THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF AEROSPACE ENGINEERING

THE NONLINEAR PROPAGATION OF WAVES THROUGH A TURBULENT ATMOSPHERE

EVAN WARD
Spring 2011

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Aerospace Engineering
with honors in Aerospace Engineering

Reviewed and approved* by the following:

Philip J. Morris
Boeing/A. D. Welliver Professor of Aerospace Engineering
Thesis Supervisor and Honors Adviser

Dennis K. McLaughlin
Professor of Aerospace Engineering
Faculty Reader

George A. Lesieutre
Professor and Head of Aerospace Engineering
Department Head

* Signatures are on file in the Schreyer Honors College.

Abstract

The nonlinear propagation of waves in a turbulent atmosphere can be described by solutions to the generalized Burgers Equation. A frequency domain algorithm is used to obtain these solutions. Additional terms that describe the effects of fluctuations in temperature and velocity can be included. A random realization of the wind and temperature profile in the atmosphere is chosen for each simulation. Many simulations are then combined to produce the average effect of the turbulence on the propagation. The turbulent fluctuations are calculated from a von Karman spectrum. A computer code was developed that implemented this algorithm to generate the pressure time history at the final range from the pressure time history at the starting range. The new solution method is applied to flattop and ramp sonic boom waveforms and to broadband jet noise. The inclusion of the atmospheric turbulence changes the waveform by adding energy at higher frequencies.

Contents

List of Figures

Nomenclature

a	Correlation length
BBF	Blackstock Bridging Function
c_0	Nominal speed of sound, m/s
c_p	Specific heat
c'	Turbulent fluctuation in speed of sound, m/s
C_T	Structure parameter for temperature variation
C_v	Structure parameter for velocity fluctuations
FFTW	Fastest Fourier Transform in the West (C library)
$F(k_n)$	Turbulence spectrum as a function of wave number
g	Gavitational acceleratoin
H	Vertical heat flux at the ground, W/m^2
ICAO	International Civil Aviation Organization
k	Wave number
k	von Karman's constant ($k = 0.4$)
K_0	$K_0 = 2\pi/L$, where L is the size of the largest eddies
KZK	Khokhlov-Zabolotskaya-Kuznetsov Equation
LSAF	Boeing Low Speed Aeroacoustic Facility
m	Wave parameter
p	Acoustic Pressure in the time domain, Pa

PBL	Planetary Boundary Layer
$\tilde{p}$	Fourier transform of acoustic pressure. Pa s
$\tilde{p}_n$	Pressure at the nth frequency
q	Pressure squared in the time domain, Pa^2
$\tilde{q}$	Fourier transform of pressure squared, Pa^2 s
r	Range from source, m
T_0	Nominal temperature
U	Real part of Fourier transform of pressure squared, Pa^2 s
$\vec{u}_\perp$	Flow of medium perpendicular to porpagation direction, m/s
u_x	Flow of medium in direction of propagation, m/s
V	Imaginary part of Fourier transform of pressure squared, Pa^2 s
w_*	Velocity scaling parameter
X	Real part of Fourier transform of pressure, Pa s
Y	Imaginary part of Fourier transform of pressure, Pa s
z	Altitude, m
z_1	Height of the PBL, m
α	Atmospheric absorption, neper/m
α_n	Random angle between 0 and $2pi$, rad
α'	Complex atmospheric absorption and dispersion
β	Coefficient of non-linearity
β_d	Atomspheric dispersion, 1/m
δ	Diffusivity of sound
Δk	Wave number step
ϵ	Nonlinearity Coefficients

Γ	The Gamma function
μ	Refractive index
μ_0^2	Room mean square value of the turbulence spectrum
ω	angular frequency, rad/s
ρ	Density, kg/m^3
ρ_0	Nominal density, kg/m^3
τ	Retarded time, s
τ	Surface stress, Pa/m^2
θ_*	Temperature scaling parameter

Acknowledgments

The author would like to thank Dr. K. Viswanathan for providing Low Speed Aeroacoustic Facility data. The author would also like to thank Dr. P. Morris for continuous advice and guidance.

1. Introduction

A recently developed frequency domain Burgers Equation solver[1] has been demonstrated to be robust, accurate and efficient for the propagation of sonic booms and broadband jet noise. It uses fewer data points in the calculations than similar methods to achieve converged results. However it has not been used for propagation through a nonuniform, turbulent atmosphere.

Past work with turbulence has used a variety of methods to simulate turbulence in a computationally efficient manner. The calculation of the changes in atmospheric properties may be accomplished by summing many distinct Fourier modes of the turbulence spectrum. Several papers use a Gaussian spectrum in their turbulence calculations[2, 3, 4]. The Gaussian spectrum is less expensive computationally but it does not include energy at higher wave numbers[5]. The energy at low wave numbers dominates the spectrum, so a Gaussian approximation is accurate for those lower wave numbers.

The von Karman spectrum is also widely used in calculating turbulent fields[6, 7]. It includes more energy at higher wave numbers and is therefore more accurate over a wider range. The increased accuracy comes at the cost of increased computational complexity and increased difficulty in choosing representative parameters for the spectrum. Both the von Karman and Gaussian spectra assume homogeneous and isotropic turbulence, which is rarely the case in the real atmosphere[5].

The homogeneous assumption can be relaxed if the spectrum parameters are allowed to vary with altitude. This is usually the case in the real atmosphere because the outer scales of turbulence increase with altitude. The change in turbulence with altitude can be modeled using the Planetary Boundary Layer (PBL). Above the PBL there is no turbulence, while the PBL itself is broken into three sections based on their characteristics. Between the ground and a tenth of the PBL height is the surface layer, which is characterized by heat convecting from the ground and mechanical wind shear. At the top of the PBL, greater than 80% of its height, is the inversion layer, which is characterized by a transition to the turbulence free atmosphere above the PBL. The most important layer for sonic boom propagation is the layer in the middle: the mixed layer. The mixed layer extends from 10% to 80% of the PBL height and is characterized by relatively constant mean velocity and temperature.[8]

Past work with turbulence has taken several different approaches to include the effects of

turbulence in sonic boom propagation including caustics, folding and turbules. Caustics are produced through the focusing effects of refraction. In a turbulent atmosphere the index of refraction changes with position and results in a complex web of caustics. Rays are then traced from the source to the receiver in two dimensions using a nonlinear transport equation. At a caustic a wave's peak overpressure is greatly increased. After propagation, the waves are summed together at the receiver to produce the resulting waveform. This approach was used by Blanc-Benon et al.[3] to propagate spark produced N waves.

Another approach that uses caustics is the folding of the sonic boom wavefront. At the cusp of a caustic the wavefront folds in on itself so there are three weaker booms propagating forward instead of one. This process repeats itself as the boom travels so that by the time it reaches the receiver it has been folded many times and is actually the sum of many micro-shocks. The caustic cusps necessary for folding are induced by turbulent changes in the refractive index, which causes the wavefront to bend and focus. This method was used by Pierce and Maglieri[4] to explain the effects of turbulence on rise time and peak overpressure.

A different method uses turbules of specific sizes to simulate a turbulent atmosphere. Between the source and the receiver a Monte Carlo method is used to generate turbules with random positions and sizes. Then for each frequency a ray is traced through the computational region. Every time the ray intersects a turbule its amplitude and phase are changed and it is scattered at an angle determined by the refractive index. The changes induced for each frequency are then compiled together and used to describe the net effect of turbulence. This approach was used by Boulanger et al.[7] to propagate a sonic boom.

All of these methods are two or three dimensional and require complex calculations (determining caustics or turbules) before they can be applied to a specific waveform. This make these methods very expensive computationally so most have only been used to propagate waves over short distances. The goal of this thesis is to develop an accurate model some of the effects of turbulence using a one dimensional equation.

2. Algorithm

2.1. Frequency Domain Burgers Equation

The generalized Burgers Equation is given as,[1]

$$\frac{\partial p}{\partial r} + \frac{m}{r}p - \frac{\delta}{2c_0^3}\frac{\partial^2 p}{\partial \tau^2} = \frac{\beta p}{\rho_0 c_0^3}\frac{\partial p}{\partial \tau} \tag{2.1}$$

where p is pressure, r is range from the source δ is the diffusivity of sound, c_0 is the nominal speed of sound, τ is retarded time, β is the coefficient of non-linearity, and ρ_0 is the nominal density. The parameter m is the wave parameter and has the value 0, 0.5, or 1 for plane, cylindrical, or spherical waves.

The range stepping part of the algorithm is derived by transforming equation 2.1 to the frequency domain[1]:

$$\frac{\partial \tilde{p}}{\partial r} + m\frac{\tilde{p}}{r} + \alpha'\tilde{p} = \frac{i\omega\epsilon}{2}\tilde{q} \tag{2.2}$$

where $\tilde{p}$ is the Fourier transform of pressure, ω is the angular frequency, and $\tilde{q}$ is the Fourier transform of the pressure squared. The atmospheric absorption and dispersion can be split into their separate parts $\alpha' = \alpha + i\beta_d$. The nonlinear coefficient, ϵ, is defined as $\epsilon = \beta/\rho_0 c_0^3$. In air the coefficient of non-linearity, β, is 1.2. Splitting equation 2.2 into real and imaginary parts gives the coupled ordinary differential equations:

$$\frac{dX}{dr} = -m\frac{X}{r} - (\alpha X - \beta_d Y) - \frac{\omega\epsilon}{2}V \tag{2.3}$$

$$\frac{dY}{dr} = -m\frac{Y}{r} - (\beta_d X + \alpha Y) + \frac{\omega\epsilon}{2}U \tag{2.4}$$

where

$$\tilde{p} = X + iY \tag{2.5}$$

$$\tilde{q} = U + iV \tag{2.6}$$

Fluctuations in the effective sound speed are included in a separate term, giving the complete equation:

$$\frac{dX}{dr} = -m\frac{X}{r} - (\alpha X - \beta_d Y) - \frac{\omega\epsilon}{2}V - \frac{c'\omega}{c_0^2}Y \tag{2.7}$$

$$\frac{dY}{dr} = -m\frac{Y}{r} - (\beta_d X + \alpha Y) + \frac{\omega\epsilon}{2}U + \frac{c'\omega}{c_0^2}X \tag{2.8}$$

where $c = c_0 + c'$, meaning that the speed of sound is equal to the average speed of sound plus a turbulent fluctuation[3]. This extra term is justified by reducing the parabolic Khokhlov-Zabolotskaya-Kuznetsov (KZK) Equation:[9, 10]

$$\frac{\partial}{\partial\tau}\left[\frac{\partial p}{\partial x} - \frac{\beta}{c_0^3\rho_0}p\frac{\partial p}{\partial\tau} - \frac{c' + u_x}{c_0^2}\frac{\partial p}{\partial\tau} + \frac{1}{c_0}\left(\vec{u}_\perp\vec{\nabla}_\perp p\right) - \frac{p}{2\rho}\frac{\partial\rho}{\partial x} - \delta\frac{\partial^2 p}{\partial\tau^2}\right] = \frac{c_0}{2}\Delta_\perp p \tag{2.9}$$

where, u_x is the flow of the medium in the direction of propagation, $\vec{u}_\perp$ are the transverse components of the flow, and ρ is the density such that $\rho = \rho_0 + \rho'$. The terms, in order, in equation 2.9 are: propagation, nonlinear distortion, sound speed and axial flow inhomogeneities, transverse flow inhomogeneities, density inhomogeneities, and absorption. The right hand side represents diffraction.[9] Reducing the equation to one dimension and assuming no mean flow or density changes gives the equation:

$$\frac{\partial p}{\partial x} - \frac{c'}{c_0^2}\frac{\partial p}{\partial\tau} - \delta\frac{\partial^2 p}{\partial\tau^2} = \frac{\beta}{c_0^3\rho_0}p\frac{\partial p}{\partial\tau} \tag{2.10}$$

which is the same as equation 2.1 constructed to ignore spreading ($m = 0$) and including sound speed fluctuations.

High frequency errors can arise because only a finite number of harmonics are used in the computation. As energy travels to higher frequencies it can cause the solution to diverge if it overwhelms the effects of atmospheric absorption and dispersion. In these cases the higher adjacent frequencies are cutoff and made to fall as equation 2.11. Usually a cutoff frequency of $0.9f_{max}$ is sufficient for a converged solution.

$$\tilde{p}_n = \tilde{p}_{n-1} \times \frac{n-1}{n} \tag{2.11}$$

The author implemented this algorithm as a C/C++ computer code. At each range step the squared pressure is interpolated to include twice as many points as the present time history. Then both the pressure and the pressure squared are transformed into the frequency

domain. The Fast Fourier Transform is computed using the FFTW library. Equations 2.7 and 2.8 are integrated using the Runge-Kutta 4,5 method from the Gnu Science Library to calculate the real and imaginary parts of the pressure at the next range step. These calculations take between a few minutes to a few days on a quad core 2.2 GHz processor, depending on the receiver distance and the number of harmonics. After the final step a Lanczos filter is applied to eliminate the Gibbs phenomenon. This process is repeated for between five and 50 random realizations of the atmosphere. This algorithm is shown graphically in figure 2.1.

2.2. Atmospheric Turbulence

For each range step a different random snapshot of the turbulent atmosphere is used. The atmosphere is assumed to be frozen for a single simulation. Then many simulations are averaged together to produce the effect of turbulence. The turbulence is expressed as variations in the effective speed of sound due to changing wind and temperature.

Computationally, turbulence is calculated from summing Fourier modes of the chosen spectrum with a random phase[5].

$$\mu(r) = \sqrt{4\pi\Delta k}\sum_{n=1}^{N} cos\left(k_n r + \alpha_n\right)\sqrt{F\left(k_n\right)k_n} \tag{2.12}$$

Here μ is the refractive index, k_n is the wave number, Δk is the step between wave numbers, and α_n is a random angle between 0 and 2π. The fluctuation in the speed of sound can then be calculated from equation 2.13.

$$c' = c_0\mu(r) \tag{2.13}$$

In this paper the von Karman spectrum is used to calculate the variation in refractive index of the atmosphere.

$$F\left(k\right) = \mu_0^2\frac{\Gamma(8/6)}{\Gamma(1/3)\pi}\frac{a^2}{(1+k^2a^2)^{8/6}} \tag{2.14}$$

where μ_0^2 is the root mean square value, and a is the correlation length. Empirically values for μ_0^2 range from 10^{-6} to 10^{-5} and a is approximately 1m[5]. The nominal sound speed is multiplied by the refractive index to find the deviation in the speed of sound.

In most computations involving turbulence the parameters μ_0 and a are kept constant. Another option is to vary the parameters with altitude and to model the variation on the PBL. In this model the von Karman spectrum is written in terms of the structure parameters[5].

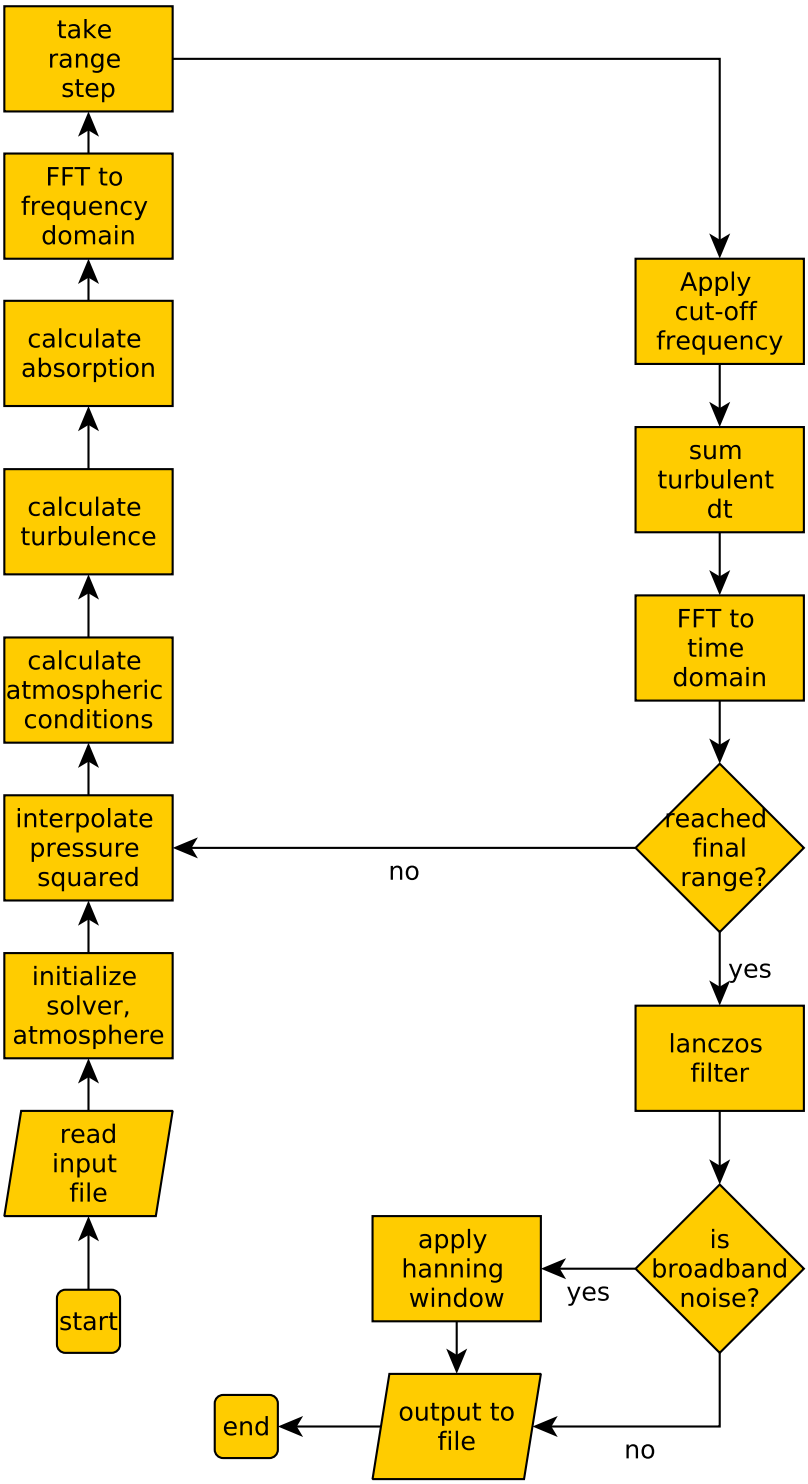


Figure 2.1.: Flowchart describing the execution path of a single simulation. Multiple simulations are then averaged together to produce turbulent waveforms. Most of the computational time is spent in calculating atmospheric absorption and turbulence.

$$F(k) = \frac{A}{\left(k^2 + K_0^2\right)^{8/6}} \left(\frac{\Gamma\left(\frac{1}{2}\right)\Gamma\left(\frac{8}{6}\right)}{\Gamma\left(\frac{11}{6}\right)} \frac{C_T^2}{4T_0^2} + \left[\frac{\Gamma\left(\frac{3}{2}\right)\Gamma\left(\frac{8}{6}\right)}{\Gamma\left(\frac{17}{6}\right)} + \frac{k^2}{k^2 + K_0^2} \frac{\Gamma\left(\frac{1}{2}\right)\Gamma\left(\frac{14}{6}\right)}{\Gamma\left(\frac{17}{6}\right)} \right] \frac{22C_v^2}{12c_0^2} \right) \tag{2.15}$$

Here the constant $A = 5/[18\pi\Gamma(1/3)] \approx 0.03$, C_T is the structure parameter describing temperature variation, and C_v is the structure parameter for velocity fluctuations. The structure parameters are calculated using curve fits from Plotkin[8]:

$$C_v^2 = \frac{1.5 w_*^2}{z_1^{2/3}} \tag{2.16}$$

$$C_T^2 = \frac{2.67 \theta_*^2 \left(z/z_1\right)^{-4/3}}{z_1^{2/3}} \tag{2.17}$$

where z_1 is the height of the PBL, z is altitude, w_* is the velocity scaling parameter, given in equation 2.18, and θ_* is the temperature scaling parameter, given in equation 2.19.

$$w_* = \left(\frac{gHz}{k u_* c_p \rho} \right) \tag{2.18}$$

$$\theta_* = \frac{H}{c_p \rho w_*} \tag{2.19}$$

$$u_* = \sqrt{\overline{u'w'}} = \sqrt{\tau/\rho} \tag{2.20}$$

Here g is gravitational acceleration, k is von Karman's constant ($k \approx 0.4$), c_p is the specific heat, ρ is density, H is the vertical heat flux at the ground, an input parameter, and the other input parameter is τ, the surface stress. These relations are only applicable in the mixed layer ($0.1z_1$ to $0.8z_1$) of the PBL. Only the mixed layer is used in the algorithm because it has the greatest effect on sonic booms.[8]

Combining equations 2.16-2.20 to remove numerical instabilities around $H = 0$ gives equations 2.21 and 2.22, which are used in the simulation.

$$C_T^2 = 2.67 \left(\frac{k z_1^2 H^2}{c_p^2 z^5} \sqrt{\frac{\tau}{\rho^5}} \right)^{1/3} \tag{2.21}$$

$$C_v^2 = 1.5 \left(\frac{gHz}{k c_p z_1 \sqrt{\rho\tau}} \right)^{2/3} \tag{2.22}$$

As the many simulations are run the time it takes for the sound to travel from the source

to the receiver is recorded. This propagation time includes the turbulence in the speed of sound and therefore is slightly different each time. After many simulations the time histories are shifted by their propagation time and then summed to produce the averaged result.

2.3. Atmospheric Absorption

Atmospheric absorption and dispersion remove energy from the wave due to the relaxation of oxygen and nitrogen in the atmosphere.

The complex absorption is calculated from the equation given by Bass et al.[11],

$$\alpha = p_s F^2 \left\{ 1.84 \times 10^{-11} \left(\frac{T}{T_0}\right)^{1/2} p_{s0} + \left(\frac{T}{T_0}\right)^{-5/2} \left[0.01275 \frac{e^{\frac{-2239.1}{T}}}{F_{r,O} + \frac{F^2}{F_{r,O}}} + \right.\right. \\ \left.\left. + 0.1068 \frac{e^{\frac{-3352}{T}}}{F_{r,N} + \frac{F^2}{F_{r,N}}} \right] \right\} \frac{nepers}{m\,atm} \quad (2.23)$$

where the various parameters are defined by Bass et al.[12]

The atmospheric state parameters in equation (2.23) are calculated according the ICAO standard atmosphere and vary with altitude. The relative humidity is an important parameter in calculating atmospheric absorption. It is included in the calculation of relaxation frequencies for nitrogen and oxygen.

3. Validation

3.1. Without Turbulence

The algorithm is validated against the analytic Blackstock Bridging Function (BBF)[13, 14] for the case of an initial sinusoidal wave with no turbulence during propagation. This is the same validation problem used by Lee et al.[1] to validate previous versions of this algorithm. The input pressure is given by equation 3.1.

$$P = P_a \sin(2\pi f_0 t) \tag{3.1}$$

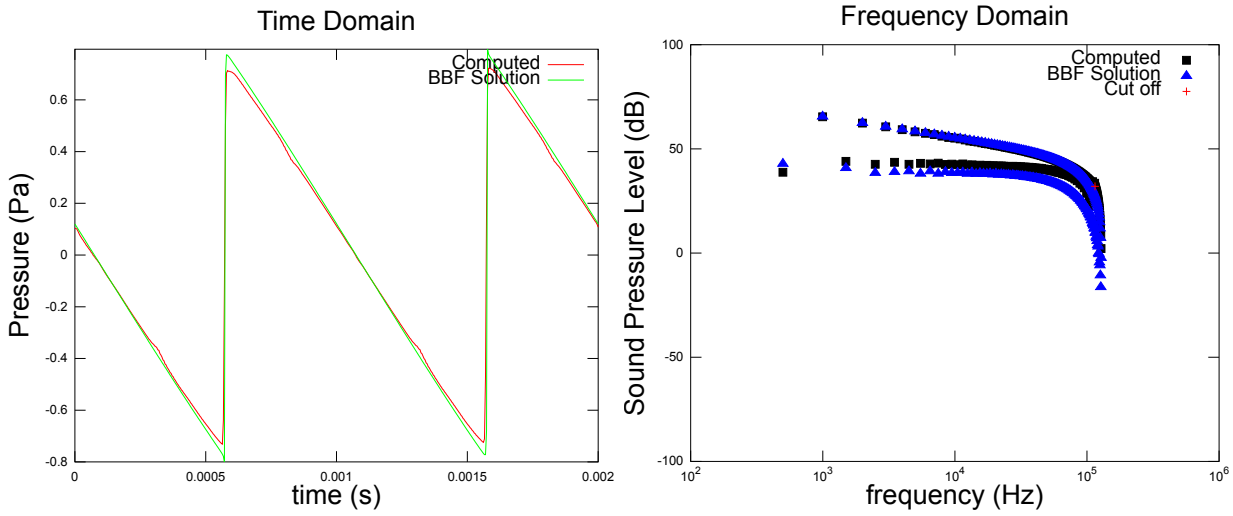
The values for P_a and f_0 are chosen to be 282.84 Pa and 1 kHz, respectively. Equation 3.1 is then discretized with a time step of 3.9139×10^{-6}s for a total of 512 points in the time domain. The wave is propagated out to 68.7m, which is three times the shock formation distance[1]. A cutoff frequency of 115 kHz is used.

Figure 3.1 shows good agreement between the Blackstock Bridging Function and the algorithm used in the present paper. The BBF shows sharper corners at the extrema of each wave because it does not include dispersion, which smooths out the wave.

3.2. With Turbulence

The algorithm is tested on an N wave, shown in figure 3.2, and compared to the results of a modified KZK solution[3]. This test includes turbulence from a von Karman spectrum. The implementation of the KZK equation by Blanc-Benon et al.[3] is two dimensional and includes the effects of caustics on the resulting waveform. Because the current algorithm is one dimensional it cannot reproduce all of the distorted waveforms seen by the KZK equation, but it does produce some similar results. In particular the current algorithm can produce the rounded waveform and the double peaked waveform, due to the waves having different propagation times.

These results are shown in figure 3.3. The time histories contain 2048 points with an uniform time step of 2.9297×10^{-8}s. The first wave shows a pressure history that peaks,



(a) (b)

Figure 3.1.: A sinusoidal wave after propagating 67 meters. Plot 3.1a shows the computed and analytical waves in the time domain, while plot 3.1b shows the waves in the frequency domain. The cut off frequency is shown for the frequency domain plot.

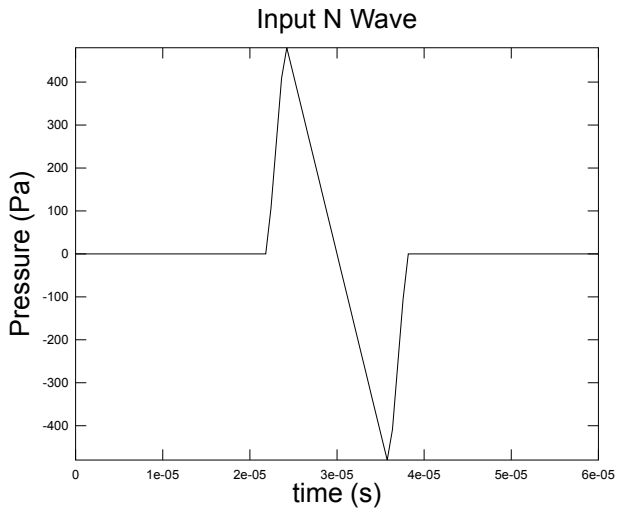
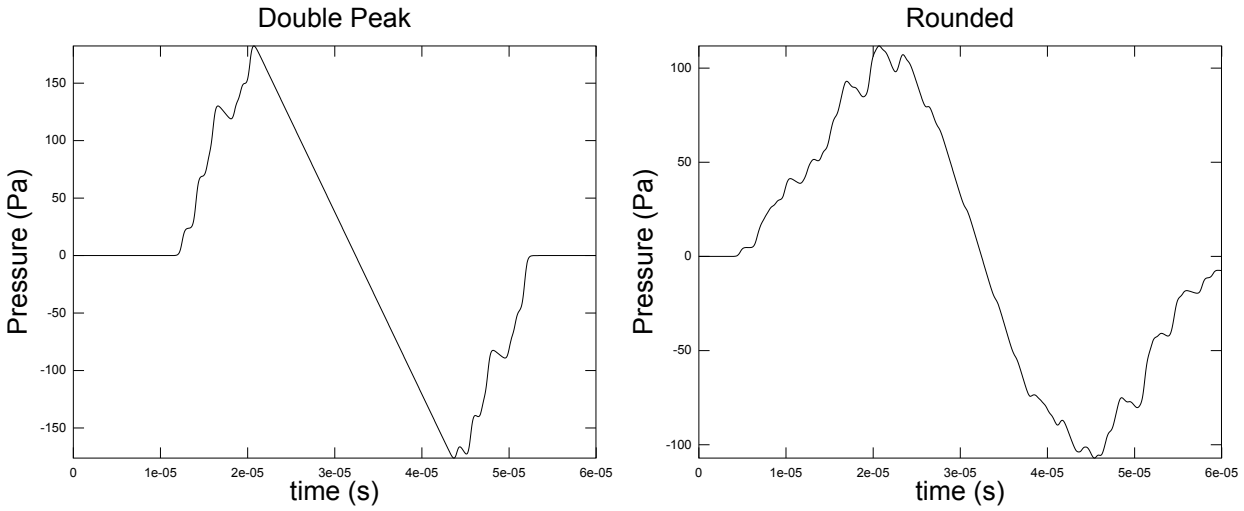


Figure 3.2.: Input N wave. It is defined as piecewise linear with a maximum over pressure of 416 Pa and a pulse duration of 16 μs.



(a) (b)

Figure 3.3.: Two examples of different waveforms caused by turbulence. Plot 3.3a shows two peaks on the way to maximum over pressure. Plot 3.3b show many steps and peaks that lead to an increased rise time.

drops and then peaks again. This type of wave was also reported by Blanc-Benon et al.[3]. The second wave is interesting in that it shows a greatly increased rise time and several steps on the way to the peak overpressure. This corresponds to the results obtained by Pierce and Maglieri[4] with wave front folding.

4. Results

4.1. Sonic Boom

Two sonic boom waveforms, shown in figure 4.1, have been used to test propagation codes[1, 15]. The flattop and ramp waves were generated by NASA Langley Research Center to challenge computer codes because of their sharp cusps and multifaceted waveforms. The boom is propagated cylindrically from the aircraft's altitude at 14630m down to ground level. The initial boom waveforms are given at 183m below the aircraft and were discretized to 16384 points, with a time step of 3.6623×10^{-5}s. The large number of points is necessary for the simulation to converge when turbulence is included. Two models turbulence are used: uniform, homogeneous turbulence, and the PBL model. The homogeneous von Karman spectrum uses a root mean square value of 1×10^{-5}, while the PBL uses parameters for the atmosphere in the early morning.

Figure 4.2 shows the flattop wave after propagation without turbulence and with the two models for turbulence presented in section 2.2. The wave with homogeneous turbulence has a longer rise time because the pressure rise is spread out over several smaller jumps. The wave without turbulence has become similar to an N wave because the positive slopes have become steeper and the negative sloped part of the wave has become less steep than the input waveform. The wave with turbulence modeled by the PBL is very similar to the wave without turbulence because the input parameters to the PBL are chosen to represent a quiet atmosphere. This low level of turbulence in chosen because any increase in turbulence strength would cause the solution to diverge. The chosen parameters reflect the PBL in the early morning and the simulated waveform compares well with experimental measurements taken under those conditions.[8]

The effect of turbulence on the ramp sonic boom waveform can be seen in figure 4.3. The results are similar to those obtained for the flat-top waveform. The addition of homogeneous turbulence has smoothed out the original distinctive features of the ramp waveform and increased the rise time. The addition of PBL turbulence has produced little change from propagation without turbulence because the PBL was chosen to simulate the atmosphere in the early morning when the atmosphere has only low levels of turbulence.

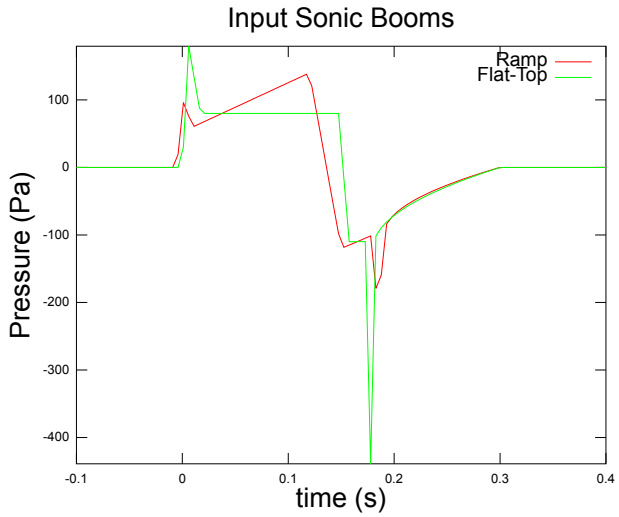


Figure 4.1.: Source waveforms for the flattop and ramp sonic booms.

4.2. Broadband Jet Noise

The algorithm has also been used to predict the sound pressure level of broadband noise from the Boeing Low Speed Aeroacoustic Facility (LSAF) if there had been turbulence. The broadband noise is given in 79801 time points with a time step of 5.2084×10^{-6}s. The supersonic jet noise is propagated from 3.23 meters to 9.32 meters from the exit plane of the nozzle and then averaged using overlapping Hanning windows.

Figure 4.4 shows the difference between jet noise propagation with turbulence, without turbulence and the experimental data. It is observed that the inclusion of turbulence results in significantly more energy at frequencies above 50 kHz. The difference between the turbulent data and the measured data is expected because the original measurements were made in an environment without ambient turbulence.

The one dimensional assumption used in this paper is a valid for broadband jet noise because the time history is of little interest. Because the input signal is already random summing small phase shifts contributes little to the waveform at the receiver. The one dimensional model does take into account the energy added or removed from the signal due to turbulence effects. These changes produce a noticeable effect at the receiver, namely the increased energy at high frequencies.

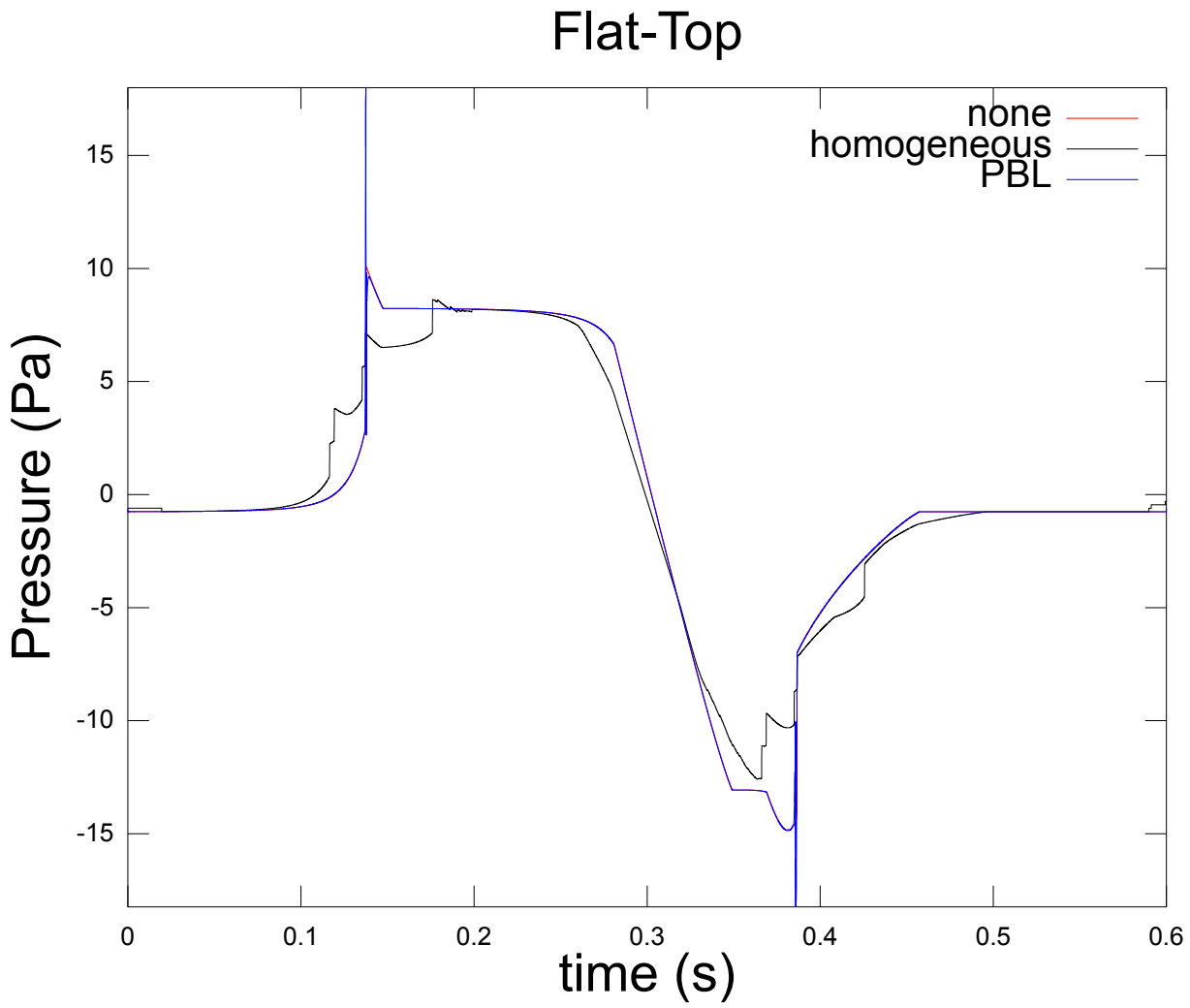


Figure 4.2.: Flattop waveform after propagation without turbulence, with homogeneous rms turbulence, and with turbulence modeled by the PBL. The PBL and no turbulence waveforms coincide.

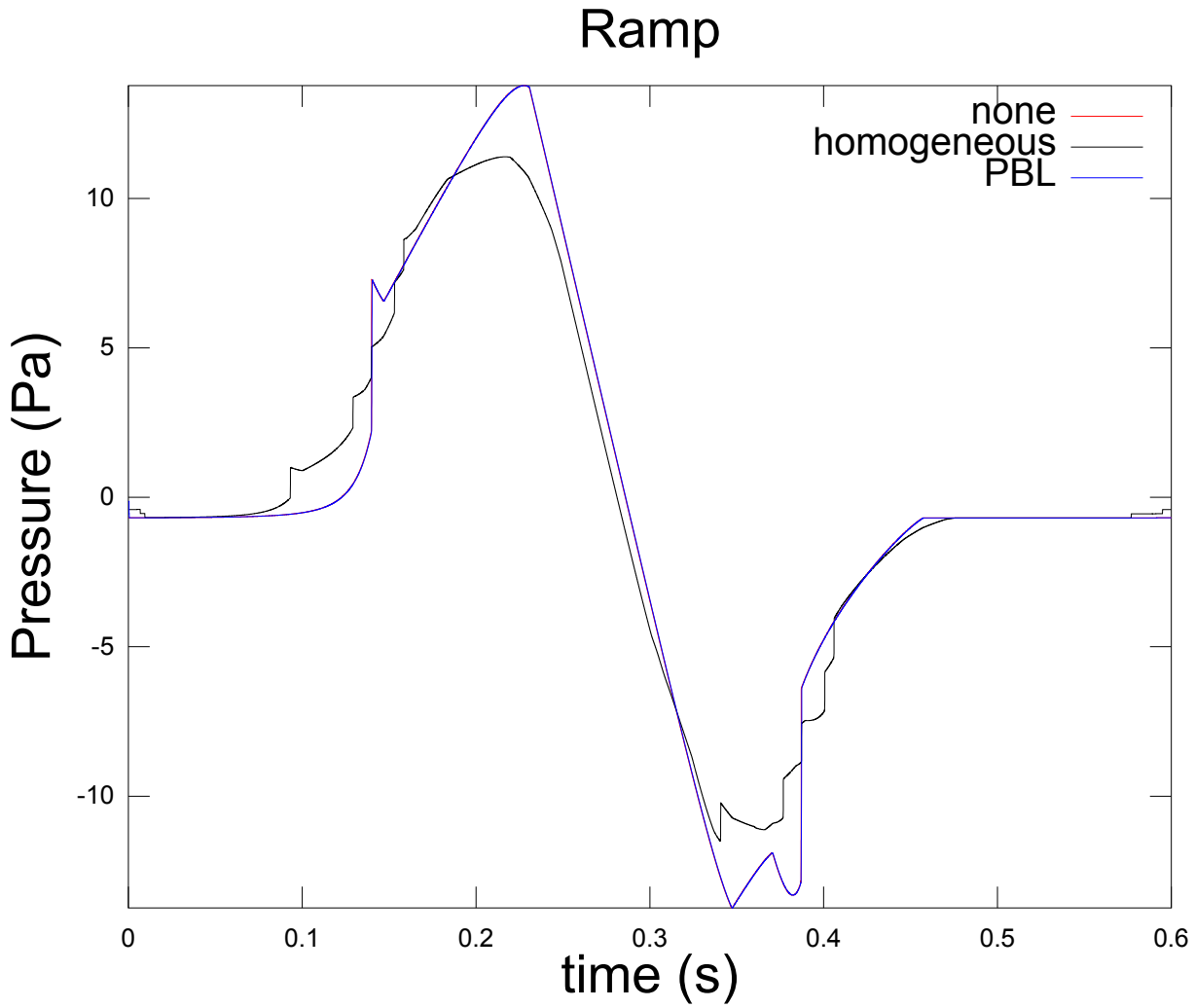
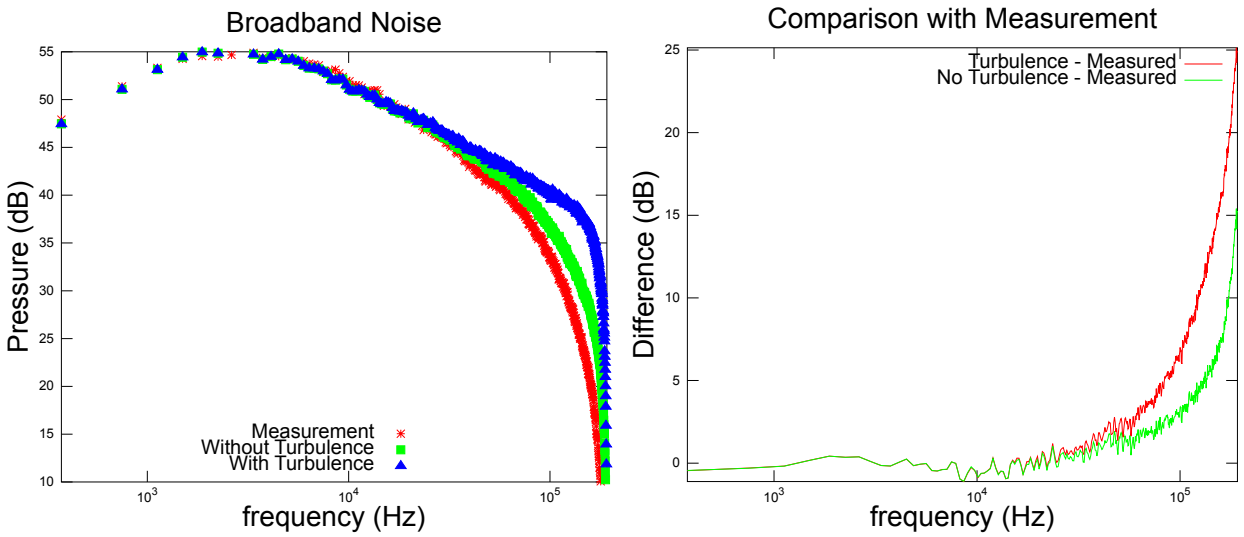


Figure 4.3.: Ramp waveform after propagation without turbulence, with homogeneous rms turbulence, and with turbulence modeled by the PBL. The PBL and no turbulence waveforms coincide. The wave without turbulence show sharper, distinct features while the wave with homogeneous turbulence is more rounded and has a longer rise time.



(a) Noise spectrum

(b) Differences between computed values and measured values.

Figure 4.4.: Broadband jet noise after propagation to 9.32m.

5. Conclusions and Future Work

The algorithm is successful in propagating a sine wave past shock formation and matches the analytic solution for this case. It can also produce N waves distorted by turbulence that have been seen in other computational experimental research. The effects of turbulence include increased rise time and multiple peaks. The algorithm is then applied to broadband jet noise. It is seen that including turbulence in the calculation increases energy at high frequencies to produce a significant difference from measured results without turbulence.

Computationally, turbulence necessitates including more harmonics and a smaller range step to keep the solution convergent. This results in much greater computational time to complete the simulation, as shown in table 5.1. The PBL is difficult to include in sonic boom propagation because it provides no turbulence for most of the propagation distance and then provides very strong turbulence for the last two kilometers. This strong kick at the end causes the solution to diverge in many cases. Homogeneous turbulence is much easier to include in the simulation because steady, low levels of turbulence are less likely to cause the solution to diverge.

The algorithm could be extended to two or three dimensional turbulence and ray tracing to provide a more complete representation of a real turbulent atmosphere. A frozen turbulent atmosphere could be generated. Then, as the rays propagate through it, the scattering angle could be recorded in addition to the propagation time. As the rays arrive at the receiver they could be summed together to account interference induced by different paths.

There are several effects not included in the present algorithm that could have significant impacts. Primarily, including the density and transverse terms canceled from equation 2.9 would provide a more complete representation of turbulence. Also, ground reflections can interfere with the incoming wave to alter the boom waveform[4]. The effects of turbulence in the inversion and surface layers in the PBL could also alter the waveform. Although the mixed layer has the greatest influence on sonic booms, the surface layer has much larger gradients and the inversion layer has large convective structures that could change the boom's shape.

Data	Turbulence type	Time Points	Distance (m)	Simulations	Run Time (s)
broadband	homogeneous	79801	6.09	1	35
broadband	none	79801	6.09	1	704
flat-top	none	16384	14444	1	22274
flat-top	homogeneous	16384	14444	5	217858
flat-top	PBL	16384	14444	5	47892*
nwave	homogeneous	2048	5	10	286
nwave	homogeneous	2048	5	50	1468
ramp	none	16384	14444	1	4844
ramp	homogeneous	16384	14444	5	69940
ramp	PBL	16384	14444	5	252842
sine	none	512	67.7	1	87

Table 5.1.: Simulation run times for different input parameters. The simulation is not deterministic if turbulence is included, which accounts for some of the variation in run times.

Bibliography

[1] Lee, S., Morris, P. J., and Brentner, K. S., "Nonlinear Acoustic Propagation Predictions with Applications to Aircraft and Helicopter Noise," The 48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, Orlando, FL, 2010.

[2] Chevret, P., Ph. Blanc-Benon, and D. Juve. "A Numerical Model for Sound Propagation through a Turbulent Atmosphere near the Ground." The Journal of the Acoustical Society of America 100.6 (1996): 3587-599. Web.

[3] Blanc-Benon, P., B. Lipkens, L. Dallois, M. F. Hamilton, and D. T. Blackstock. "Propagation of Finite Amplitude Sound through Turbulence: Modeling with Geometrical Acoustics and the Parabolic Approximation." The Journal of the Acoustical Society of America 111.1 (2002): 487. Print.

[4] Pierce, A. D., and D. J. Maglieri. "Effects of Atmospheric Irregularities on Sonic-Boom Propagation." The Journal of the Acoustical Society of America 51.2 (1972): 702-21. Print.

[5] Salomons, F. A., "Computational Atmospheric Acoustics," Kluwer Academic Publishers, Dordrecht, The Netherlands, 2001.

[6] Blanc-Benon, P. "Outdoor Sound Propagation Modeling in Complex Environments: Recent Developments in the Parabolic Equation Method." Battlefield Acoustic Sensing for ISR Applications (2006): 1-16. Web.

[7] Boulanger, P., R. Raspet, and H. E. Bass. "Sonic Boom Propagation through a Realistic Turbulent Atmosphere." The Journal of the Acoustical Society of America 98.6 (1995): 3412-417. Web.

[8] Plotkin, K. J., and M. J. Lucas. A Review of Atmospheric Turbulence. Rep. Arlington, Virginia: Wyle Laboratories, 1992. Print.

[9] Aver'yanov, M. V., V. A. Khokhlova, O. A. Sapozhnikov, Ph. Blanc-Benon, and R. O. Cleveland. "Parabolic Equation for Nonlinear Acoustic Wave Propagation in Inhomogeneous Moving Media." Acoustical Physics 52.6 (2006): 623-32. Print.

[10] Ostashev, Vladimir. Acoustics in Moving Inhomogeneous Media. London: E. Et F N. Spon, 1997. Print.

[11] Bass, H. E., Sutherland, L. C., Zuckerwar, A. J., Blackstock, D. T., Hester D. M., "Erratum: Atmospheric Absorption of Sound: Further Developments", Journal of the Acoustical Society of America, Vol. 97, 1995, pp. 680-683.

[12] Bass, H. E., L. C. Sutherland, A. J. Zuckerwar, D. T. Blackstock, and D. M. Hester. "Atmospheric Absorption of Sound: Further Developments." The Journal of the Acoustical Society of America 97.1 (1995): 680-83. Print.

[13] Blackstock, D. T., M. F. Hamilton, and A. D. Pierce (1998) chap 4. in Nonlinear Acoustics, chap. Progressive waves in lossless and lossy fluids, 1 ed., Academic Press, San Diego, pp. 65–150.

[14] Saxena, S. "A New Algorithm for Nonlinear Propagation of Broadband Jet Noise." Diss. The Pennsylvania State University, 2008. Print.

[15] Cleveland, R. O., Chambers, J. P., Bass, H. E., Raspet, R., Blackstock, D. T., and Hamilton, M. F., "Comparison of Computer Codes for the Propagation of Sonic Boom Waveforms through Isothermal Atmospheres," Journal of the Acoustical Society of America, Vol. 100, No. 5, November 1996, pp. 3017–3027.

A. Computer Code

A.1. template.cfg

```
//a template configuration file for sonic boom
//all number MUST HAVE A DECIMAL, unless stated otherwise

//configuration for the propagation algorithm
propagation:
{
  //number of runs to average together. If num_runs > 1 then the first run
  //will be without turbulence, as a baseline.
  num_runs = 1;
  //relative tolerance of integration method in propagation
  relative_tolerance = 5e-6;
  //wave parameter, aka m. 0 is plane, 0.5 is cylindrical, 1 is spherical,
  //make sure this value has a decimal
  wave_parameter = 1.;
  //frequencies higher than this times the highest frequency are made to fall
  //off as (n-1)/n
  cutoff_percent = 0.7;
  //final range to propagate the solution to, in meters
  range = 20.;
  //direction to propagate: 1 is up, -1 is down
  direction = 1;
  //if the frequency domain should be windowed to 1024 points and smoothed
  //with a hanning window after the last range step.
  smooth_final_step = false;
};

//configuration parameters to control atmospheric absorption
```

```
absorption:
{
  //ambient relative humidity
  relative_humidity = 0.686;
  //The static ambient pressure in atm
  P_ambient = 1.;
  //The static ambient temperature in Kelvin
  T_ambient = 273.;
};

//prameters to affect atmospheric turbulence
turbulence:
{
  //type of turbulence to include. A string that is either "none", "rms", or
  //"wyle". If it is "none" then the rest of the parameters in this section
  //have no effect.
  type = "wyle";
  //parameters for turbulence type "rms". This is homogenous turbulence
  //through the whole propagation distance. The settable parameters are
  //the root mean square value, and the correlation length.
  rms_average = 1e-6;
  correlation_length = 1.0;
  //parameters for turbulence type "wyle". This turbulence varies with
  //altitude acording to a model of the Planetary Boundary Layer. The
  //settable parameters are surface heat flux, surface shear stress, and
  //the hight of the PBL.
  //heat flux at earths surface in W/m^2. this changes with the sun,
  //so it is highly dependent on season, clouds, time of day
  surface_heat_flux = 200.;
  //shear stress from the wind at the earth's surface in kg/ m s^2. also know as the
  //reynolds stress. it can be calculated from average wind speed:
  //p*u^2
  surface_shear_stress = 58.1398373;
  //height of the planetary boundary layer, in m
  PBL_height = 2000.;
};
```

```
//set of values that describe the initial condition
initial_condition:
{
  //delta t, the time between two readings in the array points in seconds
  time_step = 0.1;
  //the range from the source where the measurements were taken in meters
  starting_range = 0.;
  //the altitude from sea level of the source. so that:
  //current_altitude = (+/-)range + starting_altitude
  starting_altitude = 0.;
  //the pressure time history, in Pa
  points = [5.0,6.,3.5];
};
```

A.2. main.cpp

```
/*
 * main.cpp
 *
 *  Created on: Jan 6, 2011
 *      Author: evan
 */

#include <stdio.h>
#include <time.h>
#include <libconfig.h>
#include <string.h>
#include <fstream>

#include "InitialCondition.h"
#include "Solver.h"

int ain(int argc, char * argv[]) {
```

```
config_t cfg; //configuration input
FILE * outFile; //File handle used for outputting
char * outFile_name = "result.m";//default name
InitialCondition ic;
Atmosphere_t atmosphere, *atm;
atm = &atmosphere;
Solver *solv;
double rh, P, T, H, s, z, dt, r0, m, cp, r, rt, sa, rms, a;//temporary variables
int sf;
const char * type;
long int dir, num_runs;
config_setting_t * points;
srand(time(NULL));

//parse command line arguments
if (argc < 2 || argc > 3) {
  printf("Useage: sonicBoom configuration_file [ output_file ]");
  return (EXIT_FAILURE);
}
if (argc > 2) {//if output file name was supplied
  outFile_name = argv[3];
}

//assume first argument is name of configuration file
config_init(&cfg);
if (!config_read_file(&cfg, argv[1])) {
  fprintf(stderr, "%s:%d - %s\n", argv[1], config_error_line(&cfg),
      config_error_text(&cfg));
  config_destroy(&cfg);
  return (EXIT_FAILURE);
}
//set to automatically convert form int to double
config_set_auto_convert(&cfg, CONFIG_TRUE);
//load atmospheric absorption and turbulence
if (config_lookup_float(&cfg, "absorption.relative_humidity", &rh)
    && config_lookup_float(&cfg, "absorption.P_ambient", &P)
```

```
  && config_lookup_float(&cfg, "absorption.T_ambient", &T)
  && config_lookup_string(&cfg, "turbulence.type", &type)) {
if (strcmp(type, "wyle") == 0) {
  if (config_lookup_float(&cfg, "turbulence.surface_heat_flux", &H)
      && config_lookup_float(&cfg,
          "turbulence.surface_shear_stress", &s)
      && config_lookup_float(&cfg, "turbulence.PBL_height", &z)) {
    atm->turbulence_type = TURBULENCE_WYLE;
    atm->H = H;
    atm->surface_shear_stress = s;
    atm->z1 = z;
  } else {
    //error
    fprintf(stderr, "wyle turbulence fail\n");
    config_destroy(&cfg);
    return (EXIT_FAILURE);
  }
} else if (strcmp(type, "rms") == 0) {
  if (config_lookup_float(&cfg, "turbulence.rms_average", &rms)
      && config_lookup_float(&cfg,
          "turbulence.correlation_length", &a)) {
    atm->turbulence_type = TURBULENCE_RMS;
    atm->turbulence_rms = rms;
    atm->turbulenc_correlation_length = a;
  } else {
    //error
    fprintf(stderr, "RMS turbulence fail\n");
    config_destroy(&cfg);
    return (EXIT_FAILURE);
  }
} else {
  //no turbulence
  atm->turbulence_type = TURBULENCE_NONE;
}
atm->Ps0 = P;
atm->T0 = T;
```

```
  atm->relative_humidity = rh;
} else {
  //error
  fprintf(stderr, "absorption or turbulence fail\n");
  config_destroy(&cfg);
  return (EXIT_FAILURE);
}
//load initial conditions
if (config_lookup_float(&cfg, "initial_condition.time_step", &dt)
    && config_lookup_float(&cfg, "initial_condition.starting_range",
        &r0) && config_lookup_float(&cfg,
    "initial_condition.starting_altitude", &sa) && (points
    = config_lookup(&cfg, "initial_condition.points")) != NULL) {
  ic.dt = dt;
  ic.r_0 = r0;
  ic.src_alt = sa;
  ic.num_points = config_setting_length(points);
  ic.points = new std::complex<double>[ic.num_points]; //Dynamically allocated
  for (int i = 0; i < ic.num_points; i++) {
    ic.points[i] = config_setting_get_float_elem(points, i);
  }
} else {
  //error
  fprintf(stderr, "initial conditions fail\n");
  config_destroy(&cfg);
  return (EXIT_FAILURE);
}
//propagator parameters
if (config_lookup_float(&cfg, "propagation.wave_parameter", &m)
    && config_lookup_float(&cfg, "propagation.cutoff_percent", &cp)
    && config_lookup_float(&cfg, "propagation.range", &r)
    && config_lookup_float(&cfg, "propagation.relative_tolerance", &rt)
    && config_lookup_int(&cfg, "propagation.direction", &dir)
    && config_lookup_bool(&cfg, "propagation.smooth_final_step", &sf)
    && config_lookup_int(&cfg, "propagation.num_runs", &num_runs)) {
  int turb_type = atm->turbulence_type;
```

```
atm->turbulence_type = TURBULENCE_NONE;
solv = new Solver(&ic, atm, m, cp, rt, dir);
/*output starting condition, assume name of second argument is name
 * of output file*/
outFile = fopen(outFile_name, "w");
time_t start_time = time(NULL);
fprintf(outFile, "%%Start Time: %s", ctime(&start_time));
fprintf(outFile, "%%Config File: %s\n", argv[1]);
fprintf(outFile, "initial_dt = %g;\n", ic.dt);
fprintf(outFile, "initial_y = [");
FourierArray *initial = solv->getPressure();
for (int i = 0; i < initial->size(); i++) {
  fprintf(outFile, "%g ", initial->get(i).real());
}
fprintf(outFile, "];");
//      fclose(outFile);
//      outFile = fopen("initial_spl.m", "w+");
fprintf(outFile, "initial_spl_df = %g;\n", 1 / (ic.num_points * ic.dt));
fprintf(outFile, "initial_spl_y = [");
//transform into fourier domain
initial->transformForward();
for (int i = 0; i < initial->size(); i++) {
  //convert to dB
  fprintf(outFile, "%g ", abs(initial->get(i)));
}
fprintf(outFile, "];");
//      fprintf(outFile, "initial_cutoff = %g;\n", cp);
//      fclose(outFile);
initial->transformBackward();
/*start solving*/
FourierArray * result;
double init_time_shift;
for (int i = 0; i < num_runs; i++) {
  //TODO reinitialize if not the first time
  if (i != 0 || num_runs == 1) {
    delete solv;
```

```
  atm->turbulence_type = turb_type;
  solv = new Solver(&ic, atm, m, cp, rt, dir);
}
solv->solve(r);
printf("run %d, propagation time: %g s\n", i, solv->prop_time);
if (i == 0) {//clone for first values
  result = solv->getPressure()->clone();
  result->transformBackward(); //make sure in time domain
  //divide by num runs, so it is averaged
  for (int j = 0; j < result->size(); j++) {
    result->set(j, result->get(j) / (double) num_runs);
  }
  init_time_shift = solv->prop_time;
} else {//sum for latter values
  FourierArray * tmp = solv->getPressure();
  tmp->transformBackward(); //make sure in time domain
  if (tmp->check(5e9)) {//only add in if numbers are ok
    //time shift and sum
    int time_shift = (solv->prop_time - init_time_shift)
        / ic.dt;
    //only sum overlap, should be most of it
    int j = time_shift;
    int max_j = result->size();
    if (time_shift < 0) {
      j = 0;
      max_j = tmp->size() + time_shift;
    }
    for (; j < max_j; j++) {
      //divide by num_runs to average
      result->set(j, result->get(j) + (tmp->get(j
          - time_shift) / (double) num_runs));
    }
  } else {
    i--; //redo the run
    printf("run failed check, redoing \n");
  }
```

```
  }
}//end for each run

/*output the answer*/
//      FourierArray *result = solv->getPressure();
printf("pressure time trace in %d points\n", result->size());

//      outFile = fopen("final.m", "w");
fprintf(outFile, "final_dt = %g;\n", ic.dt);
fprintf(outFile, "final_y = [");
for (int i = 0; i < result->size(); i++) {
  fprintf(outFile, "%g ", result->get(i).real());
}
fprintf(outFile, "];");
//      fclose(outFile);

//sound pressure level
//      outFile = fopen("spl.m", "w");

//transform into fourier domain
if (sf) { //if directed to smooth final step
  valarray<double> smoothed = solv->smooth();
  fprintf(outFile, "final_spl_df = %g;\n", 1
      / (smoothed.size() * ic.dt));
  fprintf(outFile, "final_spl_y = [");

  printf("smoothed\n");
  for (unsigned int i = 0; i < smoothed.size(); i++) {
    //convert to dB
    fprintf(outFile, "%g ", smoothed[i]);
  }
} else { //no smoothing
  fprintf(outFile, "final_spl_df = %g;\n", 1
      / (ic.num_points * ic.dt));
  fprintf(outFile, "final_spl_y = [");
  result->transformForward();
```

```
      for (int i = 0; i < result->size(); i++) {
        //convert to dB
        fprintf(outFile, "%g ", abs(result->get(i)));
      }
    }
    fprintf(outFile, "];\n");
    fprintf(outFile, "final_spl_cutoff = %g;\n", cp);
    time_t end_time = time(NULL);
    fprintf(outFile, "%%End Time: %s%%Total Runtime: %g s",
        ctime(&end_time), difftime(end_time, start_time));
    fclose(outFile);
  } else {
    //error
    fprintf(stderr, "propagation fail\n");
    config_destroy(&cfg);
    return (EXIT_FAILURE);
  }
  //clean up
  config_destroy(&cfg);
  delete solv;
  delete points;
}
```

A.3. Solver.h

```
/*
 * Solver.h
 *
 *  Created on: Feb 26, 2010
 *      Author: evan
 */

#ifndef SOLVER_H_
#define SOLVER_H_
```

```
#include "FourierArray.h"
#include "InitialCondition.h"
#include "Atmosphere.h"
#include <gsl/gsl_odeiv.h>
#include <valarray>

#define PI  3.14159265358979323846264338327950288419716939937510

class Solver {
private:
  //instance variables
  FourierArray * pressure, *squaredPressure; //pressure p //pressure squared p^2, sampled to 2*N
  double dt; //step size in time domain
  double cutoff; //cutoff frequency is this value times the highest frequency
  int tPts, direction; //intervals, direction of propagation( 1 is up, -1 is down)
  double m; //parameter for (plane,cylindrical, spherical waves) = (0,.5,1)
  double r_0, src_alt; //initial range, altitude of source
  double rel_tol; //relative tolerance for numerical integrator

  Atmosphere_t * atm;//atmospheric constants
  Atmosphere_Conditions_t current_conditions;
  attenuation_accel atten_accel; //used to speed up attenuation calculations
  //private member functions
  void interpolateSquarePressure(); //squares pressure and then interpolates to 2*N samples
  double static sinc(const double x);
  double static lanczos(const double x);
  double static hann(const double x);

public:
  //change in time from source to receiver from turbulent changes in the speed of sound.
    double prop_time;
  friend int burgersFunc(double r, const double pIn[], double derivs[],
      void * params); //calculates derivatives (RHS) of burgers equations for use with the gsl ode integrator
  friend int burgersJacobian(double r, const double pIn[], double * dfdy,
```

```
        double dfdr[], void *params); //calculates jacobian of burgers equations for use with
gsl ode integrater
    /**
     * Creates a problem to solve based on several input parameters
     * Params:
     *   InitialCondition * ic - the initial conditions for this problem.
     *   AtmosphericAbsorption * atmosAbsorb - the atmospheric absorption that
     *      is applied at each step.
     *   double waveParameter - the waveParameter (m) is 0, .5, or 1 for plane,
     *      cylindrical, or spherical waves, respectively.
     *   double cutoffPercent - percent of the maximum frequency where the
     *      frequency domain will be cut. Everything above this cutoff
     *      frequency will fall off as (n-1)/n. Used to eliminate high
     *      frequency noise. Should be a number between 0 (everything cutoff
     *      and 1 (nothing cutoff).
     *   double relative_tolerance - relative tolerance for the integrator
     *   int direc - direction to propagate: 1 is up, -1 is down. used in
     *      calculating altitude for atmospheric properties.
     */
    Solver(InitialCondition * ic, Atmosphere_t * atmosphere,
        double waveParameter, double cutoffPercent,
        double relative_tolerance, int direc);
    void solve(double range);
    /*returns   the presssure array, useful for outputting
     * DO NOT DELETE it will be deallocated when the Solver object is deleted
     */
    FourierArray* getPressure();
    valarray<double> smooth(); //returns a smoothed version of the frequency
    virtual ~Solver();

};

#endif /* SOLVER_H_ */
```

A.4. Solver.cpp

```
/*
 * Solver.cpp
 *
 *  Created on: Feb 26, 2010
 *      Author: evan
 */

#include "Solver.h"
#include <gsl/gsl_spline.h>
#include <gsl/gsl_matrix.h>
#include <stdexcept>
#include <omp.h>

#include <iostream>

Solver::Solver(InitialCondition * ic, Atmosphere_t * atmosphere,
    double waveParameter, double cutoffPercent, double relative_tolerance,
    int direc) {
  //initialize variables
  atm = atmosphere;
  cutoff = cutoffPercent;
  direction = direc;
  m = waveParameter; //0 is plane, .5 is cylindrical, 1 is spherical
  rel_tol = relative_tolerance;
  r_0 = ic->r_0;
  src_alt = ic->src_alt;
  dt = ic->dt;
  tPts = ic->num_points;
  prop_time = 0;
  pressure = new FourierArray(tPts);
  squaredPressure = new FourierArray(2 * tPts);
  //load ic
  //TODO optimize: don't copy, or at least copy wholesale
  for (int i = 0; i < pressure->size(); i++) {
```

```
    pressure->set(i, ic->points[i]);
  }
  interpolateSquarePressure();
}

void Solver::interpolateSquarePressure() {
  //declare arrays for interpolating
  double * x = new double[tPts];
  double * y = new double[tPts];
  //populate arrays
  for (int i = 0; i < pressure->size(); i++) {
    x[i] = dt * i;
    y[i] = pressure->get(i).real();
  }
  //setup interpolater
  //TODO   efficency make acc and spline instance variables
  gsl_interp_accel *acc = gsl_interp_accel_alloc(); //make evaluations go faster
  gsl_spline * spline = gsl_spline_alloc(gsl_interp_cspline, tPts); //set spline type and
size
  gsl_spline_init(spline, x, y, tPts); //initialize spline with data points
  //interpolate to 2*N, this will not use all of the input points
  for (int i = 0; i < squaredPressure->size(); i++) {
    double p;
    if (i % 2 == 0) {
      //use pressure
      p = pressure->get(i / 2).real();
    } else {
      //use interpolation
      p = gsl_spline_eval(spline, i * dt / 2, acc);
    }
    squaredPressure ->set(i, complex<double> (pow(p, 2), 0));
  }
  //free resources
  gsl_spline_free(spline);
  gsl_interp_accel_free(acc);
  delete[] x;
```

```
  delete[] y;
}

/*
 * computes the RHS of the Burger's function in the frequency domain.
 * Given by equation 8 in Lee, Morris, Brentner.
 * parameters:
 *   double r - current range
 *   const double *pIn - X and Y, the current values of fft of pressure
 *      pIn[0] = X[0], pIn[1] = Y[0], pIn[2] = X[1], pIn[3] = Y[1], ...
 *   double *derivs - array of the computed derivatives
 *   void *params - reference to the calling Solver, used to get problem parameters,
 *      including the squared pressure frequencies.
 * return:
 *   GSL_SUCCESS if the derivatives were computed
 */
int burgersFunc(double r, const double *pIn, double *derivs, void * params) {
  Solver* slvr = (Solver*) params;
  int length = slvr->pressure->size();
  Atmosphere_Conditions_t atm_cond = slvr->current_conditions;
  attenuation_accel * accel = &(slvr->atten_accel);
  //Calculate epsilon parameter TODO
  double epsilon = 1.2 / (atm_cond.rho * pow(atm_cond.c, 3.0));//beta/(ref density * ref
speed of sound ^3)
  //double epsilon = 1.2 / (1.21 * pow(343, 3.0));
  //use burger's equation to propagate
  int i;
  double omega;
  //absorbtion, pressure, presure squared
  complex<double> absorb, press, press_sq;
#pragma omp parallel for private(i, omega, absorb, press, press_sq)
  for (i = 0; i < length; i++) {
    omega = 2 * PI * i / (slvr->dt * slvr->tPts); //angular frequency
    absorb = get_attenuation(accel, omega / (2 * PI));//101325 Pa == 1atm
    //absorb = complex<double>(0.0,0.0);//TODO remove
    press = slvr->pressure->get(i);
```

```
    press_sq = slvr->squaredPressure->get(i);
    //alpha = absorb.real(); //real attenuation
    //betaD = absorb.imag(); //imaginary attenuation
    derivs[2 * i] = -slvr->m * press.real() / r - (absorb.real()
        * press.real() - absorb.imag() * press.imag()) - omega
        * epsilon * press_sq.imag() / 2.0 - atm_cond.dc * omega
        * press.imag() / pow(atm_cond.c, 2);

    derivs[2 * i + 1] = -slvr->m * press.imag() / r - (absorb.imag()
        * press.real() + absorb.real() * press.imag()) + omega
        * epsilon * press_sq.real() / 2.0 + atm_cond.dc * omega
        * press.real() / pow(atm_cond.c, 2);
  }
  return GSL_SUCCESS;
}

/*for the system dy_i/dr = f_i(r, y, ... , y_n)
 * computes the jacobian matrix for the system and df/dr for the system
 * params:
 *   double r - current range
 *   const double *pIn - X and Y, the current values of fft of pressure
 *     pIn[0] = X[0], pIn[1] = Y[0], pIn[2] = X[1], pIn[3] = Y[1], ...
 *   double * dfdy = jacobian matrix in row oredered form
 *     J(i, j) = dfdy[i*dimension + j]
 *   double * dfdr = derivitives of f wrt r
 *   void *params - reference to the calling Solver, used to get problem parameters
 * return:
 *   GSL_SUCCESS if the derivities were computed
 */
//int burgersJacobian(double r, const double *pIn, double * dfdy, double *dfdr,
//      void *params) {
//  Solver* slvr = (Solver*) params;
//  //create matrix view to make assigning easier
//  gsl_matrix_view dfdy_mat = gsl_matrix_view_array(dfdy, 2 * slvr->tPts, 2
//        * slvr->tPts);
//  gsl_matrix * mat = &dfdy_mat.matrix;
```

```
//   //initialize matrix
//   for (int i = 0; i < 2 * slvr->tPts; i++) {
//      for (int j = 0; j < 2 * slvr->tPts; j++) {
//         gsl_matrix_set(mat, i, j, 0.0);
//      }
//   }
//   for (int i = 0; i < 2 * slvr->tPts; i += 2) {
//      //X = pIn[i], Y = pIn[i+1]
//      //dfdr
//      dfdr[i] = slvr->m * pIn[i] / pow(r, 2);
//      dfdr[i + 1] = slvr->m * pIn[i + 1] / pow(r, 2);
//      //J
//      gsl_matrix_set(mat, i, i, -slvr->m / r - slvr->alpha);
//      gsl_matrix_set(mat, i, i + 1, slvr->betaD);
//      gsl_matrix_set(mat, i + 1, i, -slvr->betaD);
//      gsl_matrix_set(mat, i + 1, i + 1, -slvr->m / r - slvr->alpha);
//   }
//   return GSL_SUCCESS;
//}

/*steps through the algorithm until range (specified in the constructor) is reached
 *
 */
void Solver::solve(double rMax) {

  //constants through the stepping
  const gsl_odeiv_step_type *stepType = gsl_odeiv_step_rkf45; //stepping algorithm to use
  int dimension = tPts + 2; //number or equations=2(tPts/2+1)=num of real+imag values in
freq dom
  gsl_odeiv_step *stepper = gsl_odeiv_step_alloc(stepType, dimension);
  //tolerance of 10e-6, use value and derivative, scale by pressure squared values
  gsl_odeiv_control *tolerance = gsl_odeiv_control_standard_new(0, rel_tol,
      1, 0);
  gsl_odeiv_evolve *evolver = gsl_odeiv_evolve_alloc(dimension); //provides a nice way
take each step
  gsl_odeiv_system burgers = { burgersFunc, /*burgersJacobian*/0, dimension,
```

```
    this };//define system for integration
double * pressureData = (double *) pressure->getArray();//frequency domain

//define variables through stepping
double r = r_0;//Initialize to have not moved
double h = 0.01;//guess TODO guess better
current_conditions
    = get_atmosphere_conditions(atm, src_alt + direction * r);
double r_last_atm = r_0;
//solving loop, while not at desired range take a step
while (r < rMax) {
  //debug, only every 10
  //if (((int) r) % 10 == 0)
  //cout << r << endl;

  //interpolate
  interpolateSquarePressure();
  //fft
  //TODO efficiency don't transform pressure forward every time
  pressure->transformForward();
  squaredPressure->transformForward();

  /*find atmospheric conditions and turbulence for this range step,
   this is outside of actually taking the step so that the adaptive
   stepper doesn't select low turbulence conditions.
   current_ alt = src_ alt (+/-) r*/
  current_conditions = get_atmosphere_conditions(atm, src_alt
            + direction * r);
  if (r > r_last_atm + 100) {//only calc every meter
    r_last_atm = r;
    //debug
    cout << '.' << flush;
    //cout << r << "\n";//range
  }
  //      cout << current_ conditions.dc << endl;
  //sanity check
```

```
if (current_conditions.h < 0) {
  cout << "error: get_atmosphere_conditions failed, r: " << r << endl;
  exit(EXIT_FAILURE);
}
//initialize attenuation accelerator for new conditions
attenuation_accel_init(&atten_accel, atm, &current_conditions);

//rk solve ode
//TODO efficiency only solve half, then copy
int status = gsl_odeiv_evolve_apply(evolver, tolerance, stepper,
    &burgers, &r, rMax, &h, pressureData);
if (status != GSL_SUCCESS) {
  throw new runtime_error("failed to step ode");
}
//sum changes in time due to turbulent changes in sound speed
prop_time += evolver->last_step / (current_conditions.c
    + current_conditions.dc); // 1/dt (1/s)
//evolver->last_step /( current_conditions.c + current_conditions.dc); //dt = dr/dc

//smooth out high frequencies, so they don't introduce erroneous oscillations
//index of cutoff frequency, chosen to be top 10% of frequencies
int cutoffIndex = (int) (pressure->size() * cutoff);
//make sure higher frequencies die away, if they don't replace with analytic sawtooth model:
//p(w(n - 1))/p(nw) = (n-1)/n
for (int i = cutoffIndex; i < pressure->size(); i++) {
  //        //if amplitude greater than previous
  //        if (abs(pressure->get(i)) > abs(pressure->get(i - 1))) {
  //use ratio for amplitude, preserve phase
  double magnitude = abs(pressure->get(i - 1)) * (i - 1.0) / i;
  pressure->set(i, polar(magnitude, arg(pressure->get(i))));
  //        }
}
//ifft
pressure->transformBackward();
//we don't actually ifft P^2 because it is overwritten by squaring the pressure on the next iter-
ation
```

```
    squaredPressure->setIsInTimeDomain(true);
  }//repeat until r = rMax
  cout << endl;
  //prop_time = 1.0 / prop_time; //invert to get time
  //lanczos filter the results in the frequency domain
  pressure->transformForward();
  for (int i = 0; i < pressure->size(); i++) {
    //scale magnitude, leave phase unchanged
    double newMagnitude = abs(pressure->get(i)) * lanczos(((double) i)
        / pressure->size());
    pressure->set(i, polar(newMagnitude, arg(pressure->get(i))));
  }
  pressure->transformBackward();//back to time domain

  //clean up
  gsl_odeiv_evolve_free(evolver);
  gsl_odeiv_control_free(tolerance);
  gsl_odeiv_step_free(stepper);
}

/*
 * Smoothes the pressure frequency domain by
 * splitting the time domain into smaller windowed
 * sections and averaging the sections.
 * returns a smoothed frequency domain without modifying
 * the current pressure time history.
 * returns: valarray of Pressure dB re 2e-5 Pa in frequency domain
 */
#define WINDOW 1024
valarray<double> Solver::smooth() {
  //ensure pressure is in time domain
  if (!pressure->isInTimeDomain()) {
    pressure->transformBackward();
  }
  int num_windows = 2 * (pressure->size() / WINDOW) - 1;
  //create and initialize sum to 0
```

```
  valarray<double> sum(WINDOW / 2 + 1);
  FourierArray temp = FourierArray(WINDOW);
  //loop through each window
  for (int i = 0; i < num_windows; i++) {
    int offset = i * (WINDOW / 2);
    temp.setIsInTimeDomain(true);
    //loop through all points in each window
    for (int j = 0; j < WINDOW; j++) {
      //put points into each window and apply hann filter
      temp.set(j, hann(j - (WINDOW / 2)) * pressure->get(offset + j));
    }
    //transform
    temp.transformForward();
    //sum and average
    for (int j = 0; j < temp.size(); j++) {
      //add square of abs of each value, averaged over each window
      sum[j] += pow(abs(temp.get(j)), 2.0) / (double) num_windows;
    }
  }
  //sqrt to RMS average, and convert to dB re 2e-5Pa
  //sum = 20 * log10(sqrt(sum) / 2e-5);
  //not in dB
  sum = sqrt(sum);
  //clean up
  return sum;
}

/*
 * computes the hann fuction:
 * cos^2(pi*x/(2*a))
 * Using ;
 */
double Solver::hann(const double x) {
  return 0.5 * (1 + cos(2.0 * PI * x / (WINDOW)));
}
```

```
/*lanczos filter
 * for lanczos smoothing
 * window is 1
 */
double Solver::lanczos(const double x) {
  if (x == 0) {
    return 1;
  } else if (abs(x) < 1) {
    return sinc(x) * sinc(x / 1.0);
  } else {
    return 0;
  }

}

/*sinc function
 * for lanczos smoothing
 */
double Solver::sinc(const double x) {
  if (x == 0) {
    return 1.0;
  } else {
    return sin(PI * x) / (PI * x);
  }
}

FourierArray* Solver::getPressure() {
  return pressure;
}

Solver::~Solver() {
  // TODO Auto-generated destructor stub
  delete pressure;
  delete squaredPressure;
}
```

A.5. FourierArray.h

```
/*
 * FourierArray.h
 *
 *  Created on: Feb 16, 2010
 *      Author: evan
 *
 *  Discription: An array that is set up to allow fast processing from fftw3
 */

#include <complex>
#include <stdexcept>
#include <fftw3.h>

using namespace std;

//#ifndef FOURIERARRAY_H_
//#define FOURIERARRAY_H_

//template<typename double>
class FourierArray {
private:
  complex<double> * freqDomData; //frequency domain data
  double * timeDomData; //time domain data
  int n; //number of points in the time domain
  int freq_dom_length;  //num pts in frequency domain
  fftw_plan forwardPlan, backwardPlan;
  bool isTimeDomain;//if the current state of the data is in the time domain
  static bool isWisdomLoaded;
  void createPlans();
public:
  FourierArray(int);
  int size();
  complex<double> get(int);
  complex<double> * getArray(); //returns a pointer to the underlying array holding the com-
```

```
plex, frequency domain, values
  void set(int, complex<double> );
  void setIsInTimeDomain(bool);//sets the frequency/time domain state of the array
  bool isInTimeDomain();  //gets if is in time delay, if not in time domain then it is frequency
domain
  bool check(double max_value); //returns true if all values are less than max_value and there
are no inf or nan
  void transformForward();
  void transformBackward();
  FourierArray * clone();
  virtual ~FourierArray();
};

//#endif /* FOURIERARRAY_H_ */
```

A.6. FourierArray.cpp

```
/*
 * FourierArray.cpp
 *
 *  Created on: Feb 16, 2010
 *      Author: evan
 *
 *  Discription: creates an array class that is compatible with fftw3
 *    class also checks bounds when indexed
 */

#include <iostream>
#include <sstream>
#include <string>
#include "FourierArray.h"
#include <stdio.h>
#include <string.h>

bool FourierArray::isWisdomLoaded = false;
```

```
/*
 * construct an array with arrayLenth elements in the time domain
 */
FourierArray::FourierArray(int arrayLength) {
  n = arrayLength;
  freq_dom_length = (n / 2 + 1);
  isTimeDomain = true;
  freqDomData = (complex<double> *) fftw_malloc(sizeof(complex<double> )
      * freq_dom_length);
  timeDomData = (double *) fftw_malloc(sizeof(double) * n);
  forwardPlan = backwardPlan = NULL;
  createPlans();//this call will overwrite data
}

/*  will return the number of elements that the underlying array contains.
 *  The time domain has n elements and the frequency domain has n/2+1 elements.
 */
int FourierArray::size() {
  if (isTimeDomain) {
    return n;
  } else {
    return freq_dom_length;
  }
}

/*  Retrieves an element from the array.
 *  see the size() function for how many elements exist.
 *  in the time domain the imaginary part will be zero
 */
complex<double> FourierArray::get(int index) {
  if (isTimeDomain && index >= 0 && index < n) {
    return timeDomData[index];
  } else if (!isTimeDomain && index >= 0 && index < freq_dom_length) {
    return freqDomData[index];
  } else {
```

```
    stringstream message;
    message << "FourierArray::get() index out of range: " << index;
    throw out_of_range(message.str());
  }
}

/*returns a pointer to the underlying complex (frequency domain) array array
 *
 */
complex<double> * FourierArray::getArray() {
  return freqDomData;
}

/*  sets an element of the array.
 *  see size() for how many elements exist.
 *  if the array is in the time domain the imaginary part of the argument is ignored
 */
void FourierArray::set(int index, complex<double> value) {
  if (isTimeDomain && index >= 0 && index < n) {
    timeDomData[index] = value.real();
  } else if (!isTimeDomain && index >= 0 && index < freq_dom_length) {//freq domain
    freqDomData[index] = value;
  } else {
    stringstream message;
    message
        << "FourierArray::set(int, complex<double>) index out of range: "
        << index;
    throw out_of_range(message.str());
  }
}

/*  sets the time domain <-> frequency domain state of this array
 *  useful for changing the initial state or changing the data to transform forward again
 */
void FourierArray::setIsInTimeDomain(bool state) {
  isTimeDomain = state;
```

```
}

bool FourierArray::isInTimeDomain() {
  return isTimeDomain;
}

bool FourierArray::check(double max_value) {
  bool answer = true;//default to ok
  //pick array to search through
  if (isTimeDomain) { //search time domain
    for (int i = 0; i < n; i++) {
      //if too big or is nan
      if (timeDomData[i] > max_value || timeDomData[i] != timeDomData[i]) {
        answer = false;
      }
    }
  } else {//search frequency domain
    for (int i = 0; i < freq_dom_length; i++) {
      if (abs(freqDomData[i]) > max_value || freqDomData[i]
          != freqDomData[i]) {
        answer = false;
      }
    }
  }
  return answer;
}

/*
 * creates a plan for a fftw.
 * loads wisdom from the file ./fftw_wisdom if it has not been loaded yet.
 */
void FourierArray::createPlans() {
  FILE * wisFile;
  if (!isWisdomLoaded) {//if wisdom is not loaded, then load it,
    wisFile = fopen("fftw_wisdom", "r");
    if (wisFile != NULL) {
```

```
      fftw_import_wisdom_from_file(wisFile);
      isWisdomLoaded = true;
      fclose(wisFile);
    } else {
      cout << "warn: could not read wisdom from: fftw_wisdom" << endl;
    }
  }//plan it and save it

  forwardPlan = fftw_plan_dft_r2c_1d(n, timeDomData,
      (fftw_complex *) freqDomData, FFTW_EXHAUSTIVE);
  backwardPlan = fftw_plan_dft_c2r_1d(n, (fftw_complex *) freqDomData,
      timeDomData, FFTW_EXHAUSTIVE);

  wisFile = fopen("fftw_wisdom", "w");
  if (wisFile != NULL) {
    fftw_export_wisdom_to_file(wisFile);
    fclose(wisFile);
  } else {
    cout << "warn: could not write wisdom to: fftw_wisdom" << endl;
  }

}

/*
 * transforms to the frequency domain and divides by L
 */
void FourierArray::transformForward() {
  if (isTimeDomain) {//only transform if in time domain
    fftw_execute(forwardPlan);
    isTimeDomain = false;
    //fftw computes unormalized transform so we divide by number of points
    for (int i = 0; i < freq_dom_length; i++) {
      freqDomData[i] = freqDomData[i] / (double) n;
    }
  }
}
```

```
/*
 * transforms to the time domain
 */
void FourierArray::transformBackward() {
  if (!isTimeDomain) {//only transform if in frequency domain
    isTimeDomain = true;
    fftw_execute(backwardPlan);
  }

}

FourierArray * FourierArray::clone() {
  FourierArray * ans = new FourierArray(n);
  ans->isTimeDomain = isTimeDomain;
  if (isTimeDomain) {//copy time domain
    memcpy(ans->timeDomData, timeDomData, n * sizeof(double));
  } else { //copy frequency domain data
    memcpy(ans->freqDomData, freqDomData, freq_dom_length
        * sizeof(std::complex<double>));
  }
  return ans;
}

FourierArray::~FourierArray() {
  // TODO Auto-generated destructor stub
  fftw_destroy_plan(forwardPlan);
  fftw_destroy_plan(backwardPlan);
  fftw_free(timeDomData);
  fftw_free(freqDomData);
}
```

A.7. Atmosphere.h

```
/*
```

```
 * Atmosphere.h
 *
 *  Created on: Nov 16, 2010
 *      Author: evan
 */

#ifndef ATMOSPHERE_H_
#define ATMOSPHERE_H_

#include <complex>

/* this struct is returned to caller and represents
 * the state of the atmosphere at the specified height*/
struct Atmosphere_Conditions_t {
  double h; //altitude in m
  double P; //Pressure in Pascals
  double T; //temp in kelvin
  double rho; //density in kg / m^3
  double c; //speed of sound in m/s, does not include turbulenc, equivilent to c_0
  double dc; //change in speed of sound due to turbulence. s.t. c_eff = c + dc
};

/*
 * turbulence types for Atmosphere_t
 */
#define TURBULENCE_NONE 1
#define TURBULENCE_WYLE 2
#define TURBULENCE_RMS 3

/*
 * Quantities that can be taken as constant, even in a turbulent atmosphere
 */
struct Atmosphere_t {
  int turbulence_type;
  //old turbulence
  double turbulence_rms; //root mean square strength of the atmospheric turbulence
```

```
  double turbulenc_correlation_length; //characteristic length of turbulence
  //new turbulence
  double H; //surface heat flux
  double surface_shear_stress; //shear stress at h=0 in PBL
  double z1;  //height of PBL
  //Absorption
  double relative_humidity;
  double Ps0; //ambient static pressure, in atm
  double T0; //ambient static temperature, in K
};

/*
 * Stores common information to speed up calculations of atmospheric
 * attenuation for different frequencies, assuming a constant atmospheric state
 */
struct attenuation_accel {
  double T; //static temperature in K
  double h; //absolute humidity
  double Ps; //static pressure in atm
  double F_rN; //relaxation of Nitrogen
  double F_rO; //relaxation of oxygen
  double F_rN_Ps; //F_rN * Ps
  double F_rO_Ps; //F_rO * Ps
  double alphaTvb_part_no_F;
  double alpharN_part_no_F;
  double alpharO_part_no_F;
  double betad_part_no_freq;
};

/*
 * params:
 *   double altitude in m
 *   bool do_random, sets if the effective sound speed should include turbulence
 */
Atmosphere_Conditions_t get_atmosphere_conditions(Atmosphere_t *atm,
    double altitude);
```

```
/*
 * gets the absorption for a specific point
 * params:   double Ps - static pressure in atm
 *         double freq - frequency in Hz
 * returns: complex absorption in nepers per meter
 */
std::complex<double> get_attenuation(Atmosphere_t *atm, double Ps, double freq);
/*
 * gets the absorption for a specific frequency and atmospheric conditions, but
 * speeds up the calculations by pre-calculating the effects of the atmospheric
 * conditions and storing them in an accelerator.
 * params:   attenuation_accel* accel - the struct returned from
 *           attenuation_accel_init. Used to speed calculations for
 *           different frequencies at a constant atmosphere.
 *         double freq - frequency in Hz
 * returns: complex absorption in nepers per meter
 */
std::complex<double> get_attenuation(attenuation_accel* accel, double freq);
/*
 * Initializes an attenuation_accel to speed calculations for
 * constant atmospheric conditions. accel MUST already be allocated.
 */
void attenuation_accel_init(attenuation_accel* accel, Atmosphere_t *atm_const,
    Atmosphere_Conditions_t *atm_cond);

//Debug
//double von_karman(Atmosphere_t *atm, Atmosphere_Conditions_t * cond, double k) ;

#endif /* ATMOSPHERE_H_ */
```

A.8. Atmosphere.cpp

```
/*
 * Atmosphere.cpp
```

```
 *
 *  Created on: Nov 16, 2010
 *      Author: evan
 */
//implements atmoshpere.h
#include "Atmosphere.h"
#include <math.h>
#include <cstdlib>

#define PI 3.1415926535897

//constants and types for icao atmosphere
const double R = 287.05; //gas constant in J/ kg K
const double g = 9.80665; //acelleration due to gravity in m / s^2
const double CpCv = 1.4; //ratio of specific heats (gamma) for air
const double T01 = 273.16;//triple point temp
const double von_Karman_const = 0.4; //von karman constant Wyle, eq 5
const double Cp = 1.005e3; //J/kg K. from http://www.engineeringtoolbox.com/air-properties-d_156.html

struct ICAO_Layer_t {
  double h; //base height in km
  double lapse_rate; //in K / km
  double T; //base temperature in K
  double P; //base Pressure in Pa
};
/*make array of all layers for icao std atmosphere
 * data taken from wikipedia article
 */
#define ICAO_LAYERS_LENGTH 7
const ICAO_Layer_t layers[ICAO_LAYERS_LENGTH] = {
    { 0.0, -6.5, 289.15, 101325 }, //troposphere
    { 11, 0.0, 217.65, 22632 }, //tropopause
    { 20, 1.0, 217.65, 5474.9 }, //stratsphere
    { 32, 2.8, 229.65, 868.02 }, //stratosphere
    { 47, 0.0, 271.65, 110.91 }, //stratopause
```

```
    { 51, -2.8, 271.65, 66.939 }, //mesosphere
    { 71, -2.0, 215.65, 3.9564 }, //mesosphere
    };

//functions for turbulence
/* von_karman - calculates von karman spectrum as a function of wave number:
 *   F(k)
 * params:
 *   atm - atmosphere, defines:
 *     z1 - height of PBL
 *     H - heat flux at surface
 *     tau - surface shear stress
 *   cond - current atmosphere conditions, only uses:
 *     rho - density
 *     h - altitude
 *   k - wave number - independent variable
 */
double von_karman(Atmosphere_t *atm, Atmosphere_Conditions_t * cond, double k) {
  double answer = 0.0;//return value
  if (atm->turbulence_type == TURBULENCE_WYLE) {
    //calculate parameters from wyle parameters
//     double u_star = sqrt(atm->surface_shear_stress / cond->rho);
//     double w_star = pow(g * atm->H * cond->h / (von_Karman_const * u_star
//         * Cp * cond->rho), 1. / 3.);
//     double theta_star = atm->H / (Cp * cond->rho * w_star);
    double C_v_squared = 1.5 * pow(g * atm->H * cond->h / (von_Karman_const
        * Cp * atm->z1 * pow(cond->rho * atm->surface_shear_stress, 1.
        / 2.)), 2. / 3.);
    //1.5 * pow(w_star, 2) / pow(atm->z1, 2. / 3.);
    double C_T_squared = 2.67 * pow(pow(atm->z1 * atm->H / Cp, 2) * k
        * pow(atm->surface_shear_stress, 1. / 2.) / (pow(cond->h, 5)
        * pow(cond->rho, 5. / 2.)), 1. / 3.);
    //         2.67 * pow(theta_star, 2) * pow(cond->h / atm->z1,
    //         -4. / 3.) / pow(atm->z1, 2. / 3.);
    double L = cond->h; //altidute is upper bound on eddy size
    //         pow(u_star, 3) * Cp * cond->rho * cond->T
```

```
    //          / (von_Karman_const * g * atm->H);
    //equation I.53 in salomons, assume k_x = 0
    double K_0 = 2 * PI / L;
    double A = 5 / (18 * PI * tgamma(1. / 3.));
    return A / pow(pow(k, 2) + pow(K_0, 2), 8. / 6.) * (tgamma(1. / 2.)
        * tgamma(8. / 6.) * C_T_squared / (tgamma(11. / 6.) * 4 * pow(
        cond->T, 2)) + (tgamma(3. / 2.) * tgamma(8. / 6.) / tgamma(17.
        / 6.) + pow(k, 2) / (pow(k, 2) + pow(K_0, 2)) * tgamma(1. / 2.)
        * tgamma(14. / 6.) / tgamma(17. / 6.)) * 22 * C_v_squared / (12
        * cond->c));
  } else if (atm->turbulence_type == TURBULENCE_RMS) {
    //rms turbulence
    answer = atm->turbulence_rms * tgamma(8.0 / 6.0) * pow(
        atm->turbulenc_correlation_length, 2) / (tgamma(1.0 / 3.0) * PI
        * pow(
            1 + pow(k, 2) * pow(atm->turbulenc_correlation_length,
                2), 8.0 / 6.0));
  }
  return answer;
}

double refractive_index(Atmosphere_t *atm, Atmosphere_Conditions_t*cond,
    double r) {
  double dk = 0.05; //delta k
  double answer = 0;
  for (int i = 1; i <= 200; i++) {
    double kn = i * dk;
    double alpha_n = ((double) rand()) * 2 * PI / (double) RAND_MAX;
    answer += cos(kn * r + alpha_n) * sqrt(kn * von_karman(atm, cond, kn));
  }
  //TODO check the +1
  return sqrt(4 * PI * dk) * answer;
}

/*icao atmosphere implementation*/
```

```
/*
 * params:
 *   double altitude in m
 *   bool do_random, sets if the effective sound speed should include turbulence
 */
Atmosphere_Conditions_t get_atmosphere_conditions(Atmosphere_t * atm,
    double altitude) {
  //input check
  if (altitude < 0 || altitude > 86000) {
    Atmosphere_Conditions_t error = { -1.0, -1.0, -1.0, -1.0, -1.0, -1.0 };
    return error;
  }
  double altkm = altitude / 1000.; //altitude in km
  //find which layer we are in
  int layer = 0; //index to layers[]
  for (int i = 0; i < ICAO_LAYERS_LENGTH; i++) {
    if (altkm >= layers[i].h)
      layer = i;
  }
  //calculate conditions in layer
  Atmosphere_Conditions_t answer;
  answer.h = altitude;
  //calculate new temperature with linear
  answer.T = layers[layer].T + layers[layer].lapse_rate * (altkm
      - layers[layer].h);
  //calculate new pressure
  if (layers[layer].lapse_rate == 0) {
    //convert height in km to m to work with g
    answer.P = layers[layer].P * exp(-g * (altkm - layers[layer].h) * 1000
        / (R * answer.T));
  } else {
    //convert lapse rate (in km) to m
    answer.P = layers[layer].P * pow(answer.T / layers[layer].T, -g
        / (layers[layer].lapse_rate / 1000.0 * R));
  }
  //calculate density from ideal gas law
```

```
  answer.rho = answer.P / (R * answer.T);
  //sound speed
  answer.c = sqrt(CpCv * R * answer.T);
  //include turbulent fluctuations if we're asked for them
  //use the von_ karman spectrum, if in mixed layer, else no turbulence
  if (atm->turbulence_type == TURBULENCE_RMS || (atm->turbulence_type
      == TURBULENCE_WYLE && answer.h > 0.1 * atm->z1 && answer.h < 0.8
      * atm->z1)) {
    double ref_idx = refractive_index(atm, &answer, answer.h);
    answer.dc = answer.c * ref_idx;
  } else {
    answer.dc = 0;
  }
  return answer;
}

std::complex<double> get_attenuation(Atmosphere_t *atm, double Ps, double freq) {
  double T = atm->T0 * pow(Ps / atm->Ps0, (CpCv - 1) / CpCv);//tempurature
  double PsatOverPs0 = pow(10.0, -6.8348 * pow(T01 / T, 1.261) + 4.6151);
  double h = atm->relative_humidity * (PsatOverPs0) / (Ps / atm->Ps0);//absolute hu-
midity
  double FrN = (1 / atm->Ps0) * pow(atm->T0 / T, .5) * (9 + 280 * h * exp(
      -4.17 * (pow(atm->T0 / T, 1.0 / 3.0) - 1)));
  double FrO = (1 / atm->Ps0) * (24 + 4.04 * 10000 * h * (.02 + h) / (.391
      + h));
  double F = freq / Ps;
  double alphaTvb = Ps * pow(F, 2) * (1.84e-11 * pow(T / atm->T0, .5)
      * atm->Ps0);
  double alpharN = Ps * pow(F, 2) * (pow(T / atm->T0, -5.0 / 2.0) * (0.1068
      * exp(-3352.0 / T) / (FrN + pow(F, 2) / FrN)));
  double alpharO = Ps * pow(F, 2) * pow(T / atm->T0, -5.0 / 2.0) * (0.01275
      * exp(-2239.1 / T) / (FrO + pow(F, 2) / FrO));
  double alpha = alphaTvb + alpharN + alpharO;
  double betad = freq * (alpharN / (FrN * Ps) + alpharO / (FrO * Ps));
  std::complex<double> answer = std::complex<double>(alpha, betad);
  return answer;
```

```
}

std::complex<double> get_attenuation(attenuation_accel* accel, double freq) {
  //double FrN =
  //double FrO =
  double F2 = pow(freq / accel->Ps, 2); //F^2
  double alphaTvb = F2 * accel->alphaTvb_part_no_F;
  double alpharN = F2 * accel->alpharN_part_no_F / (accel->F_rN + F2
      / accel->F_rN);
  double alpharO = F2 * accel->alpharO_part_no_F / (accel->F_rO + F2
      / accel->F_rO);
  double alpha = alphaTvb + alpharN + alpharO;
  double betad = freq * (alpharN / (accel->F_rN_Ps) + alpharO
      / (accel->F_rO_Ps));
  std::complex<double> answer = std::complex<double>(alpha, betad);
  return answer;
}

void attenuation_accel_init(attenuation_accel* accel, Atmosphere_t *atm_const,
    Atmosphere_Conditions_t *atm_cond) {
  //double T = atm_const->T0 * pow(Ps / atm_const->Ps0, (CpCv - 1) / CpCv);//tempurature
  accel->T = atm_cond->T; //copy Temperature
  accel->Ps = atm_cond->P / 101325.0; //convert to atm from Pa
  double PsatOverPs0 = pow(10.0, -6.8348 * pow(T01 / accel->T, 1.261)
      + 4.6151);
  accel->h = atm_const->relative_humidity * (PsatOverPs0) / (accel->Ps
      / atm_const->Ps0);//absolute humidity
  accel->F_rN = (1 / atm_const->Ps0) * pow(atm_const->T0 / accel->T, .5) * (9
      + 280 * accel->h * exp(-4.17 * (pow(atm_const->T0 / accel->T, 1.0
          / 3.0) - 1)));
  accel->F_rO = (1 / atm_const->Ps0) * (24 + 4.04 * 10000 * accel->h * (.02
      + accel->h) / (.391 + accel->h));
  accel->alphaTvb_part_no_F = accel->Ps * (1.84e-11 * pow(accel->T
      / atm_const->T0, .5) * atm_const->Ps0);
  accel->alpharO_part_no_F = accel->Ps * pow(accel->T / atm_const->T0, -5.0
      / 2.0) * 0.01275 * exp(-2239.1 / accel->T);
```

```
  accel->alpharN_part_no_F = accel->Ps * pow(accel->T / atm_const->T0, -5.0
      / 2.0) * 0.1068 * exp(-3352.0 / accel->T);
  accel->F_rN_Ps = accel->F_rN * accel->Ps;
  accel->F_rO_Ps = accel->F_rO * accel->Ps;
}
```

A.9. InitialCondition.h

```
/*
 * InitialCondition.h
 * A stuct to hold the initial pressure time history with some meta information
 *
 *  Created on: Mar 2, 2010
 *      Author: evan
 */

#ifndef INITIALCONDITION_H_
#define INITIALCONDITION_H_

#include <complex>

struct InitialCondition{
  double dt;  //time step
  double r_0; //range that the points were measured at
  double src_alt; //altitude of the source
  int num_points; //length of the points array
  std::complex<double> * points;  //reference to array of points that is the ic in the time
domain
};

#endif /* INITIALCONDITION_H_ */
```

Vita Evan Ward

219 I Alley
State College, PA 16801
halful01@gmail.com

Education

Bachelor of Science, Aerospace Engineering with Honors in Aerospace Engineering and a Mathematics Minor, The Pennsylvania State University, University Park, PA. May, 2011.

Honors and Awards

College of Engineering Honors Program – The Leonhard Center, 2007-2011
Eagle Scout, 2006
National Merit Scholarship, Honorable Mention, 2006

Affiliations

American Institute of Aeronautics and Astronautics (AIAA)

References

Phillip J. Morris, Boeing, A.D. Welliver Professor of Aerospace Engineering, The Pennsylvania State University, University Park, PA. Tel: (814) 863-0157, Email: pjm@cac.psu.edu