_se(i,j+1,k,vpctx)+pctx->calc_se(i,j,k,vpctx)))
92 #define sek2(i,j,k) (0.5*(pctx->calc_se(i,j,k+1,vpctx)+pctx->calc_se(i,j,k,vpctx)))
93 #define SEI2FLOPS 2
94 #define SEJ2FLOPS 2
95 #define SEK2FLOPS 2
96
97    /* kappa, kappa_e, Q at whole node location */
98 #define kappa(i,j,k) (pctx->calc_kappa(i,j,k,vpctx))
99 #define kappae(i,j,k) (pctx->calc_kappae(i,j,k,vpctx))
100
101   /* vx, vy, vz at whole node locations */
102 #define vx(i,j,k) (0.5*(pctx->calc_v(i,j,k,0,vpctx)+pctx->calc_v(i-1,j,k,0,vpctx)))
103 #define vy(i,j,k) (0.5*(pctx->calc_v(i,j,k,1,vpctx)+pctx->calc_v(i,j-1,k,1,vpctx)))
104 #define vy_reflect(i,j,k) (0.0)
105 #define vz(i,j,k) (0.5*(pctx->calc_v(i,j,k,2,vpctx)+pctx->calc_v(i,j,k-1,2,vpctx)))
106 #define VXFLOPS 2
107 #define VYFLOPS 2
108 #define VZFLOPS 2
109
110   /* px, py, pz at whole node locations */
111 #define px(i,j,k) (0.5*(s[k][j][i+1][0]-s[k][j][i-1][0])*pctx->invdx)
112   //#define px_ilo(i,j,k) ((s[k][j][i+1][0]-s[k][j][i][0])*pctx->invdx)
113   //#define px_ihi(i,j,k) ((s[k][j][i][0]-s[k][j][i-1][0])*pctx->invdx)
114 #define px_ilo(i,j,k) (0.5*(s[k][j][i+1][0]-0)*pctx->invdx)
115 #define px_ihi(i,j,k) (0.5*(0-s[k][j][i-1][0])*pctx->invdx)
116 #define py(i,j,k) (0.5*(s[k][j+1][i][0]-s[k][j-1][i][0])*pctx->invdy)
117 #define py_reflect(i,j,k) (0.5*(s[k][j+1][i][0]-s[k][j][i][0])*pctx->invdy)
118 #define pz(i,j,k) (0.5*(s[k+1][j][i][0]-s[k-1][j][i][0])*pctx->invdz)
```

```
119   //#define pz_klo(i,j,k) ((s[k+1][j][i][0]-s[k][j][i][0])*pctx->invdz)
120   //#define pz_khi(i,j,k) ((s[k][j][i][0]-s[k-1][j][i][0])*pctx->invdz)
121 #define pz_klo(i,j,k) (0.5*(s[k+1][j][i][0]-0)*pctx->invdz)
122 #define pz_khi(i,j,k) (0.5*(0-s[k-1][j][i][0])*pctx->invdz)
123 #define PXFLOPS 3
124 #define PYFLOPS 3
125 #define PZFLOPS 3
126
127   /* wxx, wyy, wzz at whole node locations */
128 #define wxx(i,j,k) ((s[k][j][i][1]-s[k][j][i-1][1])*pctx->invdx)
129   //#define wxx_ilo(i,j,k) ((s[k][j][i+1][1]-s[k][j][i][1])*pctx->invdx)
130 #define wxx_ilo(i,j,k) ((s[k][j][i][1]-0)*pctx->invdx)
131 #define wyy(i,j,k) ((s[k][j][i][2]-s[k][j-1][i][2])*pctx->invdy)
132   //#define wyy_jlo(i,j,k) ((s[k][j+1][i][2]-s[k][j][i][2])*pctx->invdy)
133 #define wyy_jlo(i,j,k) ((s[k][j][i][2]-0)*pctx->invdy)
134 #define wyy_reflect(i,j,k) ((2*s[k][j][i][2])*pctx->invdy)
135 #define wzz(i,j,k) ((s[k][j][i][3]-s[k-1][j][i][3])*pctx->invdz)
136   //#define wzz_klo(i,j,k) ((s[k+1][j][i][3]-s[k][j][i][3])*pctx->invdz)
137 #define wzz_klo(i,j,k) ((s[k][j][i][3]-0)*pctx->invdz)
138 #define WXXFLOPS 2
139 #define WYYFLOPS 2
140 #define WZZFLOPS 2
141
142   /* px, py, pz at half node locations */
143 #define pxi2(i,j,k) ((s[k][j][i+1][0]-s[k][j][i][0])*pctx->invdx)
144   //#define pxi2_ihi(i,j,k) ((s[k][j][i][0]-s[k][j][i-1][0])*pctx->invdx)
145 #define pxi2_ihi(i,j,k) ((0-s[k][j][i][0])*pctx->invdx)
146 #define pyj2(i,j,k) ((s[k][j+1][i][0]-s[k][j][i][0])*pctx->invdy)
147   //#define pyj2_jhi(i,j,k) ((s[k][j][i][0]-s[k][j-1][i][0])*pctx->invdy)
148 #define pyj2_jhi(i,j,k) ((0-s[k][j][i][0])*pctx->invdy)
149 #define pzk2(i,j,k) ((s[k+1][j][i][0]-s[k][j][i][0])*pctx->invdz)
150   //#define pzk2_khi(i,j,k) ((s[k][j][i][0]-s[k-1][j][i][0])*pctx->invdz)
151 #define pzk2_khi(i,j,k) ((0.0-s[k][j][i][0])*pctx->invdz)
152 #define PXI2FLOPS 2
153 #define PYJ2FLOPS 2
154 #define PZK2FLOPS 2
155
156   /* vx, vy, vz at (i+1/2,j,k) half node locations */
157 #define vxi2(i,j,k) (pctx->calc_v(i,j,k,0,vpctx))
158 #define vyi2(i,j,k) (0.25*(pctx->calc_v(i,j,k,1,vpctx)+pctx->calc_v(i+1,j,k,1,vpctx)+
        pctx->calc_v(i,j-1,k,1,vpctx)+pctx->calc_v(i+1,j-1,k,1,vpctx)))
159 #define vyi2_reflect(i,j,k) (0.0)
160 #define vzi2(i,j,k) (0.25*(pctx->calc_v(i,j,k,2,vpctx)+pctx->calc_v(i+1,j,k,2,vpctx)+
        pctx->calc_v(i,j,k-1,2,vpctx)+pctx->calc_v(i+1,j,k-1,2,vpctx)))
161 #define VXI2FLOPS 0
162 #define VYI2FLOPS 4
163 #define VZI2FLOPS 4
164
165   /* wx, wy, wz at (i+1/2,j,k) half node locations */
166 #define wxi2(i,j,k) (s[k][j][i][1])
```

```
167 #define wyi2(i,j,k) (0.25*(s[k][j][i][2]+s[k][j][i+1][2]+s[k][j-1][i][2]+s[k][j-1][i
        +1][2]))
168 #define wyi2_reflect(i,j,k) (0.0)
169 #define wzi2(i,j,k) (0.25*(s[k][j][i][3]+s[k][j][i+1][3]+s[k-1][j][i][3]+s[k-1][j][i
        +1][3]))
170   //#define wzi2_ihi(i,j,k) (0.5*(s[k][j][i][3]+s[k-1][j][i][3]))
171   //#define wzi2_klo(i,j,k) (0.5*(s[k][j][i][3]+s[k][j][i+1][3]))
172   //#define wzi2_ihi_klo(i,j,k) ((s[k][j][i][3]))
173 #define wzi2_ihi(i,j,k) (0.25*(s[k][j][i][3]+0+s[k-1][j][i][3]+0))
174 #define wzi2_klo(i,j,k) (0.25*(s[k][j][i][3]+s[k][j][i+1][3]+0+0))
175 #define wzi2_ihi_klo(i,j,k) (0.25*(s[k][j][i][3]+0+0+0))
176 #define WXI2FLOPS 0
177 #define WYI2FLOPS 4
178 #define WZI2FLOPS 4
179
180   /* vx, vy, vz at (i,j+1/2,k) half node locations */
181 #define vxj2(i,j,k) (0.25*(pctx->calc_v(i,j,k,0,vpctx)+pctx->calc_v(i,j+1,k,0,vpctx)+
        pctx->calc_v(i-1,j+1,k,0,vpctx)+pctx->calc_v(i-1,j,k,0,vpctx)))
182 #define vyj2(i,j,k) (pctx->calc_v(i,j,k,1,vpctx))
183 #define vzj2(i,j,k) (0.25*(pctx->calc_v(i,j,k,2,vpctx)+pctx->calc_v(i,j+1,k,2,vpctx)+
        pctx->calc_v(i,j+1,k-1,2,vpctx)+pctx->calc_v(i,j,k-1,2,vpctx)))
184 #define VXJ2FLOPS 4
185 #define VYJ2FLOPS 0
186 #define VZJ2FLOPS 4
187
188   /* wx, wy, wz at (i,j+1/2,k) half node locations */
189 #define wxj2(i,j,k) (0.25*(s[k][j][i][1]+s[k][j+1][i][1]+s[k][j+1][i-1][1]+s[k][j][i
        -1][1]))
190   //#define wxj2_ilo(i,j,k) (0.5*(s[k][j][i][1]+s[k][j+1][i][1]))
191 #define wxj2_ilo(i,j,k) (0.25*(s[k][j][i][1]+s[k][j+1][i][1]+0+0))
192 #define wyj2(i,j,k) (s[k][j][i][2])
193 #define wzj2(i,j,k) (0.25*(s[k][j][i][3]+s[k][j+1][i][3]+s[k-1][j+1][i][3]+s[k-1][j][i
        ][3]))
194   //#define wzj2_klo(i,j,k) (0.5*(s[k][j][i][3]+s[k][j+1][i][3]))
195 #define wzj2_klo(i,j,k) (0.25*(s[k][j][i][3]+s[k][j+1][i][3]+0+0))
196 #define WXJ2FLOPS 4
197 #define WYJ2FLOPS 0
198 #define WZJ2FLOPS 4
199
200   /* vx, vy, vz at (i,j,k+1/2) half node locations */
201 #define vxk2(i,j,k) (0.25*(pctx->calc_v(i,j,k,0,vpctx)+pctx->calc_v(i-1,j,k,0,vpctx)+
        pctx->calc_v(i,j,k+1,0,vpctx)+pctx->calc_v(i-1,j,k+1,0,vpctx)))
202 #define vyk2(i,j,k) (0.25*(pctx->calc_v(i,j,k,1,vpctx)+pctx->calc_v(i,j-1,k,1,vpctx)+
        pctx->calc_v(i,j,k+1,1,vpctx)+pctx->calc_v(i,j-1,k+1,1,vpctx)))
203 #define vyk2_reflect(i,j,k) (0.0)
204 #define vzk2(i,j,k) (pctx->calc_v(i,j,k,2,vpctx))
205 #define VXK2FLOPS 4
206 #define VYK2FLOPS 4
207 #define VZK2FLOPS 0
208
```

```
209   /* wx, wy, wz at (i,j,k+1/2) half node locations */
210 #define wxk2(i,j,k) (0.25*(s[k][j][i][1]+s[k][j][i-1][1]+s[k+1][j][i][1]+s[k+1][j][i
        -1][1]))
211   //#define wxk2_khi(i,j,k) (0.5*(s[k][j][i][1]+s[k][j][i-1][1]))
212   //#define wxk2_ilo(i,j,k) (0.5*(s[k][j][i][1]+s[k+1][j][i][1]))
213   //#define wxk2_ilo_khi(i,j,k) ((s[k][j][i][1]))
214 #define wxk2_khi(i,j,k) (0.25*(s[k][j][i][1]+s[k][j][i-1][1]+0+0))
215 #define wxk2_ilo(i,j,k) (0.25*(s[k][j][i][1]+0+s[k+1][j][i][1]+0))
216 #define wxk2_ilo_khi(i,j,k) (0.25*(s[k][j][i][1]+0+0+0))
217 #define wyk2(i,j,k) (0.25*(s[k][j][i][2]+s[k][j-1][i][2]+s[k+1][j][i][2]+s[k+1][j-1][i
        ][2]))
218 #define wyk2_reflect(i,j,k) (0.0)
219 #define wzk2(i,j,k) (s[k][j][i][3])
220 #define WXK2FLOPS 4
221 #define WYK2FLOPS 4
222 #define WZK2FLOPS 0
223
224   /* wxx, wxy, wxz at half node locations (i+1/2,j,k) */
225 #define wxxi2(i,j,k) (0.5*(s[k][j][i+1][1]-s[k][j][i-1][1])*pctx->invdx)
226   //#define wxxi2_ihi(i,j,k) ((s[k][j][i][1]-s[k][j][i-1][1])*pctx->invdx)
227   //#define wxxi2_ilo(i,j,k) ((s[k][j][i+1][1]-s[k][j][i][1])*pctx->invdx)
228 #define wxxi2_ihi(i,j,k) (0.5*(0-s[k][j][i-1][1])*pctx->invdx)
229 #define wxxi2_ilo(i,j,k) (0.5*(s[k][j][i+1][1]-0)*pctx->invdx)
230 #define wxyi2(i,j,k) (0.5*(s[k][j+1][i][1]-s[k][j-1][i][1])*pctx->invdy)
231 #define wxyi2_reflect(i,j,k) (0.5*(s[k][j+1][i][1]-s[k][j][i][1])*pctx->invdy)
232 #define wxzi2(i,j,k) (0.5*(s[k+1][j][i][1]-s[k-1][j][i][1])*pctx->invdz)
233   //#define wxzi2_klo(i,j,k) ((s[k+1][j][i][1]-s[k][j][i][1])*pctx->invdz)
234   //#define wxzi2_khi(i,j,k) ((s[k][j][i][1]-s[k-1][j][i][1])*pctx->invdz)
235 #define wxzi2_klo(i,j,k) (0.5*(s[k+1][j][i][1]-0)*pctx->invdz)
236 #define wxzi2_khi(i,j,k) (0.5*(0-s[k-1][j][i][1])*pctx->invdz)
237 #define WXXI2FLOPS 3
238 #define WXYI2FLOPS 3
239 #define WXZI2FLOPS 3
240
241   /* wyx, wyy, wyz at half node locations (i,j+1/2,k) */
242 #define wyxj2(i,j,k) (0.5*(s[k][j][i+1][2]-s[k][j][i-1][2])*pctx->invdx)
243   //#define wyxj2_ilo(i,j,k) ((s[k][j][i+1][2]-s[k][j][i][2])*pctx->invdx)
244   //#define wyxj2_ihi(i,j,k) ((s[k][j][i][2]-s[k][j][i-1][2])*pctx->invdx)
245 #define wyxj2_ilo(i,j,k) (0.5*(s[k][j][i+1][2]-0)*pctx->invdx)
246 #define wyxj2_ihi(i,j,k) (0.5*(0-s[k][j][i-1][2])*pctx->invdx)
247 #define wyyj2(i,j,k) (0.5*(s[k][j+1][i][2]-s[k][j-1][i][2])*pctx->invdy)
248 #define wyyj2_reflect(i,j,k) (0.5*(s[k][j+1][i][2]+s[k][j][i][2])*pctx->invdy)
249 #define wyzj2(i,j,k) (0.5*(s[k+1][j][i][2]-s[k-1][j][i][2])*pctx->invdz)
250   //#define wyzj2_klo(i,j,k) (0.5*(s[k+1][j][i][2]-s[k][j][i][2])*pctx->invdz)
251   //#define wyzj2_khi(i,j,k) (0.5*(s[k][j][i][2]-s[k-1][j][i][2])*pctx->invdz)
252 #define wyzj2_klo(i,j,k) (0.5*(s[k+1][j][i][2]-0)*pctx->invdz)
253 #define wyzj2_khi(i,j,k) (0.5*(0-s[k-1][j][i][2])*pctx->invdz)
254 #define WYXJ2FLOPS 3
255 #define WYYJ2FLOPS 3
256 #define WYZJ2FLOPS 3
```

```
257
258   /* wzx, wzy, wzz at half node locations (i,j,k+1/2) */
259 #define wzxk2(i,j,k) (0.5*(s[k][j][i+1][3]-s[k][j][i-1][3])*pctx->invdx)
260   //#define wzxk2_ilo(i,j,k) ((s[k][j][i+1][3]-s[k][j][i][3])*pctx->invdx)
261   //#define wzxk2_ihi(i,j,k) ((s[k][j][i][3]-s[k][j][i-1][3])*pctx->invdx)
262 #define wzxk2_ilo(i,j,k) (0.5*(s[k][j][i+1][3]-0)*pctx->invdx)
263 #define wzxk2_ihi(i,j,k) (0.5*(0-s[k][j][i-1][3])*pctx->invdx)
264 #define wzyk2(i,j,k) (0.5*(s[k][j+1][i][3]-s[k][j-1][i][3])*pctx->invdy)
265 #define wzyk2_reflect(i,j,k) (0.5*(s[k][j+1][i][3]-s[k][j][i][3])*pctx->invdy)
266 #define wzzk2(i,j,k) (0.5*(s[k+1][j][i][3]-s[k-1][j][i][3])*pctx->invdz)
267   //#define wzzk2_klo(i,j,k) ((s[k+1][j][i][3]-s[k][j][i][3])*pctx->invdz)
268   //#define wzzk2_khi(i,j,k) ((s[k][j][i][3]-s[k-1][j][i][3])*pctx->invdz)
269 #define wzzk2_klo(i,j,k) (0.5*(s[k+1][j][i][3]-0)*pctx->invdz)
270 #define wzzk2_khi(i,j,k) (0.5*(0-s[k-1][j][i][3])*pctx->invdz)
271 #define WZXK2FLOPS 3
272 #define WZYK2FLOPS 3
273 #define WZZK2FLOPS 3
274
275   /* vxx, vxy, vxz at half node locations (i+1/2,j,k) */
276 #define vxxi2(i,j,k) (pctx->calc_dvdx(i,j,k,0,0,vpctx))
277 #define vxyi2(i,j,k) (pctx->calc_dvdy(i,j,k,0,1,vpctx))
278 #define vxzi2(i,j,k) (pctx->calc_dvdz(i,j,k,0,2,vpctx))
279 #define VXXI2FLOPS 0
280 #define VXYI2FLOPS 0
281 #define VXZI2FLOPS 0
282
283   /* vyx, vyy, vyz at half node locations (i,j+1/2,k) */
284 #define vyxj2(i,j,k) (pctx->calc_dvdx(i,j,k,1,0,vpctx))
285 #define vyyj2(i,j,k) (pctx->calc_dvdy(i,j,k,1,1,vpctx))
286 #define vyzj2(i,j,k) (pctx->calc_dvdz(i,j,k,1,2,vpctx))
287 #define VYXJ2FLOPS 0
288 #define VYYJ2FLOPS 0
289 #define VYZJ2FLOPS 0
290
291   /* vzx, vzy, vzz at half node locations (i,j,k+1/2) */
292 #define vzxk2(i,j,k) (pctx->calc_dvdx(i,j,k,2,0,vpctx))
293 #define vzyk2(i,j,k) (pctx->calc_dvdy(i,j,k,2,1,vpctx))
294 #define vzzk2(i,j,k) (pctx->calc_dvdz(i,j,k,2,2,vpctx))
295 #define VZXK2FLOPS 0
296 #define VZYK2FLOPS 0
297 #define VZZK2FLOPS 0
298
299   /* Calculate the residual on the local grid */
300   for(i=sx;i<sx+mx;i++)
301     for(j=sy;j<sy+my;j++)
302       for(k=sz;k<sz+mz;k++)
303   {
304     if (InteriorRegion(pctx,i,j,k))
305       {
306         if (TwoDimensional(pctx))
```

```
307     {
308       /* dp/dt at (i*dx,j*dy,k*dz) */
309       r[k][j][i][0]=
310         (-(vx(i,j,k)*px(i,j,k)+
311             vy(i,j,k)*py(i,j,k))
312          +kappa(i,j,k)*
313          (r[k][j][i][0]-
314           (wxx(i,j,k)+
315            wyy(i,j,k))));
316
317       PetscLogFlops(3+2+
318         (VXFLOPS+VYFLOPS+
319          PXFLOPS+PYFLOPS)-2
320         +1+1+1+2+
321         (WXXFLOPS+WYYFLOPS)-1);
322
323       /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
324       r[k][j][i][1]=
325         (-(wxi2(i,j,k)*vxxi2(i,j,k)+
326             wyi2(i,j,k)*vxyi2(i,j,k))
327          -(vxi2(i,j,k)*wxxi2(i,j,k)+
328             vyi2(i,j,k)*wxyi2(i,j,k))
329          +bi2(i,j,k)*
330          (r[k][j][i][1]-pxi2(i,j,k)));
331       PetscLogFlops(3+2+
332         (WXI2FLOPS+WYI2FLOPS+
333          VXXI2FLOPS+VXYI2FLOPS)-2
334         +1+3+2+
335         (VXI2FLOPS+VYI2FLOPS+
336          WXXI2FLOPS+WXYI2FLOPS)-2
337         +1+1+1+
338         PXI2FLOPS);
339
340       /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
341       r[k][j][i][2]=
342         (-(wxj2(i,j,k)*vyxj2(i,j,k)+
343             wyj2(i,j,k)*vyyj2(i,j,k))
344          -(vxj2(i,j,k)*wyxj2(i,j,k)+
345             vyj2(i,j,k)*wyyj2(i,j,k))
346          +bj2(i,j,k)*
347          (r[k][j][i][2]-pyj2(i,j,k)));
348       PetscLogFlops(3+2+
349         (WXJ2FLOPS+WYJ2FLOPS+
350          VYXJ2FLOPS+VYYJ2FLOPS)-2
351         +1+3+2+
352         (VXJ2FLOPS+VYJ2FLOPS+
353          WYXJ2FLOPS+WYYJ2FLOPS)-2
354         +1+1+1+
355         PYJ2FLOPS);
356
```

```
357         /* dwy/dt at (i*dx,j*dy,(k+1/2)*dz) */
358         r[k][j][i][3]=0.0;
359       }
360           else
361       {
362         /* dp/dt at (i*dx,j*dy,k*dz) */
363         r[k][j][i][0]=
364           (-(vx(i,j,k)*px(i,j,k)+
365               vy(i,j,k)*py(i,j,k)+
366               vz(i,j,k)*pz(i,j,k))
367            +kappa(i,j,k)*
368            (r[k][j][i][0]-
369             (wxx(i,j,k)+
370              wyy(i,j,k)+
371              wzz(i,j,k))));
372
373         PetscLogFlops(3+2+
374           (VXFLOPS+VYFLOPS+VZFLOPS+
375            PXFLOPS+PYFLOPS+PZFLOPS)
376           +1+1+1+2+
377           (WXXFLOPS+WYYFLOPS+WZZFLOPS));
378
379         /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
380         r[k][j][i][1]=
381           (-(wxi2(i,j,k)*vxxi2(i,j,k)+
382               wyi2(i,j,k)*vxyi2(i,j,k)+
383               wzi2(i,j,k)*vxzi2(i,j,k))
384            -(vxi2(i,j,k)*wxxi2(i,j,k)+
385              vyi2(i,j,k)*wxyi2(i,j,k)+
386              vzi2(i,j,k)*wxzi2(i,j,k))
387            +bi2(i,j,k)*
388            (r[k][j][i][1]-pxi2(i,j,k)));
389         PetscLogFlops(3+2+
390           (WXI2FLOPS+WYI2FLOPS+WZI2FLOPS+
391            VXXI2FLOPS+VXYI2FLOPS+VXZI2FLOPS)
392           +1+3+2+
393           (VXI2FLOPS+VYI2FLOPS+VZI2FLOPS+
394            WXXI2FLOPS+WXYI2FLOPS+WXZI2FLOPS)
395           +1+1+1+
396           PXI2FLOPS);
397
398         /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
399         r[k][j][i][2]=
400           (-(wxj2(i,j,k)*vyxj2(i,j,k)+
401               wyj2(i,j,k)*vyyj2(i,j,k)+
402               wzj2(i,j,k)*vyzj2(i,j,k))
403            -(vxj2(i,j,k)*wyxj2(i,j,k)+
404              vyj2(i,j,k)*wyyj2(i,j,k)+
405              vzj2(i,j,k)*wyzj2(i,j,k))
406            +bj2(i,j,k)*
```

```
407             (r[k][j][i][2]-pyj2(i,j,k)));
408         PetscLogFlops(3+2+
409           (WXJ2FLOPS+WYJ2FLOPS+WZJ2FLOPS+
410            VYXJ2FLOPS+VYYJ2FLOPS+VYZJ2FLOPS)
411           +1+3+2+
412           (VXJ2FLOPS+VYJ2FLOPS+VZJ2FLOPS+
413            WYXJ2FLOPS+WYYJ2FLOPS+WYZJ2FLOPS)
414           +1+1+1+
415           PYJ2FLOPS);
416
417         /* dwy/dt at (i*dx,j*dy,(k+1/2)*dz) */
418         r[k][j][i][3]=
419           (-(wxk2(i,j,k)*vzxk2(i,j,k)+
420               wyk2(i,j,k)*vzyk2(i,j,k)+
421               wzk2(i,j,k)*vzzk2(i,j,k))
422            -(vxk2(i,j,k)*wzxk2(i,j,k)+
423               vyk2(i,j,k)*wzyk2(i,j,k)+
424               vzk2(i,j,k)*wzzk2(i,j,k))
425            +bk2(i,j,k)*
426            (r[k][j][i][3]-pzk2(i,j,k)));
427         PetscLogFlops(3+2+
428           (WXK2FLOPS+WYK2FLOPS+WZK2FLOPS+
429            VZXK2FLOPS+VZYK2FLOPS+VZZK2FLOPS)
430           +1+3+2+
431           (VXK2FLOPS+VYK2FLOPS+VZK2FLOPS+
432            WZXK2FLOPS+WZYK2FLOPS+WZZK2FLOPS)
433           +1+1+1+
434           PZK2FLOPS);
435       }
436         }
437       else if (ReflectionRegion(pctx,i,j,k))
438         {
439           if (TwoDimensional(pctx))
440       {
441         /* A reflecting ground condition */
442         /* dp/dt at (i*dx,j*dy,k*dz) */
443         /* r[k][j][i][0] could contain a source term, unlikely but possible */
444         r[k][j][i][0] *= kappa(i,j,k);
445
446         if (ILowFace(pctx,i,j,k))
447           r[k][j][i][0] -=  vx(i,j,k)*px_ilo(i,j,k);
448         else if (IHighFace(pctx,i,j,k))
449           r[k][j][i][0] -=  vx(i,j,k)*px_ihi(i,j,k);
450         else
451           r[k][j][i][0] -=  vx(i,j,k)*px(i,j,k);
452         /* vy(i,j,k) zero on whole node for reflection */
453
454         if (ILowFace(pctx,i,j,k))
455           r[k][j][i][0] -= kappa(i,j,k)*wxx_ilo(i,j,k);
456         else
```

```
457           r[k][j][i][0] -= kappa(i,j,k)*wxx(i,j,k);
458
459         r[k][j][i][0] -= kappa(i,j,k)*wyy_reflect(i,j,k);
460
461         PetscLogFlops(3+2+
462           (VXFLOPS+
463            PXFLOPS)
464           +1+1+1+2+
465           (WXXFLOPS+WYYFLOPS));
466
467         /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
468         /* r[k][j][i][1] could contain a source term, unlikely but possible */
469         r[k][j][i][1] *= bi2(i,j,k);
470
471         r[k][j][i][1] -= wxi2(i,j,k)*vxxi2(i,j,k);
472         /* vxyi2(i,j,k) zero on ((i+1/2)*dx,j*dy,k*dz) for reflection */
473
474         if (ILowFace(pctx,i,j,k))
475           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2_ilo(i,j,k);
476         else if (IHighFace(pctx,i,j,k))
477           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2_ihi(i,j,k);
478         else
479           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2(i,j,k);
480         /* vyi2(i,j,k) zero on ((i+1/2)*dx,j*dy,k*dz) for reflection */
481
482         if (IHighFace(pctx,i,j,k))
483           r[k][j][i][1] -= bi2(i,j,k)*pxi2_ihi(i,j,k);
484         else
485           r[k][j][i][1] -= bi2(i,j,k)*pxi2(i,j,k);
486
487         PetscLogFlops(3+2+
488           (WXI2FLOPS+
489            VXXI2FLOPS)
490           +1+3+2+
491           (VXI2FLOPS+
492            WXXI2FLOPS)
493           +1+1+1+
494           PXI2FLOPS);
495
496         /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
497         /* r[k][j][i][2] could contain a source term, unlikely but possible */
498         r[k][j][i][2] *= bj2(i,j,k);
499
500         if (ILowFace(pctx,i,j,k))
501           r[k][j][i][2] -= wxj2_ilo(i,j,k)*vyxj2(i,j,k);
502         else
503           r[k][j][i][2] -= wxj2(i,j,k)*vyxj2(i,j,k);
504
505         r[k][j][i][2] -= wyj2(i,j,k)*vyyj2(i,j,k);
506
```

```
507       if (ILowFace(pctx,i,j,k))
508         r[k][j][i][2] -= vxj2(i,j,k)*wyxj2_ilo(i,j,k);
509       else if (IHighFace(pctx,i,j,k))
510         r[k][j][i][2] -= vxj2(i,j,k)*wyxj2_ihi(i,j,k);
511       else
512         r[k][j][i][2] -= vxj2(i,j,k)*wyxj2(i,j,k);
513
514       r[k][j][i][2] -= vyj2(i,j,k)*wyyj2_reflect(i,j,k);
515
516       r[k][j][i][2] -= bj2(i,j,k)*pyj2(i,j,k);
517
518       PetscLogFlops(3+2+
519         (WXJ2FLOPS+WYJ2FLOPS+
520          VYXJ2FLOPS+VYYJ2FLOPS)
521         +1+3+2+
522         (VXJ2FLOPS+VYJ2FLOPS+
523          WYXJ2FLOPS+WYYJ2FLOPS)
524         +1+1+1+
525         PYJ2FLOPS);
526
527       /* dwy/dt at (i*dx,j*dy,(k+1/2)*dz) */
528       r[k][j][i][3] = 0.0;
529     }
530       else
531     {
532       /* A reflecting ground condition */
533       /* dp/dt at (i*dx,j*dy,k*dz) */
534       /* r[k][j][i][0] could contain a source term, unlikely but possible */
535       r[k][j][i][0] *= kappa(i,j,k);
536
537       if (ILowFace(pctx,i,j,k))
538         r[k][j][i][0] -=  vx(i,j,k)*px_ilo(i,j,k);
539       else if (IHighFace(pctx,i,j,k))
540         r[k][j][i][0] -=  vx(i,j,k)*px_ihi(i,j,k);
541       else
542         r[k][j][i][0] -=  vx(i,j,k)*px(i,j,k);
543       /* vy(i,j,k) zero on whole node for reflection */
544       if (KLowFace(pctx,i,j,k))
545         r[k][j][i][0] -=  vz(i,j,k)*pz_klo(i,j,k);
546       else if (KHighFace(pctx,i,j,k))
547         r[k][j][i][0] -=  vz(i,j,k)*pz_khi(i,j,k);
548       else
549         r[k][j][i][0] -=  vz(i,j,k)*pz(i,j,k);
550
551       if (ILowFace(pctx,i,j,k))
552         r[k][j][i][0] -= kappa(i,j,k)*wxx_ilo(i,j,k);
553       else
554         r[k][j][i][0] -= kappa(i,j,k)*wxx(i,j,k);
555
556       r[k][j][i][0] -= kappa(i,j,k)*wyy_reflect(i,j,k);
```

```
557
558         if (KLowFace(pctx,i,j,k))
559           r[k][j][i][0] -= kappa(i,j,k)*wzz_klo(i,j,k);
560         else
561           r[k][j][i][0] -= kappa(i,j,k)*wzz(i,j,k);
562
563         PetscLogFlops(3+2+
564           (VXFLOPS+VZFLOPS+
565            PXFLOPS+PZFLOPS)
566           +1+1+1+2+
567           (WXXFLOPS+WYYFLOPS+WZZFLOPS));
568
569         /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
570         /* r[k][j][i][1] could contain a source term, unlikely but possible */
571         r[k][j][i][1] *= bi2(i,j,k);
572
573         r[k][j][i][1] -= wxi2(i,j,k)*vxxi2(i,j,k);
574         /* vxyi2(i,j,k) zero on ((i+1/2)*dx,j*dy,k*dz) for reflection */
575         if (IHighFace(pctx,i,j,k) && KLowFace(pctx,i,j,k))
576           r[k][j][i][1] -= wzi2_ihi_klo(i,j,k)*vxzi2(i,j,k);
577         else if (IHighFace(pctx,i,j,k))
578           r[k][j][i][1] -= wzi2_ihi(i,j,k)*vxzi2(i,j,k);
579         else if (KLowFace(pctx,i,j,k))
580           r[k][j][i][1] -= wzi2_klo(i,j,k)*vxzi2(i,j,k);
581         else
582           r[k][j][i][1] -= wzi2(i,j,k)*vxzi2(i,j,k);
583
584         if (ILowFace(pctx,i,j,k))
585           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2_ilo(i,j,k);
586         else if (IHighFace(pctx,i,j,k))
587           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2_ihi(i,j,k);
588         else
589           r[k][j][i][1] -= vxi2(i,j,k)*wxxi2(i,j,k);
590         /* vyi2(i,j,k) zero on ((i+1/2)*dx,j*dy,k*dz) for reflection */
591         if (KLowFace(pctx,i,j,k))
592           r[k][j][i][1] -= vzi2(i,j,k)*wxzi2_klo(i,j,k);
593         else if (KHighFace(pctx,i,j,k))
594           r[k][j][i][1] -= vzi2(i,j,k)*wxzi2_khi(i,j,k);
595         else
596           r[k][j][i][1] -= vzi2(i,j,k)*wxzi2(i,j,k);
597
598         if (IHighFace(pctx,i,j,k))
599           r[k][j][i][1] -= bi2(i,j,k)*pxi2_ihi(i,j,k);
600         else
601           r[k][j][i][1] -= bi2(i,j,k)*pxi2(i,j,k);
602
603         PetscLogFlops(3+2+
604           (WXI2FLOPS+WZI2FLOPS+
605            VXXI2FLOPS+VXZI2FLOPS)
606           +1+3+2+
```

```
607           (VXI2FLOPS+VZI2FLOPS+
608            WXXI2FLOPS+WXZI2FLOPS)
609           +1+1+1+
610           PXI2FLOPS);
611
612         /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
613         /* r[k][j][i][2] could contain a source term, unlikely but possible */
614         r[k][j][i][2] *= bj2(i,j,k);
615
616         if (ILowFace(pctx,i,j,k))
617           r[k][j][i][2] -= wxj2_ilo(i,j,k)*vyxj2(i,j,k);
618         else
619           r[k][j][i][2] -= wxj2(i,j,k)*vyxj2(i,j,k);
620         r[k][j][i][2] -= wyj2(i,j,k)*vyyj2(i,j,k);
621         if (KLowFace(pctx,i,j,k))
622           r[k][j][i][2] -= wzj2_klo(i,j,k)*vyzj2(i,j,k);
623         else
624           r[k][j][i][2] -= wzj2(i,j,k)*vyzj2(i,j,k);
625
626         if (ILowFace(pctx,i,j,k))
627           r[k][j][i][2] -= vxj2(i,j,k)*wyxj2_ilo(i,j,k);
628         else if (IHighFace(pctx,i,j,k))
629           r[k][j][i][2] -= vxj2(i,j,k)*wyxj2_ihi(i,j,k);
630         else
631           r[k][j][i][2] -= vxj2(i,j,k)*wyxj2(i,j,k);
632         r[k][j][i][2] -= vyj2(i,j,k)*wyyj2_reflect(i,j,k);
633         if (KLowFace(pctx,i,j,k))
634           r[k][j][i][2] -= vzj2(i,j,k)*wyzj2_klo(i,j,k);
635         else if (KHighFace(pctx,i,j,k))
636           r[k][j][i][2] -= vzj2(i,j,k)*wyzj2_khi(i,j,k);
637         else
638           r[k][j][i][2] -= vzj2(i,j,k)*wyzj2(i,j,k);
639
640         r[k][j][i][2] -= bj2(i,j,k)*pyj2(i,j,k);
641
642         PetscLogFlops(3+2+
643           (WXJ2FLOPS+WYJ2FLOPS+WZJ2FLOPS+
644            VYXJ2FLOPS+VYYJ2FLOPS+VYZJ2FLOPS)
645           +1+3+2+
646           (VXJ2FLOPS+VYJ2FLOPS+VZJ2FLOPS+
647            WYXJ2FLOPS+WYYJ2FLOPS+WYZJ2FLOPS)
648           +1+1+1+
649           PYJ2FLOPS);
650
651         /* dwy/dt at (i*dx,j*dy,(k+1/2)*dz) */
652         /* r[k][j][i][3] could contain a source term, unlikely but possible */
653         r[k][j][i][3] *= bk2(i,j,k);
654
655         if (ILowFace(pctx,i,j,k) && KHighFace(pctx,i,j,k))
656           r[k][j][i][3] -= wxk2_ilo_khi(i,j,k)*vzxk2(i,j,k);
```

```
657       else if (ILowFace(pctx,i,j,k))
658         r[k][j][i][3] -= wxk2_ilo(i,j,k)*vzxk2(i,j,k);
659       else if (KHighFace(pctx,i,j,k))
660         r[k][j][i][3] -= wxk2_khi(i,j,k)*vzxk2(i,j,k);
661       else
662         r[k][j][i][3] -= wxk2(i,j,k)*vzxk2(i,j,k);
663       /* wyk2(i,j,k) zero on i+1/2 nodes for reflection */
664       r[k][j][i][3] -= wzk2(i,j,k)*vzzk2(i,j,k);
665
666       if (ILowFace(pctx,i,j,k))
667         r[k][j][i][3] -= vxk2(i,j,k)*wzxk2_ilo(i,j,k);
668       else if (IHighFace(pctx,i,j,k))
669         r[k][j][i][3] -= vxk2(i,j,k)*wzxk2_ihi(i,j,k);
670       else
671         r[k][j][i][3] -= vxk2(i,j,k)*wzxk2(i,j,k);
672       /* vyk2(i,j,k) zero on k+1/2 nodes for reflection */
673       if (KLowFace(pctx,i,j,k))
674         r[k][j][i][3] -= vzk2(i,j,k)*wzzk2_klo(i,j,k);
675       else if (KHighFace(pctx,i,j,k))
676         r[k][j][i][3] -= vzk2(i,j,k)*wzzk2_khi(i,j,k);
677       else
678         r[k][j][i][3] -= vzk2(i,j,k)*wzzk2(i,j,k);
679
680       if (KHighFace(pctx,i,j,k))
681         r[k][j][i][3] -= bk2(i,j,k)*pzk2_khi(i,j,k);
682       else
683         r[k][j][i][3] -= bk2(i,j,k)*pzk2(i,j,k);
684
685       PetscLogFlops(3+2+
686         (WXK2FLOPS+WZK2FLOPS+
687          VZXK2FLOPS+VZZK2FLOPS)
688         +1+3+2+
689         (VXK2FLOPS+VZK2FLOPS+
690          WZXK2FLOPS+WZZK2FLOPS)
691         +1+1+1+
692         PZK2FLOPS);
693     }
694       }
695     else if (AbsorbingBCRegion(pctx,i,j,k))
696       {
697         if (TwoDimensional(pctx))
698     {
699       /* Absorbing Boundary Condition layers, including extremes of mesh */
700       /* dp/dt at (i*dx,j*dy,k*dz) */
701       r[k][j][i][0]=0.0;
702       if (ILowFace(pctx,i,j,k))
703         r[k][j][i][0] += -kappae(i,j,k)*wxx_ilo(i,j,k);
704       else
705         r[k][j][i][0] += -kappae(i,j,k)*wxx(i,j,k);
706       if (JLowFace(pctx,i,j,k))
```

```
707         r[k][j][i][0] += -kappae(i,j,k)*wyy_jlo(i,j,k);
708       else
709         r[k][j][i][0] += -kappae(i,j,k)*wyy(i,j,k);
710       PetscLogFlops(1+1+2+WXXFLOPS+WYYFLOPS);
711
712       /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
713       if (IHighFace(pctx,i,j,k))
714         r[k][j][i][1]=
715           (-sei2(i,j,k)*wxi2(i,j,k)
716            -bei2(i,j,k)*pxi2_ihi(i,j,k));
717       else
718         r[k][j][i][1]=
719           (-sei2(i,j,k)*wxi2(i,j,k)
720            -bei2(i,j,k)*pxi2(i,j,k));
721       PetscLogFlops(1+2+SEI2FLOPS+WXI2FLOPS+BI2FLOPS+PXI2FLOPS);
722
723       /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
724       if (JHighFace(pctx,i,j,k))
725         r[k][j][i][2]=
726           (-sej2(i,j,k)*wyj2(i,j,k)
727            -bej2(i,j,k)*pyj2_jhi(i,j,k));
728       else
729         r[k][j][i][2]=
730           (-sej2(i,j,k)*wyj2(i,j,k)
731            -bej2(i,j,k)*pyj2(i,j,k));
732       PetscLogFlops(1+2+SEJ2FLOPS+WYJ2FLOPS+BJ2FLOPS+PYJ2FLOPS);
733
734       r[k][j][i][3] = 0.0;
735     }
736       else
737     {
738       /* Absorbing Boundary Condition layers, including extremes of mesh */
739       /* dp/dt at (i*dx,j*dy,k*dz) */
740       r[k][j][i][0]=0.0;
741       if (ILowFace(pctx,i,j,k))
742         r[k][j][i][0] += -kappae(i,j,k)*wxx_ilo(i,j,k);
743       else
744         r[k][j][i][0] += -kappae(i,j,k)*wxx(i,j,k);
745       if (JLowFace(pctx,i,j,k))
746         r[k][j][i][0] += -kappae(i,j,k)*wyy_jlo(i,j,k);
747       else
748         r[k][j][i][0] += -kappae(i,j,k)*wyy(i,j,k);
749       if (KLowFace(pctx,i,j,k))
750         r[k][j][i][0] += -kappae(i,j,k)*wzz_klo(i,j,k);
751       else
752         r[k][j][i][0] += -kappae(i,j,k)*wzz(i,j,k);
753       PetscLogFlops(1+1+2+WXXFLOPS+WYYFLOPS+WZZFLOPS);
754
755       /* dwx/dt at ((i+1/2)*dx,j*dy,k*dz) */
756       if (IHighFace(pctx,i,j,k))
```

```
757           r[k][j][i][1]=
758             (-sei2(i,j,k)*wxi2(i,j,k)
759              -bei2(i,j,k)*pxi2_ihi(i,j,k));
760         else
761           r[k][j][i][1]=
762             (-sei2(i,j,k)*wxi2(i,j,k)
763              -bei2(i,j,k)*pxi2(i,j,k));
764         PetscLogFlops(1+2+SEI2FLOPS+WXI2FLOPS+BI2FLOPS+PXI2FLOPS);
765
766         /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
767         if (JHighFace(pctx,i,j,k))
768           r[k][j][i][2]=
769             (-sej2(i,j,k)*wyj2(i,j,k)
770              -bej2(i,j,k)*pyj2_jhi(i,j,k));
771         else
772           r[k][j][i][2]=
773             (-sej2(i,j,k)*wyj2(i,j,k)
774              -bej2(i,j,k)*pyj2(i,j,k));
775         PetscLogFlops(1+2+SEJ2FLOPS+WYJ2FLOPS+BJ2FLOPS+PYJ2FLOPS);
776
777         /* dwy/dt at (i*dx,(j+1/2)*dy,k*dz) */
778         if (KHighFace(pctx,i,j,k))
779           r[k][j][i][3]=
780             (-sek2(i,j,k)*wzk2(i,j,k)
781              -bek2(i,j,k)*pzk2_khi(i,j,k));
782         else
783           r[k][j][i][3]=
784             (-sek2(i,j,k)*wzk2(i,j,k)
785              -bek2(i,j,k)*pzk2(i,j,k));
786         PetscLogFlops(1+2+SEK2FLOPS+WZK2FLOPS+BK2FLOPS+PZK2FLOPS);
787       }
788         }
789       else
790         {
791           r[k][j][i][0]=0.0;
792           r[k][j][i][1]=0.0;
793           r[k][j][i][2]=0.0;
794           r[k][j][i][3]=0.0;
795         }
796     }
797     /* Release access to array */
798     DAVecRestoreArrayDOF(pctx->da,residual,&r);
799     DAVecRestoreArrayDOF(pctx->da,local,&s);
800
801     /* Return local vector representation to DA */
802     DARestoreLocalVector(pctx->da,&local);
803
804     PetscLogEventEnd(pctx->FormODEEvent,0,0,0,0);
805
806 //   PetscReal Min,Max;
```

```
807 //   VecMin(residual,PETSC_IGNORE,&Min);
808 //   VecMax(residual,PETSC_IGNORE,&Max);
809 //   if (Min!=Min || Max!=Max)
810 //     {
811 //       /* The solver has diverged! */
812 //       PetscPrintf(PETSC_COMM_WORLD,"The solver appears to have diverged (NaN in
        residual)!\n");
813 //       return 1;
814 //     }
815
816   return 0;
817 }
818
819 // PetscErrorCode FormJacobian(TS ts,
820 //           PetscReal t,
821 //           Vec solution,
822 //           Mat *pJ,
823 //           Mat *pPC,
824 //           MatStructure *flag,
825 //           void *vpctx)
826 // {
827 //   app_ctx *pctx=(app_ctx*)(vpctx);
828 //   PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz;
829 //   Vec local;
830 //   PetscScalar val[13*4][4];
831 //   MatStencil row,col;
832 //   PetscScalar ****s;
833 //   PetscScalar ***Q;
834 //   PetscScalar ****F;
835 //   PetscTruth Valid;
836 //
837 //   assert(0); /* This function is not complete (ReflectionRegion is currently wrong at
        i=0, k=0, i=Nx-1, k=Nz-1.) Some Jacobian terms are
838 //    not calculated correctly and use of this function
839 //    produces exponential memory usage */
840 //   assert(pctx);
841 //
842 //   PetscLogEventBegin(pctx->FormJacobianEvent,0,0,0,0);
843 //
844 //   /* Assemble the residual vector,
845 //      which is already a local vector */
846 //
847 //   /* Borrow local vector representation from DA */
848 //   DAGetLocalVector(pctx->da,&local);
849 //
850 //   /* Determine global mesh nodal extents */
851 //   DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
852 //      PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
853 //      &dof,PETSC_IGNORE,
854 //      PETSC_IGNORE,PETSC_IGNORE);
```

```
855 //
856 //    /* Gather from global solution to local */
857 //    DAGlobalToLocalBegin(pctx->da, solution ,INSERT_VALUES, local);
858 //    DAGlobalToLocalEnd(pctx->da, solution ,INSERT_VALUES, local);
859 //
860 //    /* Grab the array representation of the local solution and
861 //       residual vectors */
862 //    DAVecGetArrayDOF(pctx->da, local,&s);
863 //
864 //    /* Get local grid boundaries, which are independent of dof */
865 //    DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
866 //
867 //    /* Calculate the residual on the local grid */
868 //    for(i=sx;i<sx+mx;i++)
869 //      for(j=sy;j<sy+my;j++)
870 //        for(k=sz;k<sz+mz;k++)
871 //  {
872 //     row.i=i; row.j=j; row.k=k; row.c=0;
873 //     if (InteriorRegion(pctx,i,j,k))
874 //       {
875 //          col.i=i; col.j=j; col.k=k; col.c=0;
876 //          /* dR_(i,j,k)/dS_(i,j,k) at block (i,j,k),(i,j,k) */
877 //          val[0][0]=0.0;
878 //          val[0][1]=-kappa(i,j,k)*pctx->invdx;
879 //          val[0][2]=-kappa(i,j,k)*pctx->invdy;
880 //          val[0][3]=-kappa(i,j,k)*pctx->invdz;
881 //          val[1][0]=bi2(i,j,k)*pctx->invdx;
882 //          val[1][1]=-vxxi2(i,j,k);
883 //          val[1][2]=-0.25*vxyi2(i,j,k);
884 //          val[1][3]=-0.25*vxzi2(i,j,k);
885 //          val[2][0]=bj2(i,j,k)*pctx->invdy;
886 //          val[2][1]=-0.25*vyxj2(i,j,k);
887 //          val[2][2]=-vyyj2(i,j,k);
888 //          val[2][3]=-0.25*vyzj2(i,j,k);
889 //          val[3][0]=bk2(i,j,k)*pctx->invdz;
890 //          val[3][1]=-0.25*vzxk2(i,j,k);
891 //          val[3][2]=-0.25*vzyk2(i,j,k);
892 //          val[3][3]=-vzzk2(i,j,k);
893 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
894 //          /* Eliminate zeros in diagonal of preconditioner */
895 //          //for(d=0;d<dof;d++)
896 //          //   if (val[d][d]==0) val[d][d]=1e-3;
897 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
898 //          PetscLogFlops(3+
899 //            3+(BI2FLOPS+VXXI2FLOPS+VXYI2FLOPS+VXZI2FLOPS)+
900 //            3+(BJ2FLOPS+VYXJ2FLOPS+VYYJ2FLOPS+VYZJ2FLOPS)+
901 //            3+(BK2FLOPS+VZXK2FLOPS+VZYK2FLOPS+VZZK2FLOPS));
902 //
903 //          col.i=i+1; col.j=j; col.k=k; col.c=0;
904 //          /* dR_(i,j,k)/dS_(i+1,j,k) at block (i,j,k),(i+1,j,k) */
```

```
905 //          val[0][0]=-0.5*pctx->invdx*vx(i,j,k);
906 //          val[0][1]=0.0;
907 //          val[0][2]=0.0;
908 //          val[0][3]=0.0;
909 //          val[1][0]=-bi2(i,j,k)*pctx->invdx;
910 //          val[1][1]=-0.5*vxi2(i,j,k)*pctx->invdx;
911 //          val[1][2]=-0.25*vxyi2(i,j,k);
912 //          val[1][3]=-0.25*vxzi2(i,j,k);
913 //          val[2][0]=0.0;
914 //          val[2][1]=0.0;
915 //          val[2][2]=-0.5*vxj2(i,j,k)*pctx->invdx;
916 //          val[2][3]=0.0;
917 //          val[3][0]=0.0;
918 //          val[3][1]=0.0;
919 //          val[3][2]=0.0;
920 //          val[3][3]=-0.5*pctx->invdx*vxk2(i,j,k);
921 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
922 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
923 //
924 //          PetscLogFlops(2+(VXFLOPS)+
925 //            5+(BI2FLOPS+VXI2FLOPS+VXYI2FLOPS+VXZI2FLOPS)+
926 //            2+(VXJ2FLOPS)+
927 //            2+(VXK2FLOPS));
928 //
929 //          col.i=i-1; col.j=j; col.k=k; col.c=0;
930 //          /* dR_(i,j,k)/dS_(i-1,j,k) at block (i,j,k),(i-1,j,k) */
931 //          val[0][0]=+0.5*pctx->invdx*vx(i,j,k);
932 //          val[0][1]=kappa(i,j,k)*pctx->invdx;
933 //          val[0][2]=0.0;
934 //          val[0][3]=0.0;
935 //          val[1][0]=0.0;
936 //          val[1][1]=+0.5*vxi2(i,j,k)*pctx->invdx;
937 //          val[1][2]=0.0;
938 //          val[1][3]=0.0;
939 //          val[2][0]=0.0;
940 //          val[2][1]=-0.25*vyxj2(i,j,k);
941 //          val[2][2]=+0.5*vxj2(i,j,k)*pctx->invdx;
942 //          val[2][3]=0.0;
943 //          val[3][0]=0.0;
944 //          val[3][1]=-0.25*vzxk2(i,j,k);
945 //          val[3][2]=0.0;
946 //          val[3][3]=0.5*pctx->invdx*vxk2(i,j,k);
947 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
948 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
949 //          PetscLogFlops(3+(VXFLOPS)+
950 //            2+(VXI2FLOPS)+
951 //            3+(VYXJ2FLOPS+VXJ2FLOPS)+
952 //            3+(VZXK2FLOPS+VXK2FLOPS));
953 //
954 //          col.i=i; col.j=j+1; col.k=k; col.c=0;
```

```
955 //          /* dR_(i,j,k)/dS_(i,j+1,k) at block (i,j,k),(i,j+1,k) */
956 //          val[0][0]=-0.5*pctx->invdy*vy(i,j,k);
957 //          val[0][1]=0.0;
958 //          val[0][2]=0.0;
959 //          val[0][3]=0.0;
960 //          val[1][0]=0.0;
961 //          val[1][1]=-0.5*pctx->invdy*vyi2(i,j,k);
962 //          val[1][2]=0.0;
963 //          val[1][3]=0.0;
964 //          val[2][0]=-pctx->invdy*bj2(i,j,k);
965 //          val[2][1]=-0.25*vyxj2(i,j,k);
966 //          val[2][2]=-0.5*pctx->invdy*vyj2(i,j,k);
967 //          val[2][3]=-0.25*vyzj2(i,j,k);
968 //          val[3][0]=0.0;
969 //          val[3][1]=0.0;
970 //          val[3][2]=0.0;
971 //          val[3][3]=-0.5*pctx->invdy*vyk2(i,j,k);
972 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
973 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
974 //          PetscLogFlops(2+(VYFLOPS)+
975 //            5+(BJ2FLOPS+VYJ2FLOPS+VYXJ2FLOPS+VYZJ2FLOPS)+
976 //            2+(VYI2FLOPS)+
977 //            2+(VYK2FLOPS));
978 //
979 //          col.i=i; col.j=j-1; col.k=k; col.c=0;
980 //          /* dR_(i,j,k)/dS_(i,j-1,k) at block (i,j,k),(i,j-1,k) */
981 //          val[0][0]=0.5*pctx->invdy*vy(i,j,k);
982 //          val[0][1]=0.0;
983 //          val[0][2]=pctx->invdy*kappa(i,j,k);
984 //          val[0][3]=0.0;
985 //          val[1][0]=0.0;
986 //          val[1][1]=0.5*pctx->invdy*vyi2(i,j,k);
987 //          val[1][2]=-0.25*vxyi2(i,j,k);
988 //          val[1][3]=0.0;
989 //          val[2][0]=0.0;
990 //          val[2][1]=0.0;
991 //          val[2][2]=0.5*pctx->invdy*vyj2(i,j,k);
992 //          val[2][3]=0.0;
993 //          val[3][0]=0.0;
994 //          val[3][1]=0.0;
995 //          val[3][2]=-0.25*vzyk2(i,j,k);
996 //          val[3][3]=0.5*pctx->invdy*vyk2(i,j,k);
997 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
998 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
999 //          PetscLogFlops(3+(VYFLOPS)+
1000 //            2+(VYJ2FLOPS)+
1001 //            3+(VXYI2FLOPS+VYI2FLOPS)+
1002 //            3+(VZXK2FLOPS+VYK2FLOPS));
1003 //
1004 //          col.i=i; col.j=j; col.k=k+1; col.c=0;
```

```
1005 //          /* dR_(i,j,k)/dS_(i,j,k+1) at block (i,j,k),(i,j,k+1) */
1006 //          val[0][0]=-0.5*pctx->invdz*vz(i,j,k);
1007 //          val[0][1]=0.0;
1008 //          val[0][2]=0.0;
1009 //          val[0][3]=0.0;
1010 //          val[1][0]=0.0;
1011 //          val[1][1]=-0.5*pctx->invdz*vzi2(i,j,k);
1012 //          val[1][2]=0.0;
1013 //          val[1][3]=0.0;
1014 //          val[2][0]=0.0;
1015 //          val[2][1]=0.0;
1016 //          val[2][2]=-0.5*pctx->invdz*vzj2(i,j,k);
1017 //          val[2][3]=0.0;
1018 //          val[3][0]=-pctx->invdz*bk2(i,j,k);
1019 //          val[3][1]=-0.25*vzxk2(i,j,k);
1020 //          val[3][2]=-0.25*vzyk2(i,j,k);
1021 //          val[3][3]=-0.5*pctx->invdz*vzk2(i,j,k);
1022 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1023 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1024 //          PetscLogFlops(2+(VZFLOPS)+
1025 //            5+(BK2FLOPS+VZK2FLOPS+VZXK2FLOPS+VZYK2FLOPS)+
1026 //            2+(VZI2FLOPS)+
1027 //            2+(VZJ2FLOPS));
1028 //
1029 //          col.i=i; col.j=j; col.k=k-1; col.c=0;
1030 //          /* dR_(i,j,k)/dS_(i,j,k-1) at block (i,j,k),(i,j,k-1) */
1031 //          val[0][0]=0.5*pctx->invdz*vz(i,j,k);
1032 //          val[0][1]=0.0;
1033 //          val[0][2]=0.0;
1034 //          val[0][3]=pctx->invdz*kappa(i,j,k);
1035 //          val[1][0]=0.0;
1036 //          val[1][1]=0.5*pctx->invdz*vzi2(i,j,k);
1037 //          val[1][2]=0.0;
1038 //          val[1][3]=-0.25*vxzi2(i,j,k);
1039 //          val[2][0]=0.0;
1040 //          val[2][1]=0.0;
1041 //          val[2][2]=0.5*pctx->invdz*vzj2(i,j,k);
1042 //          val[2][3]=-0.25*vyzj2(i,j,k);
1043 //          val[3][0]=0.0;
1044 //          val[3][1]=0.0;
1045 //          val[3][2]=0.0;
1046 //          val[3][3]=0.5*pctx->invdz*vzk2(i,j,k);
1047 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1048 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1049 //          PetscLogFlops(3+(VZFLOPS)+
1050 //            2+(VZK2FLOPS)+
1051 //            3+(VXZI2FLOPS+VZI2FLOPS)+
1052 //            3+(VYZJ2FLOPS+VZJ2FLOPS));
1053 //
1054 //          col.i=i+1; col.j=j; col.k=k-1; col.c=0;
```

```
1055 //          /* dR_(i,j,k)/dS_(i+1,j,k-1) at block (i,j,k),(i+1,j,k-1) */
1056 //          val[0][0]=0.0;
1057 //          val[0][1]=0.0;
1058 //          val[0][2]=0.0;
1059 //          val[0][3]=0.0;
1060 //          val[1][0]=0.0;
1061 //          val[1][1]=0.0;
1062 //          val[1][2]=0.0;
1063 //          val[1][3]=-0.25*vxzi2(i,j,k);
1064 //          val[2][0]=0.0;
1065 //          val[2][1]=0.0;
1066 //          val[2][2]=0.0;
1067 //          val[2][3]=0.0;
1068 //          val[3][0]=0.0;
1069 //          val[3][1]=0.0;
1070 //          val[3][2]=0.0;
1071 //          val[3][3]=0.0;
1072 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1073 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1074 //          PetscLogFlops(1+VXZI2FLOPS);
1075 //
1076 //          col.i=i-1; col.j=j; col.k=k+1; col.c=0;
1077 //          /* dR_(i,j,k)/dS_(i-1,j,k+1) at block (i,j,k),(i-1,j,k+1) */
1078 //          val[0][0]=0.0;
1079 //          val[0][1]=0.0;
1080 //          val[0][2]=0.0;
1081 //          val[0][3]=0.0;
1082 //          val[1][0]=0.0;
1083 //          val[1][1]=0.0;
1084 //          val[1][2]=0.0;
1085 //          val[1][3]=0.0;
1086 //          val[2][0]=0.0;
1087 //          val[2][1]=0.0;
1088 //          val[2][2]=0.0;
1089 //          val[2][3]=0.0;
1090 //          val[3][0]=0.0;
1091 //          val[3][1]=-0.25*vzxk2(i,j,k);
1092 //          val[3][2]=0.0;
1093 //          val[3][3]=0.0;
1094 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1095 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1096 //          PetscLogFlops(1+VZXK2FLOPS);
1097 //
1098 //          col.i=i; col.j=j+1; col.k=k-1; col.c=0;
1099 //          /* dR_(i,j,k)/dS_(i,j+1,k-1) at block (i,j,k),(i,j+1,k-1) */
1100 //          val[0][0]=0.0;
1101 //          val[0][1]=0.0;
1102 //          val[0][2]=0.0;
1103 //          val[0][3]=0.0;
1104 //          val[1][0]=0.0;
```

```
1105 //          val[1][1]=0.0;
1106 //          val[1][2]=0.0;
1107 //          val[1][3]=0.0;
1108 //          val[2][0]=0.0;
1109 //          val[2][1]=0.0;
1110 //          val[2][2]=0.0;
1111 //          val[2][3]=-0.25*vyzj2(i,j,k);
1112 //          val[3][0]=0.0;
1113 //          val[3][1]=0.0;
1114 //          val[3][2]=0.0;
1115 //          val[3][3]=0.0;
1116 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1117 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1118 //          PetscLogFlops(1+VYZJ2FLOPS);
1119 //
1120 //          col.i=i; col.j=j-1; col.k=k+1; col.c=0;
1121 //          /* dR_(i,j,k)/dS_(i,j-1,k+1) at block (i,j,k),(i,j-1,k+1) */
1122 //          val[0][0]=0.0;
1123 //          val[0][1]=0.0;
1124 //          val[0][2]=0.0;
1125 //          val[0][3]=0.0;
1126 //          val[1][0]=0.0;
1127 //          val[1][1]=0.0;
1128 //          val[1][2]=0.0;
1129 //          val[1][3]=0.0;
1130 //          val[2][0]=0.0;
1131 //          val[2][1]=0.0;
1132 //          val[2][2]=0.0;
1133 //          val[2][3]=0.0;
1134 //          val[3][0]=0.0;
1135 //          val[3][1]=0.0;
1136 //          val[3][2]=-0.25*vzyk2(i,j,k);
1137 //          val[3][3]=0.0;
1138 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1139 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1140 //          PetscLogFlops(1+VZYK2FLOPS);
1141 //
1142 //          col.i=i+1; col.j=j-1; col.k=k; col.c=0;
1143 //          /* dR_(i,j,k)/dS_(i+1,j-1,k) at block (i,j,k),(i+1,j-1,k) */
1144 //          val[0][0]=0.0;
1145 //          val[0][1]=0.0;
1146 //          val[0][2]=0.0;
1147 //          val[0][3]=0.0;
1148 //          val[1][0]=0.0;
1149 //          val[1][1]=0.0;
1150 //          val[1][2]=-0.25*vxyi2(i,j,k);
1151 //          val[1][3]=0.0;
1152 //          val[2][0]=0.0;
1153 //          val[2][1]=0.0;
1154 //          val[2][2]=0.0;
```

```
1155 //          val[2][3]=0.0;
1156 //          val[3][0]=0.0;
1157 //          val[3][1]=0.0;
1158 //          val[3][2]=0.0;
1159 //          val[3][3]=0.0;
1160 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1161 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1162 //          PetscLogFlops(1+VXYI2FLOPS);
1163 //
1164 //          col.i=i-1; col.j=j+1; col.k=k; col.c=0;
1165 //          /* dR_(i,j,k)/dS_(i-1,j+1,k) at block (i,j,k),(i-1,j+1,k) */
1166 //          val[0][0]=0.0;
1167 //          val[0][1]=0.0;
1168 //          val[0][2]=0.0;
1169 //          val[0][3]=0.0;
1170 //          val[1][0]=0.0;
1171 //          val[1][1]=0.0;
1172 //          val[1][2]=0.0;
1173 //          val[1][3]=0.0;
1174 //          val[2][0]=0.0;
1175 //          val[2][1]=-0.25*vyxj2(i,j,k);
1176 //          val[2][2]=0.0;
1177 //          val[2][3]=0.0;
1178 //          val[3][0]=0.0;
1179 //          val[3][1]=0.0;
1180 //          val[3][2]=0.0;
1181 //          val[3][3]=0.0;
1182 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1183 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1184 //          PetscLogFlops(1+VYXJ2FLOPS);
1185 //        }
1186 //      else if (ReflectionRegion(pctx,i,j,k))
1187 //        {
1188 //          /* A reflecting ground condition */
1189 //          col.i=i; col.j=j; col.k=k; col.c=0;
1190 //          /* dR_(i,j,k)/dS_(i,j,k) at block (i,j,k),(i,j,k) */
1191 //          val[0][0]=0.0;
1192 //          val[0][1]=-kappa(i,j,k)*pctx->invdx;
1193 //          val[0][2]=-2*kappa(i,j,k)*pctx->invdy;
1194 //          val[0][3]=-kappa(i,j,k)*pctx->invdz;
1195 //          val[1][0]=bi2(i,j,k)*pctx->invdx;
1196 //          val[1][1]=-vxxi2(i,j,k);
1197 //          val[1][2]=0.0;
1198 //          val[1][3]=-0.25*vxzi2(i,j,k);
1199 //          val[2][0]=bj2(i,j,k)*pctx->invdy;
1200 //          val[2][1]=-0.25*vyxj2(i,j,k);
1201 //          val[2][2]=-vyyj2(i,j,k)+0.5*pctx->invdy*vyj2(i,j,k);
1202 //          val[2][3]=-0.25*vyzj2(i,j,k);
1203 //          val[3][0]=bk2(i,j,k)*pctx->invdz;
1204 //          val[3][1]=-0.25*vzxk2(i,j,k);
```

```
1205 //          val[3][2]=0.0;
1206 //          val[3][3]=-vzzk2(i,j,k);
1207 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1208 //          /* Eliminate zeros in diagonal of preconditioner */
1209 //          //for(d=0;d<dof;d++)
1210 //          //  if (val[d][d]==0) val[d][d]=1e-3;
1211 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1212 //          PetscLogFlops(4+
1213 //            3+(VYJ2FLOPS)+
1214 //            3+(BI2FLOPS+VXXI2FLOPS+VXZI2FLOPS)+
1215 //            3+(BJ2FLOPS+VYXJ2FLOPS+VYYJ2FLOPS+VYZJ2FLOPS)+
1216 //            3+(BK2FLOPS+VZXK2FLOPS+VZZK2FLOPS));
1217 //
1218 //          col.i=i+1; col.j=j; col.k=k; col.c=0;
1219 //          /* dR_(i,j,k)/dS_(i+1,j,k) at block (i,j,k),(i+1,j,k) */
1220 //          val[0][0]=-0.5*pctx->invdx*vx(i,j,k);
1221 //          val[0][1]=0.0;
1222 //          val[0][2]=0.0;
1223 //          val[0][3]=0.0;
1224 //          val[1][0]=-bi2(i,j,k)*pctx->invdx;
1225 //          val[1][1]=-0.5*vxi2(i,j,k)*pctx->invdx;
1226 //          val[1][2]=0.0;
1227 //          val[1][3]=-0.25*vxzi2(i,j,k);
1228 //          val[2][0]=0.0;
1229 //          val[2][1]=0.0;
1230 //          val[2][2]=-0.5*vxj2(i,j,k)*pctx->invdx;
1231 //          val[2][3]=0.0;
1232 //          val[3][0]=0.0;
1233 //          val[3][1]=0.0;
1234 //          val[3][2]=0.0;
1235 //          val[3][3]=-0.5*pctx->invdx*vxk2(i,j,k);
1236 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1237 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1238 //
1239 //          PetscLogFlops(2+(VXFLOPS)+
1240 //            5+(BI2FLOPS+VXI2FLOPS+VXZI2FLOPS)+
1241 //            2+(VXJ2FLOPS)+
1242 //            2+(VXK2FLOPS));
1243 //
1244 //          col.i=i-1; col.j=j; col.k=k; col.c=0;
1245 //          /* dR_(i,j,k)/dS_(i-1,j,k) at block (i,j,k),(i-1,j,k) */
1246 //          val[0][0]=+0.5*pctx->invdx*vx(i,j,k);
1247 //          val[0][1]=kappa(i,j,k)*pctx->invdx;
1248 //          val[0][2]=0.0;
1249 //          val[0][3]=0.0;
1250 //          val[1][0]=0.0;
1251 //          val[1][1]=+0.5*vxi2(i,j,k)*pctx->invdx;
1252 //          val[1][2]=0.0;
1253 //          val[1][3]=0.0;
1254 //          val[2][0]=0.0;
```

```
1255 //          val[2][1]=-0.25*vyxj2(i,j,k);
1256 //          val[2][2]=+0.5*vxj2(i,j,k)*pctx->invdx;
1257 //          val[2][3]=0.0;
1258 //          val[3][0]=0.0;
1259 //          val[3][1]=-0.25*vzxk2(i,j,k);
1260 //          val[3][2]=0.0;
1261 //          val[3][3]=0.5*pctx->invdx*vxk2(i,j,k);
1262 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1263 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1264 //          PetscLogFlops(3+(VXFLOPS)+
1265 //            2+(VXI2FLOPS)+
1266 //            3+(VYXJ2FLOPS+VXJ2FLOPS)+
1267 //            3+(VZXK2FLOPS+VXK2FLOPS));
1268 //
1269 //          col.i=i; col.j=j+1; col.k=k; col.c=0;
1270 //          /* dR_(i,j,k)/dS_(i,j+1,k) at block (i,j,k),(i,j+1,k) */
1271 //          val[0][0]=0.0;
1272 //          val[0][1]=0.0;
1273 //          val[0][2]=0.0;
1274 //          val[0][3]=0.0;
1275 //          val[1][0]=0.0;
1276 //          val[1][1]=0.0;
1277 //          val[1][2]=0.0;
1278 //          val[1][3]=0.0;
1279 //          val[2][0]=-pctx->invdy*bj2(i,j,k);
1280 //          val[2][1]=-0.25*vyxj2(i,j,k);
1281 //          val[2][2]=-0.5*pctx->invdy*vyj2(i,j,k);
1282 //          val[2][3]=-0.25*vyzj2(i,j,k);
1283 //          val[3][0]=0.0;
1284 //          val[3][1]=0.0;
1285 //          val[3][2]=0.0;
1286 //          val[3][3]=0.0;
1287 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1288 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1289 //          PetscLogFlops(5+(BJ2FLOPS+VYJ2FLOPS+VYXJ2FLOPS+VYZJ2FLOPS));
1290 //
1291 //          col.i=i; col.j=j; col.k=k+1; col.c=0;
1292 //          /* dR_(i,j,k)/dS_(i,j,k+1) at block (i,j,k),(i,j,k+1) */
1293 //          val[0][0]=-0.5*pctx->invdz*vz(i,j,k);
1294 //          val[0][1]=0.0;
1295 //          val[0][2]=0.0;
1296 //          val[0][3]=0.0;
1297 //          val[1][0]=0.0;
1298 //          val[1][1]=-0.5*pctx->invdz*vzi2(i,j,k);
1299 //          val[1][2]=0.0;
1300 //          val[1][3]=0.0;
1301 //          val[2][0]=0.0;
1302 //          val[2][1]=0.0;
1303 //          val[2][2]=-0.5*pctx->invdz*vzj2(i,j,k);
1304 //          val[2][3]=0.0;
```

```
1305 //          val[3][0]=-pctx->invdz*bk2(i,j,k);
1306 //          val[3][1]=-0.25*vzxk2(i,j,k);
1307 //          val[3][2]=0.0;
1308 //          val[3][3]=-0.5*pctx->invdz*vzk2(i,j,k);
1309 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1310 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1311 //          PetscLogFlops(2+(VZFLOPS)+
1312 //            5+(BK2FLOPS+VZK2FLOPS+VZXK2FLOPS)+
1313 //            2+(VZI2FLOPS)+
1314 //            2+(VZJ2FLOPS));
1315 //
1316 //          col.i=i; col.j=j; col.k=k-1; col.c=0;
1317 //          /* dR_(i,j,k)/dS_(i,j,k-1) at block (i,j,k),(i,j,k-1) */
1318 //          val[0][0]=0.5*pctx->invdz*vz(i,j,k);
1319 //          val[0][1]=0.0;
1320 //          val[0][2]=0.0;
1321 //          val[0][3]=pctx->invdz*kappa(i,j,k);
1322 //          val[1][0]=0.0;
1323 //          val[1][1]=0.5*pctx->invdz*vzi2(i,j,k);
1324 //          val[1][2]=0.0;
1325 //          val[1][3]=-0.25*vxzi2(i,j,k);
1326 //          val[2][0]=0.0;
1327 //          val[2][1]=0.0;
1328 //          val[2][2]=0.5*pctx->invdz*vzj2(i,j,k);
1329 //          val[2][3]=-0.25*vyzj2(i,j,k);
1330 //          val[3][0]=0.0;
1331 //          val[3][1]=0.0;
1332 //          val[3][2]=0.0;
1333 //          val[3][3]=0.5*pctx->invdz*vzk2(i,j,k);
1334 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1335 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1336 //          PetscLogFlops(3+(VZFLOPS)+
1337 //            2+(VZK2FLOPS)+
1338 //            3+(VXZI2FLOPS+VZI2FLOPS)+
1339 //            3+(VYZJ2FLOPS+VZJ2FLOPS));
1340 //
1341 //          col.i=i+1; col.j=j; col.k=k-1; col.c=0;
1342 //          /* dR_(i,j,k)/dS_(i+1,j,k-1) at block (i,j,k),(i+1,j,k-1) */
1343 //          val[0][0]=0.0;
1344 //          val[0][1]=0.0;
1345 //          val[0][2]=0.0;
1346 //          val[0][3]=0.0;
1347 //          val[1][0]=0.0;
1348 //          val[1][1]=0.0;
1349 //          val[1][2]=0.0;
1350 //          val[1][3]=-0.25*vxzi2(i,j,k);
1351 //          val[2][0]=0.0;
1352 //          val[2][1]=0.0;
1353 //          val[2][2]=0.0;
1354 //          val[2][3]=0.0;
```

```
1355 //          val[3][0]=0.0;
1356 //          val[3][1]=0.0;
1357 //          val[3][2]=0.0;
1358 //          val[3][3]=0.0;
1359 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1360 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1361 //          PetscLogFlops(1+VXZI2FLOPS);
1362 //
1363 //          col.i=i-1; col.j=j; col.k=k+1; col.c=0;
1364 //          /* dR_(i,j,k)/dS_(i-1,j,k+1) at block (i,j,k),(i-1,j,k+1) */
1365 //          val[0][0]=0.0;
1366 //          val[0][1]=0.0;
1367 //          val[0][2]=0.0;
1368 //          val[0][3]=0.0;
1369 //          val[1][0]=0.0;
1370 //          val[1][1]=0.0;
1371 //          val[1][2]=0.0;
1372 //          val[1][3]=0.0;
1373 //          val[2][0]=0.0;
1374 //          val[2][1]=0.0;
1375 //          val[2][2]=0.0;
1376 //          val[2][3]=0.0;
1377 //          val[3][0]=0.0;
1378 //          val[3][1]=-0.25*vzxk2(i,j,k);
1379 //          val[3][2]=0.0;
1380 //          val[3][3]=0.0;
1381 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1382 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1383 //          PetscLogFlops(1+VZXK2FLOPS);
1384 //
1385 //          col.i=i; col.j=j+1; col.k=k-1; col.c=0;
1386 //          /* dR_(i,j,k)/dS_(i,j+1,k-1) at block (i,j,k),(i,j+1,k-1) */
1387 //          val[0][0]=0.0;
1388 //          val[0][1]=0.0;
1389 //          val[0][2]=0.0;
1390 //          val[0][3]=0.0;
1391 //          val[1][0]=0.0;
1392 //          val[1][1]=0.0;
1393 //          val[1][2]=0.0;
1394 //          val[1][3]=0.0;
1395 //          val[2][0]=0.0;
1396 //          val[2][1]=0.0;
1397 //          val[2][2]=0.0;
1398 //          val[2][3]=-0.25*vyzj2(i,j,k);
1399 //          val[3][0]=0.0;
1400 //          val[3][1]=0.0;
1401 //          val[3][2]=0.0;
1402 //          val[3][3]=0.0;
1403 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1404 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
```

```
1405 //          PetscLogFlops(1+VYZJ2FLOPS);
1406 //
1407 //          col.i=i-1; col.j=j+1; col.k=k; col.c=0;
1408 //          /* dR_(i,j,k)/dS_(i-1,j+1,k) at block (i,j,k),(i-1,j+1,k) */
1409 //          val[0][0]=0.0;
1410 //          val[0][1]=0.0;
1411 //          val[0][2]=0.0;
1412 //          val[0][3]=0.0;
1413 //          val[1][0]=0.0;
1414 //          val[1][1]=0.0;
1415 //          val[1][2]=0.0;
1416 //          val[1][3]=0.0;
1417 //          val[2][0]=0.0;
1418 //          val[2][1]=-0.25*vyxj2(i,j,k);
1419 //          val[2][2]=0.0;
1420 //          val[2][3]=0.0;
1421 //          val[3][0]=0.0;
1422 //          val[3][1]=0.0;
1423 //          val[3][2]=0.0;
1424 //          val[3][3]=0.0;
1425 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1426 //          //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1427 //          PetscLogFlops(1+VYXJ2FLOPS);
1428 //        }
1429 //      else if (AbsorbingBCRegion(pctx,i,j,k))
1430 //        {
1431 //          /* ABC region, extreme limits of mesh */
1432 //          col.i=i; col.j=j; col.k=k; col.c=0;
1433 //          /* dR_(i,j,k)/dS_(i,j,k) at block (i,j,k),(i,j,k) */
1434 //          val[0][0]=0.0;
1435 //          if (ILowFace(pctx,i,j,k))
1436 //      val[0][1]=+pctx->invdx*kappae(i,j,k);
1437 //          else
1438 //      val[0][1]=-pctx->invdx*kappae(i,j,k);
1439 //          if (JLowFace(pctx,i,j,k))
1440 //      val[0][2]=+pctx->invdy*kappae(i,j,k);
1441 //          else
1442 //      val[0][2]=-pctx->invdy*kappae(i,j,k);
1443 //          if (KLowFace(pctx,i,j,k))
1444 //      val[0][3]=+pctx->invdz*kappae(i,j,k);
1445 //          else
1446 //      val[0][3]=-pctx->invdz*kappae(i,j,k);
1447 //          if (IHighFace(pctx,i,j,k))
1448 //      val[1][0]=-pctx->invdx*bei2(i,j,k);
1449 //          else
1450 //      val[1][0]=+pctx->invdx*bei2(i,j,k);
1451 //          val[1][1]=-sei2(i,j,k);
1452 //          val[1][2]=0.0;
1453 //          val[1][3]=0.0;
1454 //          if (JHighFace(pctx,i,j,k))
```

```
1455 //      val[2][0]=-pctx->invdy*bej2(i,j,k);
1456 //           else
1457 //      val[2][0]=+pctx->invdy*bej2(i,j,k);
1458 //           val[2][1]=0.0;
1459 //           val[2][2]=-sej2(i,j,k);
1460 //           val[2][3]=0.0;
1461 //           if (KHighFace(pctx,i,j,k))
1462 //      val[3][0]=-pctx->invdz*bek2(i,j,k);
1463 //           else
1464 //      val[3][0]=+pctx->invdz*bek2(i,j,k);
1465 //           val[3][1]=0.0;
1466 //           val[3][2]=0.0;
1467 //           val[3][3]=-sek2(i,j,k);
1468 //           MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1469 //           /* Eliminate zeros in diagonal of preconditioner */
1470 //           //for(d=0;d<dof;d++)
1471 //           //   if (val[d][d]==0) val[d][d]=1e-3;
1472 //           //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1473 //           PetscLogFlops(3+
1474 //             1+(BI2FLOPS+SEI2FLOPS)+
1475 //             1+(BJ2FLOPS+SEJ2FLOPS)+
1476 //             1+(BK2FLOPS+SEK2FLOPS));
1477 //
1478 //           if (!IHighFace(pctx,i,j,k))
1479 //      {
1480 //        col.i=i+1; col.j=j; col.k=k; col.c=0;
1481 //        /* dR_(i,j,k)/dS_(i+1,j,k) at block (i,j,k),(i+1,j,k) */
1482 //        val[0][0]=0.0;
1483 //        if (ILowFace(pctx,i,j,k))
1484 //          val[0][1]=-pctx->invdx*kappae(i,j,k);
1485 //        else
1486 //          val[0][1]=0.0;
1487 //        val[0][2]=0.0;
1488 //        val[0][3]=0.0;
1489 //        val[1][0]=-pctx->invdx*bei2(i,j,k);
1490 //        val[1][1]=0.0;
1491 //        val[1][2]=0.0;
1492 //        val[1][3]=0.0;
1493 //        val[2][0]=0.0;
1494 //        val[2][1]=0.0;
1495 //        val[2][2]=0.0;
1496 //        val[2][3]=0.0;
1497 //        val[3][0]=0.0;
1498 //        val[3][1]=0.0;
1499 //        val[3][2]=0.0;
1500 //        val[3][3]=0.0;
1501 //        MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1502 //        //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1503 //        PetscLogFlops(1+BI2FLOPS);
1504 //      }
```

```
1505 //
1506 //          if (!ILowFace(pctx,i,j,k))
1507 //      {
1508 //        col.i=i-1; col.j=j; col.k=k; col.c=0;
1509 //        /* dR_(i,j,k)/dS_(i-1,j,k) at block (i,j,k),(i-1,j,k) */
1510 //        val[0][0]=0.0;
1511 //        val[0][1]=+pctx->invdx*kappae(i,j,k);
1512 //        val[0][2]=0.0;
1513 //        val[0][3]=0.0;
1514 //        if (IHighFace(pctx,i,j,k))
1515 //          val[1][0]=-pctx->invdx*bei2(i,j,k);
1516 //        else
1517 //          val[1][0]=0.0;
1518 //        val[1][1]=0.0;
1519 //        val[1][2]=0.0;
1520 //        val[1][3]=0.0;
1521 //        val[2][0]=0.0;
1522 //        val[2][1]=0.0;
1523 //        val[2][2]=0.0;
1524 //        val[2][3]=0.0;
1525 //        val[3][0]=0.0;
1526 //        val[3][1]=0.0;
1527 //        val[3][2]=0.0;
1528 //        val[3][3]=0.0;
1529 //        MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1530 //        //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1531 //        PetscLogFlops(1);
1532 //      }
1533 //
1534 //          if (!JHighFace(pctx,i,j,k))
1535 //      {
1536 //        col.i=i; col.j=j+1; col.k=k; col.c=0;
1537 //        /* dR_(i,j,k)/dS_(i,j+1,k) at block (i,j,k),(i,j+1,k) */
1538 //        val[0][0]=0.0;
1539 //        val[0][1]=0.0;
1540 //        if (JLowFace(pctx,i,j,k))
1541 //          val[0][2]=-pctx->invdy*kappae(i,j,k);
1542 //        else
1543 //          val[0][2]=0.0;
1544 //        val[0][3]=0.0;
1545 //        val[1][0]=0.0;
1546 //        val[1][1]=0.0;
1547 //        val[1][2]=0.0;
1548 //        val[1][3]=0.0;
1549 //        val[2][0]=-pctx->invdy*bej2(i,j,k);
1550 //        val[2][1]=0.0;
1551 //        val[2][2]=0.0;
1552 //        val[2][3]=0.0;
1553 //        val[3][0]=0.0;
1554 //        val[3][1]=0.0;
```

```
1555 //        val[3][2]=0.0;
1556 //        val[3][3]=0.0;
1557 //        MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1558 //        //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1559 //        PetscLogFlops(1+BJ2FLOPS);
1560 //      }
1561 //
1562 //          if (!JLowFace(pctx,i,j,k))
1563 //      {
1564 //        col.i=i; col.j=j-1; col.k=k; col.c=0;
1565 //        /* dR_(i,j,k)/dS_(i,j-1,k) at block (i,j,k),(i,j-1,k) */
1566 //        val[0][0]=0.0;
1567 //        val[0][1]=0.0;
1568 //        val[0][2]=+pctx->invdy*kappae(i,j,k);
1569 //        val[0][3]=0.0;
1570 //        val[1][0]=0.0;
1571 //        val[1][1]=0.0;
1572 //        val[1][2]=0.0;
1573 //        val[1][3]=0.0;
1574 //        if (JHighFace(pctx,i,j,k))
1575 //          val[2][0]=-pctx->invdy*bej2(i,j,k);
1576 //        else
1577 //          val[2][0]=0.0;
1578 //        val[2][1]=0.0;
1579 //        val[2][2]=0.0;
1580 //        val[2][3]=0.0;
1581 //        val[3][0]=0.0;
1582 //        val[3][1]=0.0;
1583 //        val[3][2]=0.0;
1584 //        val[3][3]=0.0;
1585 //        MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1586 //        //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1587 //        PetscLogFlops(1);
1588 //      }
1589 //
1590 //          if (!KHighFace(pctx,i,j,k))
1591 //      {
1592 //        col.i=i; col.j=j; col.k=k+1; col.c=0;
1593 //        /* dR_(i,j,k)/dS_(i,j,k+1) at block (i,j,k),(i,j,k+1) */
1594 //        val[0][0]=0.0;
1595 //        val[0][1]=0.0;
1596 //        val[0][2]=0.0;
1597 //        if (KLowFace(pctx,i,j,k))
1598 //          val[0][3]=-pctx->invdz*kappae(i,j,k);
1599 //        else
1600 //          val[0][3]=0.0;
1601 //        val[1][0]=0.0;
1602 //        val[1][1]=0.0;
1603 //        val[1][2]=0.0;
1604 //        val[1][3]=0.0;
```

```
1605 //         val[2][0]=0.0;
1606 //         val[2][1]=0.0;
1607 //         val[2][2]=0.0;
1608 //         val[2][3]=0.0;
1609 //         val[3][0]=-pctx->invdz*bek2(i,j,k);
1610 //         val[3][1]=0.0;
1611 //         val[3][2]=0.0;
1612 //         val[3][3]=0.0;
1613 //         MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1614 //         //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1615 //         PetscLogFlops(1+BK2FLOPS);
1616 //       }
1617 //
1618 //           if (!KLowFace(pctx,i,j,k))
1619 //       {
1620 //         col.i=i; col.j=j; col.k=k-1; col.c=0;
1621 //         /* dR_(i,j,k)/dS_(i,j,k-1) at block (i,j,k),(i,j,k-1) */
1622 //         val[0][0]=0.0;
1623 //         val[0][1]=0.0;
1624 //         val[0][2]=0.0;
1625 //         val[0][3]=+pctx->invdz*kappae(i,j,k);
1626 //         val[1][0]=0.0;
1627 //         val[1][1]=0.0;
1628 //         val[1][2]=0.0;
1629 //         val[1][3]=0.0;
1630 //         val[2][0]=0.0;
1631 //         val[2][1]=0.0;
1632 //         val[2][2]=0.0;
1633 //         val[2][3]=0.0;
1634 //         if (KHighFace(pctx,i,j,k))
1635 //           val[3][0]=-pctx->invdz*bek2(i,j,k);
1636 //         else
1637 //           val[3][0]=0.0;
1638 //         val[3][1]=0.0;
1639 //         val[3][2]=0.0;
1640 //         val[3][3]=0.0;
1641 //         MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1642 //         //MatSetValuesBlockedStencil(*pPC,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1643 //         PetscLogFlops(1);
1644 //       }
1645 //         }
1646 //       else
1647 //         {
1648 //           col.i=i; col.j=j; col.k=k; col.c=0;
1649 //           /* dR_(i,j,k)/dS_(i,j,k) at block (i,j,k),(i,j,k) */
1650 //           val[0][0]=1.0;
1651 //           val[0][1]=0.0;
1652 //           val[0][2]=0.0;
1653 //           val[0][3]=0.0;
1654 //           val[1][0]=0.0;
```

```
1655 //          val[1][1]=1.0;
1656 //          val[1][2]=0.0;
1657 //          val[1][3]=0.0;
1658 //          val[2][0]=0.0;
1659 //          val[2][1]=0.0;
1660 //          val[2][2]=1.0;
1661 //          val[2][3]=0.0;
1662 //          val[3][0]=0.0;
1663 //          val[3][1]=0.0;
1664 //          val[3][2]=0.0;
1665 //          val[3][3]=1.0;
1666 //          MatSetValuesBlockedStencil(*pJ,1,&row,1,&col,&val[0][0],INSERT_VALUES);
1667 //        }
1668 //   }
1669 //    MatAssemblyBegin(*pJ,MAT_FINAL_ASSEMBLY);
1670 //    MatAssemblyEnd(*pJ,MAT_FINAL_ASSEMBLY);
1671 //    //   MatAssemblyBegin(*pPC,MAT_FINAL_ASSEMBLY);
1672 //    //   MatAssemblyEnd(*pPC,MAT_FINAL_ASSEMBLY);m, m,
1673 //
1674 //    DAVecRestoreArrayDOF(pctx->da,local,&s);
1675 //    /* Return local vector representation to DA */
1676 //    DARestoreLocalVector(pctx->da,&local);
1677 //
1678 //    /* The non-zero pattern of these matrices is unchanged from timestep to timestep */
1679 //    (*flag)=SAME_NONZERO_PATTERN;
1680 //
1681 //    /* Set a flag to through an error if the nonzero pattern does change */
1682 //    MatSetOption(*pJ,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_TRUE);
1683 //    // MatSetOption(*pPC,MAT_NEW_NONZERO_LOCATION_ERR,PETSC_TRUE);
1684 //
1685 //    PetscLogEventEnd(pctx->FormJacobianEvent,0,0,0,0);
1686 //
1687 //    return 0;
1688 // }
1689 //
1690
1691 PetscTruth InteriorRegion(app_ctx *pctx,
1692         PetscInt i,
1693         PetscInt j,
1694         PetscInt k)
1695 {
1696   if (i>pctx->i_lo && i<pctx->i_hi &&
1697       j>pctx->j_lo && j<pctx->j_hi &&
1698       (TwoDimensional(pctx) || (k>pctx->k_lo && k<pctx->k_hi)))
1699     return PETSC_TRUE;
1700   return PETSC_FALSE;
1701 }
1702
1703 PetscTruth ReflectionRegion(app_ctx *pctx,
1704           PetscInt i,
```

```
1705            PetscInt j,
1706            PetscInt k)
1707 {
1708   /* If statements should always check for InteriorRegion first before ReflectionRegion
          */
1709   bc_parameters  *bc=(bc_parameters*)pctx->pbc;
1710   assert(bc);
1711   if (bc->ReflectiveGround &&
1712       j==pctx->j_lo &&
1713       (i>=0 && i<=pctx->params->mesh.Nx-1) &&
1714       (TwoDimensional(pctx) || (k>=0 && k<=pctx->params->mesh.Nz-1)))
1715     return PETSC_TRUE;
1716   return PETSC_FALSE;
1717 }
1718
1719 PetscTruth AbsorbingBCRegion(app_ctx *pctx,
1720            PetscInt i,
1721            PetscInt j,
1722            PetscInt k)
1723 {
1724   /* If statements should always check for ReflectionRegion first before
          AbsorbingBCRegion */
1725   if ((i>=0 && i<=pctx->i_lo) ||
1726       (i>=pctx->i_hi && i<=pctx->params->mesh.Nx-1) ||
1727       (j>=0 && j<=pctx->j_lo) ||
1728       (j>=pctx->j_hi && j<=pctx->params->mesh.Ny-1) ||
1729       (!TwoDimensional(pctx) && k>=0 && k<=pctx->k_lo) ||
1730       (!TwoDimensional(pctx) && k>=pctx->k_hi && k<=pctx->params->mesh.Nz-1))
1731     return PETSC_TRUE;
1732   return PETSC_FALSE;
1733 }
1734
1735 PetscTruth TwoDimensional(app_ctx *pctx)
1736 {
1737   if (pctx->params->mesh.Nz==1) return PETSC_TRUE;
1738   return PETSC_FALSE;
1739 }
1740
1741 PetscTruth ILowFace(app_ctx *pctx,
1742         PetscInt i,
1743         PetscInt j,
1744         PetscInt k)
1745 {
1746   if (i==0) return PETSC_TRUE;
1747   return PETSC_FALSE;
1748 }
1749
1750 PetscTruth JLowFace(app_ctx *pctx,
1751         PetscInt i,
1752         PetscInt j,
```

```
1753          PetscInt k)
1754 {
1755   if (j==0) return PETSC_TRUE;
1756   return PETSC_FALSE;
1757 }
1758
1759 PetscTruth KLowFace(app_ctx *pctx,
1760          PetscInt i,
1761          PetscInt j,
1762          PetscInt k)
1763 {
1764   if (k==0) return PETSC_TRUE;
1765   return PETSC_FALSE;
1766 }
1767
1768 PetscTruth IHighFace(app_ctx *pctx,
1769          PetscInt i,
1770          PetscInt j,
1771          PetscInt k)
1772 {
1773   if (i==pctx->params->mesh.Nx-1) return PETSC_TRUE;
1774   return PETSC_FALSE;
1775 }
1776
1777 PetscTruth JHighFace(app_ctx *pctx,
1778          PetscInt i,
1779          PetscInt j,
1780          PetscInt k)
1781 {
1782   if (j==pctx->params->mesh.Ny-1) return PETSC_TRUE;
1783   return PETSC_FALSE;
1784 }
1785
1786 PetscTruth KHighFace(app_ctx *pctx,
1787          PetscInt i,
1788          PetscInt j,
1789          PetscInt k)
1790 {
1791   if (k==pctx->params->mesh.Nz-1) return PETSC_TRUE;
1792   return PETSC_FALSE;
1793 }
```

3.10 param.h

```
1 #ifndef _PARAM_H_
2 #define _PARAM_H_
3 #include<petsc.h>
4 #include<petscbag.h>
5
```

```
6
7  /*! \file param.h
8      \brief Run-time parameters header file.
9
10     This header file defines a number of parameters which control the behaviour of the
           solver.
11 */
12
13 #include"source.h"
14 #include"mean.h"
15 #include"bc.h"
16
17 /*! \brief Time march parameters
18
19   Parameters related to time marching methods. */
20 typedef struct {
21   PetscReal t0;          /*!< Initial time */
22   PetscReal dt;          /*!< Time step interval */
23   PetscReal tmax;        /*!< Final time */
24   PetscReal T;           /*!< RMS average interval */
25   PetscReal courant;     /*!< Courant number */
26   PetscInt maxsteps;     /*!< Maximum number of time steps */
27   PetscInt savefreq;     /*!< Checkpoint save interval */
28 } timemarch_parameters;
29
30 /*! \brief Mesh parameters
31
32   Parameters related to mesh dimensions. */
33 typedef struct {
34   PetscInt Nx;           /*!< Mesh dimension in x direction */
35   PetscInt Ny;           /*!< Mesh dimension in y direction */
36   PetscInt Nz;           /*!< Mesh dimension in z direction */
37   PetscReal xmax;        /*!< Maximum space dimension in x direction */
38   PetscReal xmin;        /*!< Minimum space dimension in x direction */
39   PetscReal xlayer;      /*!< Thickness of ABC region perpendicular to x */
40   PetscReal ymax;        /*!< Maximum space dimension in y direction */
41   PetscReal ymin;        /*!< Minimum space dimension in y direction */
42   PetscReal ylayer;      /*!< Thickness of ABC region perpendicular to y */
43   PetscReal zmax;        /*!< Maximum space dimension in z direction */
44   PetscReal zmin;        /*!< Minimum space dimension in z direction */
45   PetscReal zlayer;      /*!< Thickness of ABC region perpendicular to z */
46   PetscInt NodesPerWave;  /*!< Desired number of mesh nodes per wavelength */
47   PetscInt NumWavesInLayer; /*!< Desired number of wavelengths within ABC region */
48 } mesh_parameters;
49
50 /*! \brief Global parameters
51
52   Collection of global run-time parameters parameters. */
53 typedef struct {
54   mesh_parameters mesh;     /*!< Collection of mesh parameters */
```

```
55   timemarch_parameters tm; /*!< Collection of timemarching parameters */
56   mean_flow_type mean;       /*!< Mean flow type specifier */
57   source_type source;       /*!< Source type specifier */
58 } parameters;
59
60 /*! Register parameters using PetscBag, optionally loading values from existing file */
61 PetscErrorCode RegisterParameterBag(PetscBag *bag,parameters **params,PetscTruth
       LoadFromFile);
62
63 /*! Save parameter values to file. */
64 void SaveParameterBag(PetscBag bag);
65
66 /*! Check validity of current parameters */
67 PetscTruth ParametersAreIncorrect(parameters *params);
68
69 #endif
```

3.11 param.c

```
1 #include<assert.h>
2 #include<stdlib.h>
3 #include<petsc.h>
4 #include"param.h"
5
6 const char *MeanChoices[]={"mean_uniform","power_law","Mean Flow Choices","",0};
7 const char *SourceChoices[]={"none","mono_ping","monopole","dipole","monopole_series","
      dipole_series","Source Choices","",0};
8 const char ParametersFile[]="parameters.dat";
9
10 PetscErrorCode RegisterParameterBag(PetscBag *bag,parameters **params,PetscTruth
       LoadFromFile)
11 {
12   PetscErrorCode ierr;
13   assert(bag);
14   assert(params);
15
16   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(parameters),bag); CHKERRQ(ierr);
17   ierr=PetscBagGetData(*bag,(void **)params); CHKERRQ(ierr);
18
19   ierr=PetscBagSetName(*bag,"Parameters","runtime parameters for the acoustic solver");
20   /* Mesh parameters */
21   ierr=PetscBagRegisterInt(*bag,&(*params)->mesh.Nx,11,"N_x","Number of mesh nodes in x-
       direction"); CHKERRQ(ierr);
22   ierr=PetscBagRegisterInt(*bag,&(*params)->mesh.Nx,11,"N_x","Number of mesh nodes in x-
       direction"); CHKERRQ(ierr);
23   ierr=PetscBagRegisterInt(*bag,&(*params)->mesh.Ny,11,"N_y","Number of mesh nodes in y-
       direction"); CHKERRQ(ierr);
24   ierr=PetscBagRegisterInt(*bag,&(*params)->mesh.Nz,11,"N_z","Number of mesh nodes in z-
       direction"); CHKERRQ(ierr);
```

```
25  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.xmin,-100,"x_min","Lower x-direction
        limit on flow region"); CHKERRQ(ierr);
26  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.ymin,-100,"y_min","Lower y-direction
        limit on flow region"); CHKERRQ(ierr);
27  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.zmin,-100,"z_min","Lower z-direction
        limit on flow region"); CHKERRQ(ierr);
28  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.xmax,+100,"x_max","Upper x-direction
        limit on flow region"); CHKERRQ(ierr);
29  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.ymax,+100,"y_max","Upper y-direction
        limit on flow region"); CHKERRQ(ierr);
30  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.zmax,+100,"z_max","Upper z-direction
        limit on flow region"); CHKERRQ(ierr);
31  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.xlayer,0,"x_layer","ABC layer
        thickness in x-direction"); CHKERRQ(ierr);
32  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.ylayer,0,"y_layer","ABC layer
        thickness in y-direction"); CHKERRQ(ierr);
33  ierr=PetscBagRegisterReal(*bag,&(*params)->mesh.zlayer,0,"z_layer","ABC layer
        thickness in z-direction"); CHKERRQ(ierr);
34
35  /* Time march parameters */
36  ierr=PetscBagRegisterReal(*bag,&(*params)->tm.dt,1,"dt","Time step interval (s), if
        using fixed time step"); CHKERRQ(ierr);
37  ierr=PetscBagRegisterReal(*bag,&(*params)->tm.t0,0,"t_0","Initial time (s)"); CHKERRQ(
        ierr);
38  ierr=PetscBagRegisterReal(*bag,&(*params)->tm.tmax,1,"t_max","Final time (s)");
        CHKERRQ(ierr);
39  ierr=PetscBagRegisterReal(*bag,&(*params)->tm.T,0.1,"T","RMS average period of sound
        pressure (s)"); CHKERRQ(ierr);
40  ierr=PetscBagRegisterReal(*bag,&(*params)->tm.courant,0.5,"courant","Courant number");
        CHKERRQ(ierr);
41  ierr=PetscBagRegisterInt(*bag,&(*params)->tm.maxsteps,100000,"max_steps","Maximum
        number of time steps"); CHKERRQ(ierr);
42  ierr=PetscBagRegisterInt(*bag,&(*params)->tm.savefreq,0,"save_freq","Timestep
        frequency of checkpoint save"); CHKERRQ(ierr);
43
44  /* Mean flow type */
45  ierr=PetscBagRegisterEnum(*bag,&(*params)->mean,MeanChoices,(PetscEnum)MEAN_UNIFORM,"
        mean","Choose mean flow method"); CHKERRQ(ierr);
46  /* Source type */
47  ierr=PetscBagRegisterEnum(*bag,&(*params)->source,SourceChoices,(PetscEnum)
        SOURCE_MONO_PING,"source","Choose source method"); CHKERRQ(ierr);
48
49  (*params)->mesh.NodesPerWave=1;
50  (*params)->mesh.NumWavesInLayer=0;
51
52  if (LoadFromFile)
53    {
54      PetscViewer bagviewer;
55      if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,ParametersFile,FILE_MODE_READ,&
            bagviewer))
```

```
56   {
57     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",ParametersFile);
58     CHKERRQ(ierr);
59   }
60       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
61       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
62       ierr=PetscBagGetData(*bag,(void**)params); CHKERRQ(ierr);
63       ierr=PetscPrintf(PETSC_COMM_WORLD,"Loaded parameters:\n"); CHKERRQ(ierr);
64       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
65     }
66   return 0;
67 }
68
69 void SaveParameterBag(PetscBag bag)
70 {
71   PetscViewer bagviewer;
72   /* Save the runtime parameters to a binary file */
73   PetscViewerBinaryOpen(PETSC_COMM_WORLD,ParametersFile,FILE_MODE_WRITE,&bagviewer);
74   PetscBagView(bag,bagviewer);
75   PetscViewerDestroy(bagviewer);
76 }
77
78 PetscTruth ParametersAreIncorrect(parameters *params)
79 {
80   unsigned long MaxIndex=1<<(sizeof(PetscInt)*8-1);
81   unsigned long NumUnknowns=4*params->mesh.Nx*params->mesh.Ny*params->mesh.Nz;
82   assert(params);
83
84   if (params->mesh.Nx<3 || params->mesh.Ny<3 || params->mesh.Nz<1)
85     {
86       PetscPrintf(PETSC_COMM_WORLD,"Mesh dimensions must be greater than 3 in at least
              the x and y dimension.\n");
87       return PETSC_TRUE;
88     }
89
90   if (params->tm.courant<=0)
91     {
92       PetscPrintf(PETSC_COMM_WORLD,"Courant number should be >0, e.g. 0.85\n");
93       return PETSC_TRUE;
94     }
95
96   if (NumUnknowns>MaxIndex)
97     {
98       PetscPrintf(PETSC_COMM_WORLD,"Problem size will exceed maximum number of unknowns
              available on this system (size of PetscInt is %d)\n",sizeof(PetscInt));
99       return PETSC_TRUE;
100    }
101
102  if (params->mesh.xmax<params->mesh.xmin ||
103      params->mesh.ymax<params->mesh.ymin ||
```

```
104       (params->mesh.Nz>1 && params->mesh.zmax<params->mesh.zmin))
105     {
106       PetscPrintf(PETSC_COMM_WORLD,"Flow domain must span a non-zero interval in each
              dimension.\n");
107       PetscPrintf(PETSC_COMM_WORLD," xmin: %g m to xmax: %g m\n",params->mesh.xmin,
              params->mesh.xmax);
108       PetscPrintf(PETSC_COMM_WORLD," ymin: %g m to ymax: %g m\n",params->mesh.ymin,
              params->mesh.ymax);
109       PetscPrintf(PETSC_COMM_WORLD," zmin: %g m to zmax: %g m\n",params->mesh.zmin,
              params->mesh.zmax);
110       return PETSC_TRUE;
111     }
112
113   if (params->mesh.xlayer*2>(params->mesh.xmax-params->mesh.xmin) ||
114       params->mesh.ylayer*2>(params->mesh.ymax-params->mesh.ymin) ||
115       params->mesh.zlayer*2>(params->mesh.zmax-params->mesh.zmin) ||
116       params->mesh.xmin+params->mesh.xlayer>params->mesh.xmax-params->mesh.xlayer ||
117       params->mesh.ymin+params->mesh.ylayer>params->mesh.ymax-params->mesh.ylayer ||
118       params->mesh.zmin+params->mesh.zlayer>params->mesh.zmax-params->mesh.zlayer)
119     {
120       PetscPrintf(PETSC_COMM_WORLD,"Flow domain must span at least 2x the requested
              boundary condition layers.\n");
121       PetscPrintf(PETSC_COMM_WORLD," xmin: %g m to xmax: %g m\n",params->mesh.xmin,
              params->mesh.xmax);
122       PetscPrintf(PETSC_COMM_WORLD," ymin: %g m to ymax: %g m\n",params->mesh.ymin,
              params->mesh.ymax);
123       PetscPrintf(PETSC_COMM_WORLD," zmin: %g m to zmax: %g m\n",params->mesh.zmin,
              params->mesh.zmax);
124       PetscPrintf(PETSC_COMM_WORLD," xlayer: %g m\n",params->mesh.xlayer);
125       PetscPrintf(PETSC_COMM_WORLD," ylayer: %g m\n",params->mesh.ylayer);
126       PetscPrintf(PETSC_COMM_WORLD," zlayer: %g m\n",params->mesh.zlayer);
127       return PETSC_TRUE;
128     }
129
130   return PETSC_FALSE;
131 }
```

3.12 setup.h

```
1 #ifndef _SETUP_H_
2 #define _SETUP_H_
3
4 #include"ode.h"
5
6 /*! \file setup.h
7     \brief Setup for data structures
8
9 Declares functions for allocating and deallocating data structures based upon run-time
      parameters.
```

```
10 */
11
12 /*! Create the necessary data structures to time march the solution */
13 PetscErrorCode CreateStorage(Vec *psol,
14            app_ctx *pctx);
15
16 /*! Destroy the necessary data structures to time march the solution */
17 PetscErrorCode DestroyStorage(Vec psol,
18             app_ctx *context);
19
20 #endif
```

3.13 setup.c

```
1 #include<assert.h>
2 #include<petsc.h>
3
4 #include"ode.h"
5 #include"setup.h"
6 #include"math.h"
7 #include"io.h"
8
9 PetscErrorCode CreateStorage(Vec *psolution,
10            app_ctx *pctx)
11 {
12   const int dof=4;
13   const int stencilwidth=1;
14   DAStencilType stenciltype=DA_STENCIL_BOX;
15   PetscInt sx,sy,sz,mx,my,mz;
16   PetscInt low,high,globallen,locallen;
17   PetscInt mlow,mhigh,mM,mN;
18   int rank,size;
19   ISColoring iscoloring;
20   parameters *params=pctx->params;
21   PetscErrorCode ierr;
22
23   MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
24   MPI_Comm_size(PETSC_COMM_WORLD,&size);
25
26   assert(pctx);
27   assert(params);
28   assert(psolution);
29
30   PetscLogEventBegin(pctx->SetupEvent,0,0,0,0);
31   /* Vectors for intermediate calculations are stored under the app_ctx type.
32      These vectors are created once at the start of the code and destroyed at
33      the end of the program.
34   */
35
```

```
36  if (pctx->psource && pctx->GetSourceLowestFrequency)
37    {
38      assert(pctx->GetSourceLowestFrequency);
39
40      PetscReal flo=pctx->GetSourceLowestFrequency(pctx->psource);
41      PetscReal fhi=pctx->GetSourceHighestFrequency(pctx->psource);
42
43      assert(pctx->properties);
44      PetscReal c=GetSpeedOfSound(pctx->properties);
45
46      PetscReal lambda_max=c/flo;
47      PetscPrintf(PETSC_COMM_WORLD,"Longest wavelength: %g m\n",lambda_max);
48
49      PetscReal lambda_min=c/fhi;
50      PetscPrintf(PETSC_COMM_WORLD,"Shortest wavelength: %g m\n",lambda_min);
51
52      if (pctx->params->mesh.NumWavesInLayer>0)
53  {
54    PetscReal len=((PetscReal)pctx->params->mesh.NumWavesInLayer)*lambda_max;
55    PetscPrintf(PETSC_COMM_WORLD,"Automatic BC layer dimension using wave length of %g m
          .\n",lambda_max);
56    params->mesh.xlayer=len;
57    params->mesh.ylayer=len;
58    params->mesh.zlayer=len;
59  }
60
61      if (pctx->params->mesh.NodesPerWave>1)
62  {
63    PetscReal delta=lambda_min/((PetscReal)(pctx->params->mesh.NodesPerWave-1));
64    PetscPrintf(PETSC_COMM_WORLD,"Automatic domain spanning using wave length of %g m.\n
          ",lambda_min);
65    params->mesh.xmin=-0.5*delta*params->mesh.Nx;  params->mesh.xmax=+0.5*delta*params->
          mesh.Nx;
66    bc_parameters *pbc=(bc_parameters*)(pctx->pbc);
67    PetscReal r0=0.5;
68    if (params->mesh.ylayer>0 && !pbc->NoGround)
69      r0=params->mesh.ylayer/(delta*params->mesh.Ny);
70    params->mesh.ymin=-r0*delta*params->mesh.Ny;  params->mesh.ymax=+(1-r0)*delta*params
          ->mesh.Ny;
71    if (!TwoDimensional(pctx))
72      params->mesh.zmin=-0.5*delta*params->mesh.Nz;  params->mesh.zmax=+0.5*delta*params
            ->mesh.Nz;
73  }
74
75      if (params->mesh.xlayer*2>(params->mesh.xmax-params->mesh.xmin) ||
76    params->mesh.ylayer*2>(params->mesh.ymax-params->mesh.ymin) ||
77    params->mesh.zlayer*2>(params->mesh.zmax-params->mesh.zmin) ||
78    params->mesh.xmin+params->mesh.xlayer>params->mesh.xmax-params->mesh.xlayer ||
79    params->mesh.ymin+params->mesh.ylayer>params->mesh.ymax-params->mesh.ylayer ||
80    params->mesh.zmin+params->mesh.zlayer>params->mesh.zmax-params->mesh.zlayer)
```

```
81  {
82    PetscPrintf(PETSC_COMM_WORLD,"Flow domain must span at least 2x the requested
          boundary condition layers.\n");
83    PetscPrintf(PETSC_COMM_WORLD," xmin: %g m to xmax: %g m\n",params->mesh.xmin,params
          ->mesh.xmax);
84    PetscPrintf(PETSC_COMM_WORLD," ymin: %g m to ymax: %g m\n",params->mesh.ymin,params
          ->mesh.ymax);
85    PetscPrintf(PETSC_COMM_WORLD," zmin: %g m to zmax: %g m\n",params->mesh.zmin,params
          ->mesh.zmax);
86    PetscPrintf(PETSC_COMM_WORLD," xlayer: %g m\n",params->mesh.xlayer);
87    PetscPrintf(PETSC_COMM_WORLD," ylayer: %g m\n",params->mesh.ylayer);
88    PetscPrintf(PETSC_COMM_WORLD," zlayer: %g m\n",params->mesh.zlayer);
89    return 1;
90  }
91
92    }
93
94  if (TwoDimensional(pctx))
95    {
96      PetscPrintf(PETSC_COMM_WORLD,"Resetting z min/max for 2D calculation.\n");
97      params->mesh.zmin = 0.0;
98      params->mesh.zmax = 1.0;
99      params->mesh.zlayer = 0.0;
100   }
101
102 /* Note that a distributed array represents the mesh distributed across multiple
        processors.
103  */
104 pctx->invdx=(params->mesh.Nx-1)/(params->mesh.xmax-params->mesh.xmin);
105 pctx->invdy=(params->mesh.Ny-1)/(params->mesh.ymax-params->mesh.ymin);
106 if (!TwoDimensional(pctx))
107   pctx->invdz=(params->mesh.Nz-1)/(params->mesh.zmax-params->mesh.zmin);
108 else
109   pctx->invdz=1.0;
110
111 pctx->dx=(params->mesh.xmax-params->mesh.xmin)/(params->mesh.Nx-1);
112 pctx->dy=(params->mesh.ymax-params->mesh.ymin)/(params->mesh.Ny-1);
113 if (!TwoDimensional(pctx))
114   pctx->dz=(params->mesh.zmax-params->mesh.zmin)/(params->mesh.Nz-1);
115 else
116   pctx->dz=1.0;
117
118 pctx->i_lo=0;
119 pctx->j_lo=0;
120 pctx->k_lo=0;
121 pctx->i_hi=params->mesh.Nx-1;
122 pctx->j_hi=params->mesh.Ny-1;
123 pctx->k_hi=params->mesh.Nz-1;
124
125 if (params->mesh.xlayer>0)
```

```
126     {
127       pctx->i_lo=(int)(params->mesh.xlayer/
128             (params->mesh.xmax-params->mesh.xmin)*
129             ((double)params->mesh.Nx));
130       pctx->i_hi=params->mesh.Nx-1-pctx->i_lo;
131     }
132
133   if (params->mesh.ylayer>0)
134     {
135       pctx->j_lo=(int)(params->mesh.ylayer/
136             (params->mesh.ymax-params->mesh.ymin)*
137             ((double)params->mesh.Ny));
138       pctx->j_hi=params->mesh.Ny-1-pctx->j_lo;
139       bc_parameters  *bc=(bc_parameters*)pctx->pbc;
140       assert(bc);
141       if (bc->ReflectiveGround) pctx->j_lo=0;
142     }
143
144   if (!TwoDimensional(pctx) && params->mesh.zlayer>0)
145     {
146       pctx->k_lo=(int)(params->mesh.zlayer/
147             (params->mesh.zmax-params->mesh.zmin)*
148             ((double)params->mesh.Nz));
149       pctx->k_hi=params->mesh.Nz-1-pctx->k_lo;
150     }
151
152   assert(pctx->params->tm.dt);
153   if (pctx->params->tm.dt<0) pctx->params->tm.dt=0;
154
155   PetscReal dr=(TwoDimensional(pctx)?1.0/sqrt(1.0/(pctx->dx*pctx->dx)+1.0/(pctx->dy*pctx
          ->dy)):1.0/sqrt(1.0/(pctx->dx*pctx->dx)+1.0/(pctx->dy*pctx->dy)+1.0/(pctx->dz*pctx
          ->dz)));
156   assert(pctx->properties);
157   PetscReal c=GetSpeedOfSound(pctx->properties);
158   PetscReal dt=pctx->params->tm.courant*dr/c;
159   assert(pctx->params);
160   if (pctx->params->tm.dt>dt)
161     {
162       pctx->params->tm.dt=dt;
163       PetscPrintf(PETSC_COMM_WORLD,"Reducing time step to %g s.\n",dt);
164     }
165
166   /* Mesh distributed array for dof=4 */
167   ierr=DACreate3d(PETSC_COMM_WORLD,DA_NONPERIODIC,stenciltype,
168       params->mesh.Nx,params->mesh.Ny,params->mesh.Nz,
169       PETSC_DECIDE,PETSC_DECIDE,PETSC_DECIDE,
170       dof,
171       stencilwidth,
172       PETSC_NULL,PETSC_NULL,PETSC_NULL,
173       &(pctx->da));
```

```
174   CHKERRQ(ierr);
175
176   ierr=DASetUniformCoordinates(pctx->da,
177                params->mesh.xmin, params->mesh.xmax,
178                params->mesh.ymin, params->mesh.ymax,
179                params->mesh.zmin, params->mesh.zmax);
180   CHKERRQ(ierr);
181
182   /* A vector to store the solution */
183   ierr=DACreateGlobalVector(pctx->da,psolution);
184   CHKERRQ(ierr);
185
186   PetscPrintf(PETSC_COMM_WORLD,"Fluid mesh dimensions:\n");
187   PetscPrintf(PETSC_COMM_WORLD," mesh region 0,0,0 to %d,%d,%d\n",
188         params->mesh.Nx-1,params->mesh.Ny-1,params->mesh.Nz-1);
189   PetscPrintf(PETSC_COMM_WORLD," interior mesh boundary %d,%d,%d to %d,%d,%d\n",
190         pctx->i_lo,pctx->j_lo,pctx->k_lo,
191         pctx->i_hi,pctx->j_hi,pctx->k_hi);
192   PetscPrintf(PETSC_COMM_WORLD," domain %g,%g,%g to %g,%g,%g\n",
193         params->mesh.xmin,params->mesh.ymin,params->mesh.zmin,
194         params->mesh.xmax,params->mesh.ymax,params->mesh.zmax);
195   PetscPrintf(PETSC_COMM_WORLD," interior domain %g,%g,%g to %g,%g,%g\n",
196         params->mesh.xmin+params->mesh.xlayer,params->mesh.ymin+params->mesh.ylayer,
                params->mesh.zmin+params->mesh.zlayer,
197         params->mesh.xmax-params->mesh.xlayer,params->mesh.ymax-params->mesh.ylayer,
                params->mesh.zmax-params->mesh.zlayer);
198   //  DAView(pctx->da,PETSC_VIEWER_STDOUT_WORLD);
199
200   PetscInt M,N,P,m,n,p;
201   /* Get the global and local dimensions of solution DA */
202   DAGetInfo(pctx->da,PETSC_IGNORE,
203       &M,&N,&P,
204       &m,&n,&p,
205       PETSC_IGNORE,
206       PETSC_IGNORE,
207       PETSC_IGNORE,PETSC_IGNORE);
208
209   bc_parameters *bc=(bc_parameters*)pctx->pbc;
210   assert(bc);
211
212   if (pctx->j_lo>0  || bc->ReflectiveGround || pctx->MaxAbsPressure)
213     {
214       bc_parameters *pbc=(bc_parameters*)(pctx->pbc);
215       assert(pbc);
216       if (pctx->DoProbe && pctx->rms_slice==0)
217   if (pbc->NoGround)
218     pctx->rms_slice = N/2;
219       else
220     pctx->rms_slice = pctx->j_lo;
221
```

```
222       PetscInt ilo=pctx->i_lo+1;   /* use pctx->i_lo to trim to non-ABC region of ground */
223       PetscInt ihi=pctx->i_hi-1;   /* use pctx->i_hi to trim to non-ABC region of ground */
224       PetscInt jlo=(pctx->rms_slice>0?pctx->rms_slice:pctx->j_lo);
225       PetscInt jhi=(pctx->rms_slice>0?pctx->rms_slice:pctx->j_lo);
226       PetscInt klo=pctx->k_lo+1;   /* use pctx->k_lo to trim to non-ABC region of ground */
227       PetscInt khi=pctx->k_hi-1;   /* use pctx->k_hi to trim to non-ABC region of ground */
228
229       if (TwoDimensional(pctx))
230   {
231     ilo=0;
232     ihi=M-1;
233     jlo=0;
234     jhi=N-1;
235     klo=0;
236     khi=0;
237     if (pctx->DoProbe && pctx->rms_slice==0)
238       pctx->rms_slice = 0;
239     if (pctx->MaxAbsPressure)
240       PetscPrintf(PETSC_COMM_WORLD,"Transmission loss calculated for flow domain.\n",jlo);
241   }
242       else
243   {
244     if (pctx->MaxAbsPressure)
245       PetscPrintf(PETSC_COMM_WORLD,"Transmission loss calculated along plane along j=%d\n",jlo);
246     else
247       PetscPrintf(PETSC_COMM_WORLD,"RMS pressure calculated along plane along j=%d\n",jlo);
248   }
249
250       CreateSliceScatter(pctx,*psolution,ilo,ihi,jlo,jhi,klo,khi,0,0,&pctx->dagp,&pctx->currp,&pctx->groundp);
251       CreateSliceScatter(pctx,*psolution,ilo,ihi,jlo,jhi,klo,khi,1,3,&pctx->dagw,&pctx->currw,&pctx->groundw);
252
253       ierr=DACreateGlobalVector(pctx->dagp,&pctx->prms); CHKERRQ(ierr);
254       ierr=DACreateGlobalVector(pctx->dagw,&pctx->I); CHKERRQ(ierr);
255       if (pctx->MaxAbsPressure) ierr=DACreateGlobalVector(pctx->dagp,&pctx->pmaxabs); CHKERRQ(ierr);
256
257       VecSet(pctx->prms,0.0);
258       VecSet(pctx->I,0.0);
259       if (pctx->MaxAbsPressure) VecSet(pctx->pmaxabs,0.0);
260
261       if ((pctx->rms_slice>0 && !TwoDimensional(pctx)) ||
```

```
262     (pctx->DoProbe && TwoDimensional(pctx)))
263
264   {
265     parameters *param=pctx->params;
266     assert(params);
267     assert(pctx->psource);
268
269     /* x,y from lower left corner of SPL slice */
270     PetscReal x0=param->mesh.xmin+ilo*pctx->dx;
271     PetscReal y0=(TwoDimensional(pctx)?param->mesh.ymin+jlo*pctx->dy:param->mesh.zmin+
          klo*pctx->dz);
272     PetscReal x=pctx->GetOrigin(pctx->psource,0)+pctx->RMSProbeDistance;
273     PetscReal y=(TwoDimensional(pctx)?pctx->GetOrigin(pctx->psource,1):pctx->GetOrigin(
          pctx->psource,2));
274     PetscReal dx=pctx->dx;
275     PetscReal dy=(TwoDimensional(pctx)?pctx->dy:pctx->dz);
276     PetscReal xmax=param->mesh.xmax-param->mesh.xlayer;
277     PetscReal ymax=(TwoDimensional(pctx)?param->mesh.ymax-param->mesh.ylayer:param->mesh
          .zmax-param->mesh.zlayer);
278
279     assert(x>=x0);
280     assert(y>=y0);
281     assert(x<=xmax);
282     assert(y<=ymax);
283
284     pctx->probe_li=(PetscInt)((ihi-ilo)*(x-x0)/(pctx->dx*(ihi-ilo)));
285     if (!TwoDimensional(pctx))
286       pctx->probe_lj=(PetscInt)((khi-klo)*(y-y0)/(pctx->dz*(khi-klo)));
287     else
288       pctx->probe_lj=(PetscInt)((jhi-jlo)*(y-y0)/(pctx->dy*(jhi-jlo)));
289
290     assert(pctx->probe_li>=0);
291     assert(pctx->probe_lj>=0);
292     assert(pctx->probe_li<=ihi-ilo);
293     assert((!TwoDimensional(pctx) && pctx->probe_lj<=khi-klo) ||
294      (TwoDimensional(pctx)  && pctx->probe_lj<=jhi-jlo));
295
296     PetscReal px,py;
297     px=fmax(0.0,fmin(1.0,(x-(x0+pctx->probe_li*dx))/dx));
298     py=fmax(0.0,fmin(1.0,(y-(y0+pctx->probe_lj*dy))/dy));
299
300     assert(px>=0.0 && px<=1.0);
301     assert(py>=0.0 && py<=1.0);
302
303     pctx->probe_w[0][0] = (1-px)*(1-py);
304     pctx->probe_w[1][0] = px*(1-py);
305     pctx->probe_w[0][1] = (1-px)*py;
306     pctx->probe_w[1][1] = px*py;
307
```

```
308       PetscPrintf(PETSC_COMM_WORLD,"Probe in plane j=%d at (%g,%g). Local nodes %d,%d %g %
              g\n",pctx->rms_slice,x,y,pctx->probe_li,pctx->probe_lj,px,py);
309     }
310       }
311
312     pctx->IntegrationStarted=PETSC_FALSE;
313     pctx->IntensityStarted=PETSC_FALSE;
314     pctx->tstart=0.0;
315     pctx->tlast=0.0;
316
317     pctx->SWL=0;
318     pctx->tHistory=NULL;
319     pctx->PowerHistory=NULL;
320     pctx->pHistory=NULL;
321     pctx->HistoryAllocLen=0;
322
323     PetscLogEventEnd(pctx->SetupEvent,0,0,0,0);
324     return 0;
325 }
326
327 PetscErrorCode DestroyStorage(Vec solution,
328               app_ctx *pctx)
329 {
330     int i;
331     PetscTruth Valid;
332     PetscErrorCode ierr;
333
334     assert(pctx);
335
336     PetscLogEventBegin(pctx->SetupEvent,0,0,0,0);
337
338     ierr=VecDestroy(solution); CHKERRQ(ierr);
339
340     ierr=DADestroy(pctx->da); CHKERRQ(ierr);
341
342     bc_parameters *bc=(bc_parameters*)pctx->pbc;
343     assert(bc);
344
345     if (pctx->j_lo>0 || bc->ReflectiveGround)
346       {
347         ierr=VecScatterDestroy(pctx->groundw); CHKERRQ(ierr);
348         ierr=VecScatterDestroy(pctx->groundp); CHKERRQ(ierr);
349
350         ierr=VecDestroy(pctx->I); CHKERRQ(ierr);
351         ierr=VecDestroy(pctx->prms); CHKERRQ(ierr);
352
353         ierr=VecDestroy(pctx->currw); CHKERRQ(ierr);
354         ierr=VecDestroy(pctx->currp); CHKERRQ(ierr);
355
356         ierr=DADestroy(pctx->dagw); CHKERRQ(ierr);
```

```
357         ierr=DADestroy(pctx->dagp); CHKERRQ(ierr);
358       }
359
360     if (pctx->pHistory) PetscFree(pctx->pHistory); pctx->pHistory=NULL;
361     if (pctx->PowerHistory) PetscFree(pctx->PowerHistory); pctx->PowerHistory=NULL;
362     if (pctx->tHistory) PetscFree(pctx->tHistory); pctx->tHistory=NULL;
363     pctx->HistoryAllocLen=0;
364
365     PetscLogEventEnd(pctx->SetupEvent,0,0,0,0);
366     return 0;
367 }
```

3.14 source.h

```
 1 #ifndef _SOURCE_H_
 2 #define _SOURCE_H_
 3
 4 #include<petsc.h>
 5 #include<petscbag.h>
 6
 7 /*! \file source.h
 8     \brief Source term parameters.
 9
10 This header file defines the parameters and functions for applying source terms.
11 */
12
13 /*! \brief Mesh location and interpolation
14
15   A structure for storing interpolation coefficients for distribution of a value across
       a cube of mesh points */
16
17 typedef struct {
18   PetscReal X;            /*!< source origin in x */
19   PetscReal Y;            /*!< source origin in y */
20   PetscReal Z;            /*!< source origin in z */
21   PetscInt li;            /*!< lowest mesh index along x bounding source origin */
22   PetscInt lj;            /*!< lowest mesh index along y bounding source origin */
23   PetscInt lk;            /*!< lowest mesh index along z bounding source origin */
24   PetscReal w[2][2][2];   /*!< interpolation weightings for mesh points about origin */
25 } interpolation_parameters;
26
27 /*! \brief Montone source
28
29   A monopole source term at a point in space. */
30 typedef struct {
31   PetscReal A;            /*!< source amplitude (1/s)*/
32   PetscReal f;            /*!< source frequency (Hz) */
33   PetscReal phi;          /*!< source phase angle (radians) */
34   interpolation_parameters Location;
```

```
35 } harmonic_parameters;
36
37 /*! \brief Calibrated monopole source
38
39 A calibrated monopole source term at a point in space. */
40 typedef struct {
41   PetscReal MeasuredSWL;         /*!< SWL (dB) of source as measured */
42   PetscReal MeasuredSPL;         /*!< SPL (dB) of source as measured */
43   PetscReal Distance;            /*!< distance from measured source */
44   harmonic_parameters Harmonic;   /*!< montone source parameters */
45 } monopole_parameters;
46
47 /*! \brief Dipole source
48
49   A dipole source term at a point in space. */
50 typedef struct {
51   PetscReal MeasuredSWL;         /*!< SWL (dB) of source as measured */
52   PetscReal MeasuredSPL;         /*!< SWL (dB) of source as measured */
53   PetscReal Distance;            /*!< SWL (dB) of source as measured */
54   PetscReal ObserverX;           /*!< direction of observer in x */
55   PetscReal ObserverY;           /*!< direction of observer in y */
56   PetscReal ObserverZ;           /*!< direction of observer in z */
57   PetscReal DirectionX;          /*!< direction of dipole axis in x */
58   PetscReal DirectionY;          /*!< direction of dipole axis in y */
59   PetscReal DirectionZ;          /*!< direction of dipole axis in z */
60   harmonic_parameters Harmonic1,Harmonic2;   /*!< source parameters */
61 } dipole_parameters;
62
63 /*! \brief A noise spectrum profile
64
65   A spectrum of noise for a range of frequencies. */
66 typedef struct {
67   PetscInt N;             /*!< Length of series */
68   harmonic_parameters *Harmonic; /*!< An array of harmonic sources */
69   PetscReal *deltaf;
70   interpolation_parameters Location;
71   char InputFile[1024];  /*!< File which contains series data in CSV format */
72   PetscTruth CSVFile;
73   PetscTruth InputAWeighted;
74 } monopole_series_parameters;
75
76 /*! Available source term types */
77 typedef enum {
78   SOURCE_NONE=0,               /*!< No source */
79   SOURCE_MONO_PING=1,          /*!< Short duration monopole, ramped start and end */
80   SOURCE_MONOPOLE=2,           /*!< Calibrated monopole source with initial ramp */
81   SOURCE_DIPOLE=3,             /*!< Dipole source with initial ramp */
82   SOURCE_MONOPOLE_SERIES=4,    /*! < Fourier series of monopole sources */
83   SOURCE_DIPOLE_SERIES=5       /*!< Fourier series of dipole sources */
84 } source_type;
```

```
85
86 /*! Save source term parameter values to file. */
87 void SaveSourceBag(PetscBag bag);
88
89 /*! Register short-duration monopole source parameters, and optionally load values from
       file. */
90 PetscErrorCode RegisterMonopolePingBag(PetscBag *bag,void **params,PetscTruth
       LoadFromFile);
91
92 PetscErrorCode InitializeMonopolePingSource(void *vpctx);
93
94 /*! Calculate source term contributions to RHS for a short-duration pulse at time t. */
95 void MonopolePing(PetscReal t,Vec R,void *ctx);
96
97 /*! Get frequency of monopole source */
98 PetscReal GetMonopolePingFrequency(void *vpctx);
99
100 /*! Get location of monopole source */
101 PetscReal GetMonopolePingOrigin(void *vpctx,PetscInt i);
102
103 /*! Register continuous, calibrated monopole source parameters, and optionally load
       values from file. */
104 PetscErrorCode RegisterMonopoleBag(PetscBag *bag,void **params,PetscTruth LoadFromFile);
105
106 /*! \brief Initialize calibrated monopole source parameters
107
108   Assumes a spherical monopole source in quiescent fluid. The amplitude of the pressure
        distribution at distance \f$r\f$ is
109   \f$ \rho_0 c \frac{Q k}{4\pi r}\f$, where \f$k=\frac{\omega}{c}\f$ and \f$Q\f$ is the
        volumetric flow rate of the source.
110   As an RMS average of a sinusoid pressure is \f$p_{rms}=\frac{1}{\sqrt{2}}p\f$, the
        volumetric flow rate can be
111   determined as \f$ \frac{2\sqrt{2}r p_{rms}}{\rho_0 f}\f$.
112  */
113 PetscErrorCode InitializeMonopoleSource(void *vpctx);
114
115 /*! Calculate source term contributions to RHS for a continuous, calibrated monopole
       source at time t. */
116 void Monopole(PetscReal t,Vec R,void *ctx);
117
118 /*! Get frequency of calibrated monopole source */
119 PetscReal GetMonopoleFrequency(void *vpctx);
120
121 /*! Get location of calibrated monopole source */
122 PetscReal GetMonopoleOrigin(void *vpctx,PetscInt i);
123
124 PetscErrorCode InitializeMonopoleFamily(harmonic_parameters *psource,void *vpctx);
125
126 /*! Register continuous dipole source parameters, and optionally load values from file.
       */
```

```
127 PetscErrorCode RegisterDipoleBag(PetscBag *bag,void **params,PetscTruth LoadFromFile);
128
129 /*! Calculate source term contributions to RHS for a continuous dipole source at time t.
      */
130 void Dipole(PetscReal t,Vec R,void *ctx);
131
132 /*! Initialize dipole source parameters */
133 PetscErrorCode InitializeDipoleSource(void *vpctx);
134
135 /*! Get frequency of dipole source */
136 PetscReal GetDipoleFrequency(void *vpctx);
137
138 /*! Get location of dipole source */
139 PetscReal GetDipoleOrigin(void *vpctx,PetscInt i);
140
141
142 /*! Compute the AWeighting gain (dB) */
143 PetscReal GetAWeighting(PetscReal f);
144
145 /*! Register continuous, calibrated monopole source parameters, and optionally load
      values from file. */
146 PetscErrorCode RegisterMonopoleSeriesBag(PetscBag *bag,void **params,PetscTruth
      LoadFromFile);
147
148 /*! \brief Initialize calibrated monopole source parameters
149  */
150 PetscErrorCode InitializeMonopoleSeriesSource(void *vpctx);
151
152 /*! Calculate source term contributions to RHS for a continuous, calibrated monopole
      source at time t. */
153 void MonopoleSeries(PetscReal t,Vec R,void *ctx);
154
155 /*! Get frequency of calibrated monopole source */
156 PetscReal GetMonopoleSeriesLowestFrequency(void *vpctx);
157 PetscReal GetMonopoleSeriesHighestFrequency(void *vpctx);
158
159 /*! Get location of calibrated monopole source */
160 PetscReal GetMonopoleSeriesOrigin(void *vpctx,PetscInt i);
161
162 /*! \brief Deallocate calibrated monopole source parameters
163  */
164 PetscErrorCode DestroyMonopoleSeriesSource(void *vpctx);
165
166 #endif
```

3.15 source.c

```
1 #include<stdlib.h>
2 #include<assert.h>
```

```
3 #include<petsc.h>
4
5 #include"ode.h"
6 #include"source.h"
7
8 const char SourceParametersFile[]="sparameters.dat";
9
10 PetscErrorCode CalculateMeshLocation(interpolation_parameters *interp,
11              app_ctx *pctx)
12 {
13   parameters *param=pctx->params;
14
15   assert(interp);
16   assert(pctx);
17   assert(param);
18
19   PetscReal dx=(param->mesh.xmax-param->mesh.xmin)/(param->mesh.Nx-1);
20   PetscReal dy=(param->mesh.ymax-param->mesh.ymin)/(param->mesh.Ny-1);
21   PetscReal dz=1.0;
22   if (!TwoDimensional(pctx))
23     dz=(param->mesh.zmax-param->mesh.zmin)/(param->mesh.Nz-1);
24
25   if (interp->X<param->mesh.xmin || interp->X>param->mesh.xmax ||
26       interp->Y<param->mesh.ymin || interp->Y>param->mesh.ymax ||
27       interp->Z<param->mesh.zmin || interp->Z>param->mesh.zmax)
28     {
29       PetscPrintf(PETSC_COMM_WORLD,"Harmonic origin outside flow domain!\n");
30       return 1;
31     }
32
33   /* Note that wz values are recorded for i+dx/2,j,k */
34   interp->li=(PetscInt)((param->mesh.Nx-1)*(interp->X-param->mesh.xmin)/(param->mesh.
         xmax-param->mesh.xmin));
35   interp->lj=(PetscInt)((param->mesh.Ny-1)*(interp->Y-param->mesh.ymin)/(param->mesh.
         ymax-param->mesh.ymin));
36   if (!TwoDimensional(pctx))
37     interp->lk=(PetscInt)((param->mesh.Nz-1)*(interp->Z-param->mesh.zmin)/(param->mesh.
           zmax-param->mesh.zmin));
38   else
39     interp->lk=0;
40
41   assert(interp->li<param->mesh.Nx-1);
42   assert(interp->lj<param->mesh.Ny-1);
43   assert((TwoDimensional(pctx) && interp->lk==0) || (!TwoDimensional(pctx) && interp->lk
         <param->mesh.Nz-1));
44
45   PetscReal px,py,pz;
46   px=fmax(0.0,fmin(1.0,(interp->X-(param->mesh.xmin+interp->li*dx))/dx));
47   py=fmax(0.0,fmin(1.0,(interp->Y-(param->mesh.ymin+interp->lj*dy))/dy));
48   if (!TwoDimensional(pctx))
```

```
49      pz=fmax(0.0,fmin(1.0,(interp->Z-(param->mesh.zmin+interp->lk*dz))/dz));
50    else
51      pz=0.0;
52
53    assert(px>=0.0 && px<=1.0);
54    assert(py>=0.0 && py<=1.0);
55    assert(pz>=0.0 && pz<=1.0);
56
57    /* li,lj,lk */
58    interp->w[0][0][0] = (1-px)*(1-py)*(1-pz);
59    /* li+1,lj,lk */
60    interp->w[1][0][0] = px*(1-py)*(1-pz);
61    /* li,lj+1,lk */
62    interp->w[0][1][0] = (1-px)*py*(1-pz);
63    /* li+1,lj+1,lk */
64    interp->w[1][1][0] = px*py*(1-pz);
65
66    if (!TwoDimensional(pctx))
67      {
68        /* li,lj,lk+ */
69        interp->w[0][0][1] = (1-px)*(1-py)*pz;
70        /* li+1,lj,lk+1 */
71        interp->w[1][0][1] = px*(1-py)*pz;
72        /* li,lj+1,lk+1 */
73        interp->w[0][1][1] = (1-px)*py*pz;
74        /* li+1,lj+1,lk+1 */
75        interp->w[1][1][1] = px*py*pz;
76      }
77
78    PetscPrintf(PETSC_COMM_WORLD,"Source origin %g,%g,%g between %d,%d,%d and %d,%d,%d (%g
          ,%g,%g)\n",interp->X,interp->Y,interp->Z,interp->li,interp->lj,interp->lk,interp->
          li+1,interp->lj+1,interp->lk+1,px,py,pz);
79    return 0;
80  }
81
82  void SaveSourceBag(PetscBag bag)
83  {
84    PetscViewer bagviewer;
85    /* Save the runtime parameters to a binary file */
86    PetscViewerBinaryOpen(PETSC_COMM_WORLD,SourceParametersFile,FILE_MODE_WRITE,&bagviewer
          );
87    PetscBagView(bag,bagviewer);
88    PetscViewerDestroy(bagviewer);
89  }
90
91  PetscReal MonopoleSignalWithRamp(PetscReal t,
92            harmonic_parameters *params,
93            PetscInt RampIn,
94            PetscInt Duration,
95            PetscInt RampOut)
```

```
96 {
97   PetscReal Q=0;
98   PetscReal T1,T2,T;
99   PetscReal scale=1.0;
100
101   assert(params);
102
103   T1=((double)(RampIn))/params->f;
104   T2=((double)(RampOut))/params->f;
105   T=((double)(Duration))/params->f;
106
107   if (t>=0 && t<T1)
108     {
109       Q=0.5*params->A*(1-cos(M_PI*t/T1))*cos(2*M_PI*params->f*t+params->phi);
110       PetscLogFlops(9);
111     }
112   else if (t>=T1 && (((T==0) || (T2==0)) || (T>0 && T2>0 && t<=(T-T2))))
113     {
114       Q=params->A*cos(2*M_PI*params->f*t+params->phi);
115       PetscLogFlops(4);
116     }
117   else if (T>0 && T2>0 && (t>(T-T2) && t<=T))
118     {
119       Q=0.5*params->A*(1+cos(M_PI*(t-(T-T2))/T2))*cos(2*M_PI*params->f*t+params->phi);
120       PetscLogFlops(9);
121     }
122   return Q;
123 }
124
125 PetscErrorCode RegisterMonopolePingBag(PetscBag *bag,void **pparams,PetscTruth
      LoadFromFile)
126 {
127   return RegisterMonopoleBag(bag,pparams,LoadFromFile);
128 }
129
130 PetscErrorCode InitializeMonopolePingSource(void *vpctx)
131 {
132   app_ctx *pctx=(app_ctx*)vpctx;
133   parameters *param=pctx->params;
134   harmonic_parameters *psource=(harmonic_parameters*)pctx->psource;
135
136   assert(psource);
137
138   PetscPrintf(PETSC_COMM_WORLD,"Monopole source amplitude %g.\n",psource->A);
139
140   return InitializeMonopoleFamily(psource,vpctx);
141 }
142
143
144 void MonopolePing(PetscReal t,
```

```
145         Vec R,
146         void *vpctx)
147 {
148   app_ctx *pctx=(app_ctx*)vpctx;
149   PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz;
150   PetscScalar ****r;
151   Vec local;
152   monopole_parameters *psource=(monopole_parameters*)(pctx->psource);
153   assert(psource);
154   interpolation_parameters *params=&(psource->Harmonic.Location);
155
156   assert(pctx);
157   assert(params);
158
159   PetscLogEventBegin(pctx->CalcSourceEvent,0,0,0,0);
160
161   /* Determine global mesh nodal extents */
162   DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
163         PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
164         PETSC_IGNORE,PETSC_IGNORE,
165         PETSC_IGNORE,PETSC_IGNORE);
166
167   /* Get local grid boundaries, which are independent of dof */
168   DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
169
170   DAVecGetArrayDOF(pctx->da,R,&r);
171
172   /* R is previously zeroed */
173   PetscReal Mag=MonopoleSignalWithRamp(t,&(psource->Harmonic),3,10,3)*pctx->invdx*pctx->
         invdy*pctx->invdz;
174   PetscInt kN=(TwoDimensional(pctx)?1:2);
175   for(k=0;k<kN;k++)
176     for(j=0;j<2;j++)
177       for(i=0;i<2;i++)
178   {
179     if ((params->li+i>=sx && params->li+i<sx+mx) &&
180         (params->lj+j>=sy && params->lj+j<sy+my) &&
181         (params->lk+k>=sz && params->lk+k<sz+mz))
182       {
183         r[params->lk+k][params->lj+j][params->li+i][0]=params->w[i][j][k]*Mag;
184       }
185   }
186
187   /* Release access to array */
188   DAVecRestoreArrayDOF(pctx->da,R,&r);
189
190   PetscLogEventEnd(pctx->CalcSourceEvent,0,0,0,0);
191 }
192
193 PetscReal GetMonopolePingFrequency(void *psource)
```

```
194 {
195   harmonic_parameters *pmono=(harmonic_parameters*)psource;
196
197   return pmono->f;
198 }
199
200 PetscReal GetMonopolePingOrigin(void *psource,PetscInt d)
201 {
202   harmonic_parameters *pmono=(harmonic_parameters*)psource;
203
204   assert(d>=0 && d<=2);
205   switch(d)
206     {
207     case 0:
208       return pmono->Location.X; break;
209     case 1:
210       return pmono->Location.Y; break;
211     case 2:
212       return pmono->Location.Z; break;
213     };
214   return 0;
215 }
216
217
218 PetscErrorCode RegisterDipoleBag(PetscBag *bag,void **pparams,PetscTruth LoadFromFile)
219 {
220   PetscErrorCode ierr=0;
221   assert(bag);
222   assert(pparams);
223
224   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(dipole_parameters),bag); CHKERRQ(ierr);
225   ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
226   ierr=PetscBagSetName(*bag,"Dipole Source Parameters","runtime parameters for a simple
        dipole source"); CHKERRQ(ierr);
227
228   dipole_parameters *psource=(dipole_parameters*)(*pparams);
229   /* Monopole source parameters */
230   ierr=PetscBagRegisterReal(*bag,&psource->MeasuredSPL,0,"measured_spl","Dipole source
        sound pressure level in dipole direction (dB)"); CHKERRQ(ierr);
231   ierr=PetscBagRegisterReal(*bag,&psource->Distance,0,"measured_distance","Dipole source
         distance (m)"); CHKERRQ(ierr);
232   ierr=PetscBagRegisterReal(*bag,&psource->MeasuredSWL,0,"measured_swl","Dipole source
        sound power level (dB)"); CHKERRQ(ierr);
233   ierr=PetscBagRegisterReal(*bag,&psource->ObserverX,0,"dipole_observer_x","Dipole
        source observer direction along x-axis"); CHKERRQ(ierr);
234   ierr=PetscBagRegisterReal(*bag,&psource->ObserverY,0,"dipole_observer_y","Dipole
        source observer direction along y-axis"); CHKERRQ(ierr);
235   ierr=PetscBagRegisterReal(*bag,&psource->ObserverZ,0,"dipole_observer_z","Dipole
        source observer direction along z-axis"); CHKERRQ(ierr);
```

```
236   ierr=PetscBagRegisterReal(*bag,&psource->DirectionX,1,"dipole_direction_x","Dipole
          source axis direction along x-axis"); CHKERRQ(ierr);
237   ierr=PetscBagRegisterReal(*bag,&psource->DirectionY,0,"dipole_direction_y","Dipole
          source axis direction along y-axis"); CHKERRQ(ierr);
238   ierr=PetscBagRegisterReal(*bag,&psource->DirectionZ,0,"dipole_direction_z","Dipole
          source axis direction along z-axis"); CHKERRQ(ierr);
239   ierr=PetscBagRegisterReal(*bag,&psource->Harmonic1.A,0,"dipole_A","Dipole source
          amplitude"); CHKERRQ(ierr);
240   ierr=PetscBagRegisterReal(*bag,&psource->Harmonic1.f,100,"dipole_f","Dipole source
          frequency (Hz)"); CHKERRQ(ierr);
241   ierr=PetscBagRegisterReal(*bag,&psource->Harmonic1.Location.X,0,"dipole_origin_x","
          Dipole source origin along x-axis (m)"); CHKERRQ(ierr);
242   ierr=PetscBagRegisterReal(*bag,&psource->Harmonic1.Location.Y,0,"dipole_origin_y","
          Dipole source origin along y-axis (m)"); CHKERRQ(ierr);
243   ierr=PetscBagRegisterReal(*bag,&psource->Harmonic1.Location.Z,0,"dipole_origin_z","
          Dipole source origin along z-axis (m)"); CHKERRQ(ierr);
244
245
246   if (LoadFromFile)
247     {
248       PetscViewer bagviewer;
249       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,SourceParametersFile,
              FILE_MODE_READ,&bagviewer))
250   {
251     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",SourceParametersFile);
252     CHKERRQ(ierr);
253   }
254       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
255       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
256       ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
257       PetscPrintf(PETSC_COMM_WORLD,"Loaded dipole source parameters:\n");
258       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
259     }
260
261   psource->Harmonic2.A=psource->Harmonic1.A;
262   psource->Harmonic2.f=psource->Harmonic1.f;
263   psource->Harmonic2.phi=M_PI;
264   psource->Harmonic1.phi=0;
265   psource->Harmonic2.Location.X=psource->Harmonic1.Location.X;
266   psource->Harmonic2.Location.Y=psource->Harmonic1.Location.Y;
267   psource->Harmonic2.Location.Z=psource->Harmonic1.Location.Z;
268
269   return 0;
270 }
271
272 void Dipole(PetscReal t,
273       Vec R,
274       void *vpctx)
275 {
276   app_ctx *pctx=(app_ctx*)vpctx;
```

```
277   dipole_parameters *params=(dipole_parameters*)(pctx->psource);
278   PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz,s;
279   PetscScalar ****r;
280   Vec local;
281
282   assert(pctx);
283   assert(params);
284
285   PetscLogEventBegin(pctx->CalcSourceEvent,0,0,0,0);
286
287   /* Determine global mesh nodal extents */
288   DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
289       PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
290       PETSC_IGNORE,PETSC_IGNORE,
291       PETSC_IGNORE,PETSC_IGNORE);
292
293   /* Get local grid boundaries, which are independent of dof */
294   DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
295
296   DAVecGetArrayDOF(pctx->da,R,&r);
297
298   /* R is previously zeroed */
299   PetscReal Mag1=MonopoleSignalWithRamp(t,&(params->Harmonic1),3,0,0)*pctx->invdx*pctx->
          invdy*pctx->invdz;
300   PetscReal Mag2=MonopoleSignalWithRamp(t,&(params->Harmonic2),3,0,0)*pctx->invdx*pctx->
          invdy*pctx->invdz;
301   assert(pctx->properties);
302   for(i=0;i<2;i++)
303     for(j=0;j<2;j++)
304       for(k=0;k<2;k++)
305   {
306     if ((params->Harmonic1.Location.li+i>=sx && params->Harmonic1.Location.li+i<sx+mx)
            &&
307         (params->Harmonic1.Location.lj+j>=sy && params->Harmonic1.Location.lj+j<sy+my)
                &&
308         (params->Harmonic1.Location.lk+k>=sz && params->Harmonic1.Location.lk+k<sz+mz))
309       {
310         r[params->Harmonic1.Location.lk+k][params->Harmonic1.Location.lj+j][params->
                Harmonic1.Location.li+i][0] += Mag1*params->Harmonic1.Location.w[i][j][k];
311       }
312     if ((params->Harmonic2.Location.li+i>=sx && params->Harmonic2.Location.li+i<sx+mx)
            &&
313         (params->Harmonic2.Location.lj+j>=sy && params->Harmonic2.Location.lj+j<sy+my)
                &&
314         (params->Harmonic2.Location.lk+k>=sz && params->Harmonic2.Location.lk+k<sz+mz))
315       {
316         r[params->Harmonic2.Location.lk+k][params->Harmonic2.Location.lj+j][params->
                Harmonic2.Location.li+i][0] += Mag2*params->Harmonic2.Location.w[i][j][k];
317       }
318   }
```

```
319   /* Release access to array */
320   DAVecRestoreArrayDOF(pctx->da,R,&r);
321
322   PetscLogEventEnd(pctx->CalcSourceEvent,0,0,0,0);
323
324 }
325
326 PetscErrorCode InitializeDipoleSource(void *vpctx)
327 {
328   PetscErrorCode ierr=0;
329   const PetscReal pref=20E-6;
330   const PetscReal Wref=1E-12;
331   app_ctx *pctx=(app_ctx*)vpctx;
332   parameters *param=pctx->params;
333   dipole_parameters *psource=(dipole_parameters*)pctx->psource;
334   parameters *params=pctx->params;
335   assert(params);
336
337   PetscReal dx=(params->mesh.xmax-params->mesh.xmin)/(params->mesh.Nx-1);
338   PetscReal dy=(params->mesh.ymax-params->mesh.ymin)/(params->mesh.Ny-1);
339   PetscReal dz=1.0;
340   if (!TwoDimensional(pctx))
341     dz=(params->mesh.zmax-params->mesh.zmin)/(params->mesh.Nz-1);
342
343   PetscReal d=sqrt(dx*dx+dy*dy+dz*dz);
344
345   /* Using two monopoles, separated by distance d and 180 deg out of phase, to produce
        dipole */
346
347   /* Normalize dipole axis */
348   PetscReal invlen1=1.0/sqrt(psource->DirectionX*psource->DirectionX+
349             psource->DirectionY*psource->DirectionY+
350             psource->DirectionZ*psource->DirectionZ);
351   psource->DirectionX *= invlen1;
352   psource->DirectionY *= invlen1;
353   psource->DirectionZ *= invlen1;
354
355   /* Normalize observer axis */
356   PetscReal len2=sqrt(psource->ObserverX*psource->ObserverX+
357           psource->ObserverY*psource->ObserverY+
358           psource->ObserverZ*psource->ObserverZ);
359   if (len2==0)
360     {
361       psource->ObserverX=psource->DirectionX;
362       psource->ObserverY=psource->DirectionY;
363       psource->ObserverZ=psource->DirectionZ;
364     }
365   else
366     {
367       PetscReal invlen2=1.0/len2;
```

```
368         psource->ObserverX *= invlen2;
369         psource->ObserverY *= invlen2;
370         psource->ObserverZ *= invlen2;
371       }
372
373     PetscReal c = GetSpeedOfSound(pctx->properties);
374     PetscReal f = psource->Harmonic1.f;
375
376     if (psource->Harmonic1.A==0)
377       {
378         if (psource->MeasuredSWL==0)
379     {
380       /* If p_rms is known at distance r from source in observer direction o, for dipole
              axis d, then the amplitude of the
381          source is (8*pi*r^2*p_rms)/(o.d*sqrt(2*(1+k^2*r^2))) */
382       PetscReal prms = pref*pow(10.0,0.05*psource->MeasuredSPL);
383       PetscReal k=2*M_PI*f/c;
384       PetscReal r=psource->Distance;
385       PetscReal r2=r*r;
386       PetscReal k2=k*k;
387       PetscReal vecdot=fabs(psource->ObserverX*psource->DirectionX+
388           psource->ObserverY*psource->DirectionY+
389           psource->ObserverZ*psource->DirectionZ);
390       psource->Harmonic1.A = (8*M_PI*r2*prms)/(sqrt(2*(1+k2*r2))*vecdot);
391       PetscPrintf(PETSC_COMM_WORLD,"Dipole (as measured SPL %g dB at %g m in direction %g
              %g %g) source amplitude %g with dipole axis %g %g %g.\n",psource->MeasuredSPL,
              psource->Distance,psource->ObserverX,psource->ObserverY,psource->ObserverZ,
              psource->Harmonic1.A,psource->DirectionX,psource->DirectionY,psource->DirectionZ
              );
392     }
393         else
394     {
395       /* If power W is known, the amplitude of dual volumetric sources is sqrt((6 pi W)/(
              rho c))*(1/(k^2 d)) */
396       PetscReal W = Wref*pow(10.0,0.1*psource->MeasuredSWL);
397       PetscReal k = 2*M_PI*f/c;
398       psource->Harmonic1.A = sqrt((6*M_PI*W)/(GetSpeedOfSound(pctx->properties)*GetDensity
              (pctx->properties)))/(k*k*d);
399       PetscPrintf(PETSC_COMM_WORLD,"Dipole (as measured SWL %g dB) as dual monopole
              sources with amplitude %g and separation %g m along x-axis.\n",psource->
              MeasuredSWL,psource->Harmonic1.A,d);
400       psource->MeasuredSPL=psource->MeasuredSWL+10*log10(1/(4*M_PI*psource->Distance*
              psource->Distance));
401     }
402       }
403     else
404       {
405         PetscPrintf(PETSC_COMM_WORLD,"Dipole amplitude %g.\n",psource->Harmonic1.A);
406       }
407
```

```
408   psource->Harmonic2.A=psource->Harmonic1.A;
409   psource->Harmonic2.f=psource->Harmonic1.f;
410   psource->Harmonic2.phi=M_PI; psource->Harmonic1.phi=0;
411
412   psource->Harmonic1.Location.X -= 0.5*d*psource->DirectionX;
413   psource->Harmonic1.Location.Y -= 0.5*d*psource->DirectionY;
414   psource->Harmonic1.Location.Z -= 0.5*d*psource->DirectionZ;
415
416   psource->Harmonic2.Location.X += 0.5*d*psource->DirectionX;
417   psource->Harmonic2.Location.Y += 0.5*d*psource->DirectionY;
418   psource->Harmonic2.Location.Z += 0.5*d*psource->DirectionZ;
419
420   ierr=CalculateMeshLocation(&(psource->Harmonic1.Location),pctx); CHKERRQ(ierr);
421   ierr=CalculateMeshLocation(&(psource->Harmonic2.Location),pctx); CHKERRQ(ierr);
422
423   return 0;
424 }
425
426 PetscReal GetDipoleFrequency(void *psource)
427 {
428   dipole_parameters *pdipole=(dipole_parameters*)psource;
429
430   assert(pdipole);
431   assert(pdipole->Harmonic1.f==pdipole->Harmonic2.f);
432
433   return pdipole->Harmonic1.f;
434  }
435
436 PetscReal GetDipoleOrigin(void *psource,PetscInt d)
437 {
438   dipole_parameters *pdipole=(dipole_parameters*)psource;
439
440   assert(d>=0 && d<=2);
441   switch(d)
442     {
443     case 0:
444       return 0.5*(pdipole->Harmonic1.Location.X+pdipole->Harmonic2.Location.X); break;
445     case 1:
446       return 0.5*(pdipole->Harmonic1.Location.Y+pdipole->Harmonic2.Location.Y); break;
447     case 2:
448       return 0.5*(pdipole->Harmonic1.Location.Z+pdipole->Harmonic2.Location.Z); break;
449     };
450   return 0;
451 }
452
453
454 PetscErrorCode RegisterMonopoleBag(PetscBag *bag,void **pparams,PetscTruth LoadFromFile)
455 {
456   PetscErrorCode ierr=0;
457   assert(bag);
```

```
458   assert(pparams);
459
460   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(monopole_parameters),bag); CHKERRQ(ierr);
461   ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
462   ierr=PetscBagSetName(*bag,"Measured Monopole Source Parameters","runtime parameters
          for a monopole source"); CHKERRQ(ierr);
463
464   monopole_parameters *pmono=(monopole_parameters*)(*pparams);
465   /* Monopole source parameters */
466   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.A,0,"mono_A","Monopole source
          amplitude"); CHKERRQ(ierr);
467   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.f,100,"mono_f","Measured monopole
          source frequency (Hz)"); CHKERRQ(ierr);
468   ierr=PetscBagRegisterReal(*bag,&pmono->MeasuredSPL,0,"measured_spl","Monopole source
          measured sound pressure level (dB) at point of measurement"); CHKERRQ(ierr);
469   ierr=PetscBagRegisterReal(*bag,&pmono->Distance,1,"measured_distance","Distance from
          source (m) at point of measurement"); CHKERRQ(ierr);
470   ierr=PetscBagRegisterReal(*bag,&pmono->MeasuredSWL,1,"measured_swl","Monopole source
          sound power level (dB)"); CHKERRQ(ierr);
471   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.phi,0,"mono_phi","Monopole source
          phase (radians)"); CHKERRQ(ierr);
472   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.Location.X,0,"mono_origin_x","Monopole
           source origin along x-axis (m)"); CHKERRQ(ierr);
473   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.Location.Y,0,"mono_origin_y","Monopole
           source origin along y-axis (m)"); CHKERRQ(ierr);
474   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.Location.Z,0,"mono_origin_z","Monopole
           source origin along z-axis (m)"); CHKERRQ(ierr);
475
476   if (LoadFromFile)
477     {
478       PetscViewer bagviewer;
479       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,SourceParametersFile,
              FILE_MODE_READ,&bagviewer))
480   {
481     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",SourceParametersFile);
482    CHKERRQ(ierr);
483   }
484       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
485       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
486       ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
487       PetscPrintf(PETSC_COMM_WORLD,"Loaded monopole source parameters:\n");
488       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
489     }
490   return 0;
491 }
492
493 PetscErrorCode InitializeMonopoleFamily(harmonic_parameters *psource,void *vpctx)
494 {
495   PetscErrorCode ierr=0;
496   app_ctx *pctx=(app_ctx*)vpctx;
```

```
497   parameters *param=pctx->params;
498
499   ierr=CalculateMeshLocation(&(psource->Location),pctx); CHKERRQ(ierr);
500
501   PetscPrintf(PETSC_COMM_WORLD,"Monopole origin between mesh nodes %d,%d,%d and %d,%d,%d
          \n",psource->Location.li,psource->Location.lj,psource->Location.lk,psource->
          Location.li+1,psource->Location.lj+1,psource->Location.lk+1);
502
503   return 0;
504 }
505
506 PetscErrorCode InitializeMonopoleSource(void *vpctx)
507 {
508   const PetscReal pref=20E-6;
509   const PetscReal Wref=1E-12;
510   app_ctx *pctx=(app_ctx*)vpctx;
511   parameters *param=pctx->params;
512   monopole_parameters *psource=(monopole_parameters*)pctx->psource;
513
514   if (psource->Distance<=0)
515     {
516       PetscPrintf(PETSC_COMM_WORLD,"Distance from measured source must be >0.\n");
517       return 1;
518     }
519
520   if (psource->Harmonic.A==0)
521     {
522       if (psource->MeasuredSWL==0)
523   {
524     /* If p_rms at distance r is known, the amplitude of the volumetric source is (2
          sqrt(2) r p_rms)/(rho f) */
525     PetscReal prms = pref*pow(10.0,0.05*psource->MeasuredSPL);
526     psource->Harmonic.A = 2*sqrt(2)*prms*psource->Distance/(GetDensity(pctx->properties)
          *psource->Harmonic.f);
527     PetscPrintf(PETSC_COMM_WORLD,"Monopole (as measured SPL %g dB at %g m) source
          amplitude %g.\n",psource->MeasuredSPL,psource->Distance,psource->Harmonic.A);
528   }
529       else
530   {
531     /* If power W is known, the amplitude of the volumetric source is sqrt((2 c W)/(rho
          pi f^2)) */
532     PetscReal W = Wref*pow(10.0,0.1*psource->MeasuredSWL);
533     psource->Harmonic.A = sqrt(2*W*GetSpeedOfSound(pctx->properties)/GetDensity(pctx->
          properties)/M_PI)/psource->Harmonic.f;
534     PetscPrintf(PETSC_COMM_WORLD,"Monopole (as measured SWL %g dB) source amplitude %g.\
          n",psource->MeasuredSWL,psource->Harmonic.A);
535     psource->MeasuredSPL=psource->MeasuredSWL+10*log10(1/(4*M_PI*psource->Distance*
          psource->Distance));
536   }
537     }
```

```
538   else
539     {
540       PetscPrintf(PETSC_COMM_WORLD,"Monopole source amplitude %g.\n",psource->Harmonic.A
              );
541     }
542
543   return InitializeMonopoleFamily(&(psource->Harmonic),vpctx);
544 }
545
546 void Monopole(PetscReal t,
547         Vec R,
548         void *vpctx)
549 {
550   app_ctx *pctx=(app_ctx*)vpctx;
551   PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz;
552   PetscScalar ****r;
553   Vec local;
554   monopole_parameters *params=(monopole_parameters*)pctx->psource;
555
556   assert(pctx);
557   assert(params);
558
559   PetscLogEventBegin(pctx->CalcSourceEvent,0,0,0,0);
560
561   /* Determine global mesh nodal extents */
562   DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
563       PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
564       PETSC_IGNORE,PETSC_IGNORE,
565       PETSC_IGNORE,PETSC_IGNORE);
566
567   /* Get local grid boundaries, which are independent of dof */
568   DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
569
570   DAVecGetArrayDOF(pctx->da,R,&r);
571
572   /* R is previously zeroed */
573   PetscReal Mag=MonopoleSignalWithRamp(t,&(params->Harmonic),3,0,0)*pctx->invdx*pctx->
          invdy*pctx->invdz;
574   PetscInt kN=(TwoDimensional(pctx)?1:2);
575   for(k=0;k<kN;k++)
576     for(j=0;j<2;j++)
577       for(i=0;i<2;i++)
578   {
579     if ((params->Harmonic.Location.li+i>=sx && params->Harmonic.Location.li+i<sx+mx) &&
580         (params->Harmonic.Location.lj+j>=sy && params->Harmonic.Location.lj+j<sy+my) &&
581         (params->Harmonic.Location.lk+k>=sz && params->Harmonic.Location.lk+k<sz+mz))
582       {
583         r[params->Harmonic.Location.lk+k][params->Harmonic.Location.lj+j][params->
              Harmonic.Location.li+i][0] = params->Harmonic.Location.w[i][j][k]*Mag;
584       }
```

```
585   }
586   /* Release access to array */
587   DAVecRestoreArrayDOF(pctx->da,R,&r);
588
589   PetscLogEventEnd(pctx->CalcSourceEvent,0,0,0,0);
590 }
591
592 PetscReal GetMonopoleFrequency(void *psource)
593 {
594   monopole_parameters *pmono=(monopole_parameters*)psource;
595
596   assert(pmono);
597
598   return pmono->Harmonic.f;
599 }
600
601
602 PetscReal GetMonopoleOrigin(void *psource,PetscInt d)
603 {
604   monopole_parameters *pmono=(monopole_parameters*)psource;
605
606   assert(d>=0 && d<=2);
607   switch(d)
608     {
609     case 0:
610       return pmono->Harmonic.Location.X; break;
611     case 1:
612       return pmono->Harmonic.Location.Y; break;
613     case 2:
614       return pmono->Harmonic.Location.Z; break;
615     };
616   return 0;
617 }
618
619
620 PetscReal GetAWeighting(PetscReal f)
621 {
622   const PetscReal C1=12200*12200;
623   const PetscReal C2=20.6*20.6;
624   const PetscReal C3=107.7*107.7;
625   const PetscReal C4=737.9*737.9;
626   PetscReal f2=f*f;
627   PetscReal RA=(C1*f2*f2)/((f2+C2)*sqrt((f2+C3)*(f2+C4))*(f2+C1));
628   return 2.0+20.0*log10(RA);
629 }
630
631 PetscErrorCode RegisterMonopoleSeriesBag(PetscBag *bag,void **pparams,PetscTruth
      LoadFromFile)
632 {
633   PetscErrorCode ierr=0;
```

```
634   assert(bag);
635   assert(pparams);
636
637   ierr=PetscBagCreate(PETSC_COMM_WORLD,sizeof(monopole_series_parameters),bag); CHKERRQ(
          ierr);
638   ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
639   ierr=PetscBagSetName(*bag,"Measured Monopole Source Parameters","runtime parameters
          for a monopole source"); CHKERRQ(ierr);
640
641   monopole_series_parameters *pmono=(monopole_series_parameters*)(*pparams);
642   /* Monopole source parameters */
643   ierr=PetscBagRegisterString(*bag,&pmono->InputFile,1024,"mono.csv","mono_series_input"
          ,"Monopole series input data file"); CHKERRQ(ierr);
644   ierr=PetscBagRegisterTruth(*bag,&pmono->CSVFile,PETSC_TRUE,"mono_series_csv_input","
          Indicate monopole series input data file is in CSV format."); CHKERRQ(ierr);
645   ierr=PetscBagRegisterTruth(*bag,&pmono->InputAWeighted,PETSC_TRUE,"
          mono_series_A_weighted_input","Indicate monopole series input data file is in CSV
          format."); CHKERRQ(ierr);
646  //  ierr=PetscBagRegisterReal(*bag,&pmono->deltaf,100,"mono_deltaf","Monopole series
         frequency interval (Hz)"); CHKERRQ(ierr);
647 //   ierr=PetscBagRegisterReal(*bag,&pmono->MeasuredSWL,1,"measured_swl","Monopole source
         sound power level (dB)"); CHKERRQ(ierr);
648 //   ierr=PetscBagRegisterReal(*bag,&pmono->Harmonic.phi,0,"mono_phi","Monopole source
        phase (radians)"); CHKERRQ(ierr);
649   ierr=PetscBagRegisterReal(*bag,&pmono->Location.X,0,"mono_origin_x","Monopole source
          origin along x-axis (m)"); CHKERRQ(ierr);
650   ierr=PetscBagRegisterReal(*bag,&pmono->Location.Y,0,"mono_origin_y","Monopole source
          origin along y-axis (m)"); CHKERRQ(ierr);
651   ierr=PetscBagRegisterReal(*bag,&pmono->Location.Z,0,"mono_origin_z","Monopole source
          origin along z-axis (m)"); CHKERRQ(ierr);
652
653   if (LoadFromFile)
654     {
655       PetscViewer bagviewer;
656       if (ierr=PetscViewerBinaryOpen(PETSC_COMM_WORLD,SourceParametersFile,
              FILE_MODE_READ,&bagviewer))
657   {
658     PetscPrintf(PETSC_COMM_WORLD,"Unable to load %s.\n",SourceParametersFile);
659     CHKERRQ(ierr);
660   }
661       ierr=PetscBagLoad(bagviewer,bag); CHKERRQ(ierr);
662       ierr=PetscViewerDestroy(bagviewer); CHKERRQ(ierr);
663       ierr=PetscBagGetData(*bag,pparams); CHKERRQ(ierr);
664       PetscPrintf(PETSC_COMM_WORLD,"Loaded monopole series source parameters:\n");
665       ierr=PetscBagView(*bag,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
666     }
667   return 0;
668 }
669
670 PetscErrorCode InitializeMonopoleSeriesSource(void *vpctx)
```

```
671 {
672   const PetscReal Wref=1E-12;
673   PetscErrorCode ierr;
674   app_ctx *pctx=(app_ctx*)vpctx;
675   parameters *param=pctx->params;
676   monopole_series_parameters *psource=(monopole_series_parameters*)pctx->psource;
677
678   /* Read in parameters from file */
679   FILE *fp=fopen(psource->InputFile,"r");
680   if (!fp)
681     {
682       PetscPrintf(PETSC_COMM_WORLD,"Unable to open %s for reading monopole series data.\
          n",psource->InputFile);
683       return 1;
684     }
685
686   psource->N=0;
687   PetscInt NAlloc=10;
688   PetscReal *SWL,*f,*deltaf;
689   PetscMalloc(sizeof(PetscReal)*NAlloc,&SWL);
690   PetscMalloc(sizeof(PetscReal)*NAlloc,&f);
691   PetscMalloc(sizeof(PetscReal)*NAlloc,&deltaf);
692
693   while(!feof(fp))
694     {
695       int rc;
696       float f1,f2,f3;
697       if (psource->CSVFile==PETSC_TRUE)
698   rc=fscanf(fp,"%f,%f,%f\n",&f1,&f2,&f3);
699       else
700   rc=fscanf(fp,"%f %f %f\n",&f1,&f2,&f3);
701       if (rc==3)
702   {
703     f[psource->N]=f1;
704     deltaf[psource->N]=f2;
705     SWL[psource->N]=f3;
706     if (psource->InputAWeighted==PETSC_TRUE)
707       SWL[psource->N] -= GetAWeighting(f[psource->N]);
708     PetscPrintf(PETSC_COMM_WORLD,"Input %5d: f= %g Hz deltaf=%g Hz SWL=%g dBA SWL=%g dB\
          n",psource->N,f[psource->N],deltaf[psource->N],SWL[psource->N]+GetAWeighting(f[
          psource->N]),SWL[psource->N]);
709     psource->N++;
710     if (psource->N>=NAlloc)
711       {
712         PetscInt newNAlloc = 2*NAlloc;
713         PetscReal *newSWL,*newf,*newdeltaf;
714         PetscMalloc(sizeof(PetscReal)*newNAlloc,&newSWL);
715         PetscMalloc(sizeof(PetscReal)*newNAlloc,&newf);
716         PetscMalloc(sizeof(PetscReal)*newNAlloc,&newdeltaf);
717         memcpy(newSWL,SWL,sizeof(PetscReal)*NAlloc);
```

```
718           memcpy(newf,f,sizeof(PetscReal)*NAlloc);
719           memcpy(newdeltaf,deltaf,sizeof(PetscReal)*NAlloc);
720           PetscFree(SWL);
721           PetscFree(f);
722           PetscFree(deltaf);
723           SWL=newSWL;
724           f=newf;
725           deltaf=newdeltaf;
726           NAlloc=newNAlloc;
727         }
728     }
729         else
730     {
731       PetscPrintf(PETSC_COMM_WORLD,"Error reading data from file %s\n",psource->InputFile)
              ;
732       fclose(fp);
733       return 1;
734     }
735       }
736
737     fclose(fp);
738
739     PetscInt i;
740
741     PetscReal rho=GetDensity(pctx->properties);
742     PetscReal c=GetSpeedOfSound(pctx->properties);
743
744     ierr=PetscMalloc(sizeof(harmonic_parameters)*psource->N,&(psource->Harmonic)); CHKERRQ
          (ierr);
745     ierr=PetscMalloc(sizeof(PetscReal)*psource->N,&(psource->deltaf)); CHKERRQ(ierr);
746
747     for(i=0;i<psource->N;i++)
748       {
749         PetscReal df=deltaf[i];
750         //        PetscReal df0=(m==0?0:deltaf[m-1]);
751         //        PetscReal df1=(m==params->N-1?0:deltaf[m]);
752         //        PetscReal df=0.5*(df0+df1);
753         PetscReal invscale=1.0/(2*M_PI*df);
754         /* If power W is known, the amplitude of the volumetric source is sqrt((2 c W)/(
                rho pi f^2)) */
755         PetscReal W = Wref*pow(10.0,0.1*SWL[i]);
756         psource->Harmonic[i].A = sqrt(2*W*c/rho/M_PI)/f[i]*invscale;
757         psource->Harmonic[i].f = f[i];
758         psource->Harmonic[i].phi = 0;
759         psource->Harmonic[i].Location.X=psource->Location.X;
760         psource->Harmonic[i].Location.Y=psource->Location.Y;
761         psource->Harmonic[i].Location.Z=psource->Location.Z;
762         psource->deltaf[i]=deltaf[i];
763         PetscPrintf(PETSC_COMM_WORLD,"Monopole series (as measured SWL[%d]=%g dB) source
              amplitude A[%d]=%g.\n",i,SWL[i],i,psource->Harmonic[i].A);
```

```
764       }
765
766     PetscFree(SWL);
767     PetscFree(f);
768     PetscFree(deltaf);
769
770     for(i=0;i<psource->N;i++)
771       { ierr=CalculateMeshLocation(&(psource->Harmonic[i].Location),pctx); CHKERRQ(ierr);
              }
772
773     return 0;
774 }
775
776 /*! Calculate source term contributions to RHS for a continuous, calibrated monopole
        source at time t. */
777 void MonopoleSeries(PetscReal t,Vec R,void *vpctx)
778 {
779     app_ctx *pctx=(app_ctx*)vpctx;
780     PetscInt i,j,k,d,dof,Nx,Ny,Nz,sx,sy,sz,mx,my,mz;
781     PetscScalar ****r;
782     Vec local;
783     monopole_series_parameters *params=(monopole_series_parameters*)pctx->psource;
784
785     assert(pctx);
786     assert(params);
787
788     PetscLogEventBegin(pctx->CalcSourceEvent,0,0,0,0);
789
790     /* Determine global mesh nodal extents */
791     DAGetInfo(pctx->da,PETSC_IGNORE,&Nx,&Ny,&Nz,
792         PETSC_IGNORE,PETSC_IGNORE,PETSC_IGNORE,
793         PETSC_IGNORE,PETSC_IGNORE,
794         PETSC_IGNORE,PETSC_IGNORE);
795
796     /* Get local grid boundaries, which are independent of dof */
797     DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
798
799     DAVecGetArrayDOF(pctx->da,R,&r);
800
801     int rank; MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
802
803     assert(params->Harmonic);
804     PetscInt m;
805     for(m=0;m<params->N;m++)
806       {
807         PetscReal df=params->deltaf[m];
808         //        PetscReal df0=(m==0?0:params->deltaf[m-1]);
809         //        PetscReal df1=(m==params->N-1?0:params->deltaf[m]);
810         //        PetscReal df=0.5*(df0+df1);
```

```
811         PetscReal Mag=2*M_PI*df*params->Harmonic[m].A*cos(2*M_PI*params->Harmonic[m].f*t)*
                pctx->invdx*pctx->invdy*pctx->invdz;
812         PetscInt kN=(TwoDimensional(pctx)?1:2);
813         for(k=0;k<kN;k++)
814     for(j=0;j<2;j++)
815       for(i=0;i<2;i++)
816         {
817           if ((params->Harmonic[m].Location.li+i>=sx && params->Harmonic[m].Location.li+i<
                  sx+mx) &&
818         (params->Harmonic[m].Location.lj+j>=sy && params->Harmonic[m].Location.lj+j<sy+my)
                &&
819         (params->Harmonic[m].Location.lk+k>=sz && params->Harmonic[m].Location.lk+k<sz+mz)
                )
820       {
821         r[params->Harmonic[m].Location.lk+k][params->Harmonic[m].Location.lj+j][params->
                Harmonic[m].Location.li+i][0] += params->Harmonic[m].Location.w[i][j][k]*Mag;
822       }
823         }
824       }
825     /* Release access to array */
826     DAVecRestoreArrayDOF(pctx->da,R,&r);
827
828     PetscLogEventEnd(pctx->CalcSourceEvent,0,0,0,0);
829
830 }
831
832 /*! Get frequency of calibrated monopole source */
833 PetscReal GetMonopoleSeriesLowestFrequency(void *vpctx)
834 {
835     PetscReal MinFrequency=1E6;
836     monopole_series_parameters *psource=(monopole_series_parameters*)vpctx;
837     PetscInt i,N;
838
839     for(i=0;i<psource->N;i++)
840       MinFrequency=(MinFrequency<psource->Harmonic[i].f?MinFrequency:psource->Harmonic[i].
              f);
841
842     return MinFrequency;
843 }
844
845 /*! Get frequency of calibrated monopole source */
846 PetscReal GetMonopoleSeriesHighestFrequency(void *vpctx)
847 {
848     PetscReal MaxFrequency=0;
849     monopole_series_parameters *psource=(monopole_series_parameters*)vpctx;
850     PetscInt i,N;
851
852     for(i=0;i<psource->N;i++)
853       MaxFrequency=(MaxFrequency>psource->Harmonic[i].f?MaxFrequency:psource->Harmonic[i].
              f);
```

```
854
855   return MaxFrequency;
856 }
857
858 /*! Get location of calibrated monopole source */
859 PetscReal GetMonopoleSeriesOrigin(void *vpctx,PetscInt d)
860 {
861   monopole_series_parameters *psource=(monopole_series_parameters*)vpctx;
862   PetscInt i;
863   PetscReal Sum=0.0,Denom=0.0;
864
865   assert(d>=0 && d<=2);
866   switch(d)
867     {
868     case 0:
869       for(i=0;i<psource->N;i++)
870   {
871     Sum += psource->Harmonic[i].Location.X; Denom ++;
872   }
873       break;
874     case 1:
875       for(i=0;i<psource->N;i++)
876   {
877     Sum += psource->Harmonic[i].Location.Y; Denom ++;
878   }
879       break;
880     case 2:
881       for(i=0;i<psource->N;i++)
882   {
883     Sum += psource->Harmonic[i].Location.Z; Denom ++;
884   }
885       break;
886     };
887   assert(Denom);
888   return Sum/Denom;
889 }
890
891 PetscErrorCode DestroyMonopoleSeriesSource(void *vpctx)
892 {
893   PetscErrorCode ierr;
894   app_ctx *pctx=(app_ctx*)vpctx;
895   parameters *param=pctx->params;
896   monopole_series_parameters *psource=(monopole_series_parameters*)pctx->psource;
897
898   if (psource->Harmonic) ierr=PetscFree(psource->Harmonic); CHKERRQ(ierr);
899   if (psource->deltaf) ierr=PetscFree(psource->deltaf); CHKERRQ(ierr);
900   psource->N=0;
901
902   return 0;
903 }
```

3.16 timemarch.h

```
1 #ifndef _TIMEMARCH_H_
2 #define _TIMEMARCH_H_
3
4 #include"param.h"
5 #include"ode.h"
6
7 /*! \file timemarch.h
8     \brief Time marching methods.
9
10 This header file declares the functions for time marching the solution from an initial
       value over time \f$t_0\f$ to \f$t_{max}\f$.
11 */
12
13 /*! March the solution from time \f$t_0\f$ to \f$t_{max}\f$. */
14 PetscErrorCode Solve(timemarch_parameters *params, /*!< [in] time marching parameters */
15           app_ctx *pctx,      /*!< [in] application context*/
16           Vec solution        /*!< [in,out] solution vector */
17           );
18
19 /*! Monitor the solution, including checkpointing as required. */
20 PetscErrorCode MonitorFxn(TS ts,
21          PetscInt iter,
22          PetscReal t,
23          Vec solution,
24          void* pctx);
25
26 /*! Deallocate storage created by MonitorFxn. */
27 PetscErrorCode MonitorFree(void*);
28
29 /*! Handle a signal, checkpointing as required. */
30 PetscErrorCode CheckpointHandler(int signum, void *vpctx);
31
32 /*! Handle additional signals of SIGUSR1 and SIGUSR2. */
33 void UserSignals(int);
34 #endif
```

3.17 timemarch.c

```
1 #include<assert.h>
2 #include<signal.h>
3 #include<petsc.h>
4
5 #include"config.h"
6 #include"timemarch.h"
7 #include"ode.h"
8 #include"ts.h"
9
```

```
10 app_ctx *active_ctx=NULL;
11
12 PetscErrorCode Solve(timemarch_parameters *params,
13          app_ctx *pctx,
14          Vec solution)
15 {
16   PetscErrorCode ierr;
17   TS ts;
18
19   assert(params);
20   assert(pctx);
21
22   active_ctx=pctx;
23
24   TSCreate(PETSC_COMM_WORLD,&ts);
25   TSSetProblemType(ts,TS_NONLINEAR);
26
27   /* Register our own time march methods */
28   TSRegister("RK4","","RK4Create",Create_RK4);
29
30   TSSetType(ts,TSRUNGE_KUTTA);
31   TSRKSetTolerance(ts,1e-4);
32
33   TSSetInitialTimeStep(ts,params->t0,params->dt);
34
35   TSSetDuration(ts,params->maxsteps,params->tmax);
36
37   SetupODE(pctx);
38
39   TSSetSolution(ts,solution);
40
41   TSSetRHSFunction(ts,FormODE,(void*)pctx);
42
43   TSSetFromOptions(ts);
44
45   PetscPushSignalHandler(CheckpointHandler,(void*)pctx);
46   /* PETSc currently ignores SIGUSR1/USR2 */
47   signal(SIGUSR1,UserSignals);
48   signal(SIGUSR2,UserSignals);
49
50   pctx->terminate=PETSC_FALSE;
51   pctx->checkpoint=PETSC_FALSE;
52
53   TSMonitorSet(ts,MonitorFxn,(void*)pctx,MonitorFree);
54
55   int NOptionsLeft=0;
56   PetscOptionsAllUsed(&NOptionsLeft);
57   if (NOptionsLeft>0)
58     {
59       PetscPrintf(PETSC_COMM_WORLD,"Some options were specified but unused!\n");
```

```
60        PetscOptionsLeft();
61        return 1;
62      }
63
64    ierr=TSSolve(ts,solution); CHKERRQ(ierr);
65
66    PetscInt TSMaxSteps,TSTimeStepNumber;
67    PetscReal TSMaxTime,TSTime;
68    TSGetDuration(ts,&TSMaxSteps,&TSMaxTime);
69    TSGetTimeStepNumber(ts,&TSTimeStepNumber);
70    TSGetTime(ts,&TSTime);
71
72    PetscPrintf(PETSC_COMM_WORLD,"Final time %g s, maximum time %g s,\n",TSTime,TSMaxTime)
          ;
73    PetscPrintf(PETSC_COMM_WORLD,"       step %d, maximum steps %d.\n",TSTimeStepNumber,
          TSMaxSteps);
74    PetscPrintf(PETSC_COMM_WORLD,"Average wall clock compute time of %g s per iteration.\n
          ",pctx->AvgSecondsPerIter);
75    PetscPrintf(PETSC_COMM_WORLD,"Average wall clock compute time of %g s per iteration
          per node.\n",pctx->AvgSecondsPerIter/((PetscReal)pctx->params->mesh.Nx*pctx->
          params->mesh.Ny*pctx->params->mesh.Nz));
76
77  //  TSView(ts,PETSC_VIEWER_STDOUT_WORLD);
78
79    PetscPopSignalHandler();
80    signal(SIGUSR1,0);
81    signal(SIGUSR2,0);
82
83    TSDestroy(ts);
84
85    active_ctx=NULL;
86
87    return 0;
88  }
89
90  PetscErrorCode MonitorFxn(TS ts,
91          PetscInt iter,
92          PetscReal t,
93          Vec solution,
94          void *vpctx)
95  {
96    app_ctx *pctx=(app_ctx*)(vpctx);
97    static PetscTruth FirstTime=PETSC_TRUE;
98    static PetscReal tstart=0;
99    static unsigned short LastTimePerCent=0;
100   static unsigned short LastIterPerCent=0;
101   static PetscLogDouble PrevTimeOfDay=0;
102   PetscInt IterInterval=100;
103   PetscLogDouble CurrentTimeOfDay,ElapsedTime;
104   PetscReal Norm1,Norm2;
```

```
105
106   assert(pctx);
107   assert(pctx->params);
108
109   if (FirstTime==PETSC_TRUE)
110     {
111       FirstTime=PETSC_FALSE;
112       tstart=pctx->params->tm.t0;
113       LastTimePerCent=0;
114        LastIterPerCent=0;
115     }
116
117   unsigned short IterPerCent=(unsigned short)(100.0*((float)iter)/((float)pctx->params->
          tm.maxsteps));
118   unsigned short TimePerCent=(unsigned short)(100.0*(t-tstart)/(pctx->params->tm.tmax-
          tstart));
119
120   PetscGetTime(&CurrentTimeOfDay);
121   if (CurrentTimeOfDay>=PrevTimeOfDay)
122     {
123       ElapsedTime=CurrentTimeOfDay-PrevTimeOfDay;
124     }
125   else
126     {
127       /* Midnight roll-over */
128       ElapsedTime=(86400-PrevTimeOfDay)+CurrentTimeOfDay;
129     }
130   PrevTimeOfDay=CurrentTimeOfDay;
131
132   if (iter>0)
133     pctx->AvgSecondsPerIter=(pctx->AvgSecondsPerIter*((PetscLogDouble)(iter-1))+
134             ElapsedTime)/((PetscLogDouble)iter);
135
136   IterInterval=(PetscInt)(600.0/pctx->AvgSecondsPerIter);
137
138   if (TimePerCent!=LastTimePerCent ||
139       IterPerCent!=LastIterPerCent ||
140       (IterInterval>0 && iter%IterInterval==0))
141     {
142       PetscPrintf(PETSC_COMM_WORLD,"iter. %6d: time %6.4E s, %3d%% of final time",iter,t
              ,TimePerCent);
143       LastTimePerCent=TimePerCent;
144       LastIterPerCent=IterPerCent;
145       int RemainingSeconds=0;
146       if (IterPerCent>TimePerCent)
147   {
148     PetscPrintf(PETSC_COMM_WORLD,", %3d%% of maximum number of iterations",IterPerCent);
149     RemainingSeconds=(int)(((PetscLogDouble)(pctx->params->tm.maxsteps-iter))*pctx->
            AvgSecondsPerIter);
150   }
```

```
151       else
152   {
153     RemainingSeconds=(int)(((PetscLogDouble)iter)*((pctx->params->tm.tmax-tstart)/(t-
          tstart)-1)*pctx->AvgSecondsPerIter);
154   }
155       if (RemainingSeconds>0)
156   {
157     PetscPrintf(PETSC_COMM_WORLD,", completion in about ");
158     unsigned int Days=RemainingSeconds/86400; RemainingSeconds -= Days*86400;
159     unsigned int Hours=RemainingSeconds/3600; RemainingSeconds -= Hours*3600;
160     unsigned int Minutes=RemainingSeconds/60; RemainingSeconds -= Minutes*60;
161     unsigned int Seconds=RemainingSeconds;
162     if (Days>0) PetscPrintf(PETSC_COMM_WORLD,"%2d d, ",Days);
163     if (Hours>0) PetscPrintf(PETSC_COMM_WORLD,"%2d h, ",Hours);
164     if (Minutes>0) PetscPrintf(PETSC_COMM_WORLD,"%2d m, ",Minutes);
165     PetscPrintf(PETSC_COMM_WORLD,"%2d s",Seconds);
166   }
167       PetscPrintf(PETSC_COMM_WORLD,".\n");
168
169     }
170
171   /* Update the tlast in parameters so that we can restart from last completed time step
        */
172   pctx->params->tm.t0=t;
173
174   int rank; MPI_Comm_rank(PETSC_COMM_WORLD,&rank);
175
176   if (t>=pctx->params->tm.tmax-pctx->params->tm.T && t<=pctx->params->tm.tmax)
177     {
178       /* Scatter pressure values from the solution to the local ground plane
179    representation */
180       if (!pctx->IntegrationStarted)
181   {
182     bc_parameters *bc=(bc_parameters*)pctx->pbc;
183     assert(bc);
184     if (pctx->j_lo>0 || TwoDimensional(pctx) || bc->ReflectiveGround)
185       {
186         /* Initialize vectors to zero */
187         VecSet(pctx->prms,0.0);
188         VecSet(pctx->I,0.0);
189         VecSet(pctx->currp,0.0);
190         VecSet(pctx->currw,0.0);
191       }
192
193     /* Record time at start of integration */
194     pctx->tstart=t;
195
196     pctx->dtlast=0;
197
198     /* Initialize sound power level */
```

```
199     pctx->SWL=0;
200
201     pctx->tHistory=NULL;
202     pctx->PowerHistory=NULL;
203
204     /* Estimate the number of integrations steps remaining */
205     if (!rank)
206       {
207         pctx->HistoryAllocLen=(PetscInt)(fmax(128,fmin(1024,1.25*(pctx->params->tm.tmax-
                pctx->tstart)/pctx->params->tm.dt)));
208         PetscMalloc(sizeof(PetscReal)*pctx->HistoryAllocLen,&pctx->tHistory);
209         PetscMalloc(sizeof(PetscReal)*pctx->HistoryAllocLen,&pctx->PowerHistory);
210         pctx->NHistory=0;
211         if (pctx->DoProbe)
212     PetscMalloc(sizeof(PetscReal)*pctx->HistoryAllocLen,&pctx->pHistory);
213       }
214   }
215       /* Integrate by trapezoidal rule with respect to time */
216
217       PetscScalar dt=(t==pctx->params->tm.tmax?0:(t-pctx->tlast));
218       PetscScalar tscale=0.5*(dt+pctx->dtlast);
219
220       double probe_value=0.0;
221
222       bc_parameters *bc=(bc_parameters*)pctx->pbc;
223       assert(bc);
224
225       if (pctx->j_lo>0 || TwoDimensional(pctx) || bc->ReflectiveGround || pctx->
              MaxAbsPressure)
226   {
227     /* Calculate RMS average of pressure */
228
229     /* Scatter current solution into curr */
230     VecScatterBegin(pctx->groundp,solution,pctx->currp,INSERT_VALUES,SCATTER_FORWARD);
231     VecScatterEnd(pctx->groundp,solution,pctx->currp,INSERT_VALUES,SCATTER_FORWARD);
232     VecScatterBegin(pctx->groundw,solution,pctx->currw,INSERT_VALUES,SCATTER_FORWARD);
233     VecScatterEnd(pctx->groundw,solution,pctx->currw,INSERT_VALUES,SCATTER_FORWARD);
234
235     /* Update max abs pressure */
236     if (pctx->MaxAbsPressure) VecPointwiseMaxAbs(pctx->pmaxabs,pctx->pmaxabs,pctx->currp
            );
237
238     PetscScalar *I,*prms,*currp,*currw,*lastp,*lastw;
239     PetscInt i,d,np,nw;
240
241     VecGetLocalSize(pctx->currp,&np);
242     VecGetLocalSize(pctx->currw,&nw);
243
244     assert(3*np==nw);
245
```

```
246     VecGetArray(pctx->I,&I);
247     VecGetArray(pctx->prms,&prms);
248     VecGetArray(pctx->currp,&currp);
249     VecGetArray(pctx->currw,&currw);
250
251     for(i=0;i<np;i++)
252       {
253         prms[i] += tscale*(currp[i]*currp[i]);
254       }
255
256     for(i=0;i<nw/3;i++)
257       for(d=0;d<3;d++)
258         {
259     I[i*3+d] += tscale*(currp[i]*currw[i*3+d]);
260         }
261
262
263     VecRestoreArray(pctx->currw,&currw);
264     VecRestoreArray(pctx->currp,&currp);
265     VecRestoreArray(pctx->prms,&prms);
266     VecRestoreArray(pctx->I,&I);
267
268     if (pctx->DoProbe)
269       {
270         double probe_out=0.0;
271         PetscScalar **p;
272         DAVecGetArray(pctx->dagp,pctx->currp,&p);
273
274         PetscInt sx,sy,sz,mx,my,mz;
275         /* Get local grid boundaries, which are independent of dof */
276         DAGetCorners(pctx->dagp,&sx,&sy,&sz,&mx,&my,&mz);
277
278         PetscInt i,j;
279         for(j=0;j<2;j++)
280     for(i=0;i<2;i++)
281       {
282         if ((pctx->probe_li+i>=sx && pctx->probe_li+i<sx+mx) &&
283       (pctx->probe_lj+j>=sy && pctx->probe_lj+j<sy+my))
284           {
285       probe_out += pctx->probe_w[i][j]*p[pctx->probe_lj+j][pctx->probe_li+i];
286           }
287       }
288         DAVecRestoreArray(pctx->dagp,pctx->currp,&p);
289         MPI_Reduce(&probe_out,&probe_value,1,MPI_DOUBLE,MPI_SUM,0,PETSC_COMM_WORLD);
290       }
291   }
292
293       Vec local;
294       PetscScalar ****s;
295
```

```
296       /* Borrow local vector representation from DA */
297       DAGetLocalVector(pctx->da,&local);
298
299       /* Gather from global solution to local, ensure ghosts are up-to-date */
300       DAGlobalToLocalBegin(pctx->da,solution,INSERT_VALUES,local);
301       DAGlobalToLocalEnd(pctx->da,solution,INSERT_VALUES,local);
302
303       /* Grab the array representation of the local solution */
304       DAVecGetArrayDOF(pctx->da,local,&s);
305
306       PetscInt i,j,k,sx,sy,sz,mx,my,mz;
307       DAGetCorners(pctx->da,&sx,&sy,&sz,&mx,&my,&mz);
308
309       PetscInt ilo=(pctx->i_lo>sx?pctx->i_lo:sx);
310       PetscInt jlo=(pctx->j_lo>sy?pctx->j_lo:sy);
311       PetscInt klo=(pctx->k_lo>sz?pctx->k_lo:sz);
312       PetscInt ihi=(pctx->i_hi<sx+mx?pctx->i_hi:sx+mx);
313       PetscInt jhi=(pctx->j_hi<sy+my?pctx->j_hi:sy+my);
314       PetscInt khi=(pctx->k_hi<sz+mz?pctx->k_hi:sz+mz);
315
316       PetscReal SWL=0.0;
317
318       /* ilo face */
319       if (pctx->i_lo>=sx && pctx->i_lo<sx+mx)
320   {
321     PetscReal Area=pctx->dy*pctx->dz;
322     i=pctx->i_lo;
323     for(j=jlo;j<jhi;j++)
324       for(k=klo;k<khi;k++)
325         {
326     PetscReal scale=1.0;
327     if (j==pctx->j_lo || j==pctx->j_hi) scale *= 0.5;
328     if (k==pctx->k_lo || k==pctx->k_hi) scale *= 0.5;
329     PetscReal I=-s[k][j][i][0]*s[k][j][i][1];
330    SWL += scale*Area*I;
331         }
332   }
333
334       /* ihi face */
335       if (pctx->i_hi>=sx && pctx->i_hi<sx+mx)
336   {
337     PetscReal Area=pctx->dy*pctx->dz;
338     i=pctx->i_hi;
339     for(j=jlo;j<jhi;j++)
340       for(k=klo;k<khi;k++)
341         {
342     PetscReal scale=1.0;
343     if (j==pctx->j_lo || j==pctx->j_hi) scale *= 0.5;
344     if (k==pctx->k_lo || k==pctx->k_hi) scale *= 0.5;
345     PetscReal I=+s[k][j][i][0]*s[k][j][i][1];
```

```
346     SWL += scale*Area*I;
347         }
348   }
349
350       /* jlo face */
351       if (pctx->j_lo>=sy && pctx->j_lo<sy+my)
352   {
353     PetscReal Area=pctx->dx*pctx->dz;
354     j=pctx->j_lo;
355     for(i=ilo;i<ihi;i++)
356       for(k=klo;k<khi;k++)
357         {
358     PetscReal scale=1.0;
359     if (i==pctx->i_lo || i==pctx->i_hi) scale *= 0.5;
360     if (k==pctx->k_lo || k==pctx->k_hi) scale *= 0.5;
361     PetscReal I=-s[k][j][i][0]*s[k][j][i][2];
362     SWL += scale*Area*I;
363         }
364   }
365
366       /* jhi face */
367       if (pctx->j_hi>=sy && pctx->j_hi<sy+my)
368   {
369     PetscReal Area=pctx->dx*pctx->dz;
370     j=pctx->j_hi;
371     for(i=ilo;i<ihi;i++)
372       for(k=klo;k<khi;k++)
373         {
374     PetscReal scale=1.0;
375     if (i==pctx->i_lo || i==pctx->i_hi) scale *= 0.5;
376     if (k==pctx->k_lo || k==pctx->k_hi) scale *= 0.5;
377     PetscReal I=+s[k][j][i][0]*s[k][j][i][2];
378     SWL += scale*Area*I;
379         }
380   }
381
382       /* klo face */
383       if (pctx->k_lo>=sz && pctx->k_lo<sz+mz)
384   {
385     PetscReal Area=pctx->dx*pctx->dy;
386     k=pctx->k_lo;
387     for(i=ilo;i<ihi;i++)
388       for(j=jlo;j<jhi;j++)
389         {
390     PetscReal scale=1.0;
391     if (i==pctx->i_lo || i==pctx->i_hi) scale *= 0.5;
392     if (j==pctx->j_lo || j==pctx->j_hi) scale *= 0.5;
393     PetscReal I=-s[k][j][i][0]*s[k][j][i][3];
394     SWL += scale*Area*I;
395         }
```

```
396   }
397
398       /* jhi face */
399       if (pctx->k_hi>=sz && pctx->k_hi<sz+mz)
400   {
401     PetscReal Area=pctx->dx*pctx->dy;
402     k=pctx->k_hi;
403     for(i=ilo;i<ihi;i++)
404       for(j=jlo;j<jhi;j++)
405         {
406     PetscReal scale=1.0;
407     if (i==pctx->i_lo || i==pctx->i_hi) scale *= 0.5;
408     if (j==pctx->j_lo || j==pctx->j_hi) scale *= 0.5;
409     PetscReal I=+s[k][j][i][0]*s[k][j][i][3];
410    SWL += scale*Area*I;
411         }
412   }
413
414       double SWLout=SWL,SWLin=0.0;
415       MPI_Reduce(&SWLout,&SWLin,1,MPI_DOUBLE,MPI_SUM,0,PETSC_COMM_WORLD);
416
417       if (!rank)
418   {
419     if (pctx->NHistory>=pctx->HistoryAllocLen)
420       {
421         PetscInt newHistoryAllocLen=pctx->HistoryAllocLen+1024;
422         PetscReal *newPowerHistory,*newtHistory,*newpHistory;
423         PetscMalloc(sizeof(PetscReal)*newHistoryAllocLen,&newtHistory);
424         PetscMalloc(sizeof(PetscReal)*newHistoryAllocLen,&newPowerHistory);
425         if (pctx->DoProbe) PetscMalloc(sizeof(PetscReal)*newHistoryAllocLen,&newpHistory
                );
426         memcpy(newtHistory,pctx->tHistory,sizeof(PetscReal)*pctx->NHistory);
427         memcpy(newPowerHistory,pctx->PowerHistory,sizeof(PetscReal)*pctx->NHistory);
428         if (pctx->DoProbe) memcpy(newpHistory,pctx->pHistory,sizeof(PetscReal)*pctx->
                NHistory);
429         PetscFree(pctx->tHistory);
430         PetscFree(pctx->PowerHistory);
431         if (pctx->DoProbe) PetscFree(pctx->pHistory);
432         pctx->tHistory=newtHistory;
433         pctx->PowerHistory=newPowerHistory;
434         if (pctx->DoProbe) pctx->pHistory=newpHistory;
435         pctx->HistoryAllocLen=newHistoryAllocLen;
436       }
437
438     pctx->tHistory[pctx->NHistory] = t;
439     pctx->PowerHistory[pctx->NHistory] = SWL;
440     /* Pick up probe value */
441     if (pctx->DoProbe)
442       {
443         pctx->pHistory[pctx->NHistory] = probe_value;
```

```
444         }
445       pctx->NHistory++;
446
447     }
448         pctx->SWL += tscale*SWLin;
449
450         DAVecRestoreArrayDOF(pctx->da,local,&s);
451
452         /* Return local vector representation to DA */
453         DARestoreLocalVector(pctx->da,&local);
454
455
456         /* Record last time of integration */
457         pctx->tlast=t;
458         pctx->dtlast=(t-pctx->tlast);
459
460         /* Record that RMS integration has been started */
461         pctx->IntegrationStarted=PETSC_TRUE;
462
463       }
464
465     if (pctx->params->tm.savefreq>0 && iter%pctx->params->tm.savefreq==0)
466       {
467         PetscPrintf(PETSC_COMM_WORLD,"Saving solution for iteration %d.\n",iter);
468         //SaveVTKByIter(iter,solution,pctx);
469         SaveCheckpoint(solution,pctx);
470       }
471
472     if (pctx->checkpoint)
473       {
474         PetscPrintf(PETSC_COMM_WORLD,"Saving checkpoint solution.\n");
475         SaveCheckpoint(solution,pctx);
476       }
477
478     pctx->checkpoint=PETSC_FALSE;
479
480     if (pctx->terminate)
481       {
482         PetscPrintf(PETSC_COMM_WORLD,"Terminating.\n");
483         PetscPopSignalHandler();
484         signal(SIGUSR1,0);
485         signal(SIGUSR2,0);
486         TSMonitorCancel(ts);
487         return 1;
488       }
489
490     if (pctx->SaveTransientPressure)
491       {
492         SaveVTKByIter(iter,t,solution,pctx);
493       }
```

```
494
495   if (pctx->SaveTransient)
496     {
497       SaveFullVTKByIter(iter,t,solution,pctx);
498     }
499
500   pctx->terminate=PETSC_FALSE;
501
502   PetscReal Min,Max;
503   VecMin(solution,PETSC_IGNORE,&Min);
504   VecMax(solution,PETSC_IGNORE,&Max);
505   if (Min!=Min || Max!=Max)
506     {
507       /* The solver has diverged! */
508       PetscPrintf(PETSC_COMM_WORLD,"The solver appears to have diverged (NaN in solution
              )!\n");
509       return 1;
510     }
511
512   return 0;
513 }
514
515 PetscErrorCode MonitorFree(void*vpctx)
516 {
517   app_ctx *pctx=(app_ctx*)(vpctx);
518
519   return 0;
520 }
521
522 PetscErrorCode CheckpointHandler(int signum, void *vpctx)
523 {
524   app_ctx *pctx=(app_ctx*)(vpctx);
525
526   switch (signum)
527     {
528     case SIGTERM:
529     case SIGINT:
530     case SIGABRT:
531     case SIGALRM:
532       pctx->checkpoint=PETSC_TRUE;
533       pctx->terminate=PETSC_TRUE;
534       break;
535     case SIGUSR1:
536     case SIGUSR2:
537     case SIGHUP:
538       pctx->checkpoint=PETSC_TRUE;
539       break;
540     }
541   return 0;
542 }
```

```
543
544 void UserSignals(int signum)
545 {
546   if (active_ctx)
547     {
548       switch (signum) {
549       case SIGUSR1:
550   active_ctx->checkpoint=PETSC_TRUE;
551   break;
552       case SIGUSR2:
553   active_ctx->checkpoint=PETSC_TRUE;
554   active_ctx->terminate=PETSC_TRUE;
555   break;
556       }
557     }
558 }
```

3.18 ts.h

```
1 #ifndef _TS_H_
2 #define _TS_H_
3
4 #include"petsc.h"
5
6 /*! \file ts.h
7     \brief Extensions to PETSc TS methods
8
9 Declares functions for creating and extending PETSc TS time marching methods.
10 */
11
12 /*! A fixed time-step, fourth-order Runge-Kutta time marching method. */
13 PetscErrorCode Create_RK4(TS ts);
14
15 #endif
```

3.19 ts.c

```
1 #define PETSCTS_DLL
2
3 /*
4        Code for Timestepping with explicit Euler.
5 */
6 #include "ts.h"
7 #include "private/tsimpl.h"                /*I   "petscts.h"   I*/
8
9 /*! \brief Parameters for RK4
10
11   Parameters for computing fourth-order Runge-Kutta time marching */
12 typedef struct {
```

```
13   Vec f,work,sol_np1;
14 } TS_RKN;
15
16 static PetscErrorCode SetUp_RK4(TS ts)
17 {
18   TS_RKN        *RK = (TS_RKN*)ts->data;
19   PetscErrorCode ierr;
20
21   PetscFunctionBegin;
22   ierr = VecDuplicate(ts->vec_sol,&RK->f);CHKERRQ(ierr);
23   ierr = VecDuplicate(ts->vec_sol,&RK->work);CHKERRQ(ierr);
24   ierr = VecDuplicate(ts->vec_sol,&RK->sol_np1);CHKERRQ(ierr);
25   PetscFunctionReturn(0);
26 }
27
28 static PetscErrorCode Step_RK4(TS ts,PetscInt *steps,PetscReal *ptime)
29 {
30   const PetscReal a[4]={0,0.5,0.5,1};
31   const PetscReal b[4]={1.0/6.0,1.0/3.0,1.0/3.0,1.0/6.0};
32   const PetscReal c[4]={0,0.5,0.5,1};
33   TS_RKN        *RK = (TS_RKN*)ts->data;
34   Vec             sol = ts->vec_sol, f = RK->f, work = RK->work, sol_np1 = RK->sol_np1;
35   PetscErrorCode ierr;
36   PetscInt        i,j,max_steps = ts->max_steps;
37
38   PetscFunctionBegin;
39   *steps = -ts->steps;
40   ierr = TSMonitor(ts,ts->steps,ts->ptime,sol);CHKERRQ(ierr);
41
42   for (i=0; i<max_steps; i++) {
43     PetscReal dt = ts->time_step;
44
45     if (ts->ptime+dt>ts->max_time) dt = ts->max_time-ts->ptime;
46
47     ierr = TSPreStep(ts);CHKERRQ(ierr);
48     //     ts->ptime += dt;
49     ierr = VecCopy(sol,sol_np1); CHKERRQ(ierr);
50     ierr = VecCopy(sol,work); CHKERRQ(ierr);
51     ierr = TSComputeRHSFunction(ts,ts->ptime,work,f);CHKERRQ(ierr);
52     ierr = VecAXPY(sol_np1,b[0]*dt,f);CHKERRQ(ierr);
53     for(j=1;j<4;j++)
54       {
55   ierr = VecWAXPY(work,a[j]*dt,f,sol);
56   ierr = TSComputeRHSFunction(ts,ts->ptime+c[j]*dt,work,f);CHKERRQ(ierr);
57   ierr = VecAXPY(sol_np1,b[j]*dt,f);
58       }
59     ts->ptime += dt;
60     ts->steps++;
61     ierr = VecCopy(sol_np1,sol); CHKERRQ(ierr);
62     ierr = TSPostStep(ts);CHKERRQ(ierr);
```

```
63      ierr = TSMonitor(ts,ts->steps,ts->ptime,sol);CHKERRQ(ierr);
64      if (ts->ptime >= ts->max_time) break;
65    }
66
67    *steps += ts->steps;
68    *ptime  = ts->ptime;
69    PetscFunctionReturn(0);
70  }
71
72  static PetscErrorCode Destroy_RK4(TS ts)
73  {
74    TS_RKN        *RK = (TS_RKN*)ts->data;
75    PetscErrorCode ierr;
76
77    PetscFunctionBegin;
78    if (RK->f) {ierr = VecDestroy(RK->f);CHKERRQ(ierr);}
79    if (RK->work) {ierr = VecDestroy(RK->work);CHKERRQ(ierr);}
80    if (RK->sol_np1) {ierr = VecDestroy(RK->sol_np1);CHKERRQ(ierr);}
81    ierr = PetscFree(RK);CHKERRQ(ierr);
82    PetscFunctionReturn(0);
83  }
84
85  static PetscErrorCode SetFromOptions_RK4(TS ts)
86  {
87    PetscFunctionBegin;
88    PetscFunctionReturn(0);
89  }
90
91  static PetscErrorCode View_RK4(TS ts,PetscViewer viewer)
92  {
93    PetscFunctionBegin;
94    PetscFunctionReturn(0);
95  }
96
97  PetscErrorCode PETSCTS_DLLEXPORT Create_RK4(TS ts)
98  {
99    TS_RKN        *RK;
100   PetscErrorCode ierr;
101
102   PetscFunctionBegin;
103   ts->ops->setup           = SetUp_RK4;
104   ts->ops->step            = Step_RK4;
105   ts->ops->destroy         = Destroy_RK4;
106   ts->ops->setfromoptions  = SetFromOptions_RK4;
107   ts->ops->view            = View_RK4;
108
109   ierr = PetscNewLog(ts,TS_RKN,&RK);CHKERRQ(ierr);
110   ts->data = (void*)RK;
111
112   PetscFunctionReturn(0);
```

```
113 }
```

References

[1] International organization for standarization - acoustics: Attenuation of sound during propagation outdoors (iso 9613-2). pages 1–17, Dec 1996.

[2] Government of ontario - ontario's long-term energy plan. pages 12–34, Nov 2010. Available from: http://www.mei.gov.on.ca/en/pdf/MEI_LTEP_en.pdf.

[3] Ontario chief medical officer of health - the potential health impact of wind turbines. May 2010. Available from: http://www.health.gov.on.ca/en/public/publications/ministry_reports/wind_turbine/wind_turbine.pdf.

[4] Southpoint wind - the southpoint wind 1400mw offshore wind energy project. pages 14–112, Feb 2010. Available from: http://www.southpointwind.com/files/SPW_Draft_Project_Description_Report_1400MW.pdf.

[5] Southpoint wind - the southpoint wind 30mw offshore wind energy project. pages 15–84, Feb 2010. Available from: http://www.southpointwind.com/files/SPW_Draft_Project_Description_Report_30MW.pdf.

[6] Government of ontario - environmental protection act 359-09. pages 1–10, Feb 2011. Available from: http://www.e-laws.gov.on.ca/html/regs/english/elaws_regs_090359_e.htm.

[7] Independent electricity system operator - about the ieso. 2011. Available from: http://www.ieso.ca/imoweb/about/about_the_ieso.asp.

[8] Independent electricity system operator - map. 2011. Available from: http://www.canaxenergy.com/images/chart_onneighboring.gif.

[9] Independent electricity system operator - who we are. 2011. Available from: http://ieso.ca/imoweb/siteShared/whoweare.asp?sid=md.

[10] Independent electricity system operator - wind power in ontario. page 1, Feb 2011. Available from: http://www.ieso.ca/imoweb/marketdata/windpower.asp.

[11] Leamington post - turbine issue dead in the water? pages 1–2, Feb 2011. Available from: http://www.leamingtonpostandshopper.com/ArticleDisplay.aspx?e=2536250&archive=true.

[12] National renewable energy laboratory - integrating wind and solar on the grid-nrel analysis leads the way. Continuum Magazine, 2011. Available from: `http://www.nrel.gov/continuum/integrating_wind_solar.cfm`.

[13] Norfolk county - erie shores wind farm. Signage, March 2011.

[14] Ontario power authority - homepage. Feb 2011. Available from: `http://www.powerauthority.on.ca`.

[15] Toronto western hospital - wave. Feb 2011. Available from: `http://www.usra.ca/files/images/wave%2520frequency.jpg`.

[16] J.F. Ainslie and J. Scott. Theoretical modelling of noise generated by wind turbines. *Wind Engineering*, 14(1 , 1990):9–14, 1990. Available from: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0025018536&partnerID=40&md5=86132855a4fa78b08bc6ccb1790cdc3f`.

[17] Keith Attenborough. Acoustical characteristics of rigid fibrous absorbents and granular materials. *Acoustical Society of America*, 73(3):785–799, 1983. Available from: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0020717329&partnerID=40&md5=c0962b8b7d105a6234098d7974cf2ec8`.

[18] EF Baerwald, GH D'Amours, BJ Klug, and RMR Barclay. Barotrauma is a significant cause of bat fatalities at wind turbines. *Current Biology*, 18(16):R695–R696, 2008. Available from: `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=18063636580362992522related:irtJvMPtrvoJ`.

[19] William K. Blake. *Mechanics of Flow-Induced Sound and Vibration - Vol. II Complex Flow-Structure Interactions*. Academic Press, 1986.

[20] Thomas F. Brooks and Michael A. Marcolini. Airfoil tip vortex formation noise. *American Institute of Aeronautics and Astronautics*, 24(2):246–252, 1986. Available from: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0022660745&partnerID=40&md5=a6e1f1f0bb3f42733b18c2879dcdd2e0`.

[21] Thomas F. Brooks, D. Stuart Pope, and Michael A. Marcolini. Airfoil self-noise and prediction. Technical report, NASA, 1989.

[22] Rui Cheng, Philip J. Morris, and Kenneth S. Brentner. A 3d parabolic equation method for wind turbine noise propagation in moving inhomogeneous atmosphere. In *12th AIAA/CEAS Aeroacoustics Conference (27th AIAA Aeroacoustics Conference)*, Cambridge, Massachusetts, May 2006.

[23] M.G. A. Dassen, R. Parchen, J. Bruggeman, and F. Hagg. Wind tunnel measurements of the aerodynamic noise of blade sections. In J.L. Tsipouridis, editor, *Proc. of the European Wind Energy Association Conf. & Exhibition*, volume I, pages 791–798, Thessaloniki, October 1994.

[24] Robert Doyle. Understanding ontario's electricity system. pages 1–12, Sep 2010. Available from: http://thecesh.com/wp-content/uploads/2010/09/IESO_CESH_Sept13_10.pptx.

[25] M.R. Fink and D.A. Bailey. Airframe noise reduction studies and clean-airframe noise investigation. Technical report, NASA, 1980.

[26] Jens Forssn, Martin Schiff, Eja Pedersen, and Kerstin Persson Waye. Wind turbine noise propagation over flat ground: Measurements and predictions. *Acta Acustica United with Acustica*, 96:753–760, 2010.

[27] Stephen D. Gedney. *Introduction to the Finite-Difference Time-Domain (FDTD) Method for Electromagnetics*. Morgan & Claypool, 2011.

[28] Paul Gipe and James Murphy. Ontario landowner's guide to wind energy. Technical report, Ontario Sustainable Energy Association, 2005. Available from: http://www.wind-works.org/articles/OSEA-Landowners-2005-r1-v3.pdf.

[29] Neil Glenister. Frequency weighting equations, 1998. Available from: http://www.cross-spectrum.com/audio/weighting.html.

[30] Lee Greenberg. Ontario wins court round in wind-power fight, March 2011. Available from: http://www.ottawacitizen.com/health/Ontario+wins+court+round+wind+power+fight/4379558/story.html#ixzz1FZzG6h8W.

[31] Ferdinand W. Grosveld. Prediction of broadband noise from horizontal axis wind turbines. *Propulsion and Power*, 1(4):292–299, 1985. Available from: http://www.scopus.com/inward/record.url?eid=2-s2.0-0022098313&partnerID=40&md5=33ecc436ecb46ec12ca938327ea8697f.

[32] G. Guidati, R. Barcei, and S. Wagner. An investigation of blade-tower-interaction noise (bti) for horizontal axis wind turbines in upwind and downwind configuration first step towards modeling of aeroelastic effects. In *Proc. of the 8th IEA Symposium, Joint Action, Aerodynamics of Wind Turbines*, Lyngby, November 1994.

[33] JP Harrison. Disconnect between turbine noise guidelines and health authority recommendations. pages 1–3, 2008. Available from: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=related:PhCsUMrkj1EJ.

[34] Erich Hau, Jens Langenbrinck, Wolfgang Palz, and Commission of the European Communities. *WEGA: Large Wind Turbines*. Springer, 1993.

[35] Alan S. Hersh, Paul T. Soderman, and Richard E. Hayden. Investigation of acoustic effects of leading-edge serrations on airfoils. *Aircraft*, 11(4):197–202, 1974. Available from: http://www.scopus.com/inward/record.url?eid=2-s2.0-0016047959&partnerID=40&md5=c3e283b3be3da131a08291334a4565f2.

[36] Brian Howe, Bill Gastmeier, and Nick McCabe. Wind turbines and sound: Review and best practice guidelines. Feb 2007. Available from: `http://www.canwea.ca/images/uploads/File/CanWEA_Wind_Turbine_Sound_Study_-_Final.pdf`.

[37] M.S. Howe. On the generation of side-edge flap noise. *Sound and Vibration*, 80(4):555–573, 1982. Available from: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0020087466&partnerID=40&md5=c8c4bcfaefb706d460f22d5eeb3e45bc`.

[38] M.S. Howe. Noise produced by a sawtooth trailing edge. *Acoustical Society of America*, 90(1):482–487, 1991. Available from: `http://www.scopus.com/inward/record.url?eid=2-s2.0-0025806981&partnerID=40&md5=8af5a2e1c3b5056ed72b4b82f6ab5f36`.

[39] Allan Jenkins. Wind generation activities and issues. pages 1–15, May 2008. Available from: `http://www.ieso.ca/imoweb/pubs/consult/windpower/wpsc-20080514-Item7_IESO_Integration.pdf`.

[40] L.E. Kinsler, A.R. Frey, A.B. Coppens, and J.V. Sanders. *Fundamentals of Acoustics*. Wiley and Sons, New York, 1987.

[41] L.D. Landau and E.M. Lifshitz. *Fluid Mechanics*. Pergamon, 1987.

[42] G. Leloudas, W.J. Zhu, J.N. Srensen, W.Z. Shen, and S. Hjort. Prediction and reduction of noise from a 2.3mw wind turbine. In *The Science of Making Torque from Wind*, 75, Copenhagen, Denmark, 2007. Technical University of Denmark, IOP Publishing.

[43] Chris Loken, Daniel Gruner, Leslie Groer, Richard Peltier, Neil Bunn, Michael Craig, Teresa Henriques, Jillian Dempsey, Ching-Hsing Yu, Joseph Chen, L. Jonathan Dursi, Jason Chong, Scott Northrup, Jaime Pinto, Neil Knecht, and Ramses van Zon. Scinet: Lessons learned from building a power-efficient top-20 system and data centre. In *Journal of Physics: Conference Series 256*, High Performance Computing Symposium. IOP Publishing, 2010.

[44] Lowson. Assessment and prediction of wind turbine noise. pages 1–59, December 1992. Flow Solutions Report 92/19, ETSU W/13/00284/REP.

[45] Philip M. Morse and K. Uno Ingard. *Theoretical Acoustics*. Princeton University Press, 1968.

[46] Michael Moser. *Engineering acoustics: an introduction to noise control*. 2009. Available from: `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=2608532719430273464related:uAmgDrBeMyQJ`.

[47] W Musial and S Butterfield. Future for offshore wind energy in the united states. *EnergyOcean Proceedings, June 2004, Palm Beach Florida, USA*, pages 1–5, 2004. Available from: `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=11183622522205337476related:hHcbOm41NJsJ`.

[48] P.A. Nelson and C.L. Morfey. Aerodynamic sound production in low speed flow ducts. *Sound and Vibration*, 79(2):263–289, 1981. Available from: http://www.scopus.com/inward/record.url?eid=2-s2.0-0019634767&partnerID=40&md5=1a3be59732849d0726f88cc7084040a2.

[49] T. Nomura, K. Takagi, and S. Sato. Finite element simulation of sound propagation concerning meteorological conditions. *International Journal for Numerical Methods in Fluids*, 64:1296–1318, October 2010.

[50] VE Ostashev, DK Wilson, L Liu, DF Aldridge, NP Symons, and D Marlin. Equations for finite-difference, time-domain simulation of sound propagation in moving inhomogeneous media and numerical implementation. *Acoustical Society of America*, 117:503–517, 2005. Available from: http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=18132585892545145772related:rKtpeMzio_sJ.

[51] Robert W. Paterson, Roy K. Amiet, and C. Lee Munch. Isolated airfoil – tip vortex interaction noise. In *AIAA 12th Aerospace Sciences Meeting*. American Institute of Aeronautics and Astronautics, 1974.

[52] JN Pinder. Mechanical noise from wind turbines. *Wind Engineering*, 16(3):158–168, 1992.

[53] John M. Prospathopoulos and Spyros G. Voutsinas. Noise propagation issues in wind energy applications. *American Society of Mechanical Engineers*, 127:234–241, May 2005.

[54] S.W. Rienstra and A. Hirschberg. *An Introduction to Acoustics*. Eindhoven University of Technology, 2011.

[55] Siemens. Siemens wind turbine swt-2.3-101. Available from: http://www.energy.siemens.com/fi/en/power-generation/renewables/wind-power/wind-turbines/swt-2-3-101.htm#content=Technical%20Specification.

[56] Peter J. Skirrow. Sound weighting curves. Feb 2011. Available from: http://en.wikipedia.org/wiki/File:Acoustic_weighting_curves.svg.

[57] John W. Slater. Examining spatial (grid) convergence, 2008. Available from: http://www.grc.nasa.gov/WWW/wind/valid/tutorial/spatconv.html.

[58] Eunkok Son, Hyunjung Kim, Hogeon Kim, Wooyoung Choi, and Soogab Lee. Integrated numerical method for the prediction of wind turbine noise and the long range propagation. *Current Applied Physics*, 10:316–319, 2010.

[59] S. Wagner, R. Barcei, and G. Guidati. *Wind Turbine Noise*. Springer, 1996.

[60] L. Wang and D.J. Mavriplis. Implicit solution of the unsteady euler equations for high-order accurate discontinuous galerkin discretizations. *Computational Physics*, 225(2):1994–2015, 2007. Available from: http://www.scopus.com/inward/record.url?eid=2-s2.0-34547522173&partnerID=40&md5=03966d47f627d070c6b7f88d30f64a63.

[61] Tim Weis. Wind power realities fact sheet. *Pembina Institute*, Dec 2009. Available from: `http://www.pembina.org/pub/1943`.

[62] J E Ffowcs Williams and L H Hall. Aerodynamic sound generation by turbulent flow in the vicinity of a scattering half plane. *Fluid Mechanics*, 40(04):505, 1970. Available from: `http://www.journals.cambridge.org/abstract_S0022112070000368`.

[63] DK Wilson and L Liu. Finite-difference, time-domain simulation of sound propagation in a dynamic atmosphere. 2004. Available from: `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=4630605976239540063related:XweP9Jk3Q0AJ`.

[64] Wei Jun Zhu, Nicolai Heilskov, Wen Zhong Shen, and Jens Nrkr Srensen. Modeling of aerodynamically generated noise from wind turbines. *Solar Energy Engineering*, 127:517–528, 2005.

[65] Wei Jun Zhu, Nrr Srensen, and Wen Zhong Shen. An aerodynamic noise propagation model for wind turbines. *Wind Engineering*, 29:129–143, 2005.

Acronyms

CN Crank Nicholson. 11, 13, 26

CV Control Volume. 16

DOF Degrees of Freedom. 15

FDTD Finite-Difference, Time-Domain. 1, 12, 13, 15, 16, 19, 37, 57

FEM Finite Element Method. 1

FFP Fast Fourier Program. 12

FVM Finite Volume Method. 1

GCI Grid Convergence Index. 43

GCS Grid Convergence Study. 39, 40, 42–44

HO Hydro One. 3

HPC High Performance Computing. 37

Hz hertz. 9

IESO Independant Electricity System Operator. 3

IMO Independant Electricity Market Operator. 3

ISO International Organization for Standardization. 8–11

kHz kilohertz. 9

MW megawatts. 2–4

OPA Ontario Power Authority. 3

OPG Ontario Power Generation. 3

PE parabolic Equation. 11–13

PETSc Portable Extensible Toolkit for Scientific Computation. 26, 37

PML Perfectly Matched Layer. 57

RMS Root Mean Square. 30, 45

SIL Sound Intensity Level. 38

SPL Sound Pressure Level. 2, 28, 37, 38, 45–47, 49, 53, 54

SWL Sound Power Level. 28, 37, 44, 45