INCOHERENT SCATTER MODELING OF JICAMARCA RADAR SPECTRA

BY

DEEPANSHU AGGARWAL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Erhan Kudeki

ABSTRACT

This thesis describes a project on the modeling of spectral characteristics of electron density irregularities of the topside equatorial ionosphere probed by the Jicamarca Incoherent Scatter Radar (ISR) located near Lima, Peru. The topside equatorial ionosphere is a multi-ion plasma and the spectrum of its electron density irregularities can be modeled by extending the single-ion spectral model developed by Kudeki and Milla (2011) for a collisional equatorial F-region ionosphere. This single-ion model of Kudeki and Milla captures the essential physics of the equatorial F-region ionosphere where random displacements of the dominant oxygen ions are characterized as a Brownian-motion process, while electron displacements are non-Brownian and described in terms of a numerical library (constructed using a Monte-Carlo approach) of single-electron ACFs (autocorrelation functions) parametrized by five state parameters of the F-region consisting of ionospheric electron density, geomagnetic flux density, electron and ion temperatures, and the deviation angle of the radar boresight direction from the plane perpendicular to the geomagnetic field, the so-called magnetic aspect angle. While the extension of the model to the multi-ion case is straightforward, the discrete nature of the numerical electron ACF library defined over a grid of input parameters precludes the evaluation of the extended model with arbitrary and continuously varying input parameters. To overcome this difficulty a machine learning (ML) based interpolation procedure is developed. The thesis describes the ML algorithm, the associated training and testing steps, and finally presents a suite of examples of multi-ion IS spectra obtained with the extended model.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Erhan Kudeki for his assistance in the preparation of this thesis and his guidance throughout my research. Many thanks go to graduate student Pablo Reyes for his technical support during my research. I also thank all the professors who taught me various courses and helped me develop a solid background in the field of electrical engineering. In addition, I would also like to thank my family for their constant support and motivation which made it possible for me to pursue higher education in the United States. Finally, I would like to thank all my friends at the University of Illinois for believing in me and supporting me at every step.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis describes a project on the modeling of spectral characteristics of electron density irregularities of the topside equatorial ionosphere probed by the Jicamarca Incoherent Scatter Radar (ISR) located near Lima, Peru. The topside equatorial ionosphere is a multi-ion plasma and the spectrum of its electron density irregularities can be modeled by extending the single-ion spectral model developed by [1] and [2] for a collisional equatorial F-region ionosphere. This single-ion model of [1] and [2] captures the essential physics of the equatorial F-region ionosphere where random displacements of the dominant oxygen ions are characterized as a Brownian-motion process, while electron displacements are non-Brownian and described in terms of a numerical library (constructed using a Monte-Carlo approach) of single-electron ACFs (auto-correlation functions) parametrized by five state parameters of the F-region consisting of ionospheric electron density, geomagnetic flux density, electron and ion temperatures, and the deviation angle of the radar boresight direction from the plane perpendicular to the geomagnetic field, the so-called magnetic aspect angle. While the extension of the model to the multi-ion case is straightforward, the discrete nature of the numerical electron ACF library defined over a grid of the input parameters precludes the evaluation of the extended model with arbitrary and continuously varying input parameters. To overcome this difficulty a machine learning (ML) based interpolation procedure is developed. The thesis describes the ML algorithm, the associated training and testing steps, and finally presents a suite of examples of the multi-ion IS spectra obtained with the extended model.

1.1 Overview of Past Ionospheric Experiments

The ionosphere is an ionized layer of the upper atmosphere extending from about 60 km altitude to beyond 600 km. Studies of this layer using radar scattering techniques are motivated by its impact on radio and satellite communications utilizing trans-ionospheric paths and links. Solar radiation provides the energy input for the ionization of major neutral species of the atmosphere to generate the ionosphere. The process creates electrical conductivity in the ionized atmosphere which affects the propagation of the radio waves used for communications and radar scatter applications. Thomson scattering refers to re-radiation of electromagnetic waves by oscillating electrons, and the term "incoherent scatter" applies to Thomson scattering of electromagnetic radar pulses from collections of ionospheric free electrons in thermal equilibrium. Thomson scattered signal power can be shown to be related to the variance of the spatial Fourier transform of the electron density fluctuations at the Bragg wave vector having a magnitude twice the wavenumber of the radar carrier and a direction coinciding with the propagation direction of the backscattered electromagnetic wave. The scattered power spectrum depends on electron and ion densities, temperatures, the drift velocity, and the collision frequencies between plasma particles. Radar measurements of the backscattered signal spectra provide a means of estimating these parameters and studying the physics of the ionosphere.

The incoherent scatter radar located at the Jicamarca Radio Observatory near Lima, Peru, is extensively used for the studies of the equatorial ionosphere. The radar is capable of obtaining high-resolution and high-accuracy F-region plasma drift measurements by pointing its antenna perpendicular to Earth's magnetic field $\mathbf{B}$. Such measurements can be made on "a pulse-to-pulse basis" because the bandwidth of the ISR spectrum narrows considerably at small magnetic aspect angles [2], which is the offset angle of the radar beam direction from the plane perpendicular to the geomagnetic field. Incoherent scatter spectral models for collisional and magnetized F-region plasmas at all magnetic aspect angles were developed based on a general framework described in [1] and [2]. The model in [2] is derived in terms of single particle ACFs of the electrons and ions comprising the collisional and magnetized F-region plasma. These ACFs correspond to the characteristic functions

$\langle e^{j\vec{k}\cdot\Delta\vec{r}} \rangle$ of random particle (electron, ion) displacements $\Delta\vec{r}$. While ion displacements in the F-region plasma are described by a Brownian motion process leading to an analytic ACF expression, electron displacements are a Gauss-Markov process and were described in [2] in terms of a numerical ACF library constructed by Monte-Carlo simulations. While the numerical library was originally constructed for a single-ion plasma, it can also be used in the multi-ion case in combination with Brownian ion ACF functions. ISR spectra are constructed from a linear combination of one-sided Fourier transforms of the ACFs known as Gordeyev integrals. The discretized ACFs in the numerical electron library can be viewed as vectors or tuples $Y = (y_1, y_2, \cdots, y_d)$ defined over a time-delay variable τ, each tuple being a mapping of an ionospheric state-parameter tuple $X \equiv (N_e, B, T_e, T_i, \alpha)$, where N_e is the electron density, B is the magnetic flux density, T_e and T_i are electron and ion temperatures, and α is the magnetic aspect angle. In the library provided by [2] tuples Y exist only for tuples X defined over a discrete grid.

1.2 Outline

The thesis is organized into four chapters.

Chapter 2 covers the theory of incoherent scatter process, the experimental setup of Jicamarca incoherent scatter measurements, and a summary of research of Kudeki and Milla reported in [1] and [2]. The chapter starts with a description of Thomson scattering and how that is utilized in ionospheric research. Next, it provides a short description of incoherent scatter spectral models for ionospheric F-region plasmas with O^+ ions. Finally, a generalization of the single-ion spectral model to the multi-ion case, suitable for topside F-region studies, is described — this amounts to including the effect of hydrogen and helium ions in the formulation. Sample ISR spectra for the topside plasma are presented corresponding to states X_i available in the discretized electron ACF library of [1] and [2].

Chapter 3 describes a machine learning (ML) approach to interpolate the electron ACFs Y_i corresponding to discrete grid states X_i, $i \in [1, n]$ available in the numerical library of [2] to arbitrary $Y(X)$ for X in a five-dimensional space spanned by the available X_i.

The approach includes training and testing steps and the use of a trained data object to obtain $Y(X)$ interpolants. An ML strategy known as *k-nearest neighbor* (KNN) regression was utilized in $Y(X)$ computations. KNN was implemented using the Python ML tool-kit "scikit-learn" [3]. The chapter ends with examples of ISR spectra computed using the interpolated electron ACF's $Y(X)$ obtained with the KNN method.

Chapter 4 provides the results, conclusion and the suggestions for future research that builds on the work presented in this thesis.

CHAPTER 2

INCOHERENT SCATTER THEORY

The main content for this chapter is extracted from the papers by Kudeki and Milla [1] and Milla and Kudeki [2]. It starts with a brief description of Thomson-scatter-based incoherent scatter radar (ISR) technique. Next, a general framework of incoherent scatter spectral theories expressed in terms of the characteristics functions of charged particle displacements in thermal equilibrium is presented. Ionospheric applications of the general framework are covered with a focus on single-ion O^+ plasmas dominant at F-region heights. Finally, the description of how the framework can be used to develop multiple-ion spectral models by modifying the single-ion case is presented.

2.1 Thomson Scattering and IS Signal Spectrum

Ionospheric radar scattering from a collection of free electrons can be modeled by using the Thomson-scattering result of a single free electron. When a free electron is exposed to a time-harmonic TEM wave, it accelerates and radiates a scattered TEM wave [1]. This oscillating electron acts effectively as a Hertzian dipole. Its radiation field can be obtained from the result for Hertzian dipole as

$$\mathbf{E}_r(\mathbf{r}) = \hat{\theta}\sin\theta\frac{r_e}{r}E_i e^{-jkr}, \tag{2.1}$$

where $r_e = \frac{e^2}{4\pi\epsilon_o mc^2} \approx 2.81\times10^{-15}$ m is electron radius, $\hat{\theta}$ is the unit vector in the direction of increasing θ, and E_i is the incident field phasor.

Next, consider the case when a radar antenna is radiating from the origin and the target free electron is located at some radar range r. Taking the origin as phase reference and using the radiation field (2.1), the backscattered field phasor from the electron back to the origin is expressed as

$$E_s = -\frac{r_e}{r} E_i e^{-jk_o r} = -\frac{r_e}{r} E_o(\mathbf{r}) e^{-j2k_o r}, \tag{2.2}$$

where $k_o = \omega_o / c$ is the incident wave number at the radar wave frequency ω_o and $E_o(\mathbf{r})$ is a slowly varying incident field amplitude illuminating the electron with $E_i = E_o(\mathbf{r}) e^{-jk_o r}$ [1].

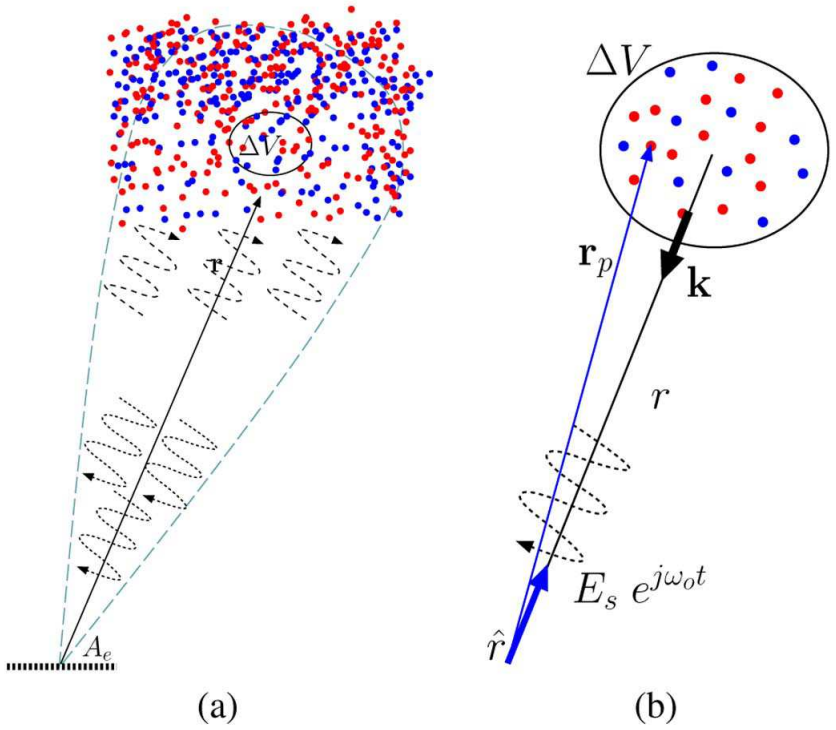

Figure 2.1 – (a) Radar scattering volume defined by the antenna (b) Geometry of a subvolume ΔV of the scattering volume

Utilizing the single electron model developed above, a model for ionospheric radar scattering from a collection of free electrons is built next. The backscattered field phasor from a subvolume ΔV as depicted in Figure 2.1 (b) is calculated by superposing the field contributions of the form (2.2) from each electron in ΔV. Thus, the backscattered signal

from the entire ΔV is

$$E_s = -\sum_{p=1}^{N_o \Delta V} \frac{r_e}{r_p} E_{op} e^{-j2k_o r_p} \approx -\frac{r_e}{r} E_o \sum_{p=1}^{N_o \Delta V} e^{-j\mathrm{k}\cdot\mathrm{r}_p}, \tag{2.3}$$

where r_p is the radar range of each electron, E_{op} is the incident field amplitude at each electron location. The right side expression in (2.3) is the paraxial approximation valid for $r > 4\Delta V^{2/3}/\lambda_o$ (far-field condition) for an antenna of size $\Delta V^{1/3}$ and operating wavelength $\lambda = \lambda_o/2$ [1]. Also, $\mathbf{k} = -2k_o\hat{r}$ denotes the Bragg vector and N_o is the average density of the electrons in the subvolume [1].

Next, considering $r_p(t)$ as the trajectories of the individual electrons, the backscattered signal (2.3) can be written as

$$E_s(t) = -\frac{r_e}{r} E_i \sum_{p=1}^{N_o \Delta V} e^{-j\mathrm{k}\cdot\mathrm{r}_p(t-\frac{r}{c})} \equiv -\frac{r_e}{r} E_i n_e(\mathbf{k}, t - \frac{r}{c}), \tag{2.4}$$

where $\frac{r}{c}$ is the propagation delay from the center of ΔV to the radar antenna and $n_e(\mathbf{k}, t) = \sum_{p=1}^{N_o \Delta V} e^{-j\mathrm{k}\cdot\mathrm{r}_p(t)}$ is the electron density denoting the 3-D spatial Fourier transform of $n_e(\mathbf{r}, t)$ [1]. Here, Equation (2.4) denotes that the scattered field amplitude from ΔV varying in time is a scaled and delayed replica of the 3-D spatial Fourier transform of the electron density. This will help to find the relationship between the electron density variations on the region probed by the radar and the radar signal statistics [1].

The density function $n_e(\mathbf{r}, t)$ and its Fourier transform $n_e(\mathbf{k}, t)$ should be modeled as random processes since individual particle trajectories $\mathbf{r}_p(t)$ are unpredictable. The radar signals will then depend on the normalized variance $\frac{1}{\Delta \mathrm{V}} \left\langle |n_e(\mathbf{k}, t)|^2 \right\rangle$ and normalized autocorrelation function (ACF) $\frac{1}{\Delta \mathrm{V}} \langle n_e^*(\mathbf{k}, t) n_e(\mathbf{k}, t + \tau) \rangle$. Also, normalized ACF can be Fourier transformed to obtain the space-time or $\mathbf{k} - \omega$ spectrum of $n_e(\mathbf{r}, t)$ as

$$\left\langle |n_e(\mathbf{k}, \omega)|^2 \right\rangle \equiv \int d\tau e^{-j\omega\tau} \frac{1}{\Delta V} \times \left\langle n_e^*(\mathbf{k}, t - \frac{r}{c}) n_e(\mathbf{k}, t - \frac{r}{c} + \tau) \right\rangle. \tag{2.5}$$

Next, following [1], the $\mathbf{k}-\omega$ spectrum (2.5) can be expanded as

$$\left\langle |n_e(\mathbf{k},\omega)|^2 \right\rangle \equiv \int d\tau e^{-j\omega\tau} \frac{1}{\Delta V} \times \sum_{p=1}^{N_o\Delta V} \sum_{q=1}^{N_o\Delta V} \left\langle e^{-j\mathrm{k}\cdot\mathrm{r}_p(t-\frac{\mathrm{r}}{c})} e^{-j\mathrm{k}\cdot\mathrm{r}_q(t-\frac{\mathrm{r}}{c}+\tau)} \right\rangle. \tag{2.6}$$

Here, assuming that the individual electrons in ΔV follow independent random trajectories, so that Equation (2.6) is zero for $q = p$, leads to

$$\left\langle |n_e(\mathbf{k},\omega)|^2 \right\rangle \equiv N_o \int d\tau e^{-j\omega\tau} \left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle \equiv \left\langle |n_{te}(\mathbf{k},\omega)|^2 \right\rangle, \tag{2.7}$$

where $\Delta\mathrm{r} \equiv \mathbf{r}_p(t-\frac{\mathrm{r}}{c}+\tau) - \mathbf{r}_p(t-\frac{\mathrm{r}}{c})$ is the random displacement vector of the electrons.

Next, the description of collective interaction effects on ionospheric backscatter is presented. Ionospheric plasmas consist of charged particles with random motions of thermal origin [1]. Including the space-charge imbalances that perturb the trajectories of the electron motion, the electron density in the volume ΔV can be written as

$$n_e(\mathbf{r},t) = n_{te}(\mathbf{r},t) + \delta n_e(\mathbf{r},t), \tag{2.8}$$

and similarly the positive ion density within ΔV can be written as

$$n_i(\mathbf{r},t) = n_{ti}(\mathbf{r},t) + \delta n_i(\mathbf{r},t), \tag{2.9}$$

where $\delta n_e(\mathbf{r},t)$ and $\delta n_i(\mathbf{r},t)$ account for the density charges caused by macroscopic forces, also called collective interactions.

Next, it is shown that the spectrum of electron density fluctuations is a weighted superposition of electron density spectrum and ion density spectrum in the absence of the collective interactions. Following [1], implementing the electrical circuit model for ionospheric plasma in terms of n_e and n_i as well as their associated currents, and applying Kirchhoff's current law to the circuit, the electron density fluctuation amplitude can be found using

$$n_e(\mathbf{k},\omega) = \frac{(j\omega\epsilon_o + \sigma_i) n_{te}(\mathbf{k},\omega) + \sigma_e n_{ti}(\mathbf{k},\omega)}{j\omega\epsilon_o + \sigma_e + \sigma_i}. \tag{2.10}$$

By squaring and averaging the electron density fluctuation (2.10), the electron density spectrum can be written as

$$\left\langle |n_e(\mathbf{k},\omega)|^2 \right\rangle = \frac{|j\omega\epsilon_o + \sigma_i|^2 \left\langle |n_{te}(\mathbf{k},\omega)|^2 \right\rangle}{|j\omega\epsilon_o + \sigma_e + \sigma_i|^2} + \frac{|\sigma_e|^2 \left\langle |n_{ti}(\mathbf{k},\omega)|^2 \right\rangle}{|j\omega\epsilon_o + \sigma_e + \sigma_i|^2}. \tag{2.11}$$

The electron density spectrum (2.11) is a sum of electron and ion-line components proportional to $\left\langle |n_{te}(\mathbf{k},\omega)|^2 \right\rangle$ and $\left\langle |n_{ti}(\mathbf{k},\omega)|^2 \right\rangle$ respectively. It is a very important result which is used in [1] and [2] to generate spectrum plots for single ion O^+. The electron density spectrum (2.11) can be modified to obtain multi-ion case which is covered in Section 2.3. To use the electron density spectrum (2.11) for calculations, the conductivity expression $\sigma_{e,i}(\mathbf{k},\omega)$ has to be known. This expression can be derived by analyzing the links between $\sigma_{e,i}(\mathbf{k},\omega)$ and $e^{j\mathbf{k}\cdot\Delta\mathbf{r}}$ statistics of the plasma particles.

As mentioned in [1], there are three relationships that play crucial roles in forming the density spectra $\left\langle |n_{te,ti}(\mathbf{k},\omega)|^2 \right\rangle$. The relationships are:

- Nyquist noise theorem
- Fluctuation-dissipation theorem
- Kramers-Kronig

The Nyquist theorem describes the thermal noise and its spectrum. The fluctuation-dissipation theorem generalizes the Nyquist noise theorem to all dissipative systems in terms of their equivalent circuit models [1]. Specifically it requires that ionospheric current fluctuations must have two-sided spectra

$$\frac{\omega^2}{k^2} e^2 \left\langle |n_{te,i}(\mathbf{k},\omega)|^2 \right\rangle = 2KT_{e,i}\mathrm{Re}\left\{\sigma_{e,i}(\mathbf{k},\omega)\right\} \tag{2.12}$$

per unit bandwidth and per species. The two-sided spectra (2.12) show that by knowing $\left\langle |n_{te,i}(\mathbf{k},\omega)|^2 \right\rangle$, i.e. $\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_{e,i}} \right\rangle$ using the $\mathbf{k}-\omega$ spectrum (2.7), the real part of conductivities $\mathrm{Re}\left\{\sigma_{e,i}(\mathbf{k},\omega)\right\}$ can be calculated. Finally, Kramers-Kronig relations show that the

imaginary part of conductivities $\mathrm{Im}\{\sigma_{e,i}(\mathbf{k},\omega)\}$ can be calculated by taking the Hilbert transform of $\mathrm{Re}\{\sigma_{e,i}(\mathbf{k},\omega)\}$. Hence, once $\left\langle |n_{te,i}(\mathbf{k},\omega)|^2 \right\rangle$ and $\mathrm{Re}\{\sigma_{e,i}(\mathbf{k},\omega)\}$ are known, $\sigma_{e,i}(\mathbf{k},\omega)$ can be calculated.

So, to compute the electron density fluctuation spectrum, it is sufficient to find the characteristic function $\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_{e,i}} \right\rangle$ of particle displacement $\Delta\mathrm{r}_{e,i}$ in the absence of collective interaction. From now on, these functions will be called single-particle ACFs (autocorrelation functions) as $e^{j\mathbf{k}\cdot\mathbf{r}_e}$ represent backscattered radar signal from an electron.

The characteristic function of a particle s (e or i) can be used to define an expression called the Gordeyev integral:

$$J_s(\omega) = \int_0^\infty d\tau e^{-j\omega\tau} \left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_s} \right\rangle, \tag{2.13}$$

and following [1], we have

$$\frac{\left\langle |n_{ts}(\mathbf{k},\omega)|^2 \right\rangle}{N_o} = 2\mathrm{Re}\left\{ \int_0^\infty d\tau e^{-j\omega\tau} \left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_s} \right\rangle \right\} = 2\mathrm{Re}\left\{ J_s(\omega_s) \right\} \tag{2.14}$$

$$\frac{\sigma_s(\mathbf{k},\omega)}{j\omega\epsilon_o} = \frac{1 - j\omega_s J_s(\omega_s)}{k^2 h_s^2}, \tag{2.15}$$

where ω_s is a Doppler-shifted frequency in the radar frame of species s, $h_s \equiv \sqrt{\epsilon_o K T_s / N_o e^2}$ is the Debye length. The electron density spectrum expression (2.11) together with (2.13) - (2.15) constitutes the general framework of ISR spectral models. The Gordeyev integral $J(\omega)$ mentioned above is calculated by numerical integration. Specifically, the technique of chirp-z transform can be used here with the ACFs [1].

2.2 Incoherent Scatter Spectral Model: Ionospheric Applications

The ionospheric applications of the general framework developed in Section 2.1 are presented in this section. The single particle ACF $\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle$ discussed in the previous

section can be computed by knowing either the pdf $f(\Delta \mathrm{r})$ or the set of realizations of $\Delta \mathrm{r}$ for a given time delay τ. The $f(\Delta \mathrm{r})$ and $\Delta \mathrm{r}$ tell us the dynamics of the random particle motions in the ionosphere.

There are different types of ionospheric plasmas considered in [1]. We will see how the general framework developed in Section 2.1 can be used to develop different spectral models. The different kinds of ionospheric plasma are

- Collisionless non-magnetized plasma
- Collisional non-magnetized plasma
- Magnetized plasma

In collisionless non-magnetized plasma, the charge particles will travel along a straight line with random velocities (let us denote it with v) giving rise to displacement vectors τ

$$\Delta \mathrm{r} = \mathrm{v}\tau \tag{2.16}$$

over the interval τ. In a Maxwellian plasma this will lead to Gaussian distributed displacements $\Delta \mathrm{r}$ with pdf $f(\Delta r)$ as

$$f(\Delta \mathrm{r}) = \frac{e^{\frac{-\Delta \mathrm{r}^2}{2\langle \Delta \mathrm{r}^2 \rangle}}}{\sqrt{2\pi \langle \Delta \mathrm{r}^2 \rangle}} \tag{2.17}$$

and a variance

$$\langle \Delta \mathrm{r}^2 \rangle = \langle v^2 \rangle \tau^2 = C^2 \tau^2, \tag{2.18}$$

where $C = \sqrt{KT/m}$ is the thermal speed of the charge carrier. This gives us the single particle ACF as

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = \int e^{j\mathbf{k}\cdot\Delta\mathbf{r}} f(\Delta \mathrm{r}) d(\Delta \mathrm{r}) = e^{-\frac{1}{2}k^2 C^2 \tau^2}. \tag{2.19}$$

The single particle ACF (2.19) is the most basic incoherent scatter spectral model [1].

Now, moving towards a more realistic plasma model, consider a collisional non-magnetized

plasma. Consider two cases: one where collision frequency $\nu \ll kC$ and the other where $\nu \gg kC$. The expression (2.19) should also be valid for collisional non-magnetized plasmas as long as $\nu \ll kC$. Long-range Coulomb collisions between electrons and ions are modeled as a Brownian motion process [1], which is given by Gaussian $f(\Delta \mathrm{r})$ with a variance

$$\left\langle \Delta \mathrm{r}^2 \right\rangle = \frac{2C^2}{\nu^2}(\nu\tau - 1 + e^{-\nu\tau}). \tag{2.20}$$

Based on variance (2.20), the ACF is given by

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = e^{-\frac{kC^2}{\nu^2}(\nu\tau - 1 + e^{-\nu\tau})} \tag{2.21}$$

for $\nu \ll kC$ and

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = e^{-\frac{kC^2}{\nu}\tau} \tag{2.22}$$

for $\nu \gg kC$. In case of $\nu \ll kC$, the average particle moves a distance of many wavelengths in between successive collisions, whereas when $\nu \gg kC$, the average particle moves across only a small fraction of a wavelength between successive collisions.

Binary collisions of charge carriers with atoms and molecules can be modeled as a Poisson process. As discussed in [1], the collisional spectra will exhibit minor differences between binary and Coulomb collisions except in the case when $\nu \gg kC$ and $\nu \ll kC$.

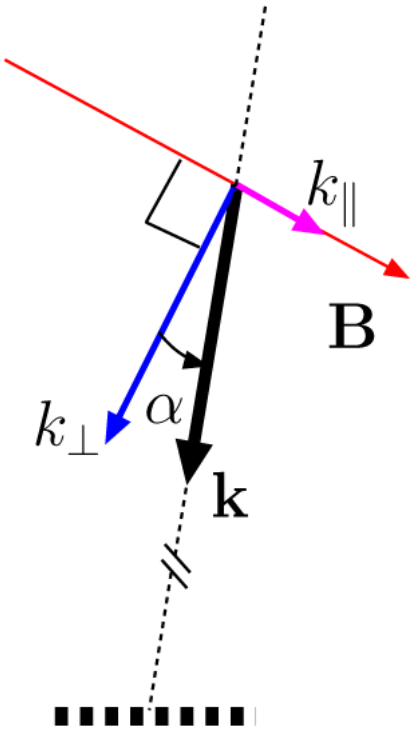

Figure 2.2 – Backscattering in magnetized ionosphere

Now, following [1] introduce the effect of an ambient magnetic field $\vec{B}$. Express the Bragg wave vector as $\mathbf{k} = \hat{b}k_{\parallel} + \hat{p}k_{\perp}$, where $\hat{b}$ and $\hat{p}$ are orthogonal vectors on the **k-B** plane (parallel and perpendicular to **B**) as shown in Figure 2.2. The single particle ACF in this case can be expressed as

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = \left\langle e^{jk_{\parallel}\Delta \mathrm{l}} \times e^{jk_{\perp}\Delta \mathrm{p}} \right\rangle, \tag{2.23}$$

where Δl and Δp are particle displacements along unit vectors $\hat{b}$ and $\hat{p}$. Following [1], in case of collisionless ionosphere,

$$\left\langle \Delta \mathrm{l}^2 \right\rangle = C^2\tau^2, \tag{2.24}$$

$$\left\langle \Delta \mathrm{p}^2 \right\rangle = \frac{4C^2}{\Omega^2}\sin^2(\Omega\tau/2) \tag{2.25}$$

and using Equations (2.24) and (2.25), the single-particle ACF for electrons and ions will be

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = e^{-\frac{1}{2}k_{\parallel}^2 C^2\tau^2} \times e^{-\frac{2k_{\perp}^2 C^2}{\Omega^2}\sin^2(\Omega\tau/2)}, \tag{2.26}$$

where $\Omega = qB/m$ is the particle gyrofrequency.

Finally, consider the effect of Coulomb collisions in a magnetized ionosphere. The Coulomb collision model was developed by Kudeki, Bhattacharyya and Woodman [4] using Fokker-Planck kinetic equation that is consistent with the Brownian motion process. The model leads to independent and Gaussian particle displacements given by

$$\left\langle \Delta \mathrm{l}^2 \right\rangle = \frac{2C^2}{\nu^2}(\nu\tau - 1 + e^{-\nu\tau}) \tag{2.27}$$

$$\left\langle \Delta \mathrm{p}^2 \right\rangle = \frac{2C^2}{\nu^2 + \Omega^2}(\cos(2\gamma) + \nu\tau - e^{-\nu\tau}\cos(\Omega\tau - 2\gamma)) \tag{2.28}$$

where $\gamma = \tan^{-1}\nu/\Omega$. Applying these variances in Equation (2.23) assuming independent Gaussian random variables, it follows that

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle \to \exp\left[-\frac{k^2C^2}{\Omega^2 + \nu^2}(\cos(2\gamma) + \nu\tau - e^{-\nu\tau}\cos(\Omega\tau - 2\gamma)) \right]. \tag{2.29}$$

2.3 Single and Multiple Ion(s) ACFs Plots

The incoherent scatter radar at Jicamarca used to study the F-region ionospheric plasma above the equator is a 50-MHz radar system. Jicamarca plasma drift measurements are conducted using antenna beams pointed almost perpendicularly to Earth's magnetic field. Theory presented [1] and [2] shows how to model the spectrum of incoherent scattered radar signal detected from the collisional, magnetized, and single-ion (ionized oxygen atoms) F-region plasma at small magnetic aspect angles. The application of the model in the topside ionosphere containing ionized hydrogen and helium atoms requires the following steps:

- Compute the Gordeyev integral (2.13) for electron, hydrogen, helium and oxygen.
- Replace the Gordeyev integral in [1] and [2] with a weighted sum of the oxygen,

helium, and hydrogen Gordeyev integrals to get the spectrum. The multi-ion ISR spectrum can be written as

$$\left\langle \left|n_e(\mathbf{k},\omega)\right|^2 \right\rangle = \frac{\left|j\omega\epsilon_o + \sum \sigma_i\right|^2 \left\langle \left|n_{te}(\mathbf{k},\omega)\right|^2 \right\rangle}{\left|j\omega\epsilon_o + \sigma_e + \sum \sigma_i\right|^2} + \frac{\left|\sigma_e\right|^2 \left\langle \left|\sum n_{ti}(\mathbf{k},\omega)\right|^2 \right\rangle}{\left|j\omega\epsilon_o + \sigma_e + \sum \sigma_i\right|^2}, \tag{2.30}$$

where $i \in [1, 2, 3]$ is for ions oxygen, hydrogen and helium. The Gordeyev integral for species s (electrons or ions) is a one-sided Fourier transform of the species single-particle ACF which in turn is dependent on Coulomb collision frequencies $\nu_{s/e}$ with electrons and $\nu_{s/i}$ with ions. The Coulomb collision of species s with species p is calculated by defining species and plasma Debye lengths [2]

$$h_s = \sqrt{\frac{\epsilon_o K T_s}{N_s q_s^2}} \tag{2.31}$$

and

$$h_D = \frac{1}{\sqrt{\sum_s h_s^{-2}}}, \tag{2.32}$$

and a minimum impact parameter

$$b_{min,s/p} = \frac{q_s q_p}{12\pi\epsilon_o m_{sp} C_{sp}^2}, \tag{2.33}$$

where $m_{sp} = \frac{m_s m_p}{m_s + m_p}$ is a reduced mass and $C_{sp}^2 = C_s^2 + C_p^2$, and logarithm of plasma parameter

$$\log_e\left(\Lambda_{s/p}\right) = \log_e\left(\frac{h_D}{b_{min,s/p}}\right). \tag{2.34}$$

With these inputs, the Coulomb collision frequency is calculated as [5]

$$\nu_{s/p} = \frac{N_p q_s^2 q_p^2 \log_e\left(\Lambda_{s/p}\right)}{3(2\pi)^{\frac{3}{2}} \epsilon_o^2 m_s m_{sp} C_{sp}^3}. \tag{2.35}$$

Once the collision frequencies $\nu_{e/p}$ are available, the electron friction coefficients $\nu_\parallel$ and $\nu_\perp$ in the direction parallel and perpendicular to $\mathbf{B}$ can be calculated using Equations (49)

and (50) of [2], namely $\nu_{\parallel} = \nu_{e/O} + \nu_{e/H_e} + \nu_{e/H}$ and $\nu_{\perp} = \nu_{\parallel} + \nu_{e/e}$. The Coulomb collision frequency for ionic species s is given by $\nu_s = \sum_x \nu_{s/x}$, where x is the set of all the elements including electrons and species s.

Following [2], the autocorrelation function in a magnetized and collisional plasma is calculated as

$$\left\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \right\rangle = e^{-\frac{1}{2}k^2 \sin^2\alpha \left\langle \Delta r_{\parallel}^2 \right\rangle} \times e^{-\frac{1}{2}k^2 \cos^2\alpha \left\langle \Delta r_{\perp}^2 \right\rangle}, \tag{2.36}$$

where

$$\begin{aligned} \left\langle \Delta r_{\parallel}^2 \right\rangle &= \frac{2C^2}{\nu_{\parallel}^2} \left(\nu_{\parallel}\tau - 1 + e^{-\nu_{\parallel}\tau} \right) \\ \left\langle \Delta r_{\perp}^2 \right\rangle &= \frac{2C^2}{\nu_{\perp}^2 + \Omega^2} \left(\cos(2\gamma) + \nu_{\perp}\tau - e^{-\nu_{\perp}\tau} \cos(\Omega\tau - 2\gamma) \right) \end{aligned} \tag{2.37}$$

in terms of parameters already obtained earlier in this section, and the Gordeyev integral (2.13) can be obtained by taking the chirp-z transform [6] of the autocorrelation function.

Figures 2.3 - 2.6 show the electron and ion Gordeyev integral plots, the overall ISR spectrum, and the corresponding ACF for a particular ionospheric state X_i from the numerical ACF library, as specified in the captions of Figures 2.7 and 2.8. Figure 2.7 shows the $(\mathbf{k} - \omega)$ spectrum of electron density fluctuations obtained with the ISR autocorrelation function according to the electron density spectrum (2.30). To calculate the ISR autocorrelation function shown in Figure 2.8, we take the chirp-z transform [6] of the ISR ion-line spectrum.

The numerical electron ACF library used above was created with a high sampling rate that generated 131072 time samples for the ACF data. A comparison of electron Gordeyev integral with a downsampling (DS) of 0, 100, 1000 and 10000 is performed. Figures 2.9 - 2.12 correspond to DS rate of 0, 100, 1000 and 10000, and show that using fewer time samples would not affect the ACF plots up to a certain level of DS. It can be clearly seen that with a DS of 10000, the ACF plot became sharp at the edges; i.e., it lost its smoothness.

Given the above results, it was decided that it is useful to produce a 1:1000 downsampled version of the numerical ACF library. The total size of this new library on the disk

is 110 MB as compared to 93 GB for the original library. The library is saved on the disk using a Python's module called *pickle*. The library can be loaded whenever needed using the same module. This new library will be utilized in the machine learning tool presented in Chapter 3 and 4.

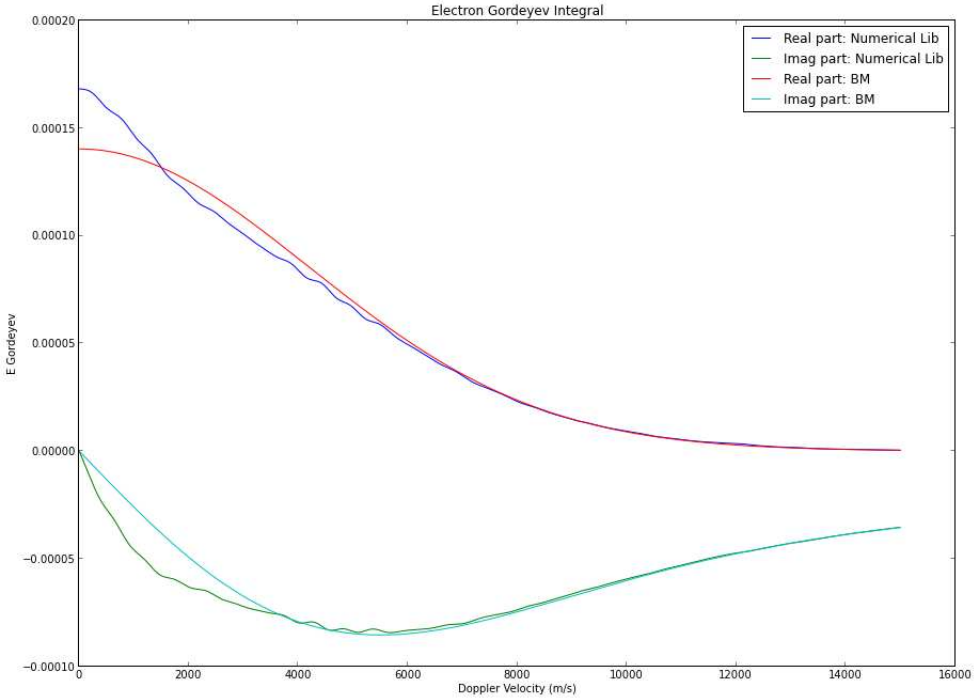

Figure 2.3 – Electron Gordeyev integral (Ge, GB)

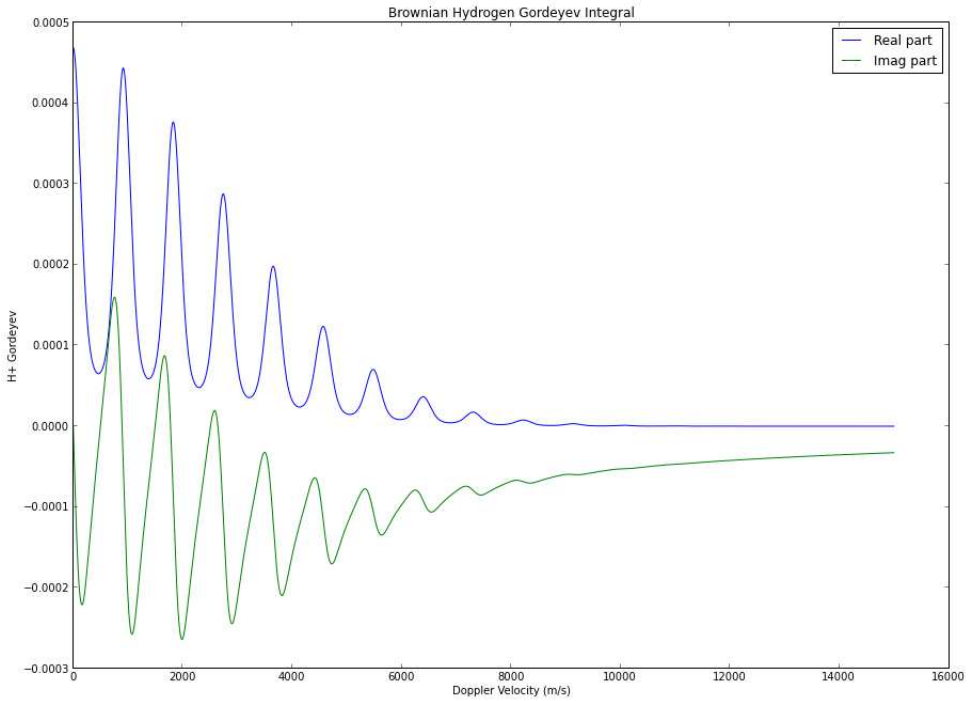

Figure 2.4 – Hydrogen Gordeyev integral (GH)

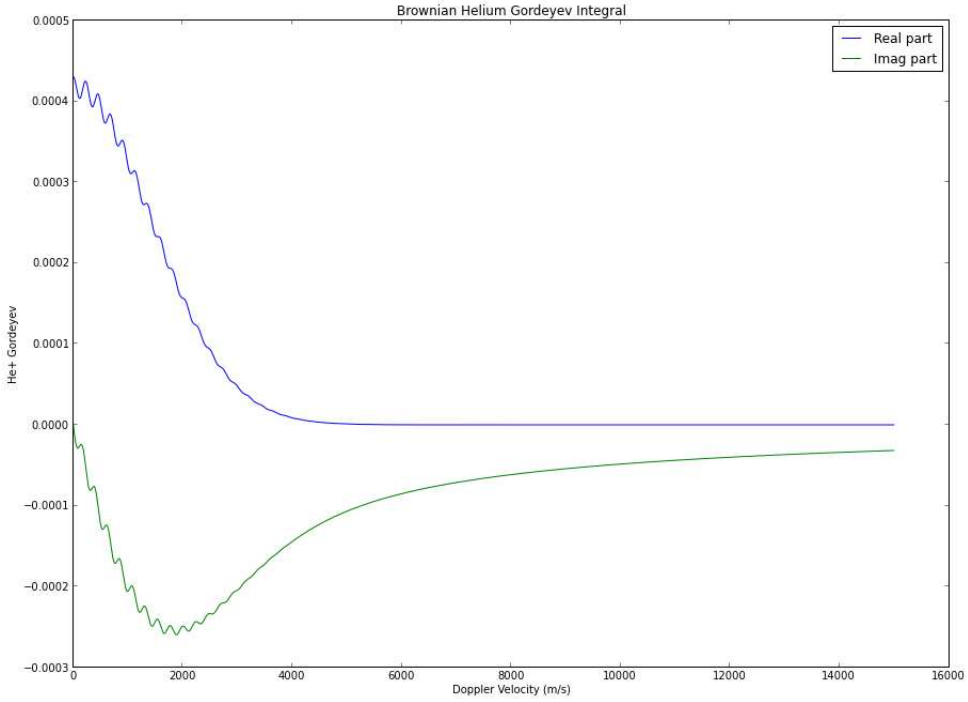

Figure 2.5 – Helium Gordeyev integral (G4)

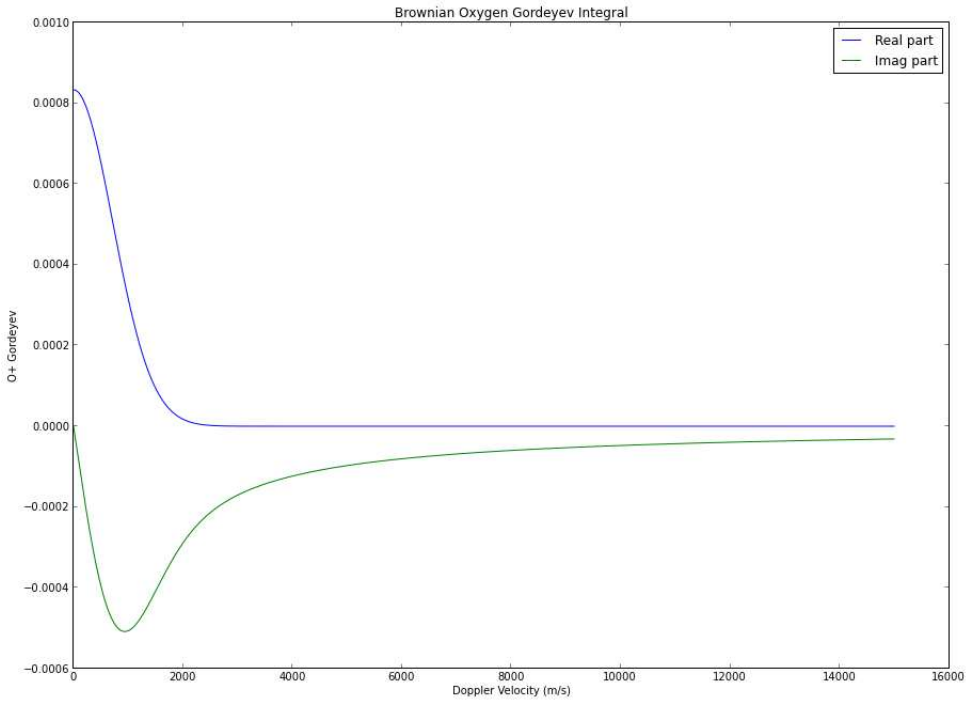

Figure 2.6 – Oxygen Gordeyev integral (GO)

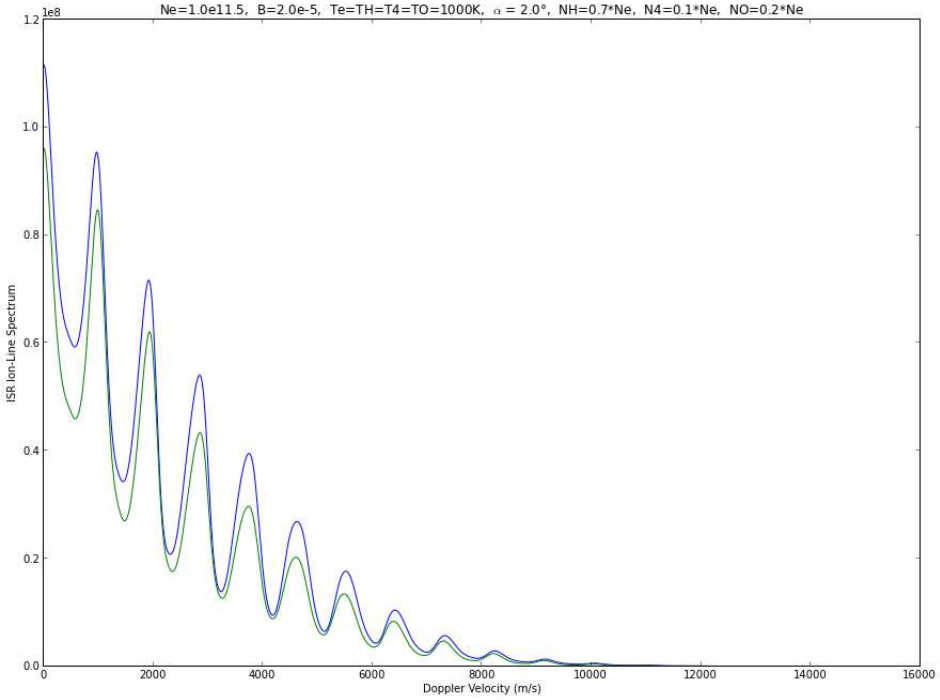

Figure 2.7 – ISR ion-line spectrum

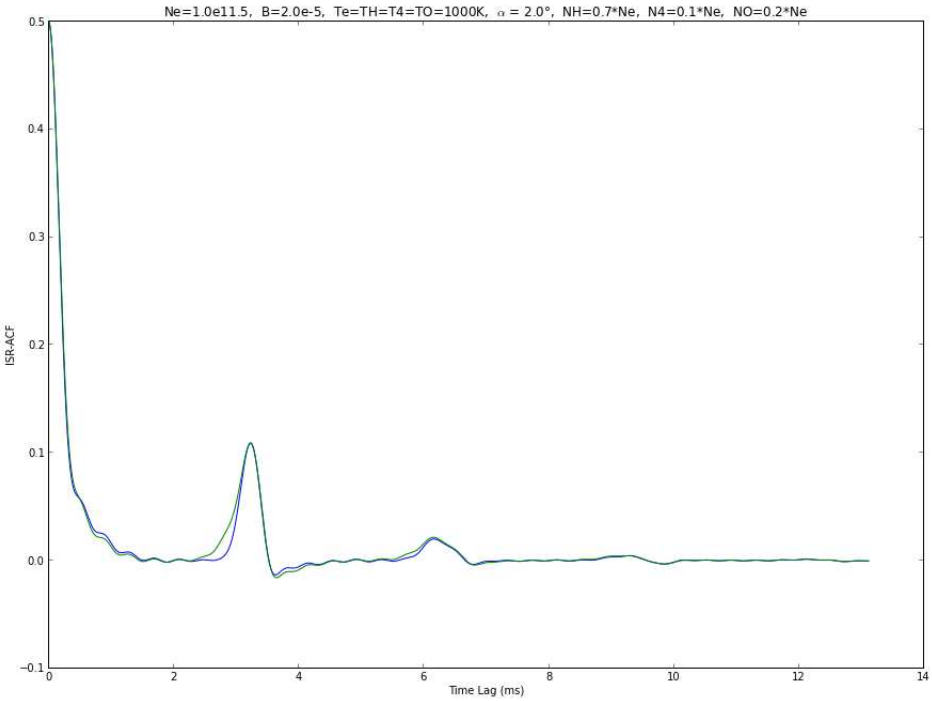

Figure 2.8 – ISR ACF

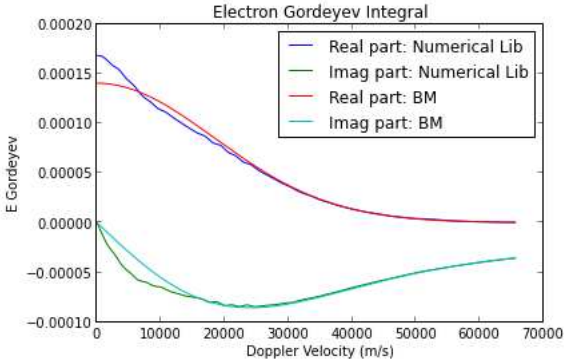

Figure 2.9 – Electron Gordeyev integral with DS = 0

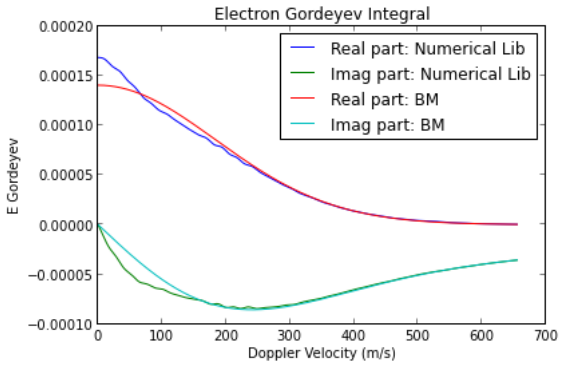

Figure 2.10 – Electron Gordeyev integral with DS = 100

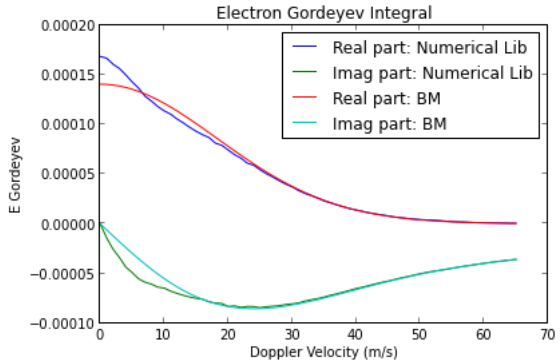

Figure 2.11 – Electron Gordeyev integral with DS = 1000

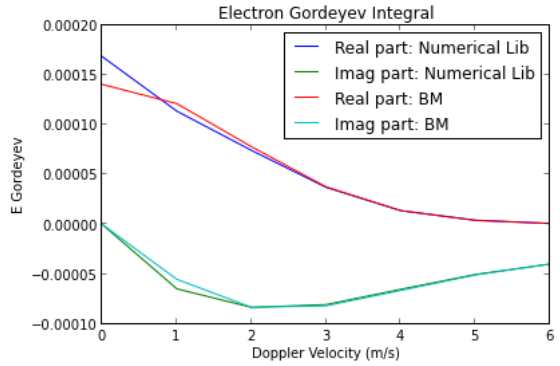

Figure 2.12 – Electron Gordeyev integral with DS = 10000

CHAPTER 3

MACHINE LEARNING APPLICATION

In this chapter a machine learning (ML) procedure is presented to interpolate the electron ACF Y_i corresponding to discrete grid states X_i, $i \in [1, n]$ available in the numerical library of [2] to arbitrary $Y(X)$ for X in a five-dimensional space spanned by the available X_i. As mentioned in Chapter 1, tuples $Y = (y_1, y, \cdots, y_d)$ are defined over a time-delay variable τ, where each tuple is a mapping of an ionospheric state-parameter tuple $X \equiv (N_e, B, T_e, T_i, \alpha)$. Among possible ML algorithms, one strategy, known as *k-nearest neighbor* (KNN), is examined in this chapter. Using a standard procedure in the area of ML, the algorithm is analyzed by performing training and testing steps involving a trained data object to obtain $Y(X)$ interpolants. KNN is implemented using Python's ML tool-kit "scikit-learn". The chapter ends with examples of interpolated electron ACF's $Y(X)$ obtained with the KNN method.

3.1 Machine Learning: Introduction

Machine learning (ML) is a technique of learning the underlying properties of a data set and applying them to produce new data. There are various well-known algorithms under ML including Linear Regression (LR), Support Vector Machines (SVM), Gaussian Classification and Regression (GC and GR), k-Nearest Neighbors Classification and Regression (KNN). The ML algorithm utilized in this work is the KNN regression.

In section 3.2, the input data set and the KNN algorithm are described. Section 3.3 covers the performance comparison of different weighting functions, and Section 3.4 shows

application usage of KNN method.

3.2 Machine Learning: Regression Algorithm

As mentioned in Chapter 2, the dataset for the ML task is stored on the disk as a new numerical library. The *k-nearest neighbor* (KNN) algorithm utilized in this work is applied to this numerical library in order to perform the prediction of ACF for a new set of ionospheric input parameters.

3.2.1 Regression Data Set

Regression data set consist of input tuple X and output tuple Y:

- Input tuple X consists of five parameters (N_e, B, T_e, T_i, α) where

$$\begin{aligned} N_e &\in \left[10^{11.0}, 10^{11.5}, 10^{12.0}, 10^{12.5}\right] \mathrm{m}^{-3} \equiv G_0 \\ B &\in [20000, 25000, 30000]\,\mathrm{nT} \equiv G_1 \\ T_e &\in [600, 800, 1000, \cdots, 3000]\,\mathrm{K} \equiv G_2 \\ T_i &\in [800, 1000, \cdots, 1600]\,\mathrm{K} \equiv G_3 \\ \alpha_s &\in [0.0, 0.001, \cdots, 1.0]^\circ \equiv G_{4s} \\ \alpha_l &\in [0.5, 0.6, \cdots, 90.0]^\circ \equiv G_{4l} \end{aligned} \tag{3.1}$$

  where α_s is for small aspect angles and α_l is for large aspect angles.

- Output tuple Y consists of correction factor time-series defined by $\mathrm{ACF}_n - \mathrm{ACF}_b$, where ACF_n is autocorrelation function from the numerical ACF library and ACF_b is Brownian motion calculated using autocorrelation function (2.36).

The size of X_i and Y_i vectors for the small angle sets is described by $i \in [1,\, 2,\, 3,\, \cdots,\, 22620]$ and by $i \in [1,\, 2,\, 3,\, \cdots,\, 15600]$ for large angle sets.

3.2.2 Description of KNN Algorithm

Let

$$X = (x_1,\, x_2,\, \cdots,\, x_p) \tag{3.2}$$

and

$$Y = (y_1,\, y_2,\, \cdots,\, y_d) \tag{3.3}$$

denote the independent and dependent variables in KNN regression. In KNN, a weighted average of the Y's of the k nearest neighbors of X is presented as an estimate of Y (X). The weights are chosen so that nearer neighbors contribute more to the average than distant ones. Usually, a weight function is used to specify the weights. This function depends on the Euclidean distance of the new input X from its neighbors. A step by step description of the KNN process to get the interpolated Y for a given X is as follows:

1. The Euclidean distances from the new input $X = (x_0,\, x_1,\, x_2,\, x_3,\, x_4)$ to its neighbors on the grid are calculated as

$$d = \sqrt{(l_0 - i_0) + (l_1 - i_1)^2 + \cdots + (l_4 - i_4)^2}, \tag{3.4}$$

where $i_j,\, j \in [0,\, 1,\, 2,\, 3,\, 4]$ denote the index of the neighbors in the grid defined by G_j vectors in (3.1) — for instance, $i_0 = 0$ for $N_e = 11.0$ and $i_2 = 2$ for $T_e = 1000\,\mathrm{K}$, etc. — and

$$l_j = \frac{x_j - \tilde{x}_{j,m-1}}{\tilde{x}_{j,m} - \tilde{x}_{j,m-1}} + (m_j - 1), \tag{3.5}$$

where $\tilde{x}_{j,m}$ is the smallest element of vector G_j with index m such that $\tilde{x}_{j,m} \geq x_j$. Clearly, this distance definition utilized in our implementation of KNN is based on the relative locations of the data within the discrete grid using the natural units of

the grid expressed in terms of grid indices. The definition is not influenced by the physical units and scalings of the input parameters.

2. Rank the distances d calculated for all neighbors of X and select the nearest k in terms of d.

3. The output Y corresponding to input X is obtained as the weighted average of k-nearest neighbor outputs Y_i,

$$Y = \frac{\sum Y_i W_i}{\sum W_i}, \tag{3.6}$$

where $i \in [1, 2, 3, \cdots, k]$ and W_i represent weight coefficients chosen such that they decay rapidly with increasing distance from the new input location [7]. The weight coefficients are defined as

$$W_i = f(d_i), \tag{3.7}$$

where $i \in [1, 2, 3, \cdots, k]$, d_i represents the Euclidean distances and $f(d_i)$ is a suitable weight function. Section 3.3 examines the performance of $f(d_i) = \frac{1}{d_i}$ and $f(d_i) = e^{-d_i^2}$ (inverse distance and Gaussian weight function).

Python's machine-learning library "scikit-learn" provides a module called *KNeighborsRegressor* for implementing the KNN regression steps described above.

3.3 Performance Analysis of Weighting Functions

The KNN algorithm uses a weighting function to weigh the nearest neighbors when calculating the weighted-average to generate $Y(X)$. Two weighting functions, inverse distance and Gaussian kernel, are compared in this section. The analysis is done by separating the input data set into training and testing sets. Then the $Y(X)$ interpolants

are generated for testing tuples and compared with the true $Y(X)$ from the numerical library. The comparison is done by calculating the percentage error (PE)

$$\mathrm{PE} = \sqrt{\frac{\sum \left(Y_{\mathrm{true}} - Y_{\mathrm{pred}}\right)^2}{\sum \left(Y_{\mathrm{true}}\right)^2}} \times 100 \tag{3.8}$$

between the true and the predicted values of the ACF. This analysis is performed 10 times after reshuffling X_i and Y_i sets for each test run. Also, the testing is done with $k = 16$. This is because for smaller values of k, the inverse and Gaussian weighting functions show the same results. Table 3.1 compares these two weighting functions.

Table 3.1 – Comparison of Inverse Distance and Gaussian Kernel Weighting

Test Run	Inverse Distance (PE)	Gaussian Weight (PE)
1	23.7	31.5
2	39.2	41.8
3	36.2	39.2
4	28.8	29.9
5	36.9	38.9
6	26.5	28.7
7	32.6	33.4
8	29.4	31.2
9	35.8	36.6
10	28.3	29.1

Results shown in Table 3.1 indicate that inverse distance and Gaussian weightings perform similarly, with the inverse distance giving slightly better results. We will use inverse distance in the interpolations from here on.

3.4 Application of Machine Learning on Radar Data

This section describes the tools used for KNN algorithm, application of regression analysis on the numerical library, information about regressor object and the run-time to generate a $Y\,(X)$ for a new input tuple.

3.4.1 Tools Used

The tasks of reading data from the numerical library and creating the data set for regression analysis are done using the Python programming language. The KNN regression algorithm is implemented using Python's machine learning library "scikit-learn". This library provides a module called *KNeighborsRegressor* that has several input parameters used to define a regressor object. These parameters are passed when creating the regressor object and they govern how the module will perform. The object can be saved on the disk and loaded back when needed. The main parameters that can be passed to the module are: Number of nearest neighbors (*n_ neighbors*), Weight function (*weights*), Algorithm used to compute the nearest neighbors (*algorithm*), Power parameter for the Minkowski metric (p). The values of these four parameters used in this work are:

- *n_ neighbors* - Number of neighbors to use for KNN regression. Depends on the number of dimensions D outside the model grid.
- *weights* - Weight function used in prediction. It takes a string *uniform*, *distance* or a callback function. The callback function is a user-defined function which accepts an array of distances and returns an array of the same shape containing the weights. A callback function that returns inverse-distance weights is implemented in this work.
- *algorithm* - Algorithm used to compute the nearest neighbors. It takes a string *ball_ tree*, *kd_ tree*, *brute* or *auto*. The string *auto* that is used in this work attempts to decide the most appropriate algorithm based on the values passed to "fit" method, that fits the model to the training data (X, Y).

- p - The value of 2 provides Euclidean distance metric.

3.4.2 Application of KNN

Consider the following set of input tuples:

$$\begin{aligned} X_1 &= (10^{11.0},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_2 &= (10^{11.1},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_3 &= (10^{11.2},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_4 &= (10^{11.3},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_5 &= (10^{11.4},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_6 &= (10^{11.5},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \end{aligned} \tag{3.9}$$

Here, X_1 and X_6 exist in the numerical library, but the ACFs for X_2, X_3, X_4, X_5 will be generated by adding the KNN interpolated correction factor (as described in Section 3.2.2) to the exact Brownian motion ACF. In these tuples, only the first dimension N_e has an off-grid value (varying from $10^{11.1}$ to $10^{11.4}$) and therefore $D = 1$. We used the KNN algorithm with $k = 2^D = 2$. The expected result is that ACFs corresponding to X_2, X_3, X_4, X_5 should be bracketed between X_1 and X_6. For each of X_2, X_3, X_4, X_5, the KNN algorithm returned two nearest neighbor tuples X_1 and X_6.

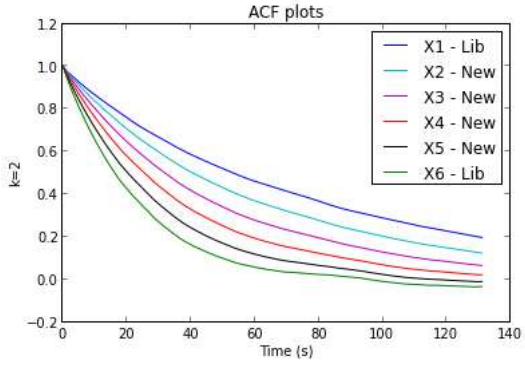

Figure 3.1 – ACF plots with the dimension N_e varying from $10^{11.0}$ to $10^{11.5}\,\mathrm{m}^{-3}$

Figure 3.1 shows the results, where the blue and green curves lying on the two extremes correspond to the ACF for X_1 and X_6 respectively. The curves for X_2, X_3, X_4, X_5 nicely sit between them.

Consider a next set of input tuples:

$$\begin{aligned} X_7 &= (10^{11.0},\ 2.5 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_8 &= (10^{11.0},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_9 &= (10^{11.25},\ 2.25 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_{10} &= (10^{11.5},\ 2.5 \times 10^{-5},\ 1000,\ 800,\ 0.0) \\ X_{11} &= (10^{11.5},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0) \end{aligned} \tag{3.10}$$

Here, X_7, X_8, X_{10} and X_{11} exist in the numerical library, but the ACF for X_9 will be generated by KNN interpolation. In these tuples, N_e and B have off-grid values and therefore $D = 2$. We used the KNN algorithm with $k = 2^D = 4$. The expected result is that ACF corresponding to X_9 should be bracketed between X_7, X_8, X_{10} and X_{11}. Figure 3.2 shows the interpolation results, where the four nearest neighbors are shown in blue, green, black and red curves. The cyan curve represents X_9, which nicely sits between the neighbors.

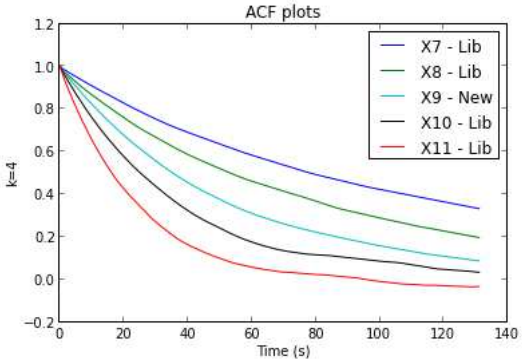

Figure 3.2 – ACF plots with the dimensions N_e and B varying

Consider a final set of input tuples:

$$X_{12} = (10^{11.0},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.001) \tag{3.11}$$

$$X_{13} = (10^{11.0},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.0016)$$
$$X_{14} = (10^{11.0},\ 2.0 \times 10^{-5},\ 1000,\ 800,\ 0.002)$$

Here, X_{12} and X_{13} exist in the numerical library, but the ACF for X_{13} will be generated by KNN interpolation. In these tuples, α has off-grid value and therefore $D = 1$. We used the KNN algorithm with $k = 2^D = 2$. The expected result is that ACF corresponding to X_{13} should be bracketed between X_{12} and X_{14}. Figure 3.3 shows the interpolation results, where the two nearest neighbors are shown in blue and red curves. The green-colored curve represents X_{13}, which nicely sits between the two neighbors.

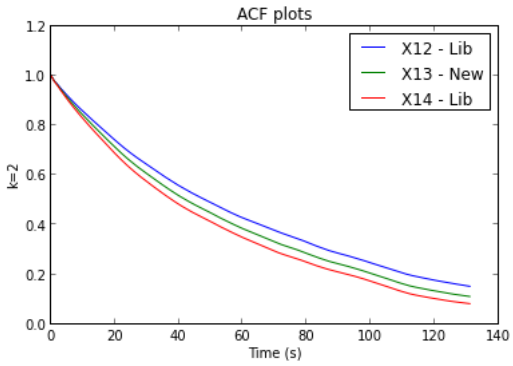

Figure 3.3 – ACF plots with the dimension α varying

3.4.3 Regressor Object and Run-Time

The regressor object returned by scikit-learn's module *KNeighborsRegressor* contains all the information needed to perform KNN regression for a given input tuple X. An example code-snippet (code provided in appendix B) to generate regressor object is

```
from sklearn.neighbors import KNeighborsRegressor
k = 5 #Number of nearest neighbors
neigh = KNeighborsRegressor(n_neighbors=k,algorithm='auto',
   weights=returnWeightsID)
neigh.fit(Xn[:,:],Y[:,:])
```

where the object name is "neigh". Here, first the *KNeighborsRegressor* module is imported from the scikit-learn library; then the regressor object is created by passing the desired

values of parameters described in Section 3.4.1. Finally, the "fit" method is called on by the object that fits the model to the input data set. Python's built-in function *dir()* can be called on by this object to see the parameters contained within it. These parameters together help the process of KNN regression whenever a new input tuple is supplied to the regressor object.

The input X has five dimensions, where each of the dimensions can be off-grid, giving the possible values of $D \in [1, 2, 3, 4, 5]$. This gives five different regressor object for each of $k \in [2, 4, 8, 16, 32]$. The size of the regressor object for different values of k is 67 MB for small-aspect angles data set, and 60 MB for large-aspect angles data set.

Next an analysis of run-time to generate the ACF plot for a new X is done with the new downsampled numerical library described at the end of Chapter 2. Table 3.2 shows that on an average it takes 0.115 seconds to generate the ACF plot.

Table 3.2 – Run-Time to Generate ACF for a New Input X

Test	Run-time (in seconds)
1	0.113
2	0.112
3	0.121
4	0.110
5	0.123
Average	**0.115**

CHAPTER 4

RESULTS, CONCLUSIONS AND FUTURE WORK

Chapter 3 presented our ML based interpolation procedure to utilize the numerical electron ACF library described in Chapter 2. Here, we will apply the procedure to calculate multi-ion ISR spectra and ACFs by utilizing input data that is "off-grid" in terms of the available numerical library. The examples will include new inputs X which have N_e, B, T_e, T_i and α lying off-grid in an incremental manner.

4.1 Multi-Ion ISR Spectra and ACF

First, a comparison of ISR spectra and ISR ACF for $N_e = 10^{11.25}\,\mathrm{m}^{-3}$ and $N_e = 10^{11.5}\,\mathrm{m}^{-3}$ is shown in Figures 4.1 and 4.2. Here, the ion composition is a 70%-20%-10% mix of hydrogen-oxygen-helium, $\alpha = 2.0^\circ$ and $T_e = T_i = 1000$ K.

Let $X_1 \equiv (10^{11.25},\, 2.0 \times 10^{-5},\, 1000,\, 1000,\, 2.0)$. Next, Figures 4.3 - 4.6 show the ISR spectra and ACF for new input tuples

$$\begin{aligned} X_2 &= (10^{11.25},\, 2.25 \times 10^{-5},\, 1000,\, 1000,\, 2.0) \\ X_3 &= (10^{11.25},\, 2.25 \times 10^{-5},\, 1100,\, 1000,\, 2.0) \\ X_4 &= (10^{11.25},\, 2.25 \times 10^{-5},\, 1100,\, 1100,\, 2.0) \\ X_5 &= (10^{11.25},\, 2.25 \times 10^{-5},\, 1100,\, 1100,\, 2.5) \end{aligned} \tag{4.1}$$

where $D = 2,\, 3,\, 4,\, 5$, respectively, and $k = 4,\, 8,\, 16,\, 32$. The same set of ISR spectra and

ACFs are overplotted in Figures 4.7 and 4.8 for comparison.

From Figures 4.7 and 4.8, it can be seen that the red curve corresponding to X_1 is quite different from the curves for X_2, X_3, X_4 and X_5. This is because X_2, X_3, X_4 and X_5 vary along the T_e, T_i and α dimensions with respect to each other, whereas the transition from X_1 to X_2 is in the B dimension. Since the magnetic field plays a significant role in the shape of ISR spectra and ACF as compared to aspect angle and electron-ion temperatures, the red curve clearly shows this fact.

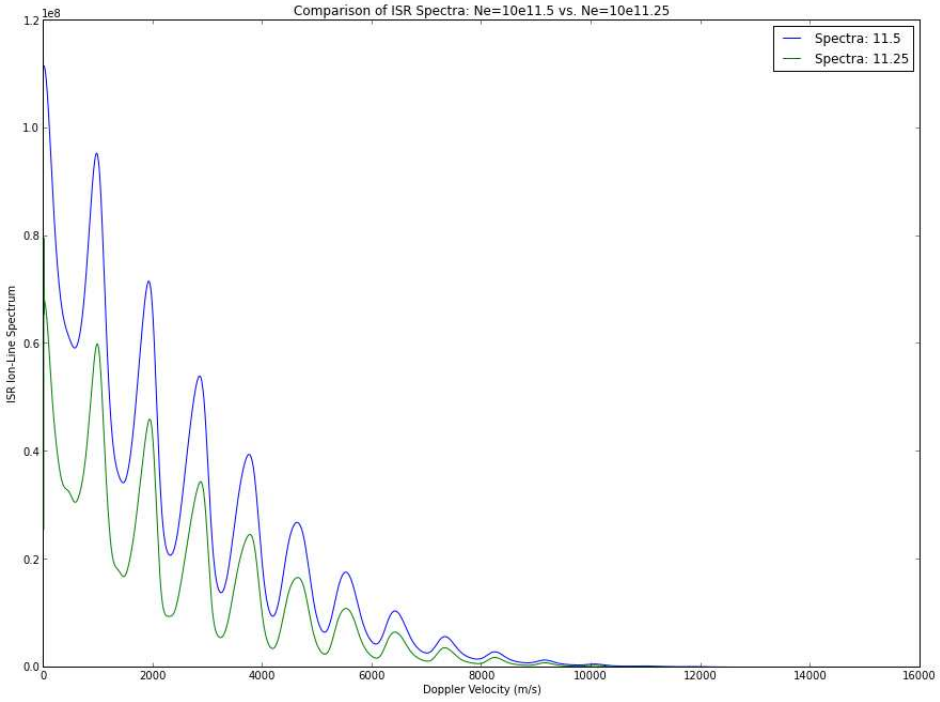

Figure 4.1 – Comparison of ISR spectra for $N_e = 10^{11.25}\,\mathrm{m}^{-3}$ and $N_e = 10^{11.5}\,\mathrm{m}^{-3}$

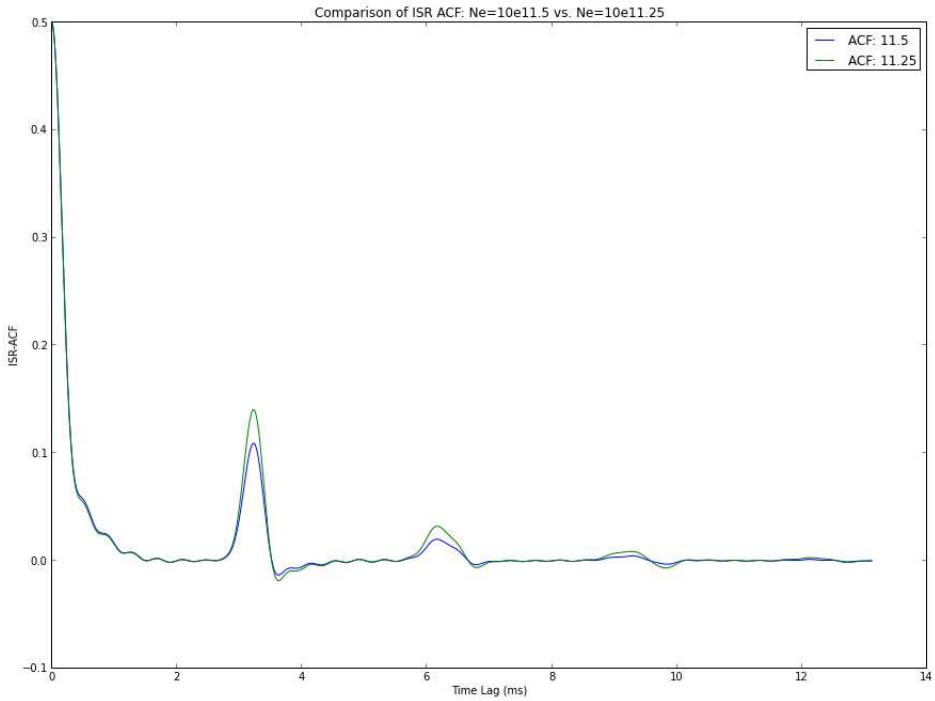

Figure 4.2 – Comparison of ISR ACF for $N_e = 10^{11.25}\,\mathrm{m}^{-3}$ and $N_e = 10^{11.5}\,\mathrm{m}^{-3}$

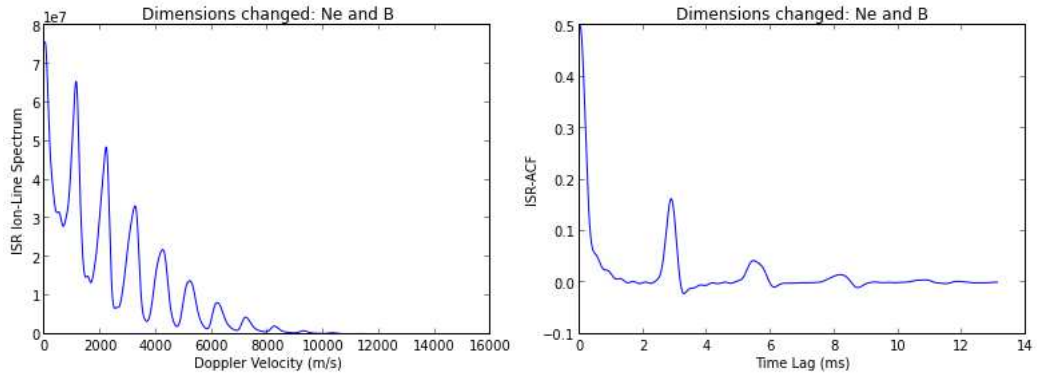

Figure 4.3 – ISR spectra and ACF for ($10^{11.25}$, 2.25×10^{-5}, 1000, 1000, 2.0) with $k = 4$

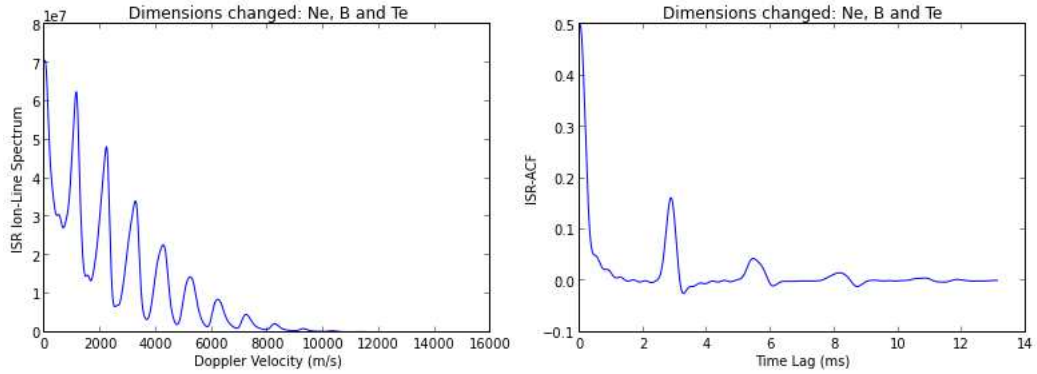

Figure 4.4 – ISR spectra and ACF for $(10^{11.25}, 2.25\times10^{-5}, 1100, 1000, 2.0)$ with $k = 8$

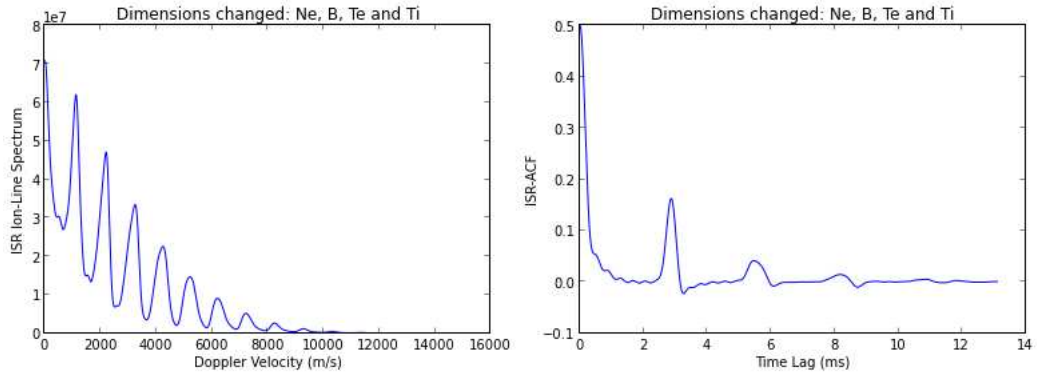

Figure 4.5 – ISR spectra and ACF for $(10^{11.25}, 2.25 \times 10^{-5}, 1100, 1100, 2.0)$ with $k = 16$

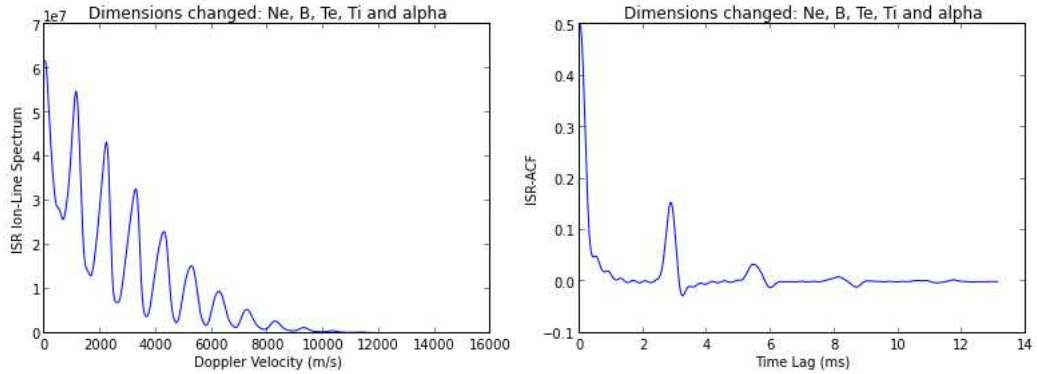

Figure 4.6 – ISR spectra and ACF for $(10^{11.25}, 2.25 \times 10^{-5}, 1100, 1100, 2.5)$ with $k = 32$

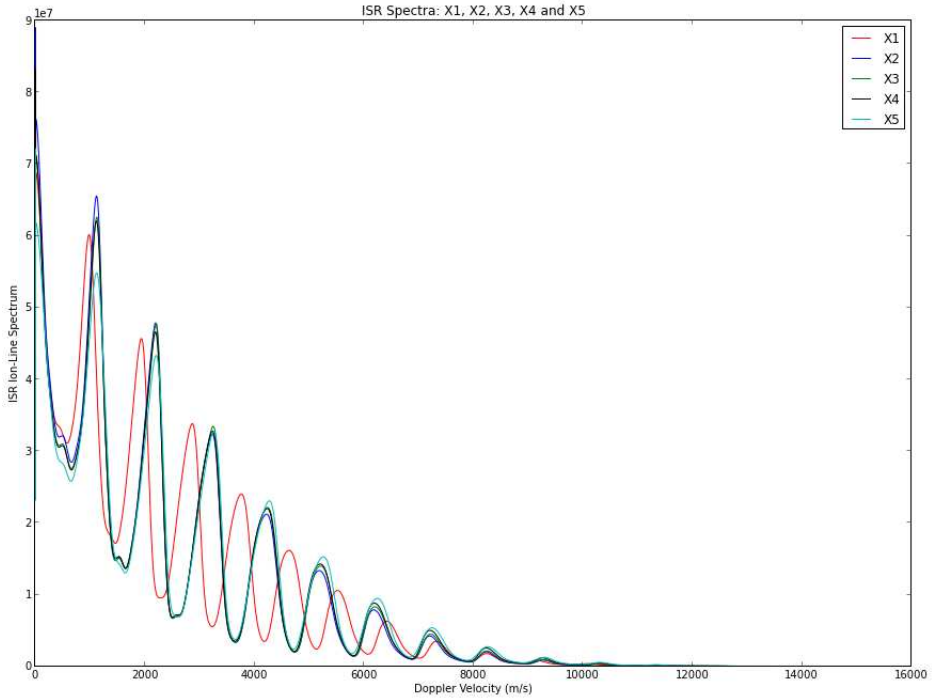

Figure 4.7 – ISR spectra for X_1, X_2, X_3, X_4 and X_5

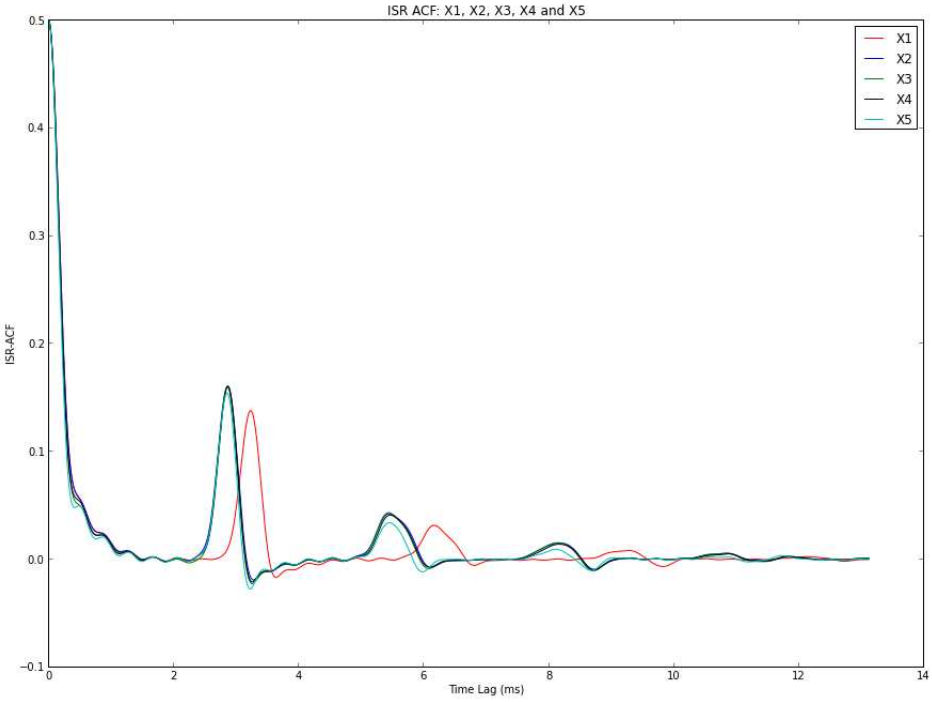

Figure 4.8 – ISR ACF for X_1, X_2, X_3, X_4 and X_5

4.2 Conclusions

This thesis covered a number of physical principles that are used in the derivation of incoherent scatter spectral models. The previous study in [1] and [2] focused on a single-ion model of the ionosphere, whereas this thesis showed the incoherent spectral results for a multi-ion ionosphere. The multi-ion case is easily derived from the single-ion case by taking into account the Coulomb collision frequencies between different ion species.

The K-Nearest Neighbor (KNN) regression tool was implemented and helped to generate and study the multi-ion IS spectra obtained with the extended model. A closer look at the original numerical ACF library revealed two important aspects. First, the ACF time-series data was created with a very high sampling rate, which is not needed; a down-sampled ACF time-series would suffice. Second, the values of five ionospheric parameters in tuple $X = (N_e, B, T_e, T_i, \alpha)$ have different measurement scales, and hence needed to be rescaled to perform the KNN algorithm. Keeping these two aspects in mind, a new numerical library was created and saved on the disk. The performance analysis with inverse distance and Gaussian weighting functions showed that inverse distance performs better than Gaussian weighting.

4.3 Future Work

The machine learning (ML) procedure presented in this thesis utilized a numerical library which is defined over a discrete grid. A new numerical library with the ionospheric parameters $(N_e, B, T_e, T_i, \alpha)$ spread over a denser and random grid can be generated for future work. In that case, a training and testing procedure to find the optimum value of k would be needed [8]. Also, having a denser and random input data set will help the interpolation procedure. We tried linear regression (LR), but it failed to show positive results. In future work, an ML algorithm based on polynomial regression (PR) can be explored. This ML procedure can also be applied to other ionospheric related input data.

APPENDIX A

SOURCE CODE: INCOHERENT SCATTERING

To calculate Chirp-z transform:

```
def chirpz(g,n,dt,dw):
    """transforms g(t) into G(w)
    g(t) is n-point array and output G(w) is (n/2)-points.
    dt and dw, sampling intervals of g(t) and G(w), are
    prescribed externally in an idependent manner
    --- see Li, Franke, Liu [1991]"""
    g[0] = (3./8.)*g[0]
    g[1] = (7./6.)*g[1]
    g[2] = (23./24.)*g[2]
    W = exp(-1j*dw*dt*arange(n)**2/2.)
    x = g*W; y = conj(W)
    x[n/2:] = 0.; y[n/2:] = y[0:n/2][::-1] # treat 2nd half
       of x and y specially
    xi = fft.fft(x); yi = fft.fft(y); G = dt*W*fft.ifft(xi*
       yi) #in MATLAB use ifft then fft (EK)
    return G[0:n/2]
```

To calculate Coulomb collision frequencies for multi-ion case:

```
# Ionospheric State
Ne=1.0e11.5 #Electron density (1/m^3)
#B=18.0e-6 #Magnetic Field (T)
B=2.0e-5

# Ion Composition
NH,N4,NO=0.70*Ne,0.10*Ne,0.20*Ne
Te,TH,T4,TO=1000,1000,1000,1000

# Physical Paramters (MKS):
```

```
me = 9.1093826e-31 # Electron mass in kg
mH,m4,mO = 1836*me,1836*4*me,1836*16*me # Ion mass

qe = 1.60217653e-19 # C (Electron charge)
K = 1.3806505e-23 # Boltzmann constant m^2*kg/(s^2*K);
eps0 = 8.854187817e-12 # F/m (Free-space permittivity)
c = 299.792458e6 # m/s (Speed of light)
re = 2.817940325e-15 # Electron radius

Ce,CH,C4,CO=sqrt(K*Te/me),sqrt(K*TH/mH),sqrt(K*T4/m4),sqrt(K
   *TO/mO) # Thermal speeds (m/s)
Omge,OmgH,Omg4,OmgO = qe*B/me,qe*B/mH,qe*B/m4,qe*B/mO # Gyro
   -frequencies

# Debye Lengths
debe,debH,deb4,debO = sqrt(eps0*K*Te/(Ne*qe**2)),sqrt(eps0*K
   *TH/(NH*qe**2)),sqrt(eps0*K*T4/(N4*qe**2)),sqrt(eps0*K*TO
   /(NO*qe**2))
debp = 1./sqrt(1./debe/debe+1./debH/debH+1./deb4/deb4+1./
   debO/debO) # Plasma Debye Length

# The following pseudocode is for Coulomb collision of
   species s with species p
#Csp=sqrt(Cs**2+Cp**2) #most probable interaction speed of s
   ={e,H,O} with p={e,H,O}
#msp=ms*mp/(ms+mp) #reduced mass for s and p
#bm_sp=qs*qp/(12*pi*eps0)/msp/Csp**2 #bmin for s and p
#log_sp=log(debp/bm_sp) #coulomb logarithm for s and p
#nusp=Np*qs**2*qp**2*log_sp/(3*(2*pi)**(3/2.)*ms*msp*Csp**3)
    # collision freq of s with p --- 2.104 Callen 2003

# electron-electron
Cee=sqrt(Ce**2+Ce**2)
mee=me*me/(me+me)
bm_ee=qe*qe/(12*pi*eps0)/mee/Cee**2
log_ee=log(debp/bm_ee)
nuee=Ne*qe**2*qe**2*log_ee/(3*(2*pi)**(3/2.)*eps0**2*me*mee*
   Cee**3)
# electron-hydrogen
CeH=sqrt(Ce**2+CH**2)
meH=me*mH/(me+mH)
bm_eH=qe*qe/(12*pi*eps0)/meH/CeH**2
log_eH=log(debp/bm_eH)
nueH=NH*qe**2*qe**2*log_eH/(3*(2*pi)**(3/2.)*eps0**2*me*meH*
   CeH**3)
# electron-helium
Ce4=sqrt(Ce**2+C4**2)
```

```
me4=me*m4/(me+m4)
bm_e4=qe*qe/(12*pi*eps0)/me4/Ce4**2
log_e4=log(debp/bm_e4)
nue4=N4*qe**2*qe**2*log_e4/(3*(2*pi)**(3/2.)*eps0**2*me*me4*
   Ce4**3)
# electron-oxygen
CeO=sqrt(Ce**2+CO**2)
meO=me*mO/(me+mO)
bm_eO=qe*qe/(12*pi*eps0)/meO/CeO**2
log_eO=log(debp/bm_eO)
nueO=NO*qe**2*qe**2*log_eO/(3*(2*pi)**(3/2.)*eps0**2*me*meO*
   CeO**3)
# electron Coulomb collision frequency
nue=nuee+nueH+nue4+nueO
nuel=nueH+nue4+nueO
nuep=nuel+nuee

# hydrogen-electron
CHe=sqrt(CH**2+Ce**2)
mHe=mH*me/(mH+me)
bm_He=qe*qe/(12*pi*eps0)/mHe/CHe**2
log_He=log(debp/bm_He)
nuHe=Ne*qe**2*qe**2*log_He/(3*(2*pi)**(3/2.)*eps0**2*mH*mHe*
   CHe**3)
# hydrogen-hydrogen
CHH=sqrt(CH**2+CH**2)
mHH=mH*mH/(mH+mH)
bm_HH=qe*qe/(12*pi*eps0)/mHH/CHH**2
log_HH=log(debp/bm_HH)
nuHH=NH*qe**2*qe**2*log_HH/(3*(2*pi)**(3/2.)*eps0**2*mH*mHH*
   CHH**3)
# hydrogen-helium
CH4=sqrt(CH**2+C4**2)
mH4=mH*m4/(mH+m4)
bm_H4=qe*qe/(12*pi*eps0)/mH4/CH4**2
log_H4=log(debp/bm_H4)
nuH4=N4*qe**2*qe**2*log_H4/(3*(2*pi)**(3/2.)*eps0**2*mH*mH4*
   CH4**3)
# hydrogen-oxygen
CHO=sqrt(CH**2+CO**2)
mHO=mH*mO/(mH+mO)
bm_HO=qe*qe/(12*pi*eps0)/mHO/CHO**2
log_HO=log(debp/bm_HO)
nuHO=NO*qe**2*qe**2*log_HO/(3*(2*pi)**(3/2.)*eps0**2*mH*mHO*
   CHO**3)
# hydrogen Coulomb collision frequency
nuH=nuHe+nuHH+nuH4+nuHO
```

```
# helium-electron
C4e=sqrt(C4**2+Ce**2)
m4e=m4*me/(m4+me)
bm_4e=qe*qe/(12*pi*eps0)/m4e/C4e**2
log_4e=log(debp/bm_4e)
nu4e=Ne*qe**2*qe**2*log_4e/(3*(2*pi)**(3/2.)*eps0**2*m4*m4e*
   C4e**3)
# helium-hydrogen
C4H=sqrt(C4**2+CH**2)
m4H=m4*mH/(m4+mH)
bm_4H=qe*qe/(12*pi*eps0)/m4H/C4H**2
log_4H=log(debp/bm_4H)
nu4H=NH*qe**2*qe**2*log_4H/(3*(2*pi)**(3/2.)*eps0**2*m4*m4H*
   C4H**3)
# helium-helium
C44=sqrt(C4**2+C4**2)
m44=m4*m4/(m4+m4)
bm_44=qe*qe/(12*pi*eps0)/m44/C44**2
log_44=log(debp/bm_44)
nu44=N4*qe**2*qe**2*log_44/(3*(2*pi)**(3/2.)*eps0**2*m4*m44*
   C44**3)
# helium-oxygen
C4O=sqrt(C4**2+CO**2)
m4O=m4*mO/(m4+mO)
bm_4O=qe*qe/(12*pi*eps0)/m4O/C4O**2
log_4O=log(debp/bm_4O)
nu4O=NO*qe**2*qe**2*log_4O/(3*(2*pi)**(3/2.)*eps0**2*m4*m4O*
   C4O**3)
# helium Coulomb collision frequency
nu4=nu4e+nu4H+nu44+nu4O

# oxygen-electron
COe=sqrt(CO**2+Ce**2)
mOe=mO*me/(mO+me)
bm_Oe=qe*qe/(12*pi*eps0)/mOe/COe**2
log_Oe=log(debp/bm_Oe)
nuOe=Ne*qe**2*qe**2*log_Oe/(3*(2*pi)**(3/2.)*eps0**2*mO*mOe*
   COe**3)
# oxygen-hydrogen
COH=sqrt(CO**2+CH**2)
mOH=mO*mH/(mO+mH)
bm_OH=qe*qe/(12*pi*eps0)/mOH/COH**2
log_OH=log(debp/bm_OH)
nuOH=NH*qe**2*qe**2*log_OH/(3*(2*pi)**(3/2.)*eps0**2*mO*mOH*
   COH**3)
# oxygen-helium
```

```
CO4=sqrt(CO**2+C4**2)
mO4=mO*m4/(mO+m4)
bm_O4=qe*qe/(12*pi*eps0)/mO4/CO4**2
log_O4=log(debp/bm_O4)
nuO4=N4*qe**2*qe**2*log_O4/(3*(2*pi)**(3/2.)*eps0**2*mO*mO4*
   CO4**3)
# oxygen-osxygen
COO=sqrt(CO**2+CO**2)
mOO=mO*mO/(mO+mO)
bm_OO=qe*qe/(12*pi*eps0)/mOO/COO**2
log_OO=log(debp/bm_OO)
nuOO=NO*qe**2*qe**2*log_OO/(3*(2*pi)**(3/2.)*eps0**2*mO*mOO*
   COO**3)
# oxygen Coulomb collision frequency
nuO=nuOe+nuOH+nuO4+nuOO
```

Loading example files from Marco's numerical library:

```
# Path to the Marco's numerical library
basepath = '/raid_b/IS_spc_model/JE-ACF-Library/'

# Full path to the specific file that user wants to
   investigate
file1 = '/raid_b/IS_spc_model/JE-ACF-Library/LN110/
   LN110Bm20Ti10/config_LN110TE10TI10M20.eh5'   # Te=Ti=1000
   K and B=2.0e-5T
file2 = '/raid_b/IS_spc_model/JE-ACF-Library/LN110/
   LN110Bm20Ti10/proc_LN110TE10TI10M20L1.eh5'   # Large
   angles file
file3 = '/raid_b/IS_spc_model/JE-ACF-Library/LN110/
   LN110Bm20Ti10/proc_LN110TE10TI10M20S1.eh5'   # Small
   angles file

import h5py
fconf = h5py.File(file1,'r')
fL1 = h5py.File(file2,'r')
fS1 = h5py.File(file3,'r')

N=131072 # comes from Marco library
dt=0.1e-6 # sampling time provided by the library (units of
   seconds)
#dt=10.0e-6 # sampling time provided by the library (units
   of seconds)
T=N*dt # total data dength in seconds
```

```
fmax=5000. # Hz units (I choose this)
df=fmax/(N/2) # in Hz units
dw=2*pi*df
w=arange(N/2)*dw

fradar=50.0e6 # Radar Frequency (Hz)
kB = 2*pi*(fradar/(c/2)) # Bragg wavenumber kB = 2*ko
aspect= 2*pi/180. # Aspect angle (rad) with 0 perp to Bs
#aspect= 0.5*pi/180. # Aspect angle (rad) with 0 perp to Bs
```

Gordeyev's Integral calculation:

```
# Electron Gordeyev from Marco library or interpolated ACF
   using KNN
Ge = chirpz(fL1['Correlations']['acf'][6,:],N,dt,dw)
#Ge = chirpz(correctedBM,N,dt,dw)

# Brownian electron Gordeyev
dtB=dt
tB = arange(N)*dtB #adjust dt such that full range of acfi
   is covered by range t
varil=((2.*Ce**2)/nuel**2)*(nuel*tB-1+exp(-nuel*tB)) # page
   -337 in ppr-II equa-43
gam=arctan(nuep/Omge)
varip=((2.*Ce**2)/(nuep**2+Omge**2))*(cos(2*gam)+nuep*tB-exp
   (-nuep*tB)*cos(Omge*tB-2*gam)) # page-337 in ppr-II equa
   -44
acfB=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.) # page-337 in ppr-II equa-42
GB = chirpz(acfB,N,dtB,dw) # Brownian electron Gordeyev
   Integral
plot(w/kB,real(GB), label='Real part')
legend()
plot(w/kB,imag(GB), label='Imag part')
legend()
xlabel('Doppler Velocity (m/s)')
ylabel('E Gordeyev')
title('Brownian Electron Gordeyev Integral')
subplots_adjust(left=0.1, bottom=0.1, right=2, top=2.2,
   wspace=0.2, hspace=0.4)
savefig("electronGordeyev.png",bbox_inches='tight') # save
   as png
```

```
# Hydrogen Gordeyev
dtH = (1./fmax)/16
tH = arange(N)*dtH #adjust dt such that full range of acfi
   is covered by range t
varil=((2.*CH**2)/nuH**2)*(nuH*tH-1+exp(-nuH*tH)) # page-337
    in ppr-II equa-43
gam=arctan(nuH/OmgH)
varip=((2.*CH**2)/(nuH**2+OmgH**2))*(cos(2*gam)+nuH*tH-exp(-
   nuH*tH)*cos(OmgH*tH-2*gam)) # page-337 in ppr-II equa-44
acfH=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.) # page-337 in ppr-II equa-42
GH = chirpz(acfH,N,dtH,dw) # Ion Gordeyev Integral
plot(w/kB,real(GH), label='Real part')
legend()
plot(w/kB,imag(GH), label='Imag part')
legend()
xlabel('Doppler Velocity (m/s)')
ylabel('H+ Gordeyev')
title('Brownian Hydrogen Gordeyev Integral')
subplots_adjust(left=0.1, bottom=0.1, right=2, top=2.2,
   wspace=0.2, hspace=0.4)
savefig("hydrogenGordeyev.png",bbox_inches='tight') # save
   as png


# Helium Gordeyev
dt4 = (1./fmax)/16
t4 = arange(N)*dt4 #adjust dt such that full range of acfi
   is covered by range t
varil=((2.*C4**2)/nu4**2)*(nu4*t4-1+exp(-nu4*t4)) # page-337
    in ppr-II equa-43
gam=arctan(nu4/Omg4)
varip=((2.*C4**2)/(nu4**2+Omg4**2))*(cos(2*gam)+nu4*t4-exp(-
   nu4*t4)*cos(Omg4*t4-2*gam)) # page-337 in ppr-II equa-44
acf4=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.) # page-337 in ppr-II equa-42
G4 = chirpz(acf4,N,dt4,dw) # Ion Gordeyev Integral
plot(w/kB,real(G4), label='Real part')
legend()
plot(w/kB,imag(G4), label='Imag part')
legend()
xlabel('Doppler Velocity (m/s)')
ylabel('He+ Gordeyev')
title('Brownian Helium Gordeyev Integral')
subplots_adjust(left=0.1, bottom=0.1, right=2, top=2.2,
   wspace=0.2, hspace=0.4)
```

```
savefig("heliumGordeyev.png",bbox_inches='tight') # save as
   png


# Oxygen Gordeyev
dtO = (1./fmax)/16
tO = arange(N)*dtO #adjust dt such that full range of acfi
   is covered by range t
varil=((2.*CO**2)/nuO**2)*(nuO*tO-1+exp(-nuO*tO)) # page-337
    in ppr-II equa-43
gam=arctan(nuO/OmgO)
varip=((2.*CO**2)/(nuO**2+OmgO**2))*(cos(2*gam)+nuO*tO-exp(-
   nuO*tO)*cos(OmgO*tO-2*gam)) # page-337 in ppr-II equa-44
acfO=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.) # page-337 in ppr-II equa-42
GO = chirpz(acfO,N,dtO,dw) # Ion Gordeyev Integral
plot(w/kB,real(GO), label='Real part')
legend()
plot(w/kB,imag(GO), label='Imag part')
legend()
xlabel('Doppler Velocity (m/s)')
ylabel('O+ Gordeyev')
title('Brownian Oxygen Gordeyev Integral')
subplots_adjust(left=0.1, bottom=0.1, right=2, top=2.2,
   wspace=0.2, hspace=0.4)
savefig("oxygenGordeyev.png",bbox_inches='tight') # save as
   png
```

ISR spectrum calculation:

```
# Admittances
yO=(1-1j*w*GO)/(kB**2*debO**2) # oxygen admittance
y4=(1-1j*w*G4)/(kB**2*deb4**2) # helium admittance
yH=(1-1j*w*GH)/(kB**2*debH**2) # hydrogen admittance
ye=(1-1j*w*Ge)/(kB**2*debe**2) # electron admittance
yB=(1-1j*w*GB)/(kB**2*debe**2) # Brownian electron
   admittance

# ISR spectrum
subplot(121)
spec=real(Ne*2*Ge)*abs((1+yH+y4+yO)/(1+ye+yH+y4+yO))**2 +
   real(NH*2*GH+N4*2*G4+NO*2*GO)*abs((ye)/(1+ye+yH+y4+yO))
   **2
plot(w/kB,spec)
```

```
xlabel('Doppler␣Velocity␣(m/s)')
ylabel('ISR␣Ion-Line␣Spectrum')
degree = unichr(176)
title(r'$\alpha$␣=␣%s%s,␣Te=1000K,␣B=2.0e-5T␣' %((aspect/pi)
   *180.,degree))

subplot(122)
acf=chirpz(spec,N/2,dw,4*dt)
lag=arange(N/4)*4*dt*1000
plot(lag,0.5*acf/acf[0])
xlabel('Time␣Lag␣(ms)')
ylabel('ISR-ACF')
degree = unichr(176)
title(r'$\alpha$␣=␣%s%s,␣Te=1000K,␣B=2.0e-5T␣' %((aspect/pi)
   *180.,degree))
subplots_adjust(left=0.1, bottom=0.1, right=2, top=1,wspace
   =0.2, hspace=0.8)

# ISR spectrum with Brownian electron
subplot(121)
spec=real(Ne*GB)*abs((1+yH+y4+yO)/(yB+yH+y4+yO))**2 + real(
   NH*2*GH+N4*2*G4+NO*2*GO)*abs((yB)/(yB+yH+y4+yO))**2
plot(w/kB,spec)
xlabel('Doppler␣Velocity␣(m/s)')
ylabel('ISR␣Ion-Line␣Spectrum')
degree = unichr(176)
title(r'$\alpha$␣=␣%s%s,␣Te=1000K,␣B=2.0e-5T␣' %((aspect/pi)
   *180.,degree))

subplot(122)
acf=chirpz(spec,N/2,dw,4*dt)
lag=arange(N/4)*4*dt*1000
plot(lag,0.5*acf/acf[0])
xlabel('Time␣Lag␣(ms)')
ylabel('ISR-ACF')
degree = unichr(176)
title(r'$\alpha$␣=␣%s%s,␣Te=1000K,␣B=2.0e-5T␣' %((aspect/pi)
   *180.,degree))
subplots_adjust(left=0.1, bottom=0.1, right=2, top=1,wspace
   =0.2, hspace=0.8)
savefig("isrIonLineSpectrumAndACF.png",bbox_inches='tight')
   # save as png
```

APPENDIX B

SOURCE CODE: REGRESSION TOOL

To calculate Coulomb collision frequencies for single-ion case:

```
# For Machine-Learning tool, we only considered Oxygen ion (
   F-region)
def parameters(Ne, B, Te, Ti):
    # Ionospheric State
    Ne=Ne # Electron density (1/m^3)
    B=B # Magnetic Field (T)

    # Ion Composition
    NO=Ne
    Te,TO=Te,Ti

    # Physical Parameters (MKS):
    me = 9.1093826e-31 # Electron mass in kg
    mO = 1836*16*me # Ion mass

    qe = 1.60217653e-19 # C (Electron charge)
    K = 1.3806505e-23 # Boltzmann constant m^2*kg/(s^2*K);
    eps0 = 8.854187817e-12 # F/m (Free-space permittivity)
    c = 299.792458e6 # m/s (Speed of light)
    re = 2.817940325e-15 # Electron radius

    Ce,CO=sqrt(K*Te/me),sqrt(K*TO/mO) # Thermal speeds (m/s)
    Omge,OmgO = qe*B/me,qe*B/mO # Gyro-frequencies

    # Debye Lengths
    debe,debO = sqrt(eps0*K*Te/(Ne*qe**2)),sqrt(eps0*K*TO/(
       NO*qe**2))
    debp = 1./sqrt(1./debe/debe+1./debO/debO) # Plasma Debye
        Length

    # the following pseudocode is for Coulomb collision of
       species s with species p
```

```
#Csp=sqrt(Cs**2+Cp**2) #most probable interaction speed
   of s={e,H,O} with p={e,H,O}
#msp=ms*mp/(ms+mp) #reduced mass for s and p
#bm_sp=qs*qp/(12*pi*eps0)/msp/Csp**2 #bmin for s and p
#log_sp=log(debp/bm_sp) #coulomb logarithm for s and p
#nusp=Np*qs**2*qp**2*log_sp/(3*(2*pi)**(3/2.)*ms*msp*Csp
   **3) # collision freq of s with p --- 2.104 Callen
   2003

# electron-electron
Cee=sqrt(Ce**2+Ce**2)
mee=me*me/(me+me)
bm_ee=qe*qe/(12*pi*eps0)/mee/Cee**2
log_ee=log(debp/bm_ee)
nuee=Ne*qe**2*qe**2*log_ee/(3*(2*pi)**(3/2.)*eps0**2*me*
   mee*Cee**3)

# electron-oxygen
CeO=sqrt(Ce**2+CO**2)
meO=me*mO/(me+mO)
bm_eO=qe*qe/(12*pi*eps0)/meO/CeO**2
log_eO=log(debp/bm_eO)
nueO=NO*qe**2*qe**2*log_eO/(3*(2*pi)**(3/2.)*eps0**2*me*
   meO*CeO**3)

# electron Coulomb collision frequency
nue=nuee+nueO
nuel=nueO
nuep=nuel+nuee

# oxygen-electron
COe=sqrt(CO**2+Ce**2)
mOe=mO*me/(mO+me)
bm_Oe=qe*qe/(12*pi*eps0)/mOe/COe**2
log_Oe=log(debp/bm_Oe)
nuOe=Ne*qe**2*qe**2*log_Oe/(3*(2*pi)**(3/2.)*eps0**2*mO*
   mOe*COe**3)
# oxygen-oxygen
COO=sqrt(CO**2+CO**2)
mOO=mO*mO/(mO+mO)
bm_OO=qe*qe/(12*pi*eps0)/mOO/COO**2
log_OO=log(debp/bm_OO)
nuOO=NO*qe**2*qe**2*log_OO/(3*(2*pi)**(3/2.)*eps0**2*mO*
   mOO*COO**3)
# oxygen Coulomb collision frequency
nuO=nuOe+nuOO
```

```
    return [nuel,nuep,Ce,Omge,c]
```

To generate learning data for regression analysis (small-angles):

```
#Regression Analysis - Small Angles

""" To generate training data by reading from the files in
   JE-ACF-Library
    This generates training data for Small-Angles files (
       only for Ti = 800K and 1200K).
    Output:
        X = Samples (States)           Format: [Ne, B, Te,
           Ti, alpha]
        Y = Labels  (Correction factor) Format: ACF_Brownian
            - ACF_Marco_Lib
"""

from jropack.jroread import *
import h5py
import sys, traceback

N=131072. #comes from Marco library
X = []
Y = []

for outer in  ['LN110','LN120','LN115','LN125']:

    for inner in [ '%s/LN%sBm20Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti16' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti16' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti16' %(outer,outer[2:])]:

        innerMost = ['/raid_b/IS_spc_model/JE-ACF-Library/%s
           ' %(inner)]
        for foldername in innerMost:
            datafiles = glob.glob1(foldername,'*S1.eh5')
```

```
datafiles.sort()
for filename in datafiles:
    counter=0
    f_S_L = h5py.File('%s/%s' %(foldername,
       filename),'r')
    if (int(float(f_S_L['Plasma']['logNe'].value
       )*10) == 115 ):
        Ne = 1.0*10**(11.5)
    elif (int(float(f_S_L['Plasma']['logNe'].
       value)*10) == 125 ):
        Ne = 1.0*10**(12.5)
    elif (int(f_S_L['Plasma']['logNe'].value) ==
        11 ):
        Ne = 1.0e11
    elif (int(f_S_L['Plasma']['logNe'].value) ==
        12 ):
        Ne = 1.0e12
    [nuel,nuep,Ce,Omge,c] = parameters(Ne, float
       (f_S_L['Plasma']['Bm'].value), int(f_S_L[
       'Plasma']['Te'].value), int(f_S_L['Plasma
       ']['Ti'].value))#(Ne, B, Te, Ti)

    if filename[21:22] == 'S':
        angle_range = 29
    else:
        angle_range = 20
    for asp in range(angle_range):
        # Generating state vector
        X.append(float(f_S_L['Plasma']['logNe'].
           value))
        X.append(float(f_S_L['Plasma']['Bm'].
           value))
        X.append(int(f_S_L['Plasma']['Te'].value
           ))
        X.append(int(f_S_L['Plasma']['Ti'].value
           ))
        if filename[21:22] == 'S':
            X.append(f_S_L['Correlations']['
               aspdegS'][asp])
        else:
            X.append(f_S_L['Correlations']['
               aspdegL'][asp])


        # Marco's Library
        if filename[21:22] == 'S':
            aspect = (f_S_L['Correlations']['
```

```
                        aspdegS'][asp])
                    dt=10.0e-6 #sampling time provided
                        by the library (units of seconds)
                        ; 10 microseconds S data
                    time_axis = arange(131072)*dt
                    tB = arange(N)*dt
                else:
                    aspect = (f_S_L['Correlations']['
                        aspdegL'][asp])
                    dt=0.1e-6 #sampling time provided by
                         the library (units of seconds);
                        0.1 microseconds S data
                    time_axis = arange(131072)*dt
                    tB = arange(N)*dt
                d = f_S_L['Correlations']['acf'][asp,:]

                # Brownian electron Gordeyev
                fradar=50.0e6
                kB = 2.0*pi*(fradar/(c/2))
                aspect=aspect*pi/180.
                varil=((2.*Ce**2)/nuel**2)*(nuel*tB-1+
                    exp(-nuel*tB))
                gam=arctan(nuep/Omge)
                varip=((2.*Ce**2)/(nuep**2+Omge**2))*(
                    cos(2*gam)+nuep*tB-exp(-nuep*tB)*cos(
                    Omge*tB-2.0*gam))
                acfB=exp(-((kB*sin(aspect))**2)*varil
                    /2.)*exp(-((kB*cos(aspect))**2)*varip
                    /2.)
                f = acfB

                corr_factor = d-f
                Y.append(corr_factor[0::1000])

#Reshape X to get: (number_of_samples, parameters)
samples = size(X)/5
X = reshape(X,(samples,5))
print 'shape(X) = ', X.shape

#Reshape Y to get: (number_of_samples, length_of_time_axis)
Y = reshape(Y,(size(X,0),132))
print 'shape(Y) = ', shape(Y)


#Create a scaled version of X
ne = [11.0, 11.5, 12.0, 12.5]
mag = [2.0e-05, 2.5e-05, 3.0e-05]
```

```
te = [600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200,
   2400, 2600, 2800, 3000]
ti = [800, 1000, 1200, 1400, 1600]
alpha = f_S_L['Correlations']['aspdegS'][:]

from copy import copy, deepcopy
Xn = deepcopy(X) #We may need X later, so Xn is an extra
   copy of X (as python by default performs shallow copy)
for i in range(4):
    for j in range(X.shape[0]):
        if X[j,0] == ne[i]:
            Xn[j,0] = i

for i in range(3):
    for j in range(X.shape[0]):
        if X[j,1] == mag[i]:
            Xn[j,1] = i

for i in range(13):
    for j in range(X.shape[0]):
        if X[j,2] == te[i]:
            Xn[j,2] = i

for i in range(5):
    for j in range(X.shape[0]):
        if X[j,3] == ti[i]:
            Xn[j,3] = i

for i in range(29):
    for j in range(X.shape[0]):
        if X[j,4] == alpha[i]:
            Xn[j,4] = i


#Save all the necessary input data on the disk
import pickle
f=open('X_dataS.pickle', 'wb')
pickle.dump(X, f)
f.close()

f=open('Y_dataS.pickle', 'wb')
pickle.dump(Y, f)
f.close()

f=open('Xn_dataS.pickle', 'wb')
pickle.dump(Xn, f)
f.close()
```

To generate learning data for regression analysis (large-angles):

```
#Regression Analysis - Large Angles

""" To generate training data by reading from the files in
   JE-ACF-Library
    This generates training data for Large-Angles files (
       only for Ti = 800K and 1200K).
    Output:
        X = Samples (States)           Format: [Ne, B, Te,
           Ti, alpha]
        Y = Labels  (Correction factor) Format: ACF_Brownian
            - ACF_Marco_Lib
"""

from jropack.jroread import *
import h5py
import sys, traceback

N=131072. #comes from Marco library
X = []
Y = []

for outer in  ['LN110','LN120','LN115','LN125']:

    for inner in [ '%s/LN%sBm20Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti08' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti12' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti14' %(outer,outer[2:]),
                   '%s/LN%sBm20Ti16' %(outer,outer[2:]),
                   '%s/LN%sBm25Ti16' %(outer,outer[2:]),
                   '%s/LN%sBm30Ti16' %(outer,outer[2:])]:

        innerMost = ['/raid_b/IS_spc_model/JE-ACF-Library/%s
           ' %(inner)]
        for foldername in innerMost:
            datafiles = glob.glob1(foldername,'*L1.eh5')
            datafiles.sort()
            for filename in datafiles:
                counter=0
                f_S_L = h5py.File('%s/%s' %(foldername,
                   filename),'r')
```

```
if (int(float(f_S_L['Plasma']['logNe'].value
   )*10) == 115 ):
    Ne = 1.0*10**(11.5)
elif (int(float(f_S_L['Plasma']['logNe'].
   value)*10) == 125 ):
    Ne = 1.0*10**(12.5)
elif (int(f_S_L['Plasma']['logNe'].value) ==
    11 ):
    Ne = 1.0e11
elif (int(f_S_L['Plasma']['logNe'].value) ==
    12 ):
    Ne = 1.0e12
[nuel,nuep,Ce,Omge,c] = parameters(Ne, float
   (f_S_L['Plasma']['Bm'].value), int(f_S_L[
   'Plasma']['Te'].value), int(f_S_L['Plasma
   ']['Ti'].value))#(Ne, B, Te, Ti)

if filename[21:22] == 'S':
    angle_range = 29
else:
    angle_range = 20
for asp in range(angle_range):
    # Generating state vector
    X.append(float(f_S_L['Plasma']['logNe'].
       value))
    X.append(float(f_S_L['Plasma']['Bm'].
       value))
    X.append(int(f_S_L['Plasma']['Te'].value
       ))
    X.append(int(f_S_L['Plasma']['Ti'].value
       ))
    if filename[21:22] == 'S':
        X.append(f_S_L['Correlations']['
           aspdegS'][asp])
    else:
        X.append(f_S_L['Correlations']['
           aspdegL'][asp])


    # Marco's Library
    if filename[21:22] == 'S':
        aspect = (f_S_L['Correlations']['
           aspdegS'][asp])
        dt=10.0e-6 #sampling time provided
           by the library (units of seconds)
           ; 10 microseconds S data
        time_axis = arange(131072)*dt
```

```
                    tB = arange(N)*dt
                else:
                    aspect = (f_S_L['Correlations']['
                       aspdegL'][asp])
                    dt=0.1e-6 #sampling time provided by
                        the library (units of seconds);
                       0.1 microseconds S data
                    time_axis = arange(131072)*dt
                    tB = arange(N)*dt
                d = f_S_L['Correlations']['acf'][asp,:]

                # Brownian electron Gordeyev
                fradar=50.0e6
                kB = 2.0*pi*(fradar/(c/2))
                aspect=aspect*pi/180.
                varil=((2.*Ce**2)/nuel**2)*(nuel*tB-1+
                   exp(-nuel*tB))
                gam=arctan(nuep/Omge)
                varip=((2.*Ce**2)/(nuep**2+Omge**2))*(
                   cos(2*gam)+nuep*tB-exp(-nuep*tB)*cos(
                   Omge*tB-2.0*gam))
                acfB=exp(-((kB*sin(aspect))**2)*varil
                   /2.)*exp(-((kB*cos(aspect))**2)*varip
                   /2.)
                f = acfB

                corr_factor = d-f
                Y.append(corr_factor[0::1000])

#Reshape X to get: (number_of_samples, parameters)
samples = size(X)/5
X = reshape(X,(samples,5))
print 'shape(X) = ', X.shape

#Reshape Y to get: (number_of_samples, length_of_time_axis)
Y = reshape(Y,(size(X,0),132))
print 'shape(Y) = ', shape(Y)

#Create a scaled version of X
ne = [11.0, 11.5, 12.0, 12.5]
mag = [2.0e-05, 2.5e-05, 3.0e-05]
te = [600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200,
   2400, 2600, 2800, 3000]
ti = [800, 1000, 1200, 1400, 1600]
alpha = f_S_L['Correlations']['aspdegL'][:]

from copy import copy, deepcopy
```

```
Xn = deepcopy(X) #We may need X later, so Xn is an extra
   copy of X (as python by default performs shallow copy)
for i in range(4):
    for j in range(X.shape[0]):
        if X[j,0] == ne[i]:
            Xn[j,0] = i

for i in range(3):
    for j in range(X.shape[0]):
        if X[j,1] == mag[i]:
            Xn[j,1] = i

for i in range(13):
    for j in range(X.shape[0]):
        if X[j,2] == te[i]:
            Xn[j,2] = i

for i in range(5):
    for j in range(X.shape[0]):
        if X[j,3] == ti[i]:
            Xn[j,3] = i

for i in range(20):
    for j in range(X.shape[0]):
        if X[j,4] == alpha[i]:
            Xn[j,4] = i

#Save all the necessary input data on the disk
import pickle
f=open('X_dataL.pickle', 'wb')
pickle.dump(X, f)
f.close()

f=open('Y_dataL.pickle', 'wb')
pickle.dump(Y, f)
f.close()

f=open('Xn_dataL.pickle', 'wb')
pickle.dump(Xn, f)
f.close()
```

To perform regression analysis using KNN regressor object (small-angles):

```
#A callback function to get inverse-distance Weights
```

```
def returnWeightsID(distances):
    weights = []
    for i in range(len(distances)):
        weights.append(1.0/(distances[i]))

    return weights

from sklearn.neighbors import KNeighborsRegressor

#User Inputs
newInput = [11.25, 2.0e-05, 1000, 800, 0.0]
NN = 2 #Number of nearest-neighbors
       #NN = 2^D, where D is the number of
       #dimensions changed in newInput
       #NN = {2,4,8,16,32}

#All the possible values of 5 parameters from numerical
   library
ne = [11.0, 11.5, 12.0, 12.5]
mag = [2.0e-05, 2.5e-05, 3.0e-05]
te = [600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200,
   2400, 2600, 2800, 3000]
ti = [800, 1000, 1200, 1400, 1600]
alpha = [0.0, 0.001, 0.002, 0.003, 0.004, 0.005, 0.006,
   0.007, 0.008, 0.009, 0.01,
          0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09,
             0.1, 0.2, 0.3, 0.4, 0.5,
          0.6, 0.7, 0.8, 0.9, 1.0]

#Get the bracketing indices for Ne, B, Te, Ti, alpha
indNe, indB, indTe, indTi, indAlpha = [], [], [], [], []
for i in range(3):
    if newInput[0] >= ne[i] and newInput[0] <= ne[i+1]:
        indNe.append(i)
        indNe.append(i+1)
        break

for i in range(2):
    if newInput[1] >= mag[i] and newInput[1] <= mag[i+1]:
        indB.append(i)
        indB.append(i+1)
        break

for i in range(12):
    if newInput[2] >= te[i] and newInput[2] <= te[i+1]:
        indTe.append(i)
        indTe.append(i+1)
```

```
        break

for i in range(4):
    if newInput[3] >= ti[i] and newInput[3] <= ti[i+1]:
        indTi.append(i)
        indTi.append(i+1)
        break

for i in range(28):
    if newInput[4] >= alpha[i] and newInput[4] <= alpha[i
       +1]:
        indAlpha.append(i)
        indAlpha.append(i+1)
        break

print indNe, indB, indTe, indTi, indAlpha

#Scale the newInput
newInputn = deepcopy(newInput)
newInputn[0] = float(newInputn[0] - ne[indNe[0]])/(ne[indNe
   [1]] - ne[indNe[0]]) + indNe[0]
newInputn[1] = float(newInputn[1] - mag[indB[0]])/(mag[indB
   [1]] - mag[indB[0]]) + indB[0]
newInputn[2] = float(newInputn[2] - te[indTe[0]])/(te[indTe
   [1]] - te[indTe[0]]) + indTe[0]
newInputn[3] = float(newInputn[3] - ti[indTi[0]])/(ti[indTi
   [1]] - ti[indTi[0]]) + indTi[0]
newInputn[4] = float(newInputn[4] - alpha[indAlpha[0]])/(
   alpha[indAlpha[1]] - alpha[indAlpha[0]]) + indAlpha[0]
print "newInputn = ", newInputn

#Get the shortest-distance in 5-dimension between newInput
   and the whole of input data set X
neigh = KNeighborsRegressor(n_neighbors=NN,algorithm='auto',
   weights=returnWeightsID)
neigh.fit(Xn[:,:],Y[:,:])
NearestPoints = neigh.kneighbors(newInputn, n_neighbors=NN,
   return_distance=True)
print NearestPoints
for i in range(NN):
    print X[NearestPoints[1][0][i],:]

#Generate the ACF from Brownian-Motion (BM)
N=131072.
aspect=newInput[4]*pi/180.
dtB=10.0e-6 #10 microseconds for Small-angles data
[nuel,nuep,Ce,Omge,c] = parameters(1.0*10**newInput[0],
```

```
   newInput[1], newInput[2], newInput[3]) #parameters(Ne, B,
    Te, Ti):
tB = arange(N)*dtB
varil=((2.*Ce**2)/nuel**2)*(nuel*tB-1+exp(-nuel*tB))
gam=arctan(nuep/Omge)
varip=((2.*Ce**2)/(nuep**2+Omge**2))*(cos(2*gam)+nuep*tB-exp
   (-nuep*tB)*cos(Omge*tB-2.0*gam))
acfB=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.)
plot(acfB[0::1000], '-r', label='BM')
legend(loc=1)
hold(True)

#Correct the BM plot by applying predicted-correction-factor
correctedBM = neigh.predict(newInputn)[0]-acfB[0::1000]
plot(correctedBM ,'-g', label='BM Corrected')
legend(loc=1)
xlabel('Time (s)')
ylabel('k=%s'%(NN))
title('ACF plot')

#savefig("ACF_SmallAngle_newTuple.png",bbox_inches='tight')
   #save as png
```

To perform regression analysis using KNN regressor object (large-angles):

```
#A callback function to get inverse-distance Weights
def returnWeightsID(distances):
    weights = []
    for i in range(len(distances)):
        weights.append(1.0/(distances[i]))

    return weights

from sklearn.neighbors import KNeighborsRegressor

#User Inputs
newInput = [11.25, 2.0e-05, 1000, 800, 45.0]
NN = 2 #Number of nearest-neighbors
       #NN = 2^D, where D is the number of
       #dimensions changed in newInput
       #NN = {2,4,8,16,32}

#All the possible values of 5 parameters from numerical
```

```
   library
ne = [11.0, 11.5, 12.0, 12.5]
mag = [2.0e-05, 2.5e-05, 3.0e-05]
te = [600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200,
   2400, 2600, 2800, 3000]
ti = [800, 1000, 1200, 1400, 1600]
alpha = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 3.0, 4.0, 5.0,
   6.0, 7.0, 8.0, 9.0,
         10.0, 15.0, 30.0, 45.0, 60.0, 90.0]

#Get the bracketing indices for Ne, B, Te, Ti, alpha
indNe, indB, indTe, indTi, indAlpha = [], [], [], [], []
for i in range(3):
    if newInput[0] >= ne[i] and newInput[0] <= ne[i+1]:
        indNe.append(i)
        indNe.append(i+1)
        break

for i in range(2):
    if newInput[1] >= mag[i] and newInput[1] <= mag[i+1]:
        indB.append(i)
        indB.append(i+1)
        break

for i in range(12):
    if newInput[2] >= te[i] and newInput[2] <= te[i+1]:
        indTe.append(i)
        indTe.append(i+1)
        break

for i in range(4):
    if newInput[3] >= ti[i] and newInput[3] <= ti[i+1]:
        indTi.append(i)
        indTi.append(i+1)
        break

for i in range(28):
    if newInput[4] >= alpha[i] and newInput[4] <= alpha[i
       +1]:
        indAlpha.append(i)
        indAlpha.append(i+1)
        break

print indNe, indB, indTe, indTi, indAlpha

#Scale the newInput
newInputn = deepcopy(newInput)
```

```
newInputn[0] = float(newInputn[0] - ne[indNe[0]])/(ne[indNe
   [1]] - ne[indNe[0]]) + indNe[0]
newInputn[1] = float(newInputn[1] - mag[indB[0]])/(mag[indB
   [1]] - mag[indB[0]]) + indB[0]
newInputn[2] = float(newInputn[2] - te[indTe[0]])/(te[indTe
   [1]] - te[indTe[0]]) + indTe[0]
newInputn[3] = float(newInputn[3] - ti[indTi[0]])/(ti[indTi
   [1]] - ti[indTi[0]]) + indTi[0]
newInputn[4] = float(newInputn[4] - alpha[indAlpha[0]])/(
   alpha[indAlpha[1]] - alpha[indAlpha[0]]) + indAlpha[0]
print "newInputn = ", newInputn

#Get the shortest-distance in 5-dimension between newInput
   and the whole of input data set X
neigh = KNeighborsRegressor(n_neighbors=NN,algorithm='auto',
   weights=returnWeightsID)
neigh.fit(Xn[:,:],Y[:,:])
NearestPoints = neigh.kneighbors(newInputn, n_neighbors=NN,
   return_distance=True)
print NearestPoints
for i in range(NN):
    print X[NearestPoints[1][0][i],:]

#Generate the ACF from Brownian-Motion (BM)
N=131072.
aspect=newInput[4]*pi/180.
dtB=0.1e-6 #0.1 microseconds for Large-angles data
[nuel,nuep,Ce,Omge,c] = parameters(1.0*10**newInput[0],
   newInput[1], newInput[2], newInput[3]) #parameters(Ne, B,
    Te, Ti):
tB = arange(N)*dtB
varil=((2.*Ce**2)/nuel**2)*(nuel*tB-1+exp(-nuel*tB))
gam=arctan(nuep/Omge)
varip=((2.*Ce**2)/(nuep**2+Omge**2))*(cos(2*gam)+nuep*tB-exp
   (-nuep*tB)*cos(Omge*tB-2.0*gam))
acfB=exp(-((kB*sin(aspect))**2)*varil/2.)*exp(-((kB*cos(
   aspect))**2)*varip/2.)
plot(acfB[0::1000], '-r', label='BM')
legend(loc=1)
hold(True)

#Correct the BM plot by applying predicted-correction-factor
correctedBM = neigh.predict(newInputn)[0]-acfB[0::1000]
plot(correctedBM ,'-g', label='BM Corrected')
legend(loc=1)
xlabel('Time (s)')
ylabel('k=%s'%(NN))
```

```
title('ACF plot')

#savefig("ACF_LargeAngle_newTuple.png",bbox_inches='tight')
   #save as png
```

REFERENCES

[1] E. Kudeki and M. A. Milla, "Incoherent scatter spectral theories—Part I: A general framework and results for small magnetic aspect angles," IEEE Trans. Geosci. Remote Sens., vol. 49, no. 2, pp. 315-326, Feb. 2011.

[2] M. A. Milla and E. Kudeki, "Incoherent scatter spectral theories—Part II: Modeling the spectrum for modes propagating perpendicular to B," IEEE Trans. Geosci. Remote Sens., vol. 49, no. 2, pp. 329-344, Feb. 2011.

[3] http://scikit-learn.org/stable/

[4] E. Kudeki, S. Bhattacharyya, and R. F. Woodman, "A new approach in incoherent scatter F region E × B drift measurements at Jicamarca," J. Geophys. Res., vol. 104, no. A12, pp. 28 145-28 162, Dec. 1999.

[5] J. D. Callen, Fundamentals of Plasma Physics, Chapter 2, Jul. 2006. [Online]. Available: http://homepages.cae.wisc.edu/~callen/book.html

[6] Y. L. Li, C. H. Liu, and S. J. Franke, "Adaptive evaluation of the Sommerfeld-type integral using the chirp z-transform," IEEE Trans. Antennas Propag., vol. 39, no. 12, pp. 1788-1791, Dec. 1991.

[7] K. Q. Weinberger and G. Gerald, Metric learning for kernel regression. [Online]. Available: http://machinelearning.wustl.edu/mlpapers/paper_files/AISTATS07_WeinbergerT.pdf

[8] G. James, D. Witten, T. Hastie, and R. Tibshirani, "Statistical Learning," in *An Introduction to Statistical Learning: With Applications in R*, Springer Texts in Statistics, vol. 103, New York: Springer, 2013, pp. 15-57.