PLASMA LINE GENERATION AND SPECTRAL ESTIMATION FROM ARECIBO OBSERVATORY RADAR DATA

BY

SHIYI YANG

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, 2018

Urbana, Illinois

Adviser:

Professor Erhan Kudeki

ABSTRACT

Incoherent scatter radar (ISR) signal spectrum is a statistical measure of Bragg scattered radio waves from thermal fluctuations of the electron density in the ionosphere. The ISR spectrum consists of up- and down-shifted electron plasma lines and a double-humped ion-line component associated with electron density waves with the governing dispersion relations of Langmuir and ion-acoustic waves, respectively. Such ISR spectral measurements can be conducted at the Arecibo Observatory, one of the most important centers in the world for research in radio astronomy, planetary radar and terrestrial aeronomy [*Altschuler*, 2002]. Although ISR measurements have been routinely taken at Arecibo since the early 1960s, full spectrum ISR measurements including the high-frequency plasma-line components became possible only very recently [*Vierinen et al.*, 2017] as a result of critical recent upgrades in hardware configuration and computing resources. This thesis describes the estimation and analysis of the full Arecibo ISR spectrum using Arecibo line- and Gregorian-feed data collected with Echotec and USRP receivers in September 2016 and processed using GPU-based parallel programming technology. In spectral analysis the "CLEAN" algorithm is used to deconvolve the measured ISR spectrograms from frequency/height mixing caused by the finite pulse length effect. CLEANed spectrograms are subsequently fitted to a Gaussian spectral model for each height to extract an estimate of the plasma-line frequency for each height.

ACKNOWLEDGMENTS

I would like to thank my adviser Prof. Erhan Kudeki for his guidance on my research. With his patient suggestions and thoughtful advice, I was able to solve the problems encountered in research and understand the obscure concepts involved during the experiment.

This thesis is based on incoherent scatter radar data collected at the Arecibo Observatory located near Arecibo, Puerto Rico. I thank the Arecibo staff scientists, engineers, and technicians who helped with the observations and initial data processing, and in particular Dr. Nestor Aponte and Phil Perillat for producing quick first-look data outputs and explaining many features and details of the experiments for us, and Arun Venkataraman for helping us transport "big data" from Arecibo to Illinois (in a multitude of ways).

I also thank Pablo Reyes for his academic advice in the ECE Illinois radar remote sensing group from day one and for teaching me how to utilize `remote2`, and Izzat Hajj for his technical support with the GPU system on `impact1`. Thanks are also due to Profs. Wen-mei Hwu and Volodymir Kindratenko for providing time and space on their experimental GPU systems at various stages of this project.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Large power VHF/UHF radar systems used with very large gain antennas in ionospheric research are known as incoherent scatter radars (ISR). The highest sensitivity ISR in operation in the world today is located at the Arecibo Observatory in Puerto Rico, and this thesis describes very broadband and high resolution scattered signal spectrum measurements conducted with the Arecibo ISR from ionospheric altitudes above about 100 km.

The mechanism underlying incoherent scattering in ISR operations is the "dipole radiation" of each free electron in the ionosphere made to oscillate by the transmitted radar pulse—this is known as the Thomson scattering process. The density of Thomson scattering free electrons in the ionosphere fluctuates as a superposition of electron density plane waves propagating in all directions across a broad spectrum of wavelengths with propagation velocities governed by the Langmuir and ion-acoustic wave dispersion relations. An ISR will only detect the superposed dipole radiation (Thomson scattering) signals of the electrons "belonging to" density waves whose wavefronts are perpendicular to the radar beam since scattering from electrons of waves propagating in other directions will be self-canceling due to the destructive interference. Furthermore, only the scattering of the electrons of waves with a wavelength equal to one half of the wavelength of the transmitted radar pulse will not self-cancel—this wave component solely responsible for the backscattered radar signal is called the "Bragg wave." The operation frequency of the Arecibo ISR is 430 MHz, which corresponds to about 70 cm wavelength, and 35 cm Bragg wavelength, meaning that Arecibo ISR will only detect signals returned from 35 cm wavelength electron density waves which are propagating parallel or anti-parallel to the direction of the radar beam. The Arecibo ISR signal spectrum will then exhibit a pair of peaks up-shifted from 430 MHz, each one caused by 35 cm Bragg waves propagating toward the radar at the ion-acoustic velocity C_s as well as the plasma-wave phase speed ω_p/k_B, respectively, where ω_p is the plasma frequency and k_B is the Bragg wavenumber, in

addition to a pair of down-shifted peaks caused by the same waves propagating in the opposite direction. The slower phase velocity peaks in the ISR spectrum relate to ion-acoustic waves in the ionospheric plasma while the fast phase velocity peaks represent electron plasma (Langmuir) waves. Landau damping and collisional damping of the scattering density waves will contribute to the broadening of each of these spectral peaks. The broadened low-frequency ion-acoustic peaks will tend to merge together to form the "double humped" ion-line feature of the ISR spectrum. Broadened electron plasma lines will be separated in the ISR spectrum from the ion line on both sides by several MHz of frequency corresponding to the "plasma frequency" of the ionosphere and will be undetactable unless the radar system bandwidth is greater than twice the plasma frequency. The shape of this entire broadband ISR spectrum, including the broadened ion and electron line components just described, has been derived by *Kudeki and Milla* [2011] and is essentially a superposition of electron and ion velocity distribution functions scaled by k_B and frequency-dependent weighting coefficients describing the collective interactions of ionospheric charged particles (electrons and ions) via polarization electric fields that they cause.

The high-power/large-aperture Arecibo ISR has an antenna bandwith exceeding the ionospheric peak plasma frequencies encountered in daytime hours. Also, with recent upgrades in its reception system, Arecibo is now capable of measuring the full ISR spectrum including both the ion- and plasma-line components described above. Such measurements have been successfully carried out at Arecibo only very recently by taking advantage of new digital- and software-based receiver technologies as well as improved computing speeds and data storage capacities. More specifically, the use of USRP hardware coupled with GNU radio toolkit is enabling the generation of terabytes of sampled radar data at 40 ns sampling rates, permitting estimation of the full ISR spectrum with a Nyquist frequency exceeding the maximum plasma frequencies encountered in the ionosphere. The introduction of GPU-based parallel computation techniques allows large-volume and high-speed spectral data analysis. The idea in parallel computation is to convert a significantly long serial data processing task into multiple parallel tasks of smaller length operations that can be quickly concluded—by generating multiple blocks and threads, the GPU allows spectral analysis of 20000 chunks of 16384-point time series simultaneously.

This thesis describes a GPU-based spectral analysis procedure we have used to compute broadband spectral maps (spectrograms) with Arecibo ISR data, includ-

ing the ion- and plasma-line features, and presents examples of computed spectral maps. The thesis also describes and discusses the convolutional distortions of the meaasured spectrograms caused by the radar pulsing scheme utilized in the radar measurements, as well as the deconvolution procedure based on the CLEAN algorithm that was developed and used to distill—from the distorted spectrograms obtained from the radar measurements—the actual underlying spectrogram of ionospheric electron density fluctuations that is geophysically significant.

There are six additional chapters in this thesis:

- Chapter 2 introduces basic concepts about the ionosphere and presents the derivation of a model equation of the spectrum of the electron density fluctuations causing the radar backscatter—a convolutionally distorted version of this spectrum also models the backscattered radar signal spectrum that can be computed from the sampled radar data.

- Chapter 3 describes the configuration of the Arecibo ISR system and the technologies utilized in the experiment including the coded-long-pulse (CLP) and uncoded-long-pulse (ULP) data collection modes.

- Chapter 4 describes the computation of the ISR and plasma-line spectrum using the GPU technology. Two types of GPU based procedures will be compared in terms of corresponding processing times.

- Chapter 5 describes the "CLEAN" algorithm used to extract—from the measured spectral maps of the radar signal—the underlying and geophysically significant spectral maps of ionospheric Bragg waves causing the radar backscatter. The procedure makes use of the 2D measurement "point spread function" in the frequency/range space that is responsible for the distortion of the Bragg wave spectral maps into measured ISR spectral maps. The derivation of point spread function (PSF) will be presented.

- Chapter 6 presents the details of the theory of plasma-line spectrum including the electron and ion Gordeyev integrals and their computation using the tabulated Dawson's integral data as well direct numerical computations using the chirp-z algorithm.

- Chapter 7 presents the conclusions of this study and ideas for future work in the general domain of ionospheric ISR spectrum estimation and spectral analysis using model fitting techniques.

- Appendices A and B present the PyCUDA and Python codes used in ISR and plasma-line spectrum estimation and the production of radar spectral maps including their inversion for ionospheric state parameters. Appendix C illustrates the codes in derivation of the plasma line from the raw data received from Arecibo Observatory.

CHAPTER 2

THE IONOSPHERE AND THE THEORY OF IONOSPHERIC INCOHERENT SCATTERING

2.1 Earth's Ionosphere

Earth's upper atmosphere is a partially ionized and electrically conducting plasma above about 50 km altitude. This deep layer of the atmosphere is called the "ionosphere" and blends into Earth's "magnetosphere" above an altitude of about 1000 km as a magnetized plasma and into the "solar wind" beyond.

Two types of classification are used to describe the properties of the ionosphere:

- Temperature profile: As shown in Figure 2.1 on the left, atmospheric temperature will decrease with height at an approximately constant rate below an altitude of about 10 km in a region known as the troposphere. Above the troposphere the temperature will increase with height throughout a region known as the stratosphere. The region of decreasing temperature above the stratosphere is known as the mesosphere, above which lies a region of increasing temperature known as the thermosphere. Stratospheric temperature increase is due to the absorption of the ultraviolet portion of solar radiation by ozone [*Kelley*, 2009]. Mesospheric temperature decrease with height above 50 km is caused by radiative cooling whereas thermospheric temperature increase is caused by daytime absorption of solar photons in UV and EUV frequency bands.

- Plasma density profile: This characteristic is defined as the number of free electrons per unit volume. As shown in Figure 2.1 on the right, ionospheric plasma density reaches its peak value at a few hundred kilometers altitude and exhibits substantial variation depending on daytime (solid curve) and nighttime (dashed curve) conditions. The daytime profile represents an equilibrium between the photo-ionization rate and recombination rate of plasma production and decay. In daytime, the solar spectrum is incident on

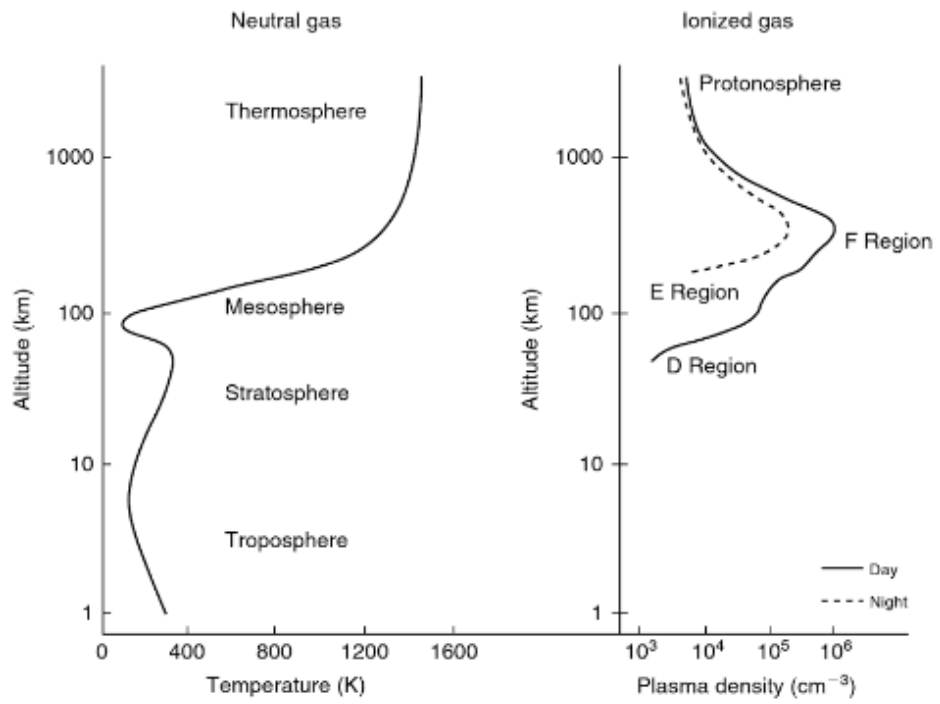


Figure 2.1: Typical profile of neutral atmospheric temperature and ionospheric plasma density with the various layers designated [*Kelley*, 2009].

a neutral atmosphere whose electron density increases exponentially with decreasing altitude. Since the photons are absorbed in the process of photoionization, the incoming beam itself decreases in intensity as it penetrates the atmosphere. The combination of decreasing solar flux, increasing neutral density, and diffusion provides a simple explanation for the basic large-scale vertical layer of ionization [*Kelley*, 2009]. From 60 km to 90 km altitude is the D region, from 90 km to 150 km is the E region, and beyond 150 km lies the F region. Peak plasma density occurs in the F region. The ion composition differs among regions; at lower altitudes, namely in the D region, it is less affected by the solar radiation. Therefore, there is a large number of neutral particles in the lower altitude. In the F region, on the other hand, due to the chemical reaction between oxygen gas, nitrogen gas and solar UV radiation, O^+ and N^+ ions will dominate the ion composition [*Kelley*, 2009].

Ionospheric plasma density drops dramatically at night at D region heights while staying nearly the same higher up in the ionosphere in the F-region; this

is due to to the dramatic difference of the recombination rates of the F- and D-region plasmas as explained in *Kelley* [2009], where atomic and molecular ions dominate, respectively. In the lower altitude D-region the recombination process may be:

$$NO^+ + e^- \rightarrow N + O,$$

and

$$O_2^+ + e^- \rightarrow O + O.$$

In F region, the dominant recombination process goes as

$$O^+ + e^- \rightarrow O + \text{photon}.$$

Without the solar radiation at night, the recombination at lower altitude will occur more frequently than at higher altitude. As a result, the plasma density in F region will still exist with a slight decrease in magnitude while the plasma density in D region will almost disappear.

2.2 Incoherent Scatter Theories

According to the incoherent scatter spectral theories [*Kudeki and Milla*, 2011], ionospheric incoherent scatter is caused by Thomson scattering of radar pulses at radio frequencies from collections of ionospheric free electrons. In this section, we will first describe the Thomson scattering process by single electrons, then study the Thomson scattering effect of multiple electrons.

2.2.1 Thomson Scattering by a Free Electron

A free electron excited by a transmitted TEM wave pulse field E_i will be forced to oscillate at the transmitted wave frequency and therefore radiate a scattered copy of the transmitted TEM wave pulse back towards the radar antenna. The oscillating electron behaves like a Hertzian dipole and, assuming a z-directed oscillation, the dipole radiation field propagating in direction θ away from the z-axis can be written as

$$\vec{E}_r(\mathbf{r}) = jk\eta_o I_o dz \frac{e^{-jkr}}{4\pi r} \sin\theta \; \hat{\theta} = \hat{\theta} \; \sin\theta \frac{r_e}{r} E_i e^{-jkr}, \tag{2.1}$$

where $\eta_o \equiv \sqrt{\frac{\mu_o}{\varepsilon_o}}$ is the intrinsic impedance of free space, θ is the zenith angle measured away and

$$r_e \equiv \frac{e^2}{4\pi\varepsilon_o mc^2} \approx 2.818 \times 10^{-15} \text{ m} \tag{2.2}$$

is the classical electron radius. This follows from modeling $I_o dz$ as $-ev$ where $v = \frac{-eE_i}{mj\omega}$ is the electron velocity phasor expressed in terms of the incident pulse field E_i assumed to point in z-direction, electron charge $-e$, electron mass m, and pulse frequency $\omega = ck$. Writing E_i seen by an electron located at position $\vec{r}$ with respect to the radar as $E_o(\vec{r})e^{-jkr}$, we can express the phasor amplitude of the scattered field 2.1 for direction $\theta = 90°$as

$$E_s = -\frac{r_e}{r}E_i e^{-jkr} = -\frac{r_e}{r}E_o(\vec{r})e^{-j2kr}. \tag{2.3}$$

This is the Thompson backscattered electric field received by the radar antenna from a single electron located at position $\vec{r}$ with respect to the radar antenna.

2.2.2 Thomson Scattering of Multiple Electrons

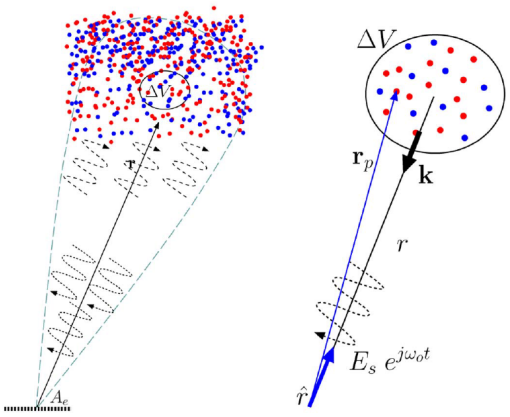


Figure 2.2: (a) Cartoon depicting radar scattering volume defined by the radar antenna beam and (b) geometry of a subvolume ΔV of the scattering volume shown in (a). Blue and red dots represent the scattering particles (electrons, mainly, since the scattering cross section of ions is negligibly small compared with that of the electrons) moving toward and away from the radar antenna, respectively. [*Kudeki and Milla*, 2011].

For a subvolume ΔV in the radar beam shown in Figure 2.2, where the spherical wave could be approximately treated as a plane wave, the total backscattered electric field from the subvolume ΔV will be the superposition of the backscattered fields (2.3) from each of the electrons contained within ΔV. This backscattered electric field can be expressed as

$$E_s = -\sum_{p=1}^{N_o \Delta V} \frac{r_e}{r_p} E_{op} e^{-j2kr_p} \approx -\frac{r_e}{r} E_o \sum_{p=1}^{N_o \Delta V} e^{j\vec{k}\cdot\vec{r}_p}, \tag{2.4}$$

where r_p and E_{op} refer to the distance and incident electric field intensity seen by the pth electron within ΔV, N_o is the average electron density within ΔV, and E_o refers to the incident electric field intensity at the center of subvolume ΔV. In addition, in (2.4) we used

$$\vec{k} \equiv -2k_o\hat{r} \tag{2.5}$$

to denote the Bragg wave vector and referred to the location of the pth electron with a position vector $\vec{r}_p$.

Given the electron density function in the subvolume ΔV,

$$n_e(\vec{r},t) \equiv \sum_{p=1}^{N_o \Delta V} \delta(\vec{r} - \vec{r}_p(t)), \tag{2.6}$$

where $\vec{r}_p(t)$ represents the electron trajectory, the spatial Fourier transform of the electron density function can be expressed as

$$\int n_e(\mathbf{r},t) e^{j\mathbf{k}\cdot\mathbf{r}} d\mathbf{r} \equiv \sum_{p=1}^{N_o \Delta V} e^{j\mathbf{k}\cdot\mathbf{r}_p(t)} \equiv n_e\mathbf{k},t. \tag{2.7}$$

And, as a consequence, we can express the Thomson backscattered field from a collection of electrons as

$$E_s(t) \approx -\frac{r_e}{r} E_i \sum_{p=1}^{N_o \Delta V} e^{j\mathbf{k}\cdot\mathbf{r}_p(t-\frac{r}{c})} \equiv -\frac{r_e}{r} E_i n_e(\mathbf{k}, t - \frac{r}{c}). \tag{2.8}$$

As shown in *Kudeki and Milla* [2011], the backscattered field (2.8) implies a baseband power spectrum (Fourier transform of the field ACF)

$$\left\langle |E_s(\omega)|^2 \right\rangle = \frac{r_e^2}{r^2} |E_i|^2 \left\langle |n_e(\vec{k},\omega)|^2 \right\rangle \Delta V \tag{2.9}$$

from subvolume ΔV and a of backscattered signal from subvolume ΔV and a total backscattered power of

$$P_r = \int \frac{d\omega}{2\pi} \int dV \, \frac{|E_i|^2/2\eta_o}{r^2} \, r_e^2 \left\langle |n_e(\vec{k},\omega)|^2 \right\rangle A_e(\theta,\phi) \tag{2.10}$$

from all the subvolumes covering the radar antenna beam — above $A_e(\theta,\phi) \equiv \frac{\lambda^2}{4\pi} G(\theta,\phi)$ denotes the effective antenna area for reception expressed in terms of antenna gain $G(\theta,\phi)$ and wavelength $\lambda = \frac{2\pi}{k}$, while $\left\langle |n_e(\vec{k},\omega)|^2 \right\rangle$ represents the electron density spectrum.

Finally, using $dV = r^2 d\Omega dr$ as well as $\frac{|E_i|^2}{2\eta_o} = \frac{P_t G(\theta,\phi)}{4\pi r^2}$ in (2.10), we obtain

$$P_r = \int \frac{d\omega}{2\pi} \int dr \int d\Omega \, \frac{P_t G(\theta,\phi)}{4\pi r^2} \, r_e^2 \left\langle |n_e(\vec{k},\omega)|^2 \right\rangle A_e(\theta,\phi). \tag{2.11}$$

This result is recognized to be a "soft target" radar equation wherein $r_e^2 \left\langle |n_e(\vec{k},\omega)|^2 \right\rangle$ stands for the backscatter radar cross-section (RCS) per unit volume, per solid angle, per unit frequency. Accordingly, using SI units, $\sigma(\vec{k},\omega) \equiv 4\pi r_e^2 \left\langle |n_e(\vec{k},\omega)|^2 \right\rangle$ is the "soft target" backscatter RCS of the ionosphere per cubic meter, per hertz.

CHAPTER 3

RADAR CONFIGURATION AT ARECIBO OBSERVATORY

The Arecibo Observatory is located near the northern coastal town of Arecibo on the island of Puerto Rico in a region populated by natural sinkholes in its terrain. One such sinkhole about 15 km inland from Arecibo houses the largest single-dish spherical reflector antenna in the world used for space research. The Arecibo reflector antenna is part of the Arecibo ISR system that was designed and built by William E. Gordon of Cornell University in the mid-1960s and maintained by Cornell University until 2011. Since 2011 the Arecibo Observatory and its ISR system have been operated under cooperative agreements with the National Science Foundation. The Arecibo facility supports three major areas of research: radio astronomy, atmospheric science, and radar astronomy. The observatory has radar transmitters with effective isotropic radiated power of 1 MW at 2380 MHz (S-band system) and 2.5 MW at 430 MHz (the ionospheric ISR system). This thesis is focused on the use the Arecibo ISR system for ionospheric measurements and research.

3.1 Arecibo ISR - System Description

The Arecibo ISR is a 430 MHz backscatter radar system using a spherical dish antenna of 305 m diameter shown in Figure 3.1. The radar transmitter generates pulses with peak power of 2.5 MW at the 430 MHz operating frequency [*Isham et al.*, 2000]. Given its very large antenna aperture and transmitted power operated within the UHF band, Arecibo ISR achieves an overall sensitivity ~ 100 larger than that of other ISR systems currently in existence. The original 430 MHz line feed of the Arecibo Observatory could detect the F-region plasma line radar scattering over a ± 7.5 MHz bandwidth. After upgrades in the year 2000, Arecibo bandwidth was broadened to ± 15 MHz, as a result of which the detection frequencies cover the range from 415 MHz to 445 MHz in ISR operations. The 430

Figure 3.1: Aerial view of the Arecibo Observatory. (Credit Arecibo Observatory/NSF.)

MHz line feed makes an efficient use of the main dish when pointed vertically, as its radiation pattern fills the available aperture. The new Gregorian feed that was added to the system during the 2000 upgrades enables dual beam operations for more efficient determinations or ionospheric plasma drifts [*Isham et al.*, 2000]. In general, ISR signal spectrum contains two distinct features of importance in ionospheric remote sensing: One is the low frequency ion-line feature, while the second one is the plasma-line peaks seen at large frequency offsets from the radar carrier as shown in Figure 3.2. Both spectral lines are characteristics of radar pulses which are Bragg backscattered from electron density waves naturally excited and dissipated within ionospheric plasmas under thermal equilibrium. The ion-line specifically is due to Bragg scattering from waves in the ion-acoustic branch and can be utilized to estimate the plasma parameters including electron and ion densities, temperatures, and ion composition. The plasma line spectrum, on the other hand, results from Bragg scattering from the Langmuir waves, and

the spectrum is intensified by inverse Landau damping from photo-electrons and can be used to measure the electron density and the electron energy spectrum [*Yngvesson and Perkins*, 1968].

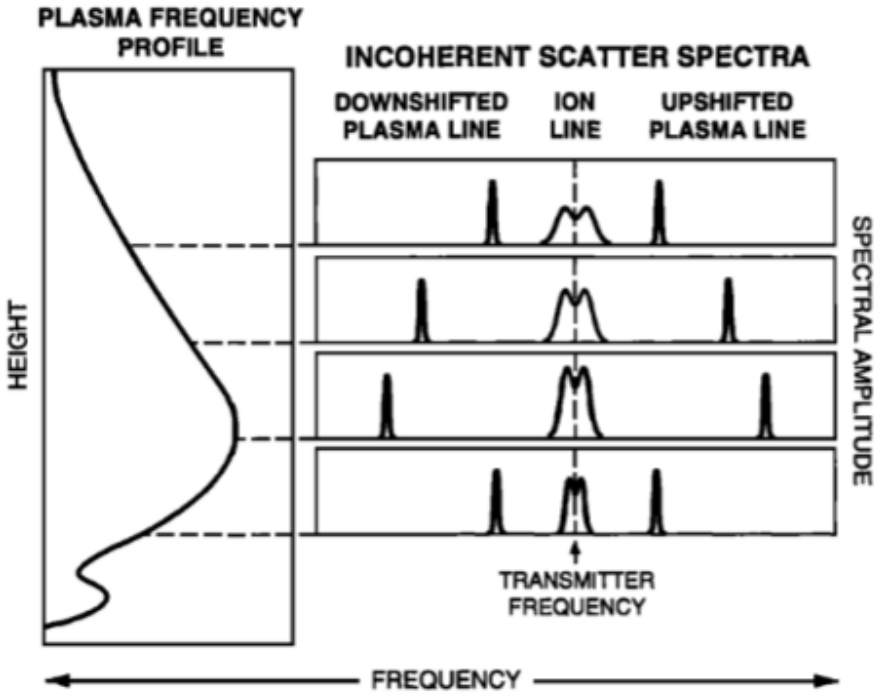


Figure 3.2: A schematic representation of the major features of the incoherent scatter radar spectrum and its relationship to the ionospheric plasma frequency profile. The ion and plasma lines arise from Bragg backscatter off thermal density fluctuations, the power of which is concentrated in the acoustic and electron Langmuir plasma wave modes, respectively [*Isham et al.*, 2000].

3.2 Arecibo ISR Data Modes

Isham et al. [2000] describe six data acquisition modes for Arecibo ISR operations; two of them, coded long pulse (CLP) and uncoded long pulse (ULP), are of importance in F-region and topside ionosphere studies. The use of CLP and ULP modes for narrowband ion-line measurements with the main Arecibo ISR receiver and broadband full spectrum measurements (including the plasma lines) with Echotek and USRP receivers will be described here.

3.2.1 Coded Long Pulse Mode

In soft target radar measurements, if the correlation times of the scattering density waves are short compared to the inter-pulse period (IPP), then "pulse-to-pulse" correlation methods cannot be used and it becomes necessary to utilize "within pulse" correlation methods with multiple samples taken from the superposed echoes of transmitted pulses whose lengths need to exceed the sampling interval by some substantial margin. Such "within pulse" operations are generally referred to as "long pulse" techniques. The disadvantage of using long pulses is accepting relatively poor radar range resolution, unless the target SNR is so strong that short baud length coding can be applied to the transmitted long pulse to produce a range resolution determined by the baud length rather than the pulse length.

The coded long pulse (CLP) mode utilized at Arecibo implements this idea and works well to probe the lower F-region altitudes of the ionosphere where the electron density is relatively large and the corresponding scatter SNR is sufficient. The mode was developed by *Sulzer* [1986] for high-resolution ion-line measurements but it also works well in high-resolution plasma line measurements, as will be shown in this thesis.

Arecibo CLP implementation for F-region measurements utilizes a 10 ms IPP and a 440 μs long transmitted rectangular pulse, described in Figure 3.3, subdivided into 220 bauds of 2 μs length within which the phase of the 430 MHz carrier is randomly assigned as 0° or 180° phase shifts. The 220 baud length random binary phasing sequence encoded in transmission is sampled and the sampled sequence is utilized to decode the echo signals to achieve a range resolution of 300 m corresponding to the 2 μs baud length (instead of 220X300 m corresponding to the 440 μs pulse length) [e.g. *Djuth et al.*, 1994].

More specifically, the CLP returns are sampled at the ion-line receiver output as (I, Q) pairs (or as $I + jQ$ complex valued voltage phasors) at 2 μs intervals matching the baud length of the binary code following an analog low-pass-filtering operation. Groups of 220 samples are conjugate multiplied with samples of the transmitted pulse taken at the same receiver output to implement an effective "pulse compression" and achieve a range resolution of 300 m for ion line measurements. These measurements detect the ion-line spectrum from a 300 m wide slab of the ionosphere without any distortion while smearing out the spectral content of 219 neighboring 300 m slabs located above and below as a white spectral background.

The binary phase coding used in the CLP technique is chosen such that its cross correlation with its shifted versions is weak due to a destructive interference effect that results in the smearing effect just mentioned and can be characterized as a convolutional distortion as discussed in Section 5.1. The spectral estimate obtained in this fashion also lacks the high-frequency plasma line features located well beyond the Nyquist frequency limit corresponding to the 2 μs sampling interval used with the ion-line receiver output.

To estimate the full ISR spectrum including the plasma-line features, the CLP returns need to be detected and sampled over a sufficiently wider baseband. This is achieved using a USRP-based receiver operated in parallel with the ion-line receiver. The sampling interval of the USRP receiver is T =40 ns corresponding to a Nyquist bandwitdh of $B = 12.5$ MHz. In spectral estimation with the USRP receiver, 11000 samples of the receiver corresponding to 440 μs of transmitted pulse length are first conjugate multiplied with 11000 samples taken during the pulse transmission time (this is the encoding phase sequence in effect) and the product sequence is FFT'ed after zero-padding to a length of 16384 samples. This is a very demanding operation that requires the use of GPU technology as described in Chapter 4.

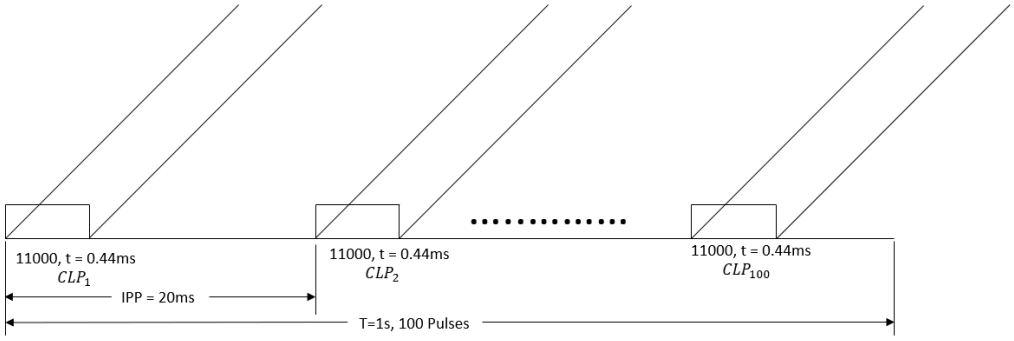


Figure 3.3: A schematic representation of the coded long pulse with 0.44 ms pulse width. In a one-second file, the total pulse number is 100 and each pulse has 11000 samples.

3.2.2 Uncoded Long Pulse Mode

The uncoded long pulse (ULP) mode is similar to CLP but no phase coding is applied. Consequently the range resolution in this mode is determined by the

pulse length rather than some baud length. ULP mode is used to complement CLP to extend the region successfully probed by CLP to higher altitudes where ionospheric densities are lower and therefore the SNR is insufficient for CLP to work. ULP works at those altitudes of weaker scattering cross sections because in the absence of coding/decoding step the spectral contributions of neighboring ionospheric slabs are not caused to smear; instead, they reinforce the ion-line spectrum within the same frequency band and stand out from the background noise level as a detectable and recognizable feature. Of course the price paid for this is range resolution related ambiguities and spectral distortions—if ionospheric parameters are changing substantially throughout the altitude region $c(t \pm \Delta t)/2$, where t is the sampling time and Δt the pulse length, then the computed spectrum will include convolutional distortions which need to be addressed in some fashion (see Chapter 5) during the data inversion stage.

Arecibo ULP implementation typically uses 500 μs pulse lengths and 20 ms IPPs and 2 μs sampling intervals as shown in Figure 3.4. With 250 samples taken per transmitted pulse echo, the ISR ion-line spectrum can be obtained using 250-point FFTs or else by Fourier transforming the lag-profile matrices constructed with auto-correlation function estimates obtained by circularly shifted products of data sample sets. Once again plasma-line components are smeared out of the ion-line estimates obtained with 2 μ samples taken at the output of the ion-lime receiver system. The full ISR spectrum including the plasma-line features can be estimated with the ULP returns sampled at the output of the USRP receiver operated in parallel with the ion-line receiver. The sampling interval of the USRP receiver is again T =40 ns corresponding to a Nyquist bandwitdh of $B = 12.5$ MHz. With the ULP mode, 12500 samples of the USRP receiver taken at T =40 ns intervals across the 500 μs long pulse length are first FFT'ed after zero padding to a 16384 length. The operations are compute-intensive and are carried out with GPUs as described in Chapter 4. A further detail of the Arecibo ULP mode is that the carrier frequency used in consecutive pulse transmissions is varied as 430 MHz $\pm$62.5 kHz from pulse to pulse. As a consequence, baseband data samples taken after consecutive pulse transmissions need to be further de-modulated by $\mp$62.5 kHz before spectral analysis. This is true with both ion-line and full spectrum analysis performed with the ion-line and the USRP receivers, respectively.

Finally, Arecibo ULP experiments are also run quite frequently using 1000 μs long uncoded pulses as shown in Figure 3.5 with 2 μs or 4 μs sampling of the ion-line receiver output. In these "long ULP" (LULP) measurements, FFT lengths will

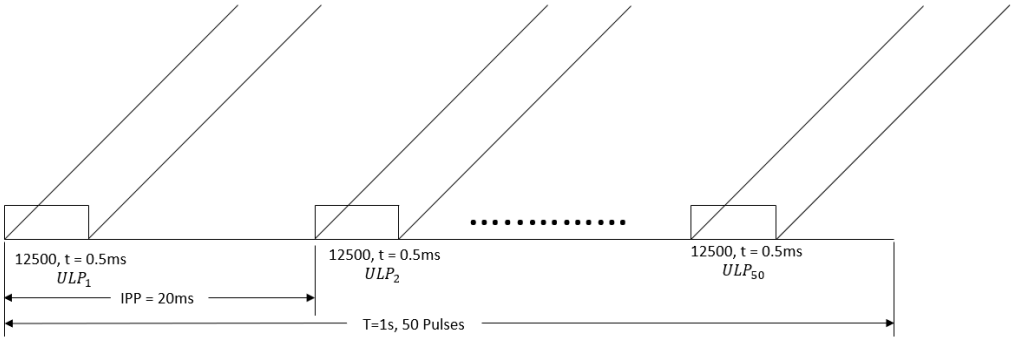


Figure 3.4: A schematic representation of the uncoded long pulse with 0.5 ms pulse width. In a one-second file, the total pulse number is 50 and each pulse has 12500 samples.

jump from 12500 to 25000 for full spectrum estimates using the USRP receiver output still sampled at 40 μs intervals.

In the next chapter we will describe how CLP and ULP modes with shorter and longer pulse lengths are interleaved in time during typical observation runs.

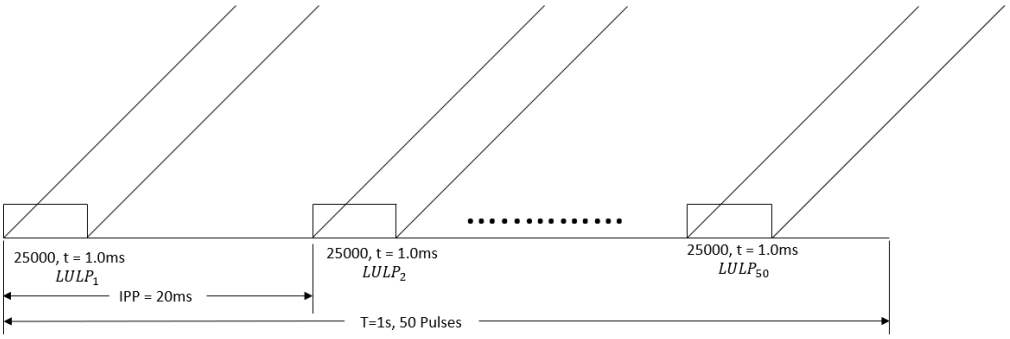


Figure 3.5: A schematic representation of the longer uncoded long pulse with 1 ms pulse width. In a one-second file, the total pulse number is 50 and each pulse has 25000 samples.

3.3 ISR Signal Processing and Spectral Estimation

The ISR receiver will effectively convert the back-scattered electric field (2.8) detected by the radar antenna into a voltage phasor

$$V(t) \equiv I(t) + jQ(t) = \ell E_s(t) = -\frac{r_e}{r} E_i \ell n_e(\mathbf{k}, t), \tag{3.1}$$

where ℓ is the antenna effective length in the direction of scattering volume ΔV, and $I(t)$ and $Q(t)$ are the in-phase and quadrature components of the voltage signal with 90° phase shift. The voltage then is sampled at $t = T$ which is the IPP in pulse-to-pulse analysis or the sampling time after pulse transmission in a long pulse experiment.

Consider an N-point voltage time series

$$V_n \equiv V(nT), n \in [0, N-1] \tag{3.2}$$

with sampling interval T forming a discrete Fourier transform pair with

$$\tilde{V}_m \equiv T \sum_{n=0}^{N-1} V_n e^{-j\frac{2\pi nm}{N}} \Leftrightarrow V_n = \frac{1}{NT} \sum_{m=0}^{N-1} \tilde{V}_m e^{j\frac{2\pi nm}{N}} \tag{3.3}$$

for $m \in [0, N-1]$, where $\frac{1}{NT} \equiv \Delta f$ is the sampling interval or "resolution" in the frequency domain.[1] The "average power" of the time series V_n is the "expected value" of

$$\frac{1}{N} \sum_{n=0}^{N-1} |V_n|^2 = \Delta f \sum_{m=0}^{N-1} \frac{|\tilde{V}_m|^2}{NT}, \tag{3.4}$$

which is the discrete form of Parseval's theorem where $\frac{|\tilde{V}_m|^2}{NT}$ is the "signal periodogram" and its expected value $\frac{\langle|\tilde{V}_m|^2\rangle}{NT}$ the "signal spectrum".

To develop a model for the signal spectrum we note that if the time series V_n is a sample sequence of a wide sense stationary (WSS) random process, then

$$\begin{aligned} \langle|\tilde{V}_m|^2\rangle &= T^2 \sum_{n=0}^{N-1} \sum_{n'=0}^{N-1} \langle V_n^* V_{n'}\rangle e^{-j\frac{2\pi m(n'-n)}{N_s}} = T^2 \sum_{p=-N}^{N} (N-|p|)\langle V_n^* V_{n+p}\rangle e^{-j\frac{2\pi mp}{N}} \\ &= NT^2 \sum_{p=-N}^{N} (1-\frac{|p|}{N})\langle V_n^* V_{n+p}\rangle e^{-j\frac{2\pi mp}{N}} \end{aligned} \tag{3.5}$$

in terms of the auto-correlation function (ACF) $\langle V_n^* V_{n+p}\rangle$ of the process. In that case the spectrum

$$\frac{\langle|\tilde{V}_m|^2\rangle}{NT} = T \sum_{p=-N}^{N} (1-\frac{|p|}{N})\langle V_n^* V_{n+p}\rangle e^{-j\frac{2\pi mp}{N}}, \tag{3.6}$$

[1] Notice $\frac{2\pi nm}{N} = \frac{2\pi nTm}{NT} = 2\pi nTm\Delta f$ in the exponents in (3.3), a discrete form of the product ωt.

forming a discrete Fourier transform pair with a triangle weighted signal ACF

$$(1-\frac{|p|}{N})\langle V_n^* V_{n+p}\rangle = \Delta f \sum_{m=0}^{N-1} \frac{\langle|\tilde{V}_m|^2\rangle}{NT} e^{j\frac{2\pi mp}{N}}, \tag{3.7}$$

which in turn reduces to (and thus verifies) the expected value of the discrete form of Parseval's theorem (3.4) if/when $p = 0$.

The WSS assumption leading to the Fourier transform pair relations (3.6) and (3.7) is in general valid for voltage time series V_n obtained in pulse-to-pulse radar experiments. Time series V_n formed in long pulse experiments, however, will in general fail to fit the WSS model (because of altitude dependence of the ionosphere from which different samples of V_n are obtained), and, as a consequence, the computed power spectrum using $\frac{|\tilde{V}_m|^2}{NT}$ will in general be an "altitude mixed" version of the discrete Fourier transform pair of true ACFs $\langle V_n^* V_{n+p}\rangle$ belonging to different altitudes as governed by some "ambiguity function" that will be derived and discussed in Chapter 5.

In practical spectral estimation with voltage time series V_n, we obtain, using an FFT routine, $\tilde{V}_m/T = \text{FFT}_m[V_n]$, and thus the spectrum, scaled by N/T, can be obtained by arithmetic averaging all $|\text{FFT}_m[V_n]|^2$ produced over some averaging time interval, and optionally this time average can be divided by N^2 if it is desirable that a sum over all m (frequency index) gives the signal power.

The choice of averaging time depends on signal strength (SNR to be specific) and how fast the spectral characteristic of the probed medium is changing. Reasonable averaging times need to be "short enough" compared to time scales over which the average properties of the scattering medium changes and "sufficiently long" in order to reduce the random fluctuations of the frequency distribution of back-scattered radar signal below the additive noise related fluctuations in the computed spectra.

Finally, the key in physical modeling of signal spectrum and ACF is the relation (3.1) according to which the average value of $|\text{FFT}_m[V_n]|^2$ will be a scaled version of the space-time average of $|\text{FFT}_m[n_e(\vec{k}, nT)]|^2$ or the Fourier transform pair of the electron density ACF $\langle n_e^*(\vec{k}, nT) n_e(\vec{k}, (n+p)T)\rangle$.

3.4 Experiment Dates and Notes

The Arecibo CLP and ULP data processed in this project were acquired at Arecibo during two three-day campaigns conducted in 2016 over six days of observations. Data are stored in binary files covering 10 seconds of CLP, 10 seconds of ULP (with 0.5 ms pulse width) and 10 seconds of L-ULP (with 1 ms pulse width). Ion-line data files contain samples taken from both the line-feed and Gregorian feed channels. Separate data files contain the Gregorian/echotek samples covering the 5.5-9.5 MHz band and the line-feed/USRP samples covering the -12.5-12.5 MHz band used for plasma line estimation. Table 3.1 illustrates the experiment dates in 2016, and Table 3.2 also shows the data collecting hours for each experiment day in 2016. Table 3.2 shows the operation hours for each experiment date in 2016.

Table 3.1: Experiment dates in 2016. The radar pulse configuration is shown in these six days of available data.

Pulse Type	CLP	ULP	Long ULP
Day of Year	210-212, 267-269		
Date	July 28-30, September 23-25		
Height Range (km)	0-1500	0-3000	0-3000
IPPs (ms)	10	20	20
Pulse Width (ms)	0.44	0.5	1.0

Table 3.2: Experiment hours for each date in 2016.

Day of Year	Date	Time
210	July 28	22:00:42-23:00:42
211	July 29	00:00:42-11:00:42, 23:27:32-23:59:58
212	July 30	00:29:58-3:29:58, 13:25:01-22:17:48
267	September 23	20:31:05-23:31:05
268	September 24	00:31:05-23:51:52
269	September 25	00:51:52-14:51:52

CHAPTER 4

COMPUTATION OF ARECIBO ISR BROADBAND SPECTROGRAMS

ISR spectral estimation is a computationally demanding task in particular when high resolution estimates of broadband spectra are desired as in Arecibo ISR applications [*Djuth et al.*, 1994]. Fortunately we are living in an era of rapid technological development and increasing computational speeds based on increasingly powerful GPU designs and GPU applications in parallel programming to process increasingly large volumes of data. In this chapter we will describe how we take advantage of new GPU and parallel programming techniques in order to produce high-resolution broadband ISR spectrograms from raw data streams collected at Arecibo and how to process the resulting spectrograms to correct for multiplicative and convolutional distortions and extract geophysical parameters by suitable fitting techniques. Our results will show that with the introduction of GPU-based coding platforms including CUDA C and PyCUDA, ISR spectral estimation resolving the high-frequency plasma-line features becomes possible within acceptable processing times. We will illustrate the use of a Tesla K40 GPU system to process both coded and uncoded long-pulse ISR data from Arecibo using several different approaches and compare the results and performances. PyCUDA gives access to NVIDIA's CUDA parallel computation API [*Klöckner et al.*, 2012] which enables efficient implementation of large-scale FFTs.

Table 4.1: Tesla K40 specification.

GPU	Tesla K40
Stream Processors	2880
Core Cloak	745 MHz
Memory Width	384 bit
Memory Clock	6 GHz
Single Precision	4.29TFLOPS
VRAM	12 GB

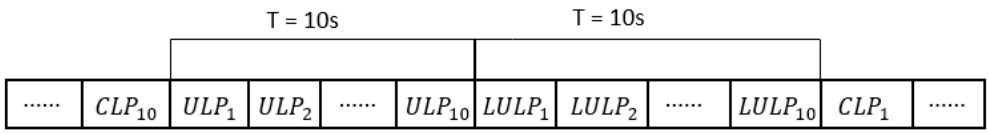


Figure 4.1: Data arrangement. There are 10 seconds CLP, 10 seconds ULP (0.5 ms puls width) and 10 seconds LULP (1 ms pulse width) files every 30 seconds.

4.1 Tesla K40 - Introduction

To process the Arecibo ISR raw data for full bandwidth spectral estimation we use a Tesla K40 system with 12 GB memory that features a 4.29 TFLOPS single precision performance. This system is hosted by a server machine named `impact1` on `csl.illinois.edu` subnet which is a Linux machine with a fast connection to `illinois.edu` network domain and to our data server `remote2` on `ece.illinois.edu` subnet that stores the USRP raw voltage data from Arecibo ISR experiments. According to Table 4.1, Tesla K40 on `impact1` features nearly 3000 stream processors and also 6 GHz memory clock, which ensures the capability to deal with large-scale FFTs.

4.2 ULP Spectrograms

Contemporary ISR experiments at Arecibo make frequent use of the coded long pulse (CLP) and uncoded long pulse (ULP) data acquisition modes. It is typical to interleave the use of these modes at 10 second intervals. Arecibo ISR data is stored in 1 s files—during a 1-min time interval 60 data files are produced. The arrangement of these stored files is 10 seconds of ULP with 0.5 ms pulse width, and 10 seconds of longer ULP with 1 ms pulse width, followed by 10 seconds of CLP files and so on, as shown in Figure 4.1, where LULP represents the uncoded long pulse with pulse width 1 ms.

We will next describe the generation of broadband ISR spectra and spectrograms using ULP mode raw data sampled at the USRP receiver output with the help of Figure 4.2. In the ULP mode the IPP is 20 ms, transmitted pulse length is 0.5 ms, and the sampling interval of the USRP receiver output is $T = 40$ ns. Therefore 500000 complex voltage raw data samples are taken in each pulse transmission period, with the first 12500 of them taken during the time of pulse trans-

mission. Consider moving the first 12500 samples of raw data V_0 to $V_{12500-1}$ to the first row of a matrix $\mathbf{V}$ depicted in Figure 4.2 to represent the voltage time series of height h_0 (assigned a zero value). Then by skipping one sample, the next 12500 samples of raw data from V_1 to V_{12500} are moved to the second row of the matrix to form the time series for $h_1 = 6$ m. The spectral measurement from 0 to 1180 km will require a 196608×12500 matrix formed in this manner. If coverage of higher altitudes is needed the matrix can be extended up to 3000 km in this manner, 3000 km being the limit imposed by the 20 ms IPP of the ULP experiment.

Here we will limit our attention to spectral estimation up to 1180 km height (plasma line is undetectably weak at higher altitudes) and thus the 2-D data matrix shown in Figure 4.2 is relevant. As shown in the figure the 2-D matrix is extended by zero padding to 2^{14} for FFT—the final dimensions are 196608×16384. The formation of the matrix shown in Figure 4.2 is carried out using the GPU mounted on `impact1` accessing the raw USRP voltage data through a file system mounted from `remote2`.

Before a 1-D parallel FFT operation is performed with the matrix, each row is multiplied by $e^{\pm j2\pi f_o nT}$, where $f_o = \pm 62.5$ kHz is the transmitted frequency offset from the 430 MHz carrier frequency utilized in the ULP mode on alternate pulses—this multiplication pulls the voltage time series in each row back to 430 MHz based baseband. Finally FFT will be implemented on each row of the matrix to derive the ULP spectrogram.

GPU-based platforms PyCUDA and CUDA C are used in applying the FFT operation on the 2-D data matrix prepared as described above. In PyCUDA implementation the "batch value" — the number of FFTs implemented simultaneously— needs to be specified. In this experiment, we choose the batch value to be some power of 2, from 1024 to 16384. Processing a 196608×16384 matrix in the GPU as a single step is not applicable because the memory space required for the operation would exceed the Tesla K40 memory capacity. We have to subdivide the 2-D matrix into several blocks, such that we can iterate each block of the spectrogram matrix to implement the parallel FFTs. When determining the batch value, we could initialize a large number such as 2^{14}, such that a smaller number of blocks for iteration is required. However, a large batch value will introduce more threads in GPU to handle the computation, which will cause more overhead in the calculation. In this case, there is a trade off between the number of iteration and the thread block required in this computation.

As shown in *Kudeki and Milla* [2006] ISR spectra $S_m \equiv \frac{\langle |\tilde{V}_m|^2 \rangle}{NT}$ estimated with

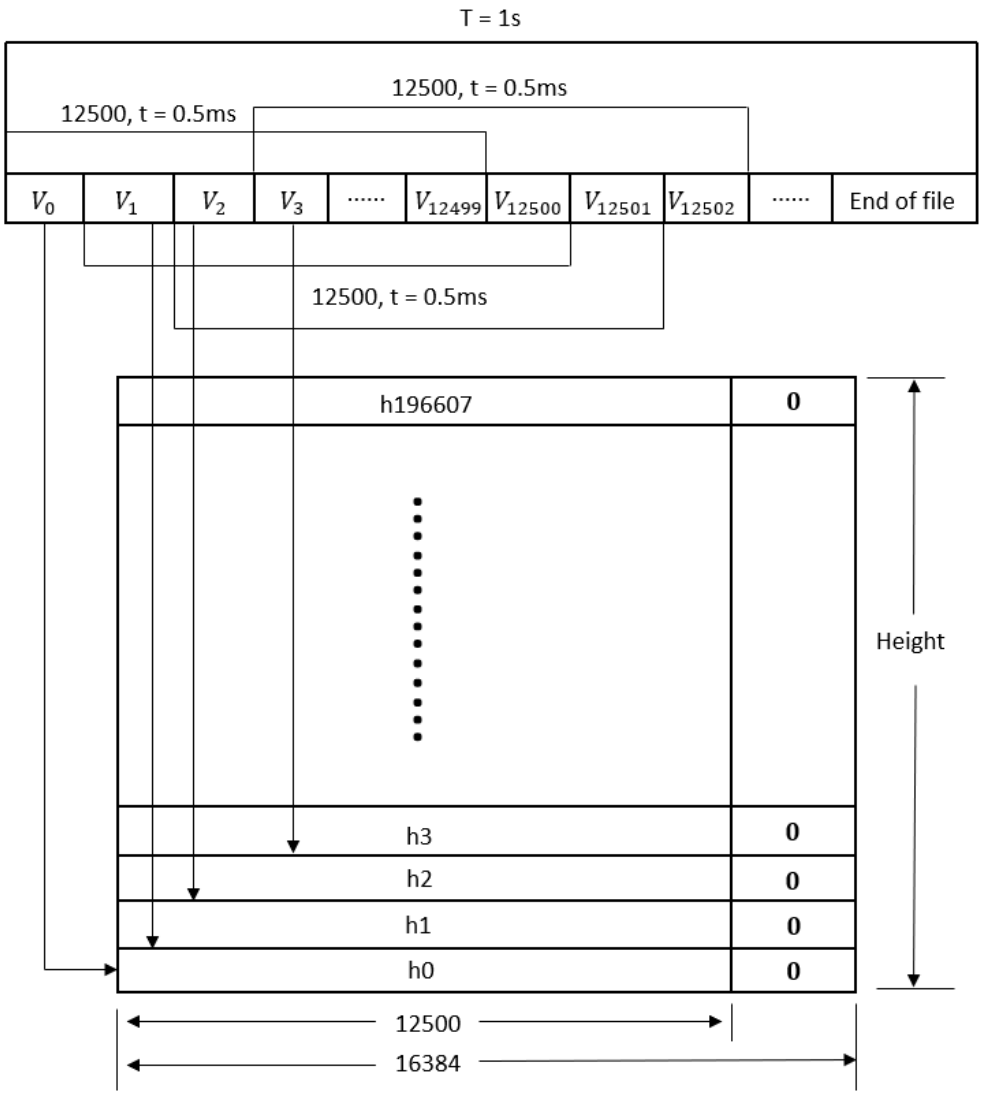
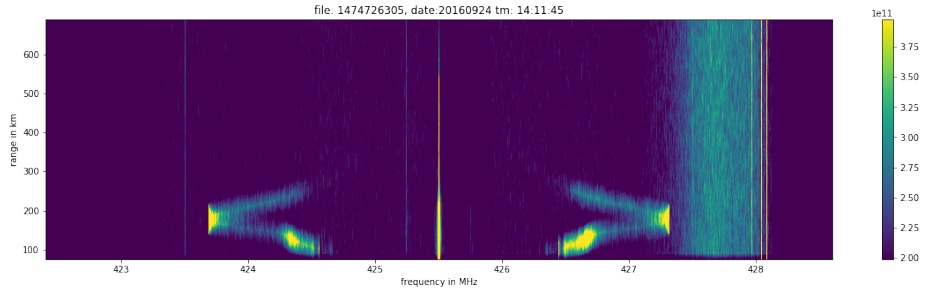


Figure 4.2: Process of spectrum generation.

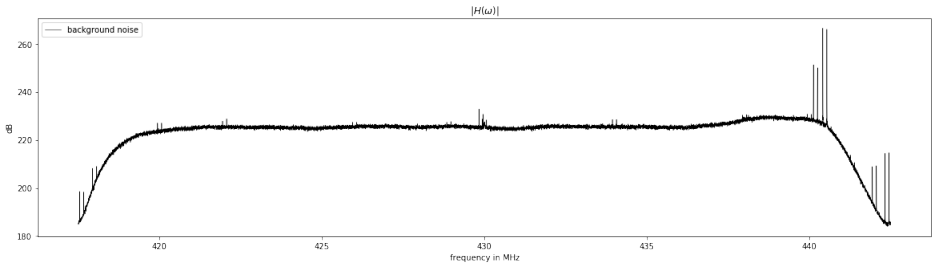
finite length time series will include a statistical estimation error with a standard deviation given by is $\delta S_m = S_m/\sqrt{I}$, if the estimate for S_m is taken as the arithmetic average of $\frac{|\tilde{V}_m|^2}{NT}$ computed with I independent and non-overlapping time series V_n using the notation introduced in Chapter 3. Alternatively, the estimate could be obtained as a result of "frequency smoothing" over I consecutive frequency bins of an $I \times N$-point $|\tilde{V}_m|^2$ (derived with a single FFT of a $I \times N$-point V_n time series) followed by decimation by a factor if I. Error standard deviation remains unchanged.

We will proceed here with our description of ISR spectrum estimation with

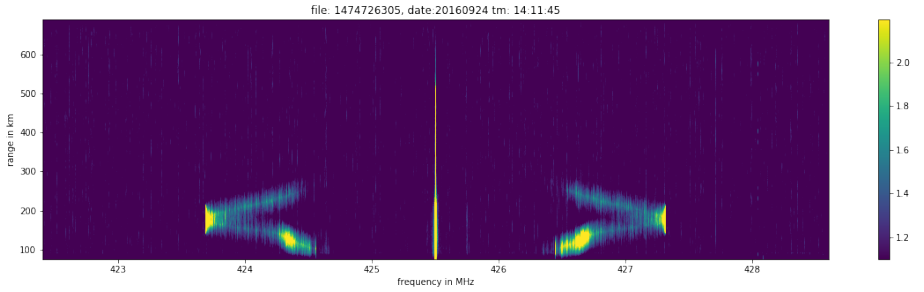
Arecibo USRP receiver data collected on 2016.09.23 from 14:11:45 to 14:11:55 LT in ULP mode where $N = 12500$ and $I = 500$ (with and I value of 50 per second of data).



(a) ULP spectrogram representing a stack of S_m estimates. The center straight line is ion-line while the curve lines are the up and down shifted plasma-lines. The altitude independent vertical band between 427.5 and 428 MHz is due to a bump in the system frequency response function



(b) Estimate of the system frequency response $H(\omega)$.



(c) ULP spectrogram $\hat{S}_m$ corrected for H_m distortions.

Figure 4.3: ULP spectrogram computation details using a batch value of 1024.

We neglect the first 12500 rows of data to avoid the cluttering below 75 km of lower altitudes such that the dimension of the matrix becomes 184108×16384. Let the spectrum S_m derived so far, by accumulating over $I = 500$ instances of the magnitude squares of the GPU computed FFTs, be samples of the ISR spectrum $\hat{S}(\omega)$ filtered through the USRP receiver frequency response function $H(\omega)$. Magnitude squared frequency response $|H(\omega)|^2$ is obtained from S_m of the high-

est altitudes in the computed spectrogram—the reason for this is the spectrum estimates from those heights are just the frequency independent background noise spectrum (sky noise plus receiver noise referred to as system noise in general) filtered by $H(\omega)$ and have the same shape as $|H(\omega)|^2$. Given $|H_m|^2$, the measured spectrum $\hat{S}_m$ is obtained as

$$\hat{S}_m = \frac{S_m}{|H_m|^2}. \tag{4.1}$$

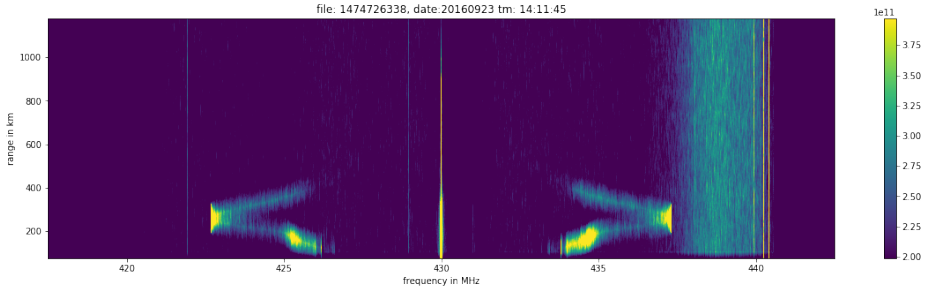
Figure 4.3(a) shows a filtered ULP spectrogram of S_m estimates, (b) the frequency response magnitude $|H_m|$ in dB derived from the top 100 altitudes of the spectrogram, and (c) the corrected spectrogram $\hat{S}_m$. The corrected spectrogram in Figure 4.3(c) shows three "lines", namely the central ion line feature of the ISR spectrum and additional up- and down-shifted plasma lines on two sides of the ion-line representing electron Langmuir waves traveling away from and towards the radar, respectively. This well formed ISR spectrogram derived from 10 s of collected raw data ($I = 500$) was computed in 761.18 seconds by using a batch value of $2^{10} = 1024$. Results of a test run conducted with batch values ranging from 2^{10} to 2^{14} are shown in Table 4.2. The table demonstrates that the batch value does not affect the time cost for the spectrum computation significantly—the overhead incurred by the threads will offset the time cost for the number of iterations.

A repeat of Figure 4.3 computed with a 16384 batch value is shown in Figure 4.4.

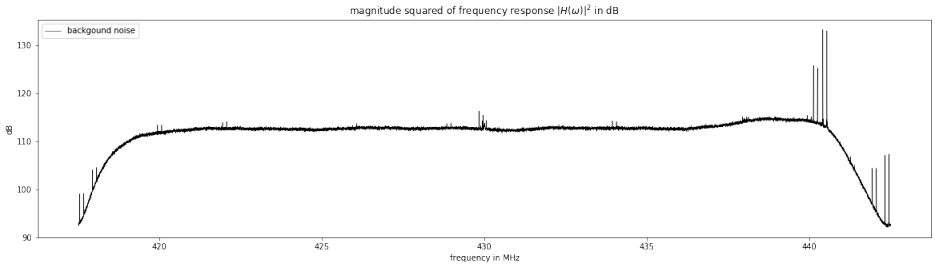
Table 4.2: Time cost in each batch value.

Batch Value	Number of Iteration	Time Cost [s]
1024	192	758.57
2048	96	755.87
4096	48	756.61
8192	24	758.57
16384	12	761.18

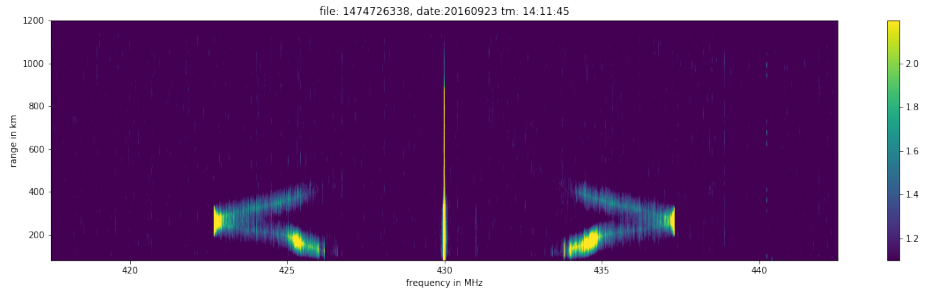
The ISR spectrograms shown in Figures 4.3 and 4.4 were computed using large data matrices of 184108×16384 dimensions and required computation times of more than 750 s, far exceeding the data collection time of 10 s. The resulting spectrogram matrices also had 184108×16384 dimensions and were impossible to display in full resolution in Matplotlib while full resolution displays in Bokeh (zoomable/expandable) were very slow to generate.



(a) ULP spectrogram representing a stack of S_m estimates. The center straight line is ion-line while the curve lines are the up and down shifted plasma-lines. The altitude independent vertical band between 427.5 and 428 MHz is due to a bump in the system frequency response function



(b) Estimate of the system frequency response $H(\omega)$.



(c) ULP spectrogram $\hat{S}_m$ corrected for H_m distortions.

Figure 4.4: ULP spectrogram computation details using a batch value of 16384.

Clearly operational needs demand faster and smaller scoped approximate methods for sprectrogram generation and display. We discuss such methods in the next section.

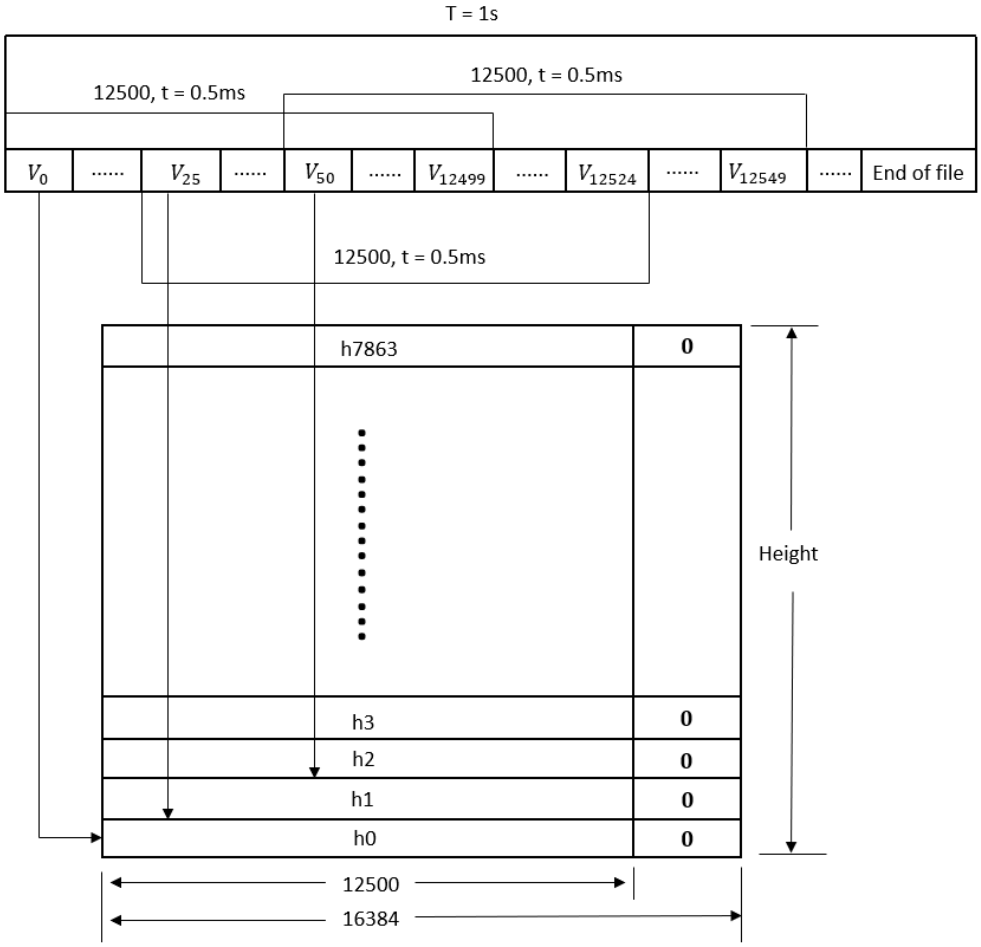
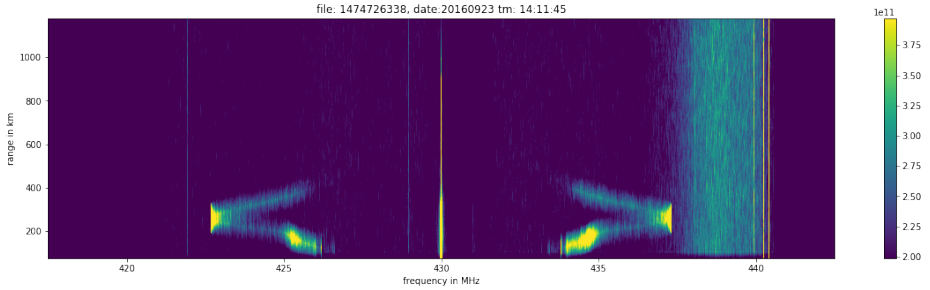


Figure 4.5: Processing FFT by skipping 25 samples between consecutive FFTs.

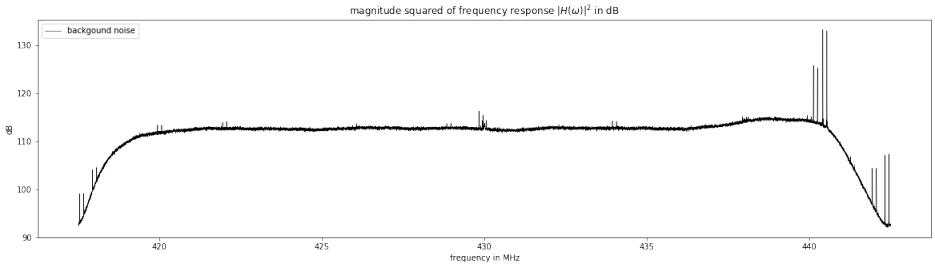
4.3 ULP Spectrogram Generation - Operational Method

In Arecibo ISR experiments 10 s of ULP mode of data is collected once every 30 s. In the last section we found out that spectrogram generation with ULP data collected in each 30 s period takes about 750 s of processing time using the GPU system mounted to the `impact1` server. Summarizing the workflow of the spectrograms shown in the previous section, GPU code operating on `impact1` server creates the raw data matrices needed for spectrogram generation from the mounted data volume served by `remote2`, and GPU based FFT results are accumulated by GPU code until the final spectrogram is passed back to `remote2` in form of an `*.np` file to be plotted in `remote2` using Matplotlib. It is possible to accelerate this process from 750 s to 30 s to match the effective collection time of the data underlying the computed spectrograms by forming 2-D data and spectrogram matrices using only "1-in-25" of the data rows included in matrix **V** depicted in Figure 4.2. That is, consider a reduced data matrix **V**' depicted in Figure 4.5 that has in its each row 12500-point time series (zero padded to 16384) consisting of samples taken at $T = 40$ ns intervals starting at row n with voltage sample V_{25*n}. Taking the FFT of this matrix in the GPU and accumulating the squared FFT's as a spectrogram over 10 s of ULP data collection taken within each 30 s time intervals we obtain a 30 s spectrogram after 30 s of computation in the GPU. The resulting `*.np` file (of 25 times smaller size) is then passed back to `remote2` and spectrogram plots such as those shown in Figure 4.6 are generated with Matplotlib.

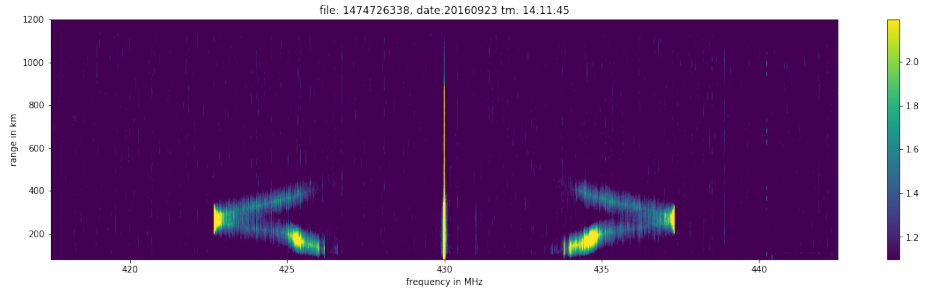
Visually, Figure 4.6, obtained after 30 s of computations looks the same as Figure 4.4 obtained after about 750 s of computations. Full resolution an accelerated spectrogram results are re-potted in Figure 4.7 for easy comparisons. Notice that in these plots we depict 2-D matrices of different sizes with equal image sizes—the equalization is done automatically by the Matplotlib `imshow` command with its default call parameters. A more direct and controlled comparison of the full resolution and accelerated spectrogram calculations is presented in Figure 4.8. Here the top row depicts the down- and up-shifted plasma line features of the accelerated spectrogram. The same features are depicted in the middle row derived from averaging and decimating the full resolution result to match the reduced dimensions of the accelerated result. Finally the bottom row shows the ratio of the results displayed in top and middle rows in dB to detect, if any, significant



(a) ULP spectrogram representing a stack of S_m estimates made using the operational approach.



(b) Estimate of the system frequency response $H(\omega)$ based on 364 top heights.



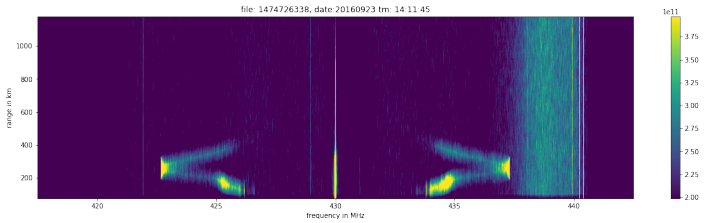
(c) ULP spectrogram $\hat{S}_m$ corrected for H_m distortions.

Figure 4.6: ULP spectrogram computation details using the accelerated procedure.

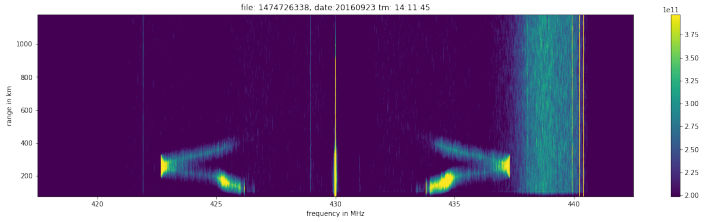
differences attained using the two approaches. Clearly we detect no significant differences between the outcomes of the two approaches.

4.4 CLP Spectrograms

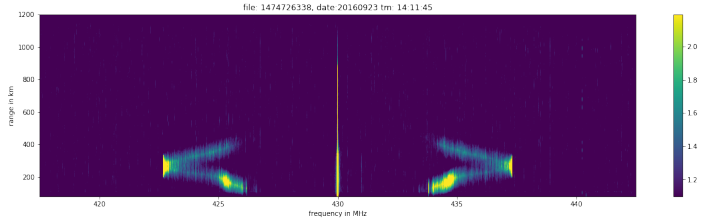
In Arecibo CLP implementation the transmitted pulse is 0.44 ms and 220 baud long and corresponds to 11000 samples taken at $T = 40$ ns intervals at the output of the USRP receiver. The IPP is 10 ms and the number of pulses recorded in each 1-second file is 100. As discussed in Section 4.3, we create a 7864×16384



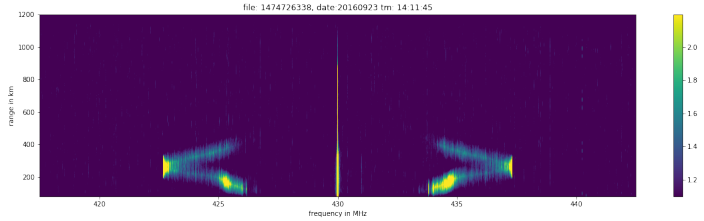
(a) ULP spectrogram representing a stack of S_m estimates made using the operational approach.



(b) ULP spectrogram representing a stack of S_m estimates made using the full resolution approach.

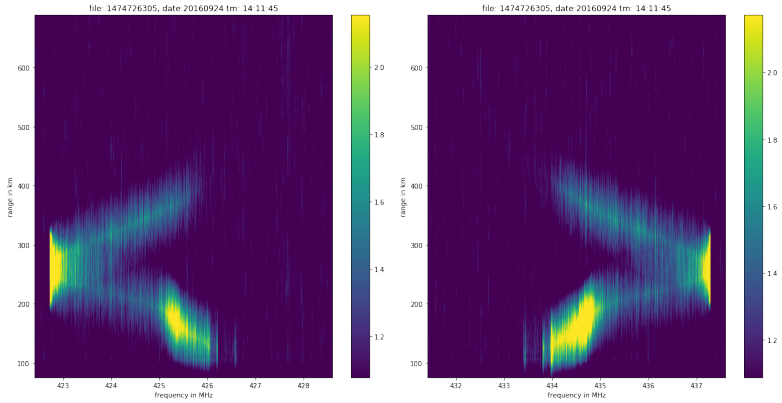


(c) ULP spectrogram $\hat{S}_m$ corrected for H_m distortions obtained with the accelerated approach.



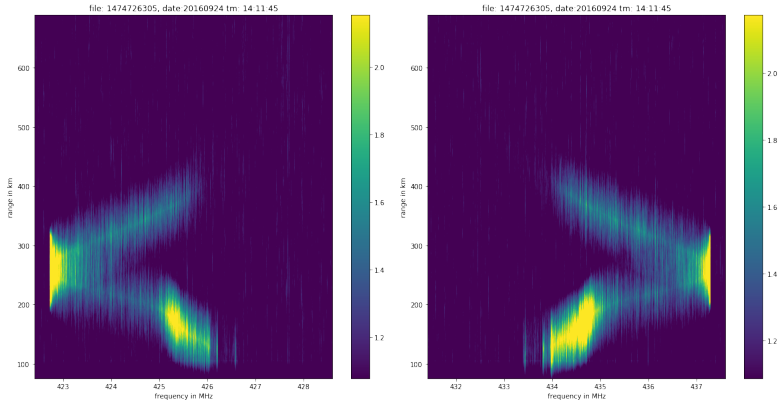
(d) ULP spectrogram $\hat{S}_m$ corrected for H_m distortions obtained with the full resolution approach.

Figure 4.7: ULP IS spectrogram comparison between accelerated and full resolution approaches.



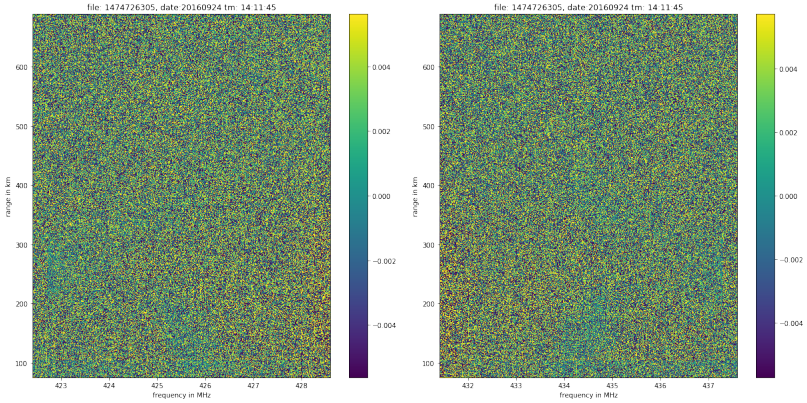
(a) Down-shifted plasma line spectrogram using accelerated procedure.

(b) Up-shifted plasma line spectrogram using accelerated procedure.



(c) Down-shifted plasma line with block average of full spectrogram.

(d) Up-shifted plasma line with block average of full spectrogram.



(e) Down-shifted plasma line ratio in dB.

(f) Up-shifted plasma line ratio in dB.

Figure 4.8: ULP IS plasma line spectrogram comparison between accelerated and full resolution approaches.

matrix for accelerated spectrogram generation as shown in Figure 4.9 after zero padding 11000 samples of data on each row to a 16384 length. The matrix is decoded by conjugate multiplying each row with the complex conjugate of the first row, a direct recording of the 220-baud pseudo-random binary +1/-1 sequence multiplying a rectangular pulse of 0.44 ms duration. Decoded matrix is FFT'ed in the GPU and its magnitude square is accumulated as the accelerated spectrogram as before. For frequency response correction we use $|H_m|^2$ values derived from the previous ULP run. An example of CLP spectrogram computed using the accelerated approach just described is shown in Figure 4.10.

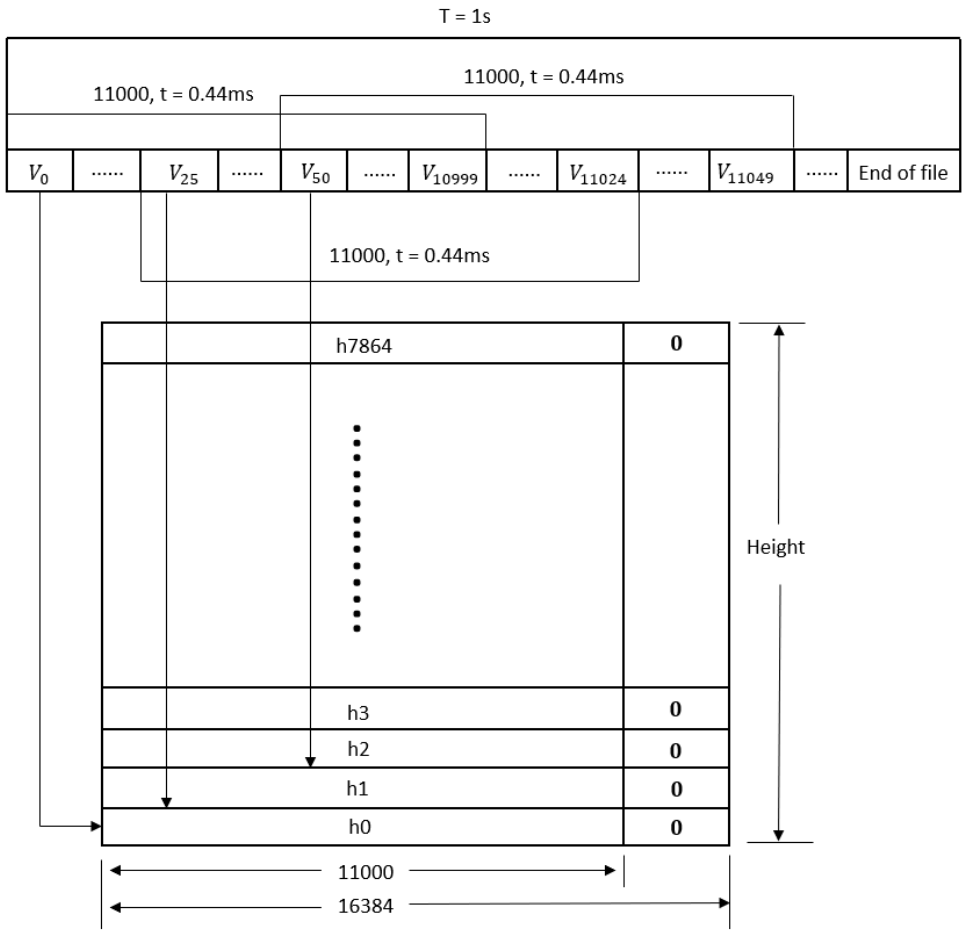
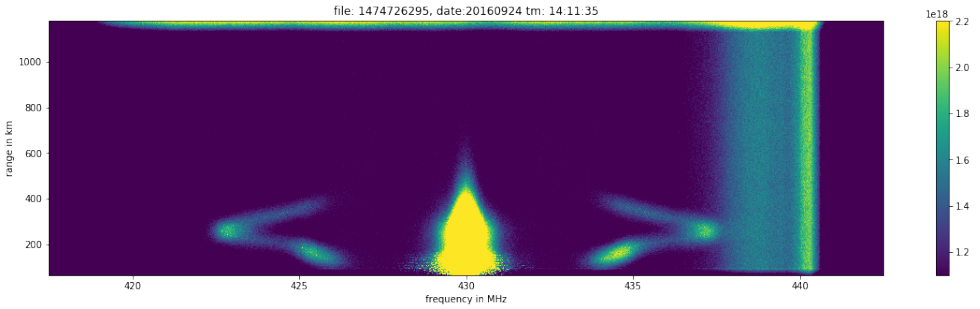
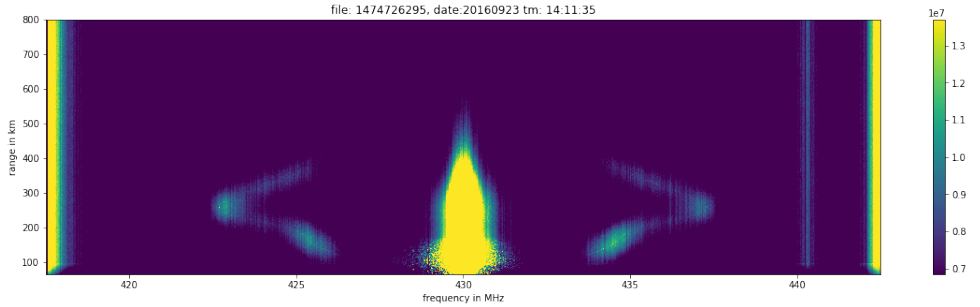


Figure 4.9: CLP plasma line generation.



(a) CLP spectrogram representing a stack of S_m estimates. The center straight line is ion-line while the curve lines are the up and down shifted plasma-lines. The altitude independent vertical band between 427.5 and 428 MHz is due to a bump in the system frequency response function.



(b) CLP spectrogram $\hat{S}_m$ corrected for H_m distortions.

Figure 4.10: CLP spectrogram computation details.

4.5 Long ULP Spectrograms

In LULP mode the transmitted pulse is 1 ms long and corresponds to 25000 samples taken at $T = 40$ ns intervals at the output of the USRP receiver. In Figure 4.11, the first 25000 samples of USRP output data V_0 to $V_{25000-1}$ are put into the first row of the matrix $\mathbf{A}$ to generate height h_0. In the accelerated approach the data matrix is filled as shown in Figure 4.11 by starting each row n with sample V_{25*n} and zero padding each row to 32768 length. The rest of processing details to obtain spectrograms is identical to the accelerated CLP method described earlier. An example of LULP spectrogram computed using the accelerated approach is shown in Figure 4.12 corresponding to LULP data recorded on 2016.09.23 from 14:11:55 to 14:12:05.

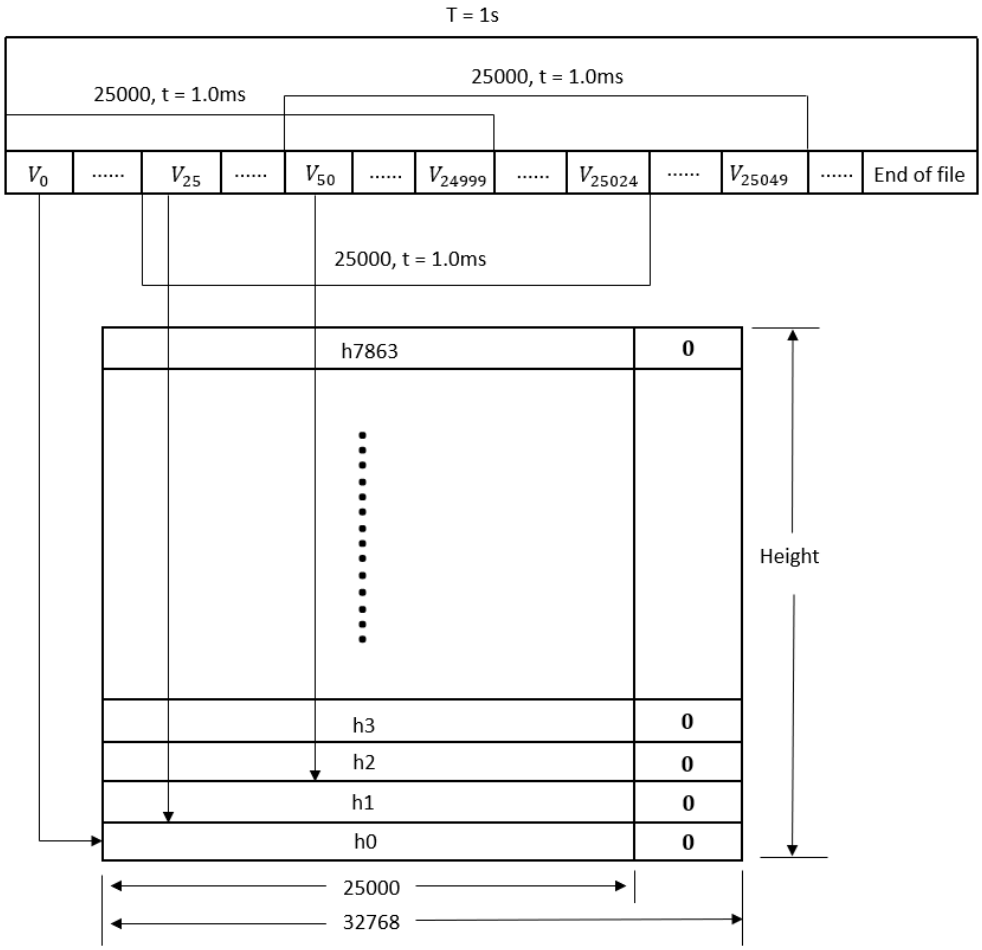
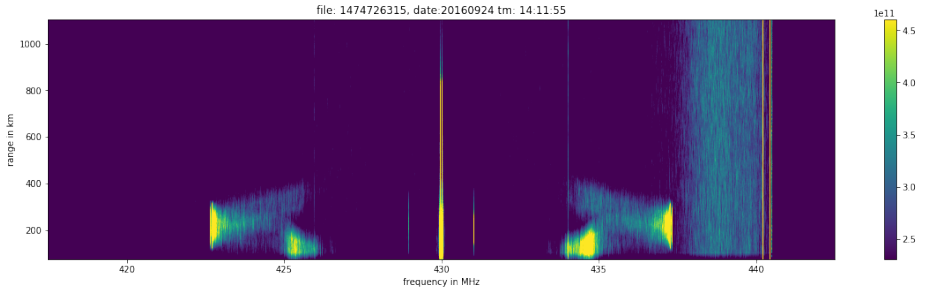
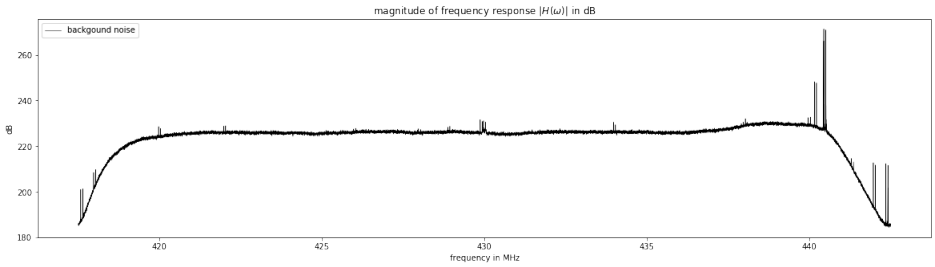


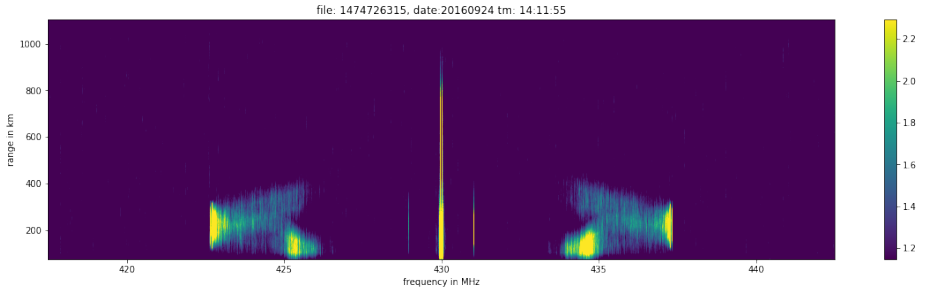
Figure 4.11: Long ULP plasma line generation.



(a) LULP spectrogram representing a stack of S_m estimates. The center straight line is ion-line while the curve lines are the up and down shifted plasma-lines. The altitude independent vertical band between 427.5 and 428 MHz is due to a bump in the system frequency response function.



(b) Estimate of the system frequency response $H(\omega)$.



(c) LULP spectrogram $\hat{S}_m$ corrected for H_m distortions.

Figure 4.12: LULP spectrogram computation details using the accelerated procedure.

Table 4.3 summarizes the time cost for each type of the transmitted radar pulse when using the accelerated operational method discussed in the last three sections.

Table 4.3: Time cost of spectrogram generation from 10 s of ULP, CLP, and LULP data records

Pulse Type	Batch Value	Time Cost [s]
Uncoded Long Pulse	7864	26.83
Coded Long Pulse	7864	63.23
Long Uncoded Long Pulse	7864	85.02

CHAPTER 5

SPECTRUM DECONVOLUTION AND PLASMA-LINE DERIVATION

5.1 Spectral Map Convolutional Distortion

Let p_0, p_1, p_2, p_3, p_4 denote the time-series voltage of a 5-baud radar pulse transmitted to the ionosphere, and V_0, V_1, V_2, V_3, V_4 represent sampled voltages at the receiver scattered from some ionospheric heights as shown in Figure 5.1. Notice that the V_n sequence consists of not only the five "blocks" of back-scattered signal from height h_0 as indicated in the figure, but also from h_1 to h_4 as well as h_{-4} to h_{-1}. This leads to a convolutional distortion where signal contributions from many subvolumes of the ionosphere are mixed. In the actual ULP mode discussed in Chapter 4, 12500 back-scattered signal samples of each row of the data matrix $\mathbf{V}$ consist of the echoes from subvolumes at heights h_{-12499} to h_{12499}. In order to recover the true sampled back-scattered radar echoes from height h_0, a deconvolution algorithm "CLEAN" must be utilized as will be discussed in this chapter.

5.2 Clean Algorithm

The CLEAN algorithm was first introduced and described by *Högbom* [1974]. CLEAN is a deconvolution algorithm to reconstruct images recorded in radio astronomy. This image restoration method is a non-linear operation, and the algorithm is based upon the fact that the image to be deconvolved is itself a convolution of a scene with multiple point sources with a point spread function (PSF) descriptive of the observing instrument. The CLEAN algorithm has been discussed in detail [*Rich et al.*, 2008] and it works as follows:

1. Finding the location of the maximum absolute brightness point source in the "dirty map" ;

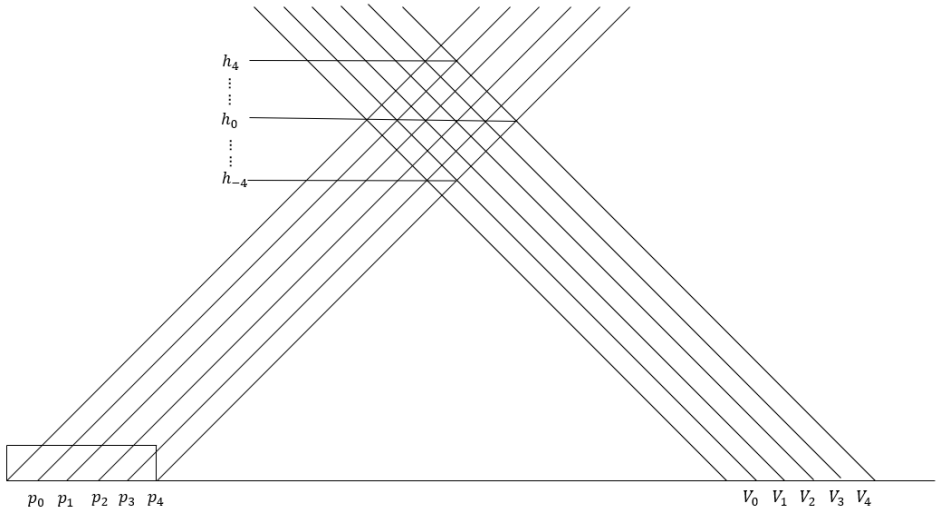


Figure 5.1: Representation of ULP radar transmission and reception.

2. Multiplication of the strength of this point source with a gain factor to generate a "CLEAN component" at this location;

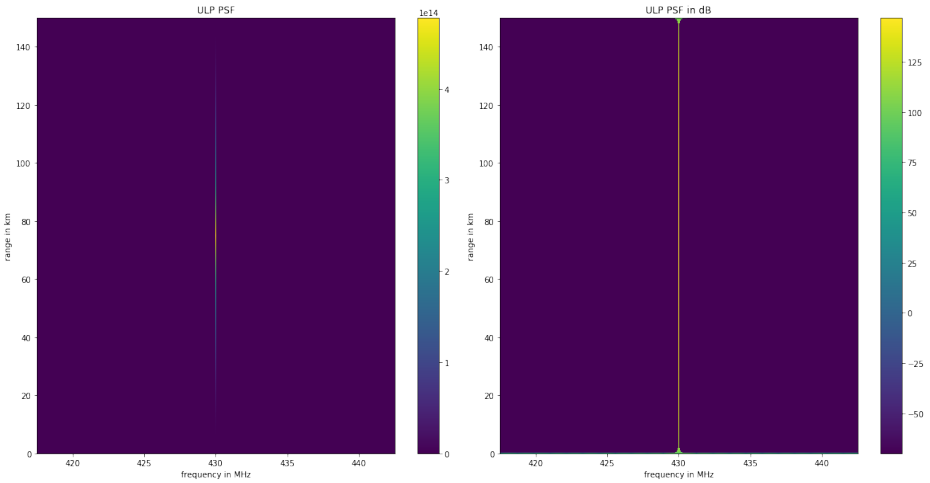
3. Convolution of the CLEAN component with the "dirty beam", and subtraction this from the dirty map, recording the position and strength of the CLEAN component subtracted;

4. Repeating step 1, 2 and 3 on the dirty map until all emission is found, or a certain flux threshold is reached, or a number of iterations has been achieved;

5. Adding the subtracted map to the CLEAN image.

As we discussed in Section 5.1, the received sampled voltages consist of radar echoes from multiple heights. Height h_0 is located at the center of the tilted 5×5 square. As a result, we create a matrix in which the transmission pulse is at the center. By going from the center to the top and bottom of the matrix each time, the rectangular pulse is multiplied with a shifted version such that the total length of the product is reduced by one as described in Figure 5.1. Then we will get a 25000×12500 matrix $\mathbf{P}$. By extending the matrix $\mathbf{P}$ by zero-padding to 25000×16384, we will implement FFT on each row of the matrix to derive the ULP PSF. After the 25000×16384 matrix is created, the resulting PSF is compressed in `*.npz` file and passed back to `remote2`, and the spectrogram is

shown in Figure 5.2. The transmitted ULP pulse is a rectangular function whose Fourier transform pair is a sinc function. A broader band of rectangular function will result in a narrower sinc function with stronger peak. As a result, the PSF is 'I' shaped and its maximum magnitude is located at the center of the image.



(a) ULP PSF in magnitude.

(b) ULP PSF in dB. We add 10^{-7} bias on the PSF spectrogram to display it in logarithmic scale.

Figure 5.2: Full spectrogram of ULP PSF.

5.3 ULP Spectrogram Deconvolution

After the point spread function is derived, the true ISR spectrogram could be recovered from the convolutional distortion by using the CLEAN algorithm. The spectrogram to deconvolve is the full spectrogram of 10-second integration on 2016.09.24 from 14:11:45 to 14:11:55. In this section, we will focus on the up-shifted plasma line region and down-shifted counterpart respectively. The "dirty images" are the two separate normalized plasma line regions and the "dirty beam" is the normalized PSF. Figure 5.3 shows the idealized CLEAN beam which is 2D Gaussian distribution. By modifying standard deviation along height and frequency respectively, we observe that the resulting cleaned spectrogram is most reasonable when $\sigma_{frequency} = 20$, and $\sigma_{height} = 10$. The full spectrogram deconvolution requires a significant amount of computing resources to process. We focus on the height of the spectrogram region from 165 km to 345 km. The gain step size

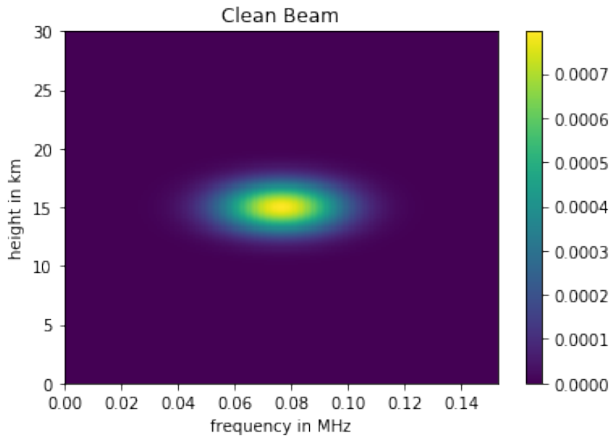
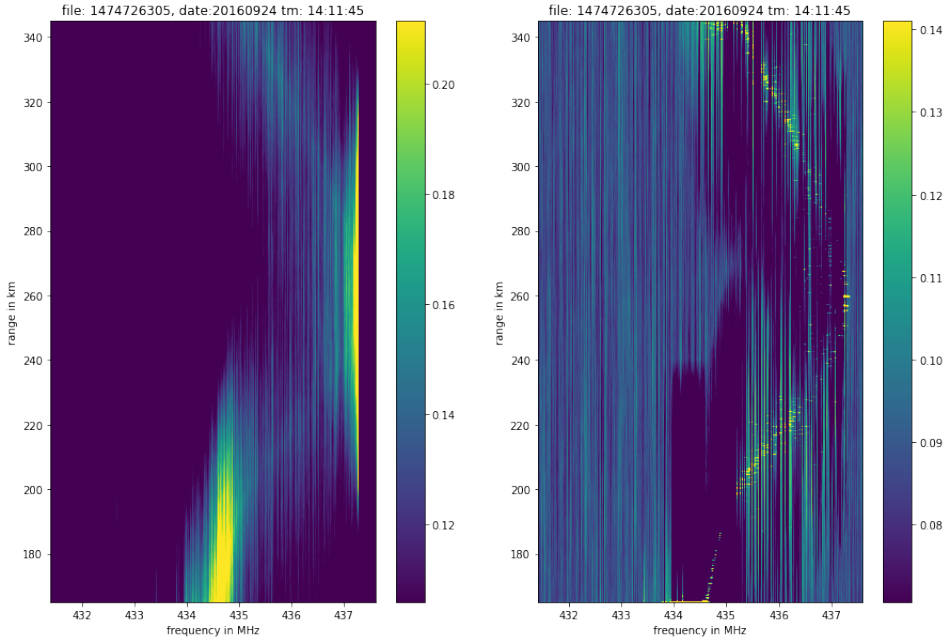


Figure 5.3: 2D Gaussian CLEAN beam.

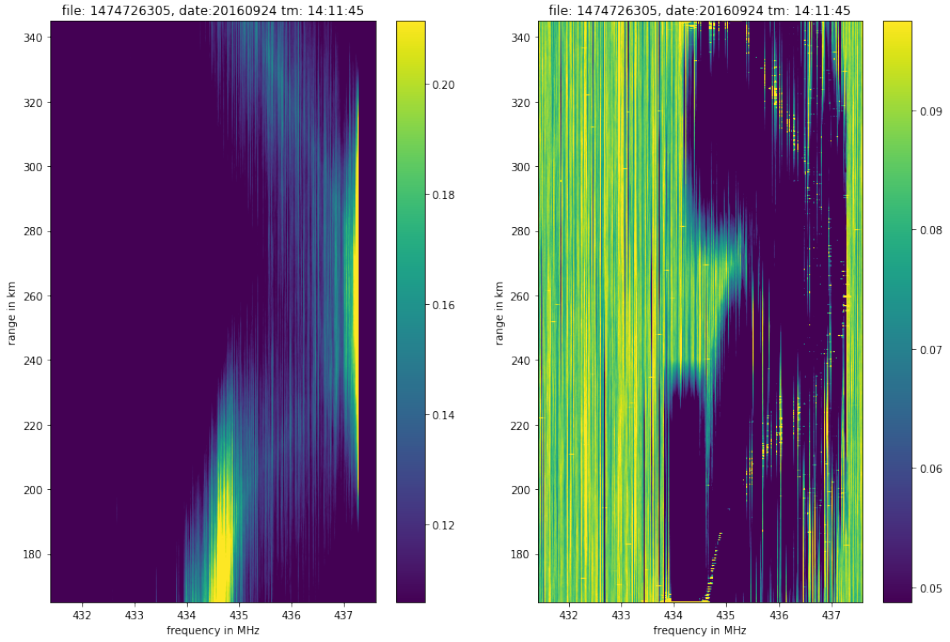
is 0.2 and the resulting spectrogram is the deconvolution of a 30000×4096 matrix and a 25000×16384 matrix. Figure 5.4 represents the up-shifted CLEANed plasma line spectrogram after the deconvolution. Visually, the CLEANed plasma line could be extracted from the distorted spectrogram when we keep iterating the CLEAN algorithm until it converges, which implies that the CLEAN algorithm is working on the full spectrogram. However, we could improve the performance of the CLEAN algorithm by modifying the gain value, which we will discuss in the next section. The 2000 iterations of cleaning require 47.88 min, and the 3207 iterations for convergence require 96.19 min. For a 30 s data collection, the data process requires over 95 min. When the height of the "dirty beam" is from 75 km to 450 km and we decrease the step size to 0.05 for better CLEANed spectrogram, the CLEAN algorithm will take an even longer time to derive the plasma line spectrogram.

However, it is possible to accelerate the CLEAN process by deconvolving the spectrogram matrices using only "1-in-25" of the data rows included in the full spectrogram matrices described in the last section. Then, the PSF is block-averaged by 25 rows such that the dimension of PSF matrices is 1000×16384, and the total data size is reduced by a factor of 25. Figure 5.5 shows the averaged PSF, and Figure 5.6 illustrates the CLEANed spectrogram with the same gain size 0.2. The height of the "dirty image" is from 75 km to 689.4 km, and the time cost for generating the two plasma line spectrograms is 6.94 min.

By visually inspecting the resulting spectrogram, we observe that a nose-shaped plasma line is extracted from the convolutional distorted spectrogram. As a result,



(a) CLEANed spectrogram after 2000 iterations.



(b) CLEANed spectrogram after 3207 iterations. The CLEAN algorithm converges.

Figure 5.4: CLEANed spectrogram from 165 km to 345 km with gain step size 0.2.

the CLEAN algorithm by using accelerated procedure works well in the spectral measurement. However, there is some information missing from 423 MHz to 423.5 MHz. According to the algorithm discussed in Section 5.2, we can change

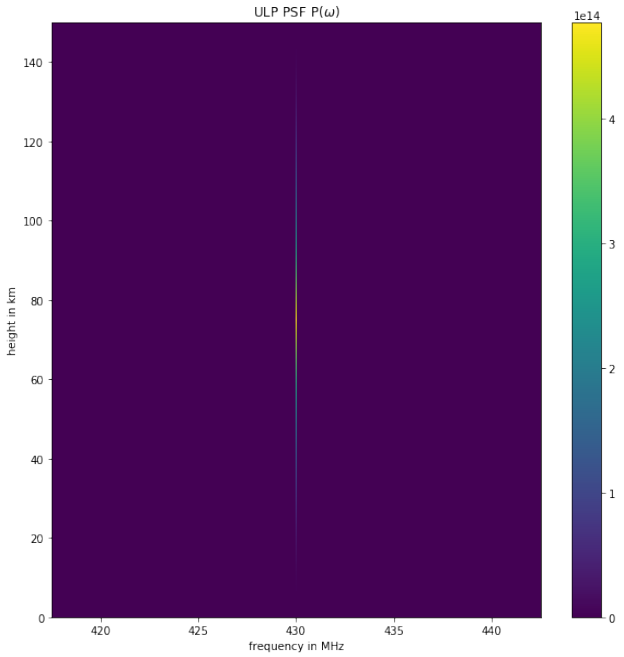


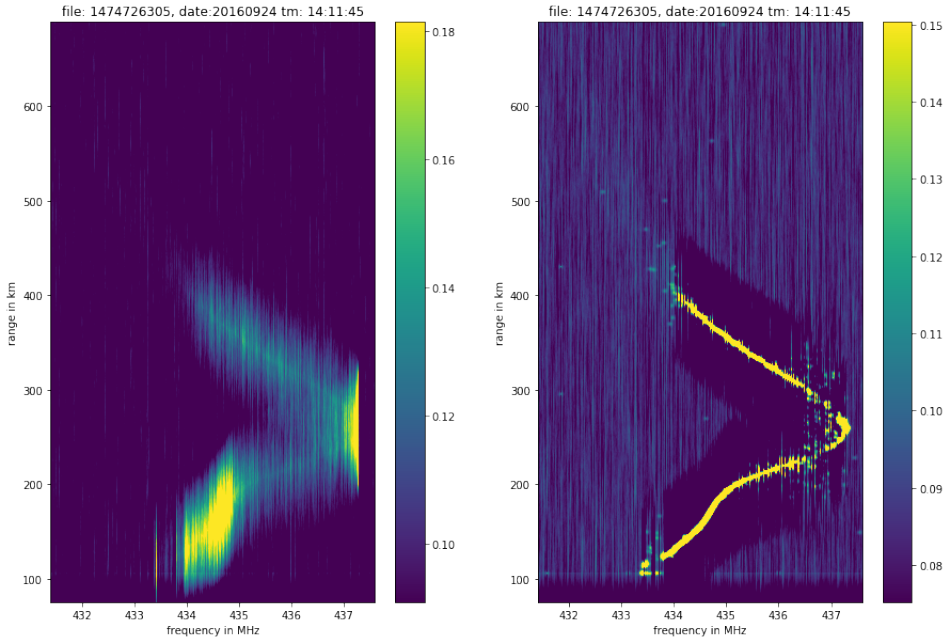
Figure 5.5: ULP PSF by using accelerated procedure.

the step size of gain to modify the CLEAN algorithm. By running the accelerated deconvolution procedure, we will investigate the resulting spectrogram with gain step 0.05.

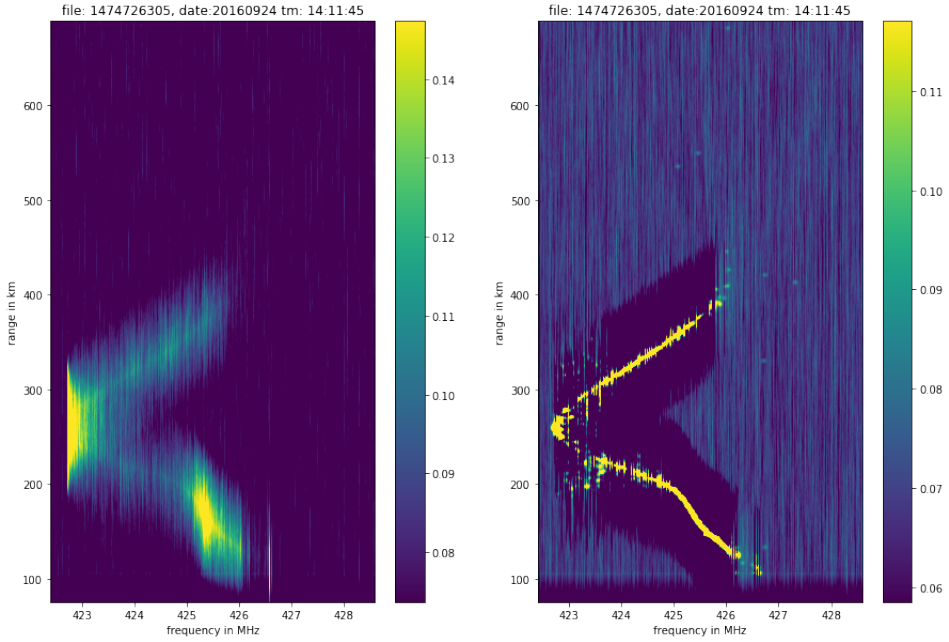
Figure 5.7 represents the CLEANed plasma line with the gain step 0.05. The resulting plasma lines are successfully derived from the distorted spectrogram, which justifies our CLEAN algorithm. The time cost for the two plasma line derivation is 7.29 min.

In order to estimate the plasma line frequency, the background noise from the spectrogram is required to be removed. We choose a range of highest altitudes' response as the background noise, and the noise of up-shifted plasma line and down-shifted counterpart is shown in Figure 5.8.

By subtracting the background noise from the deconvolved plasma line spectrogram, we could derive the true plasma line spectrogram for estimation. Figure 5.9 shows the plasma line spectrogram with background noise removed.

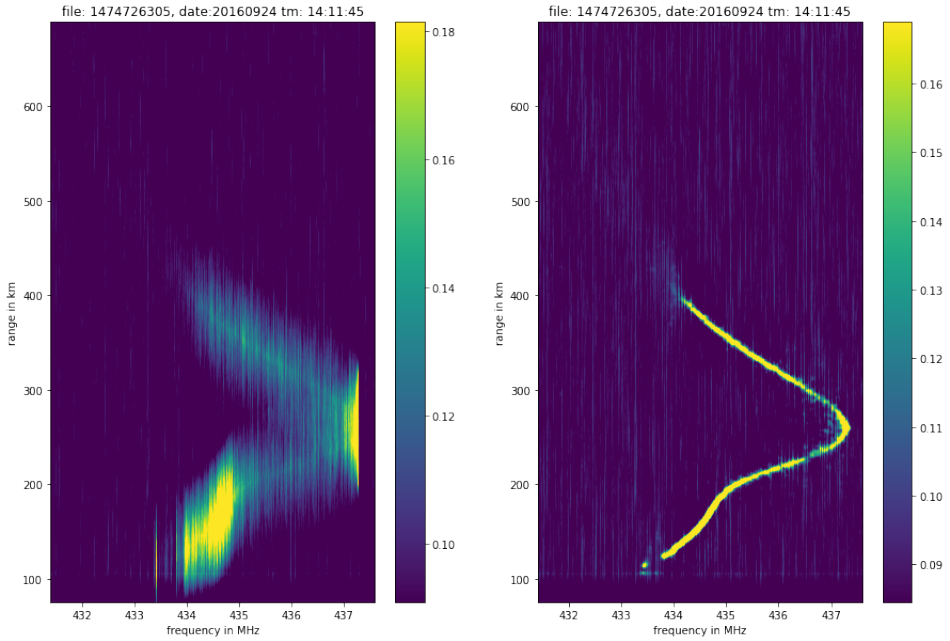


(a) The left figure is distorted up-shifted plasma line spectrogram and right figure is recovered up-shifted plasma line.

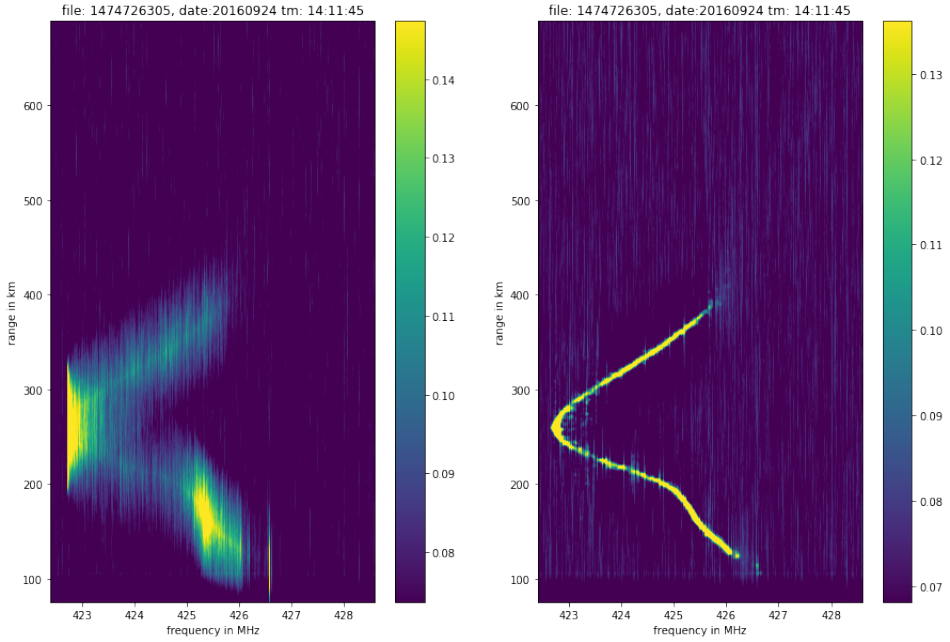


(b) The left figure is distorted down-shifted plasma line spectrogram and right figure is recovered down-shifted plasma line.

Figure 5.6: Cleaned plasma line spectrogram with gain step 0.2.

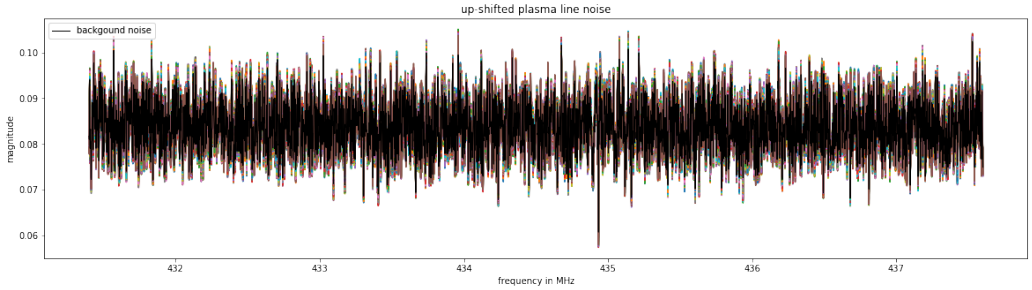


(a) The left figure is distorted up-shifted plasma line spectrogram and right figure is the recovered up-shifted plasma line.

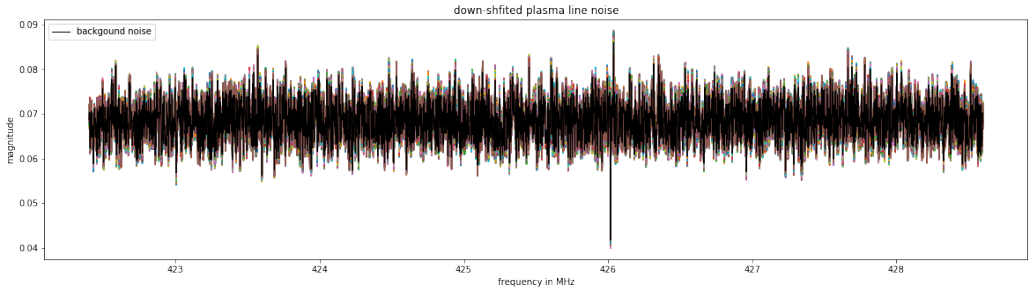


(b) The left figure is distorted down-shifted plasma line spectrogram and right figure is recovered down-shifted plasma line.

Figure 5.7: Restored plasma line spectrogram with gain step 0.05.



(a) Background noise of up-shifted plasma line spectrogram.



(b) Background noise of down-shifted plasma line spectrogram.

Figure 5.8: Background noise of CLEANed plasma line spectrogram.

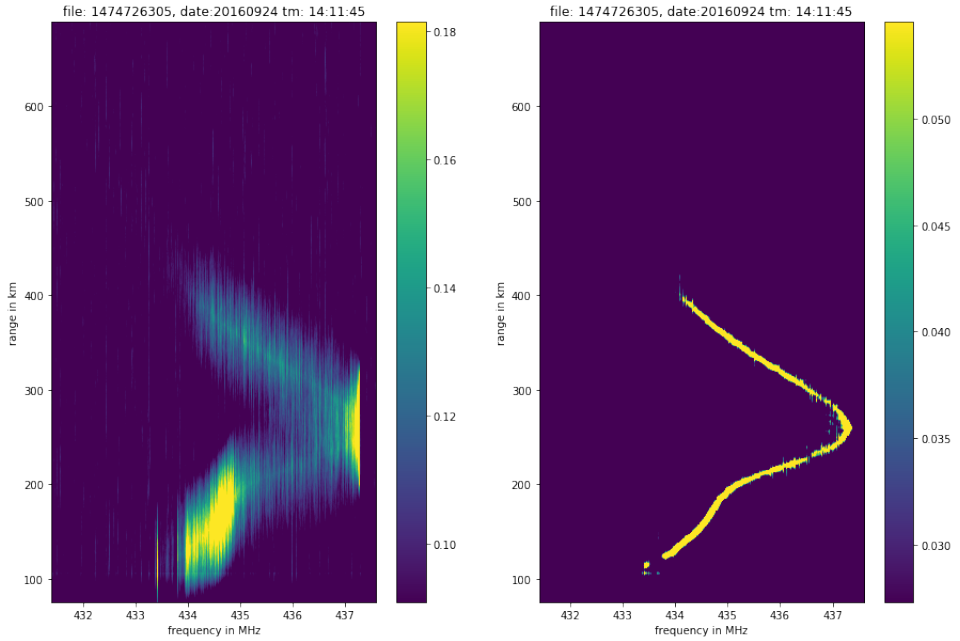
5.4 ULP Plasma Line Frequency Estimation

After we restore the true plasma line spectrogram from the convolutional distortion, we could estimate the plasma line frequency at each altitude, in which we could approximate the power spectrum with either a Gaussian distribution model or a *argmax* function. By running the plasma frequency estimation test with these two methods, we will inspect the plasma frequency prediction and the time cost for each method.

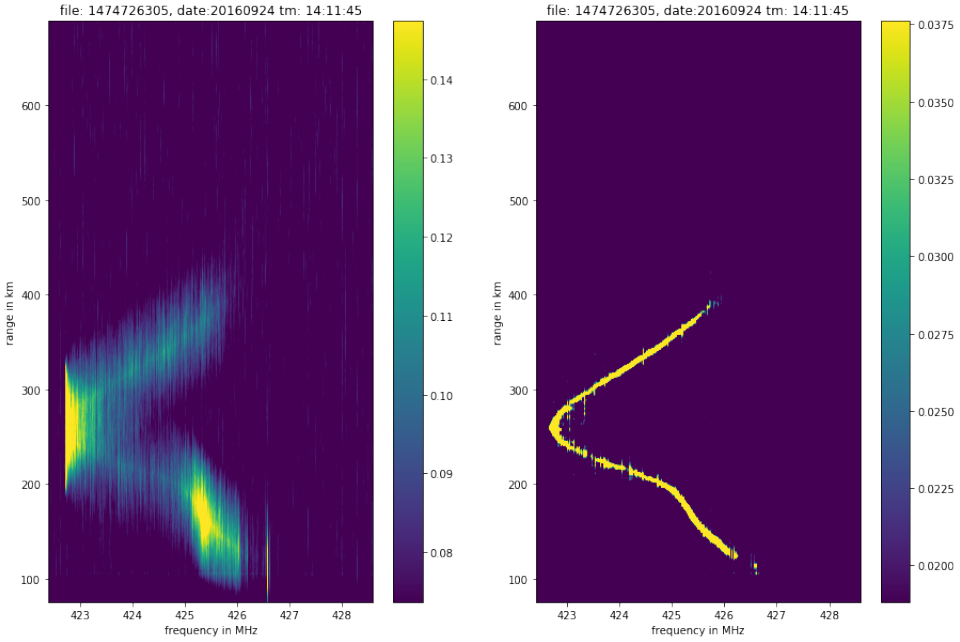
- Gaussian Fitting
  By fitting the mean μ and the standard deviation σ, we could obtain the plasma frequency at each height with a particular uncertainty. In this section, we use least-square package to optimize the parameters μ and σ such that the mean-square error is minimized:

$$\underset{\mu,\sigma}{\operatorname{argmin}} \sum_{i=1}^{N} (y^{(i)} - f(x^{(i)}, \mu, \sigma))^2, \tag{5.1}$$

  where $y^{(i)}$ is the measured spectrum and $f(x, \mu, \sigma)$ is the fitted spectrum. In this section, we choose the spectra at height 200 km, 250 km and 300 km



(a) The left figure is distorted up-shifted plasma line spectrogram and right figure is up-shifted plasma line spectrogram with noise removed.



(b) The left figure is distorted down-shifted plasma line spectrogram and right figure is down-shifted plasma line spectrogram with noise removed.

Figure 5.9: Up-shifted and down-shifted plasma line spectrograms.

respectively to illustrate the process to find the plasma frequency.

By inspecting the power spectrum of these three heights as shown in Figures 5.10 and 5.11, we obtain the measured data $y^{(i)}$. Then we use a Gaussian distribution to optimize the two parameters in the model. By minimizing the χ^2, the optimal parameters μ and σ could be derived, where μ represents the estimated plasma frequency and σ denotes the uncertainty in the estimated plasma frequency. In addition, the time cost to estimate the plasma frequency from 75 km to 450 km is 62 seconds for Gaussian fitting for one plasma line spectrogram. Then the total time cost will be around 132 seconds.

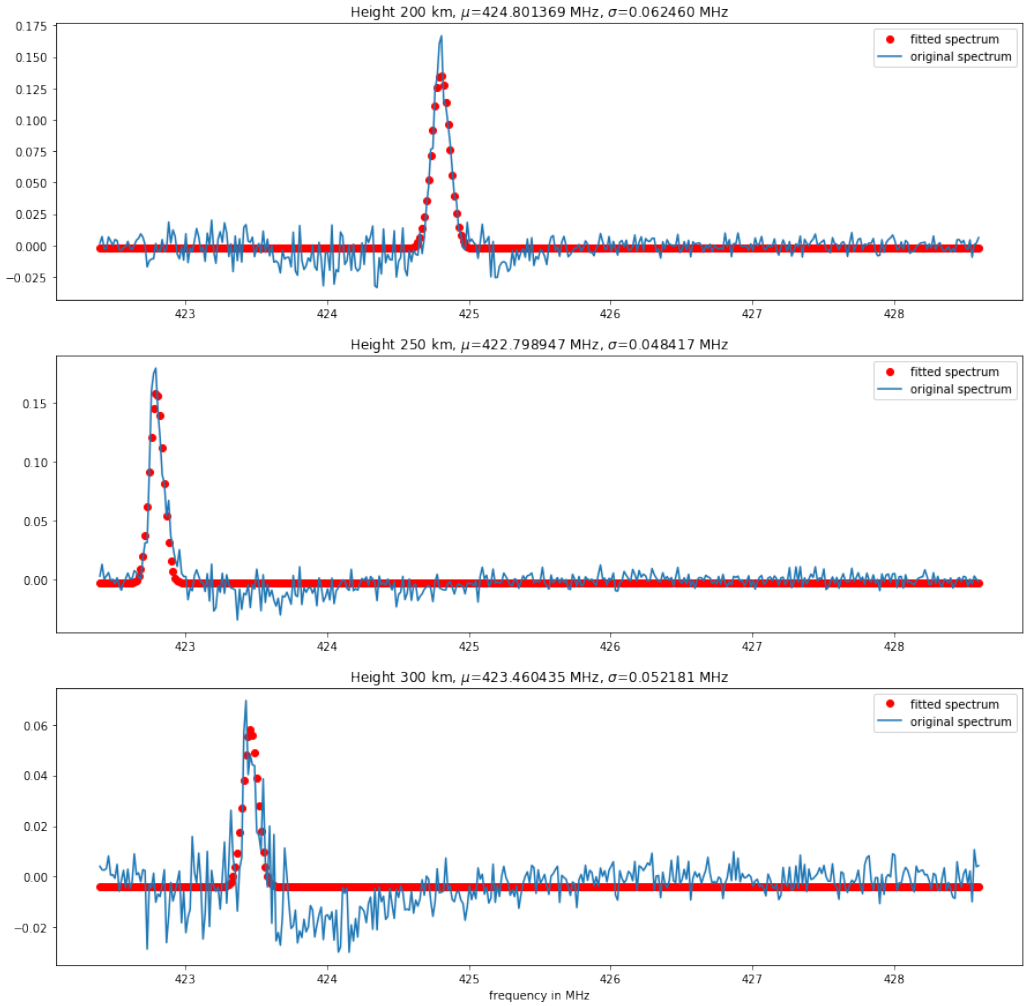


Figure 5.10: Down-shifted plasma line fitting by using Gaussian fitting at height 200km, 250km and 300km.

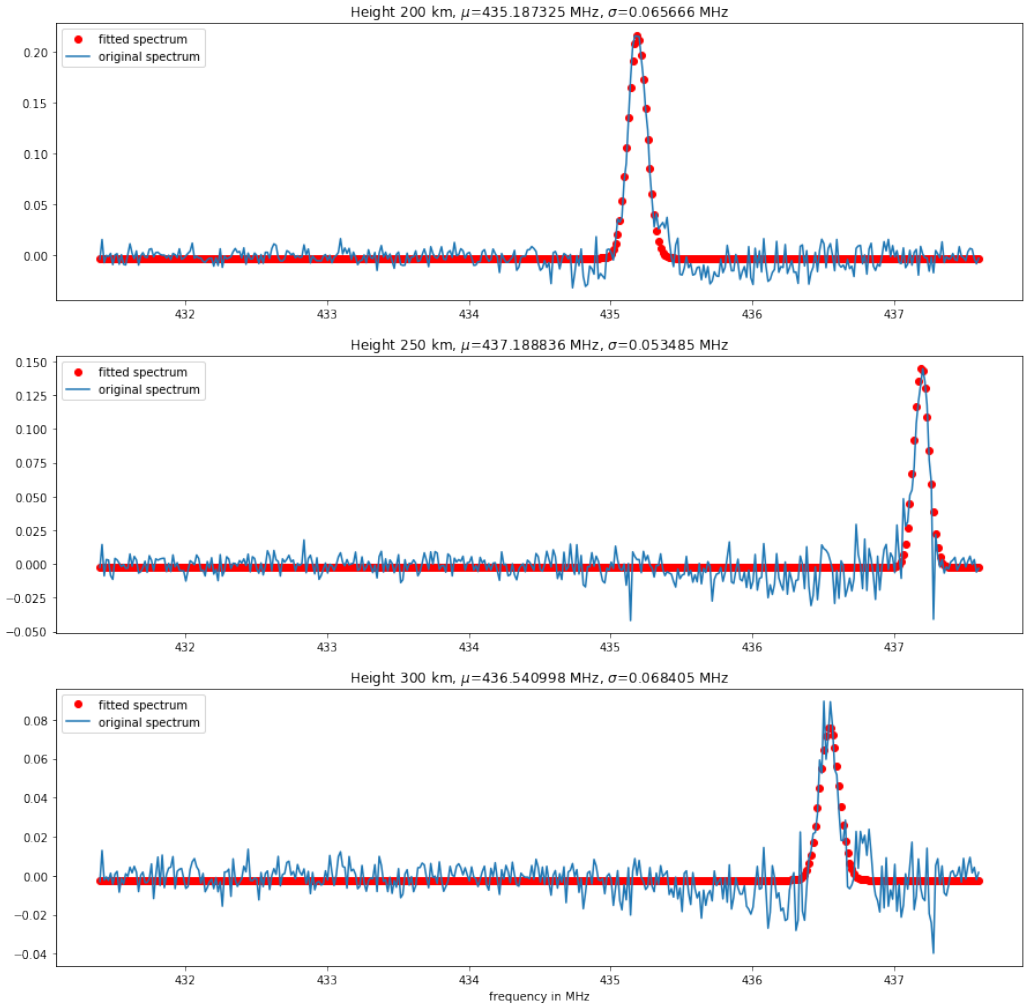


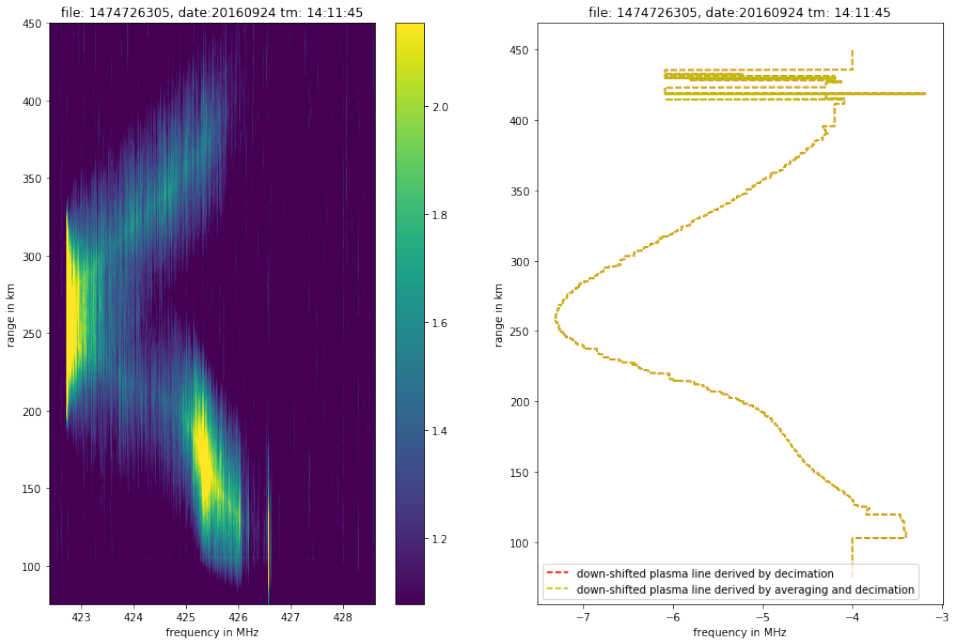
Figure 5.11: Up-shifted plasma line fitting by using Gaussian fitting at height 200km, 250km and 300km.

- *Argmax Function*

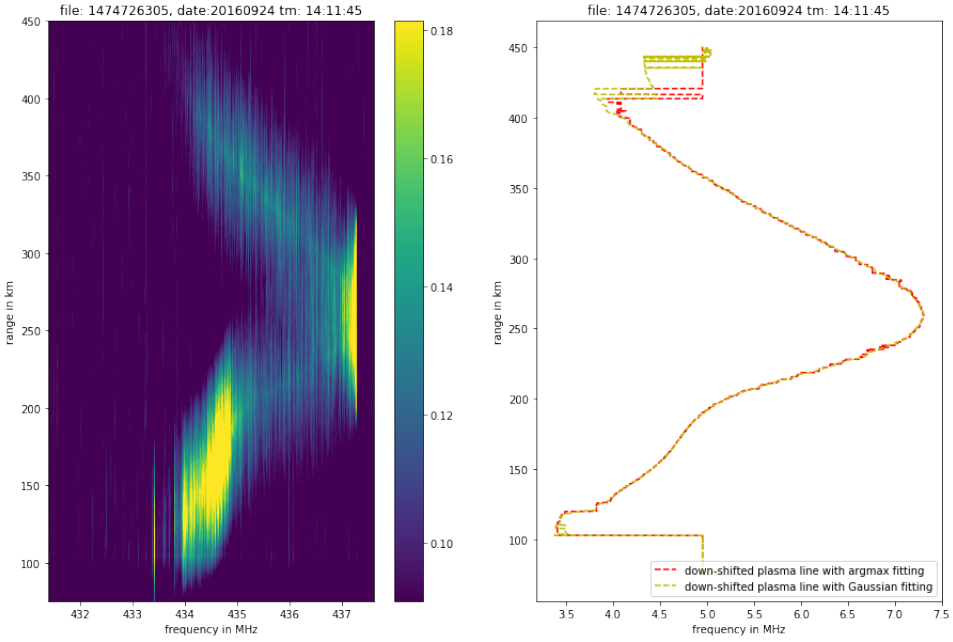
  The *argmax* function will track the frequency which has the maximum power magnitude. In each height, this frequency is the plasma line frequency. The time cost for argmax method is 0.04 seconds.

By comparing the two fitting results with the original distorted plasma line spectrogram, we conclude that the two fitting results are very close, as shown in Figure 5.12. However, since argmax function will save more time than Gaussian fitting, we will implement the argmax function in further spectral analysis.

For a 30 s data collection, the Gaussian distribution fitting will require over 2 min to obtain the plasma frequency, while the argmax function only requires 0.04



(a) Fitting down-shifted plasma line by using Gaussian fitting and argmax fitting.



(b) Fitting up-shifted plasma line by using Gaussian fitting and argmax fitting.

Figure 5.12: Fitted plasma line spectrogram by using Gaussian distribution and argmax function.

s to generate the plasma line. By visually inspecting the fitted plasma line by using above two approaches, the resulting plasma lines are very close to each other. As a result, we choose the argmax function to derive the plasma line frequency in our future spectral measurement.

5.5 Spline Fit

So far we have derived the plasma frequency by fitting the power spectrum of each height independently. However, at a given time instant, the plasma frequency should be continuous across the heights. It turns out that when fitting the power spectrum independently, there will be oscillation between consecutive points along the curve. As a result, in order to guarantee the continuity and smoothness of the plasma line spectrum, the spline interpolation is introduced. The spline interpolation is a type of interpolation which uses piece-wise polynomials to approximate the original data. By implementing the spline interpolation, the plasma line spectrum is more smooth as shown in Figure 5.13. We then compare the spline interpolation of the plasma line with the plasma line without deconvolution, and we have Figure 5.14. We have derived the ISR plasma frequency from the raw data received from Arecibo Observatory ULP measurements. Figure 5.15 demonstrates each step to generate the plasma line spectrogram and determine the plasma frequency.

5.6 CLP Spectrogram Deconvolution

Based upon Figure 5.15, we follow each step to generate the plasma line from the raw data received from the Arecibo Observatory CLP spectral measurements.

- Generate ISR CLP spectrogram and obtain the corrected CLP spectrogram for USRP receiver $H(\omega)$ described in Section 4.4.
- Use accelerated CLEAN algorithm to deconvolve the plasma line spectrogram from the convolutional distortion. The derivation of CLP PSF is similar to that of ULP except that each row of the PSF matrices will be multiplied with the complex conjugate of the transmitted CLP. Figure 5.16 shows an example of CLP PSF.

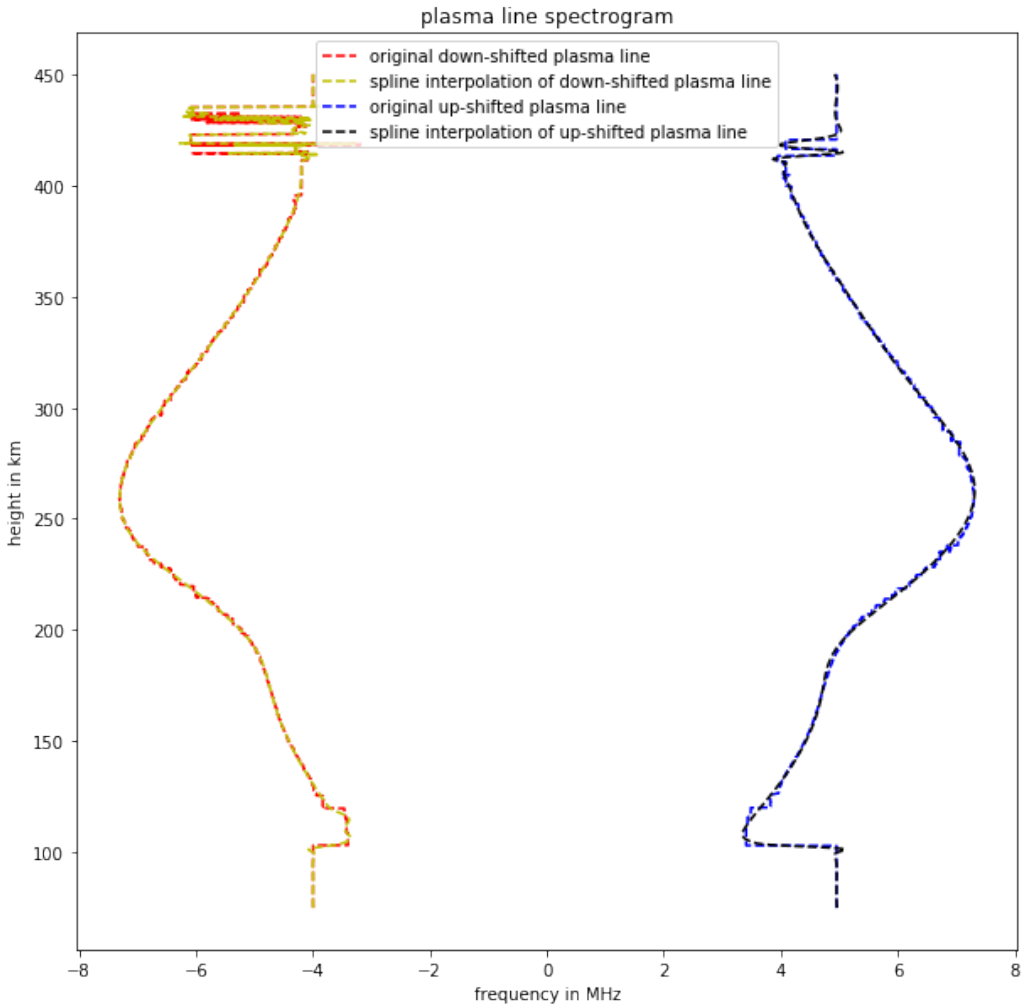
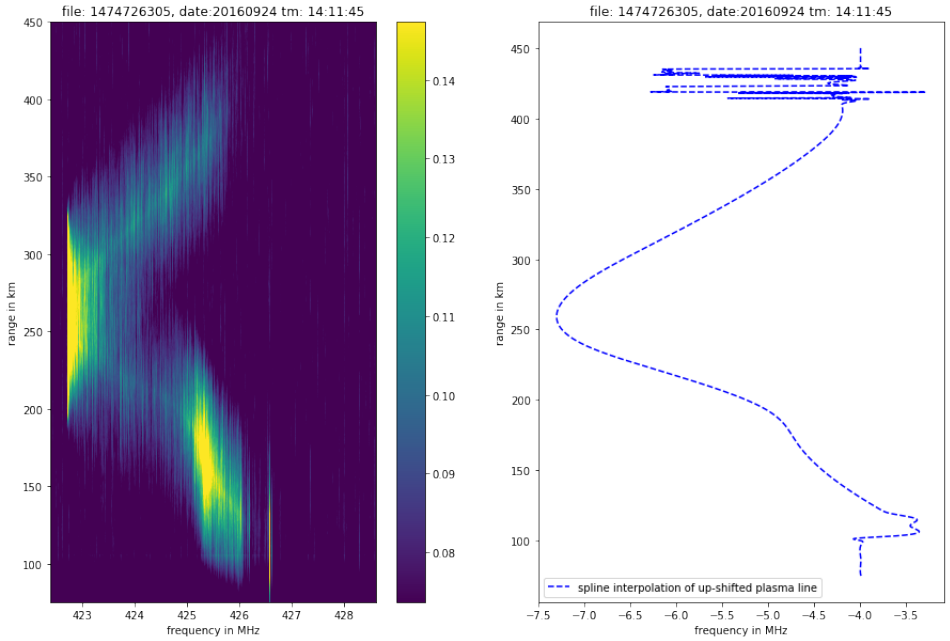


Figure 5.13: Spline interpolation of up-shifted and down-shifted plasma line spectrum.

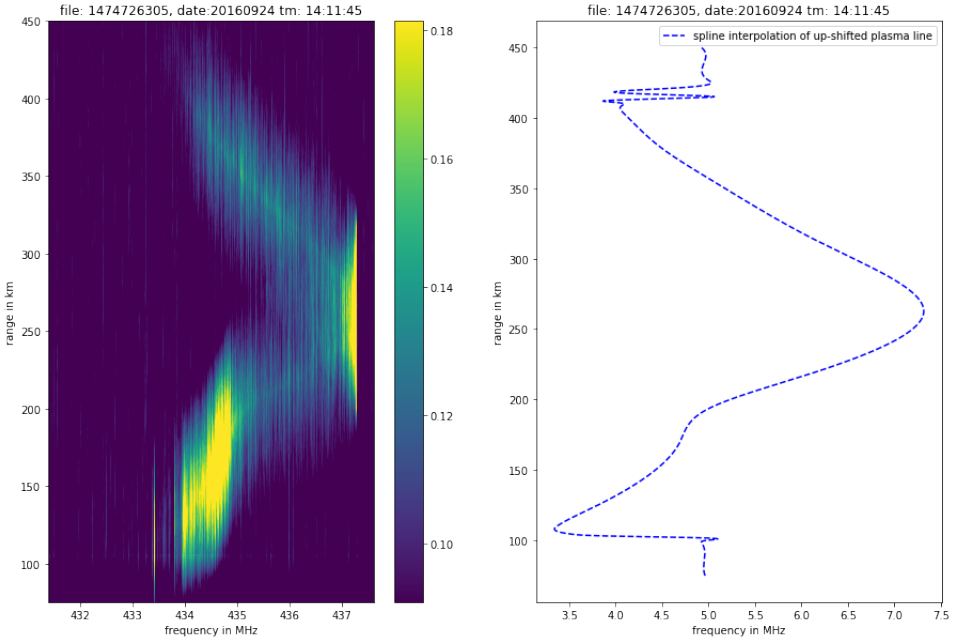
Then the gain step size is set to be 0.05 and the CLEANed spectrogram is shown in Figure 5.17

- Remove the background noise from the plasma line spectrogram, and use argmax function to estimate the plasma frequency at each height. Figure 5.18 shows the spectrogram with background noise removed and Figure 5.19 represents the estimated plasma line frequency.

- Use spline interpolation to smooth the plasma line to derive more accurate plasma frequency. Figure 5.20 illustrates the spline interpolation of the CLP plasma line.



(a) The left figure is original distorted down-shifted plasma line spectrogram and right figure is spline interpolation of the fitted plasma line.



(b) The left figure is original distorted up-shifted plasma line spectrogram and right figure is spline interpolation of the fitted plasma line.

Figure 5.14: Spline interpolation of plasma line spectrogram.

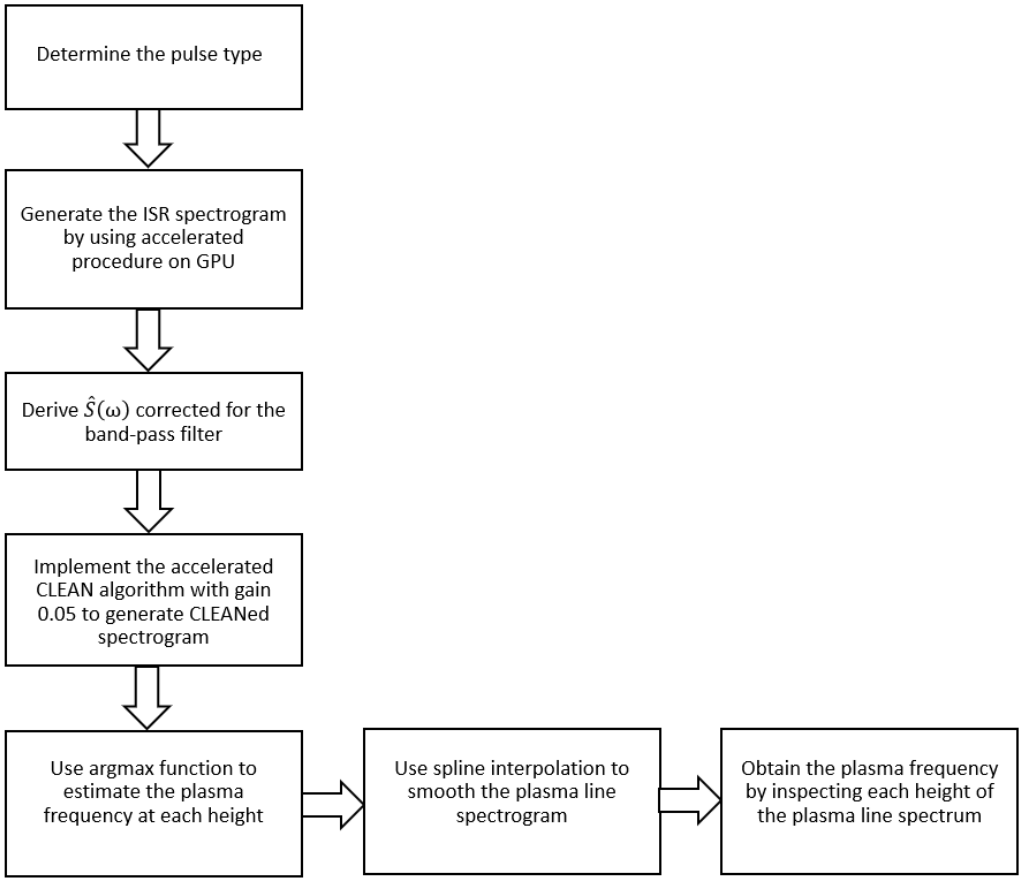


Figure 5.15: Block diagram to derive the plasma frequency from the raw data received from Arecibo Observatory.

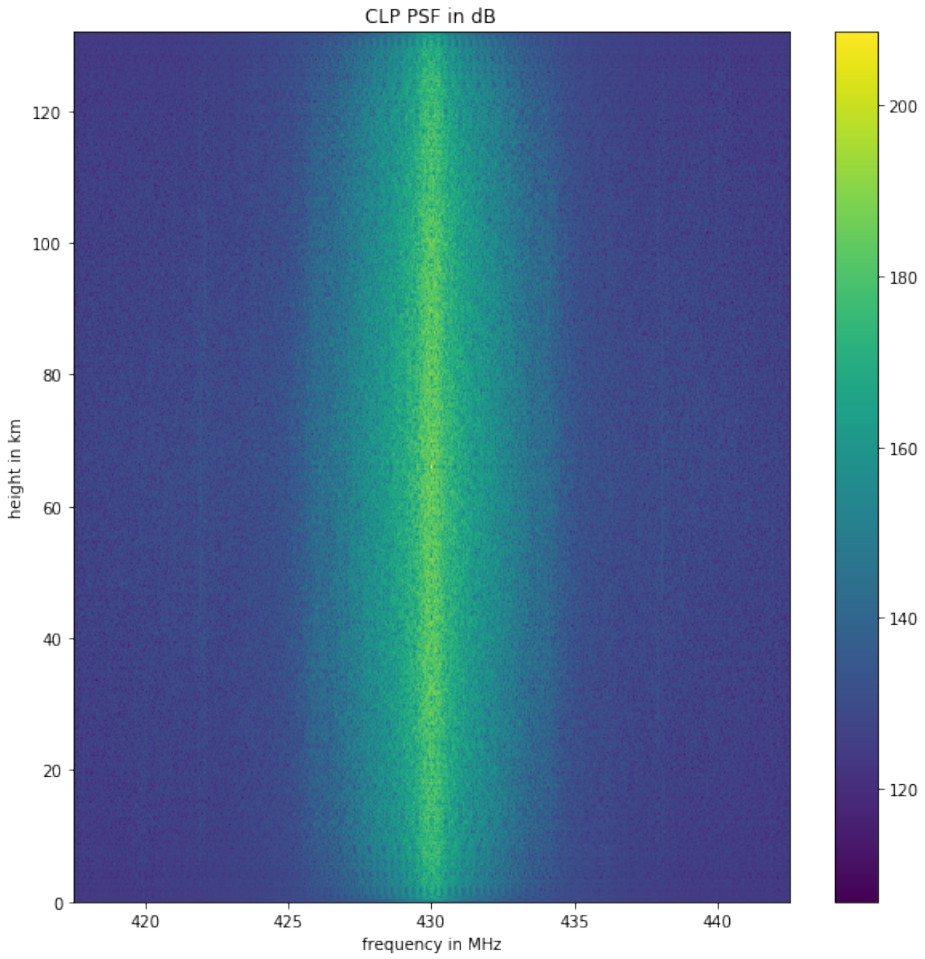
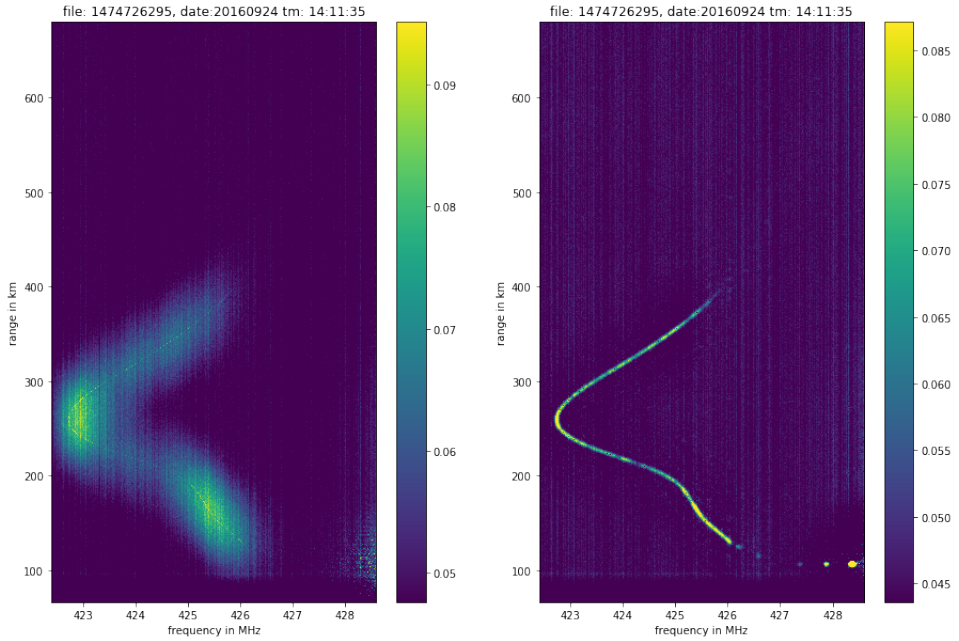
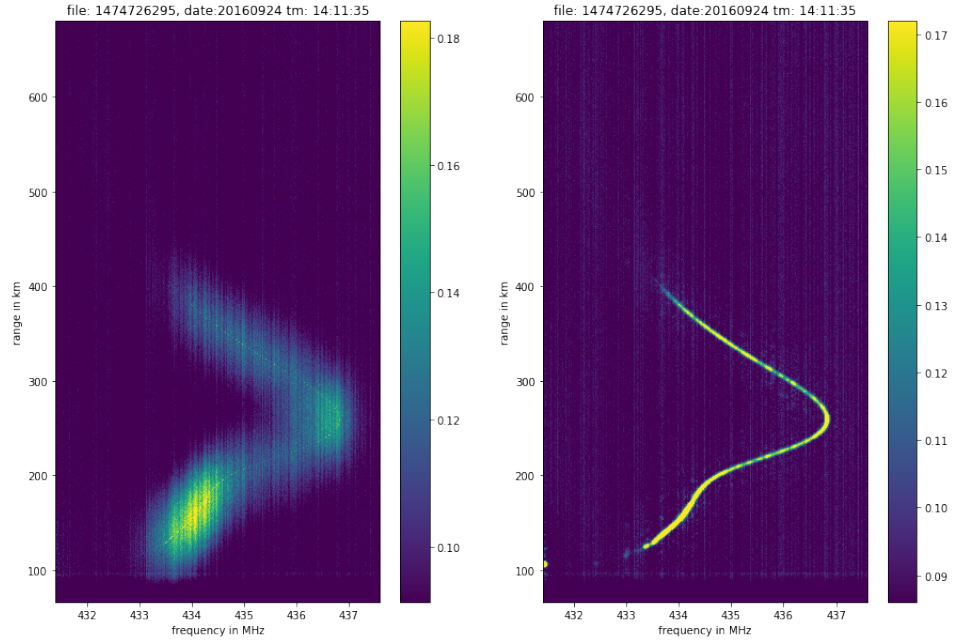


Figure 5.16: Point-spread-function of the CLP with pulse width 0.44 ms.

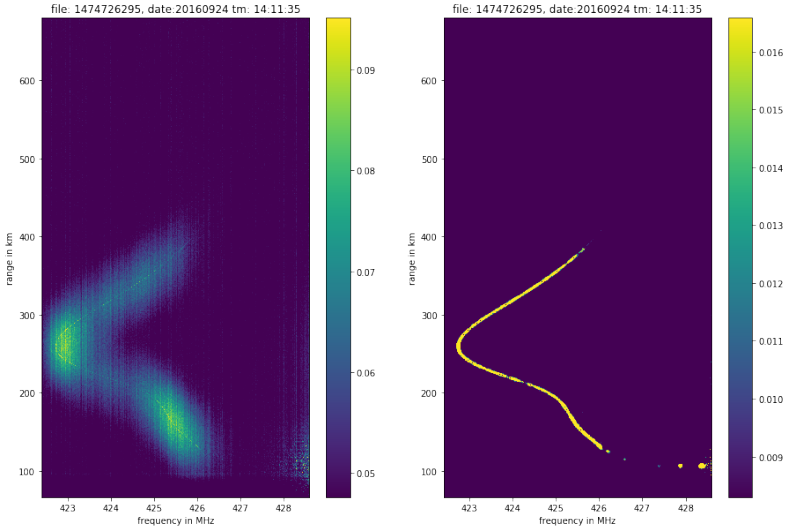


(a) The left figure distorted up-shifted plasma line spectrogram and right figure is recovered up-shifted plasma line.

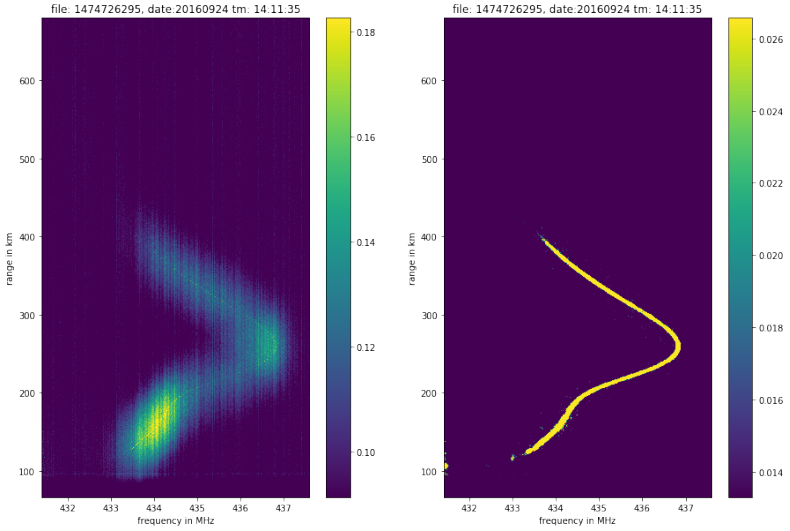


(b) The left figure is distorted down-shifted plasma line spectrogram and right figure is recovered down-shifted plasma line.

Figure 5.17: Restored plasma line spectrogram with gain step 0.05.

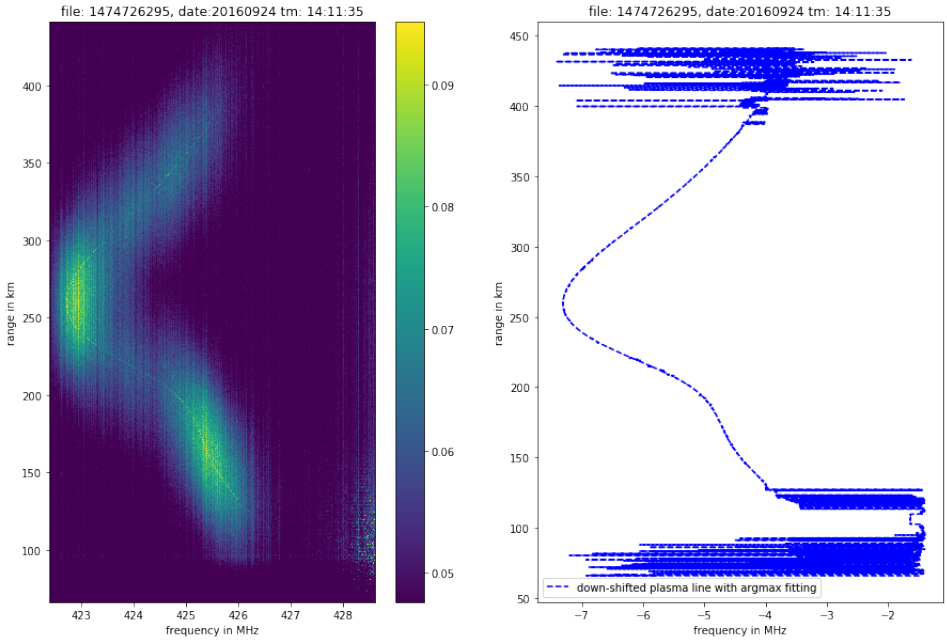


(a) Down-shifted CLEANed spectrogram with background noise removed.

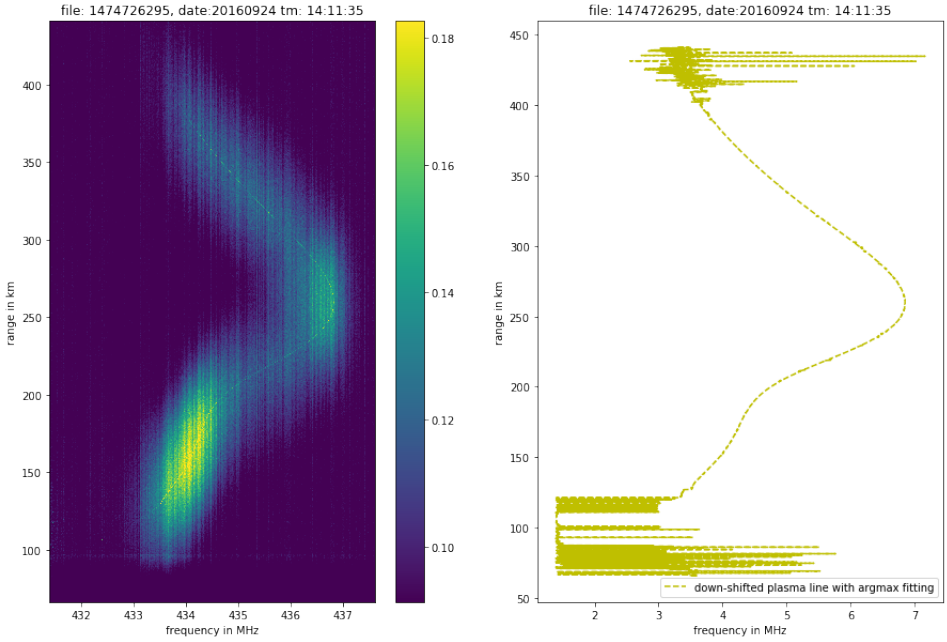


(b) Up-shifted CLEANed spectrogram with background noise removed.

Figure 5.18: Restored plasma line spectrogram without background noise.



(a) Estimated down-shifted plasma frequency.



(b) Estimated up-shifted plasma frequency.

Figure 5.19: Estimated plasma frequency.

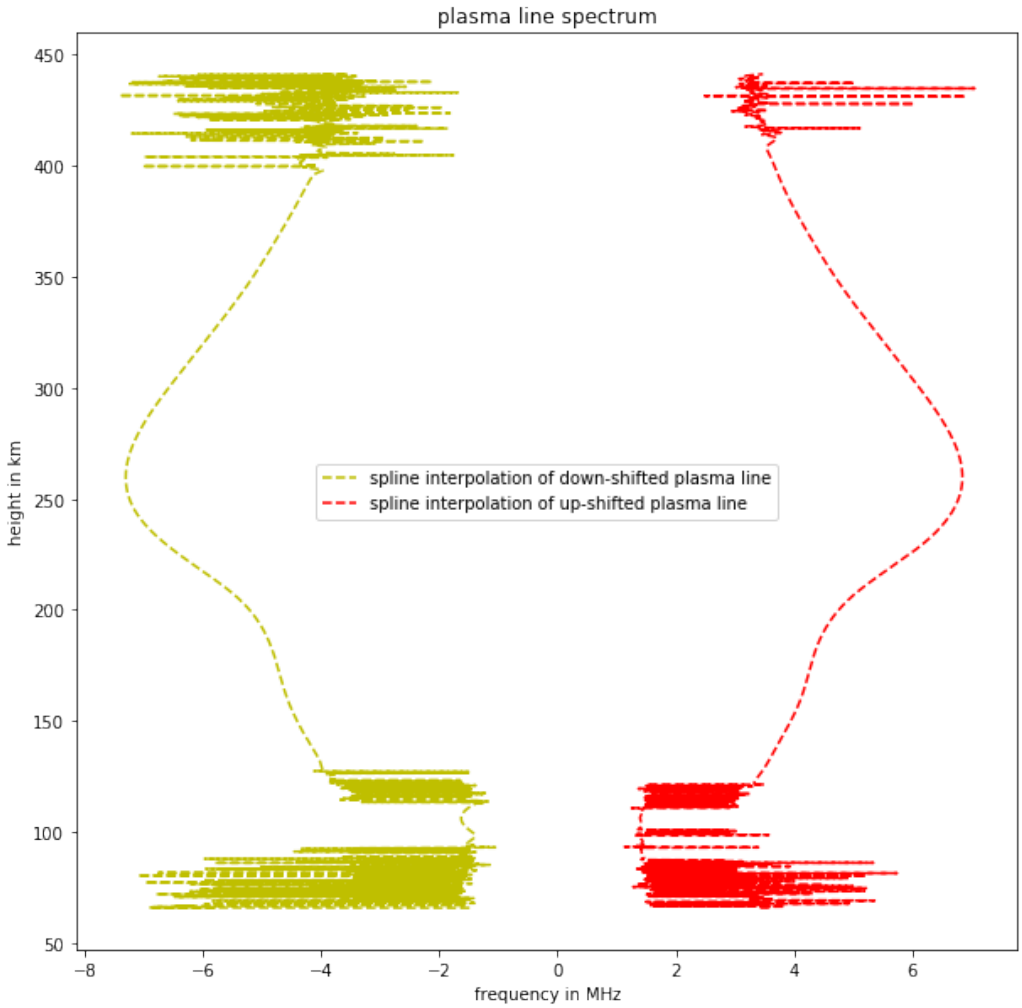


Figure 5.20: Spline interpolation of CLP plasma line.

CHAPTER 6

PLASMA-LINE PARAMETERS FITTING

With the plasma-line spectrogram, one could estimate the parameters including electron density, ion density and their corresponding component in the ionosphere by applying the general framework of incoherent scatter spectral theories according to *Kudeki and Milla* [2006]. The spectrum of electron density fluctuations in the equilibrium plasma is given by:

$$\langle |n(\mathbf{k},\omega)|^2 \rangle = \frac{|j\omega\varepsilon_o + \sigma_i| \langle |n_{te}(\mathbf{k},\omega)|^2 \rangle}{|j\omega\varepsilon_o + \sigma_e + \sigma_i|^2} + \frac{|\sigma_e| \langle |n_{ti}(\mathbf{k},\omega)|^2}{|j\omega\varepsilon + \sigma_e + \sigma_i|^2}, \tag{6.1}$$

where

$$\langle |n_{ts}(\mathbf{k},\omega)|^2 \rangle = 2N_o \Re\{J_s(\omega_s)\}, \tag{6.2}$$

and

$$\sigma_s(\mathbf{k},\omega) = \frac{1 - j\omega_s J_s(\omega_s)}{k^2 h_s^2} \cdot j\omega\varepsilon_o. \tag{6.3}$$

In the above equations, ω_s represents the Doppler-shifted frequency in the radar frame while $h_s \equiv \sqrt{\frac{\varepsilon K T_s}{N_o e^2}}$ denotes the Debye length of the species s. In addition, $J_s(\omega)$ corresponds to Gordeyev integral of species s where it could be expressed as

$$J_s(\omega) \equiv \int_0^\infty d\tau e^{-j\omega\tau} \langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_s} \rangle, \tag{6.4}$$

where $\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_s} \rangle$ denotes the single particle's auto-correlation function (ACF).

In this chapter, the non-magnetic and collision-less case will be discussed and the corresponding specification of ACF $\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}_s} \rangle$ will be derived for Gordeyev integration.

6.1 ACF derivation and Gordeyev Integral Calculation

In a non-magnetized and collisionless plasma, particles will move along the straight line trajectories with random velocities **v**. In this case, the displacement vector could be denoted as:

$$\Delta r = \mathbf{v}\tau. \tag{6.5}$$

Then, the Gaussian-distributed displacements Δr will have a probability distribution

$$f(\Delta r) = \frac{e^{-\frac{\Delta r^2}{2\langle \Delta r^2 \rangle}}}{\sqrt{2\pi\langle \Delta r^2 \rangle}}. \tag{6.6}$$

The mean square of the displacement could be expressed as

$$\langle \Delta r^2 \rangle = \langle v^2 \rangle \tau^2 = C^2\tau^2, \tag{6.7}$$

where $C \equiv \sqrt{KT/m}$ is the thermal speed of the charged carrier.

As a result, the single particle's ACF in a non-magnetized and collisionless plasma will be

$$\langle e^{j\mathbf{k}\cdot\Delta\mathbf{r}} \rangle = e^{-\frac{1}{2}k^2C^2r^2}. \tag{6.8}$$

After the generalization of single particle's ACF, we will move forward to the Gordeyev integral calculation to generate the theoretical plasma-line power spectrum.

6.2 Gordeyev Integral Calculation

The Gordeyev integral is demonstrated as

$$J(\omega) = \int_0^\infty d\tau e^{-j\omega\tau} \langle e^{j\mathbf{k}\cdot\Delta r} \rangle. \tag{6.9}$$

In this section, two algorithms will be discussed for the implementation of Gordeyev integral calculations. The first algorithm is the Dawson integral for F-region computation while the second method will be chirp-z algorithm which will be applied in higher plasma-frequency and electron density.

6.2.1 Dawson Integral

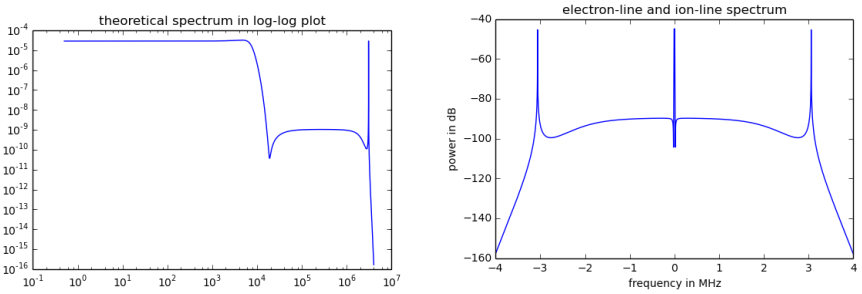
The Gordeyev integral in a non-magnetized collisionless plasma is

$$J(\omega) = \int_0^\infty d\tau e^{-j\omega\tau} \langle e^{j\mathbf{k}\cdot\Delta r} \rangle = \int_0^\infty d\tau e^{-j\omega\tau} e^{-\frac{1}{2}k^2C^2\tau^2}. \tag{6.10}$$

For the equation above, the identity as show below could used

$$jZ(\theta) \equiv \int_0^\infty dt e^{-j\theta t} e^{\frac{-t^2}{2}} = \sqrt{\pi} e^{-\theta^2} - j2e^{-\theta^2} \int_0^\theta e^{t^2} dt, \tag{6.11}$$

where $e^{-\theta^2} \int_0^\infty e^{t^2}$ is the Dawson's integral function. Figure 6.1 represents a sample theoretical plasma-line spectrum by applying the Dawson's integral.



(a) Plasma spectrum in loglog scale. (b) Plasma spectrum in dB scale.

Figure 6.1: Incoherent scatter spectral model.

In the right section of the Figure 6.1, the peak in the center region describes the ion-line component of the spectrum while another two peaks at ± 3 MHz denote the location of the plasma-line. This is the most basic incoherent scattering model. If we zoom into the center part for a closer investigation, a double humped shape will appear as show in Figure 6.2

However, the Dawson's integral will not be functioning well when the plasma frequency or the electron density is significantly high. When the electron density is increasingly higher, the plasma line in the theoretical incoherent scatter spectrum will be distorted and the Dawson's integral will present an incorrect representation. Consequently, the chirp-z algorithm will be adopted to deal with the high electron density.

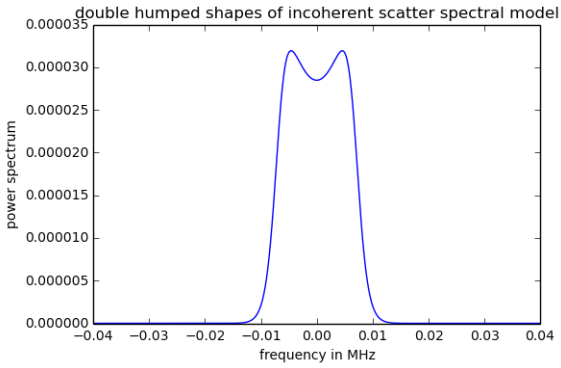


Figure 6.2: Ion line spectrum.

6.2.2 Chirp-z Calculation

The Gordeyev integral is

$$J(\omega) = \int_0^\infty d\tau e^{-j\omega\tau} \langle e^{j\mathbf{k}\cdot\Delta r} \rangle, \tag{6.12}$$

According to *Li et al.* [1991], by applying "Chirp" fast-field program, given an arbitrary integration $\int_0^{k_{max}} F(z,z_s;k)e^{-ikr}dk$ could be evaluated a N-point summation. The k_{max} in the integration is chosen such that the error produced by truncating the range of integral is negligible. The summation could be expressed as

$$S_N = \Delta k \sum_{n=0}^{N-1} F(z,z_s;n\Delta k)\exp(-in\Delta k r_o)W_N^{nm}, \tag{6.13}$$

where

$$W_N = \exp(-2ip\pi/N). \tag{6.14}$$

In our calculation, r_o is chosen as 0 so that the N-point summation becomes

$$S_N = \Delta k \sum_{n=0}^{N-1} F(z,z_s;n\Delta k)W_N^{nm}. \tag{6.15}$$

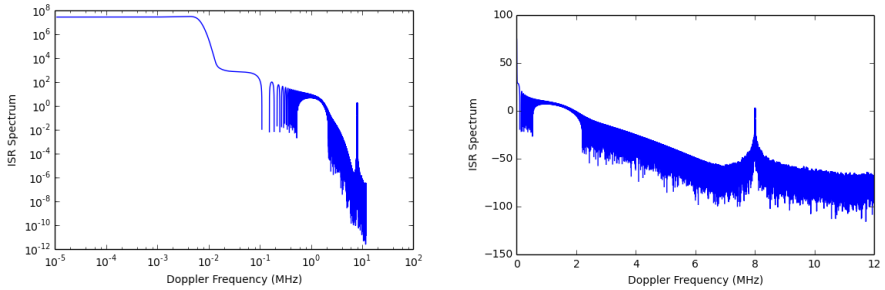
The parameter p has been introduced so that Δk and Δr could be chosen independently. By manipulating the equation, we could get the equation

$$S_N = \Delta k W_N^{m^2/2} \sum_{n=0}^{N-1} F(z,z_s;n\Delta k) \times W_N^{n^2/2} W_N^{-(m-n)^2/2}. \tag{6.16}$$

If we define $F(z, z_s; n\Delta k)W_N^{n^2/2}$ as a new sequence X_n and $W_N^{-n^2/2}$ as Y_n, then the summation can be treated as the discrete convolution of X_n and Y_n. Thus, the convolution can be expressed as:

$$S_N = \Delta k W_N^{(m^2/2)} \text{IFFT}\{\text{FFT}[X_n]\text{FFT}[Y_n]\}. \tag{6.17}$$

Consequently, the Gordeyev integral calculation could be evaluated by the high speed convolution. Figure 6.3 represents the theoretical incoherent scatter spectra.



(a) Plasma spectrum in loglog scale. (b) Plasma spectrum in dB scale.

Figure 6.3: Incoherent scatter spectral model.

Similarly, Figure 6.4 shows the ion spectrum component if we take a closer look at the center region.

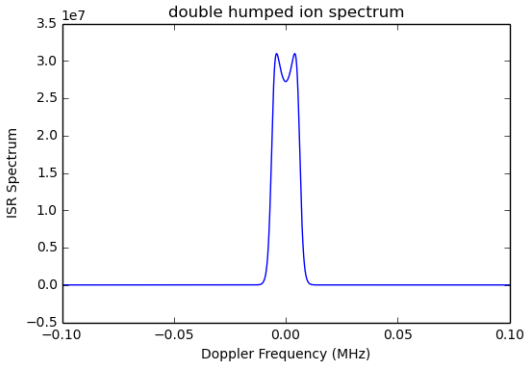


Figure 6.4: Ion line spectrum.

According to the observation, the Chirp-z method could generate a decent and plasma-line spectrogram efficiently. When we are dealing with high electron density or plasma frequency, we could adopt the Chirp-z algorithm to generate the theoretical incoherent scatter spectra.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The ISR spectrogram generation requires heavy computation which could not be easily handled due to the limitation of technology decades ago. However, thanks to the development of the graphics processing unit and introduction of parallel programing, we can efficiently generate the ISR spectrogram by implementing parallel FFTs in GPU. By implementing the full spectrogram analysis and accelerated procedure, the time cost for spectrogram generation and spectral measurement have been discussed. We find out that the optimal algorithm to process the ISR ULP and CLP data is to use an accelerated procedure.

In both ULP and CLP mode spectrogram processing, we use an accelerated operational method by skipping every 25 heights prior to FFT and spectrogram accumulation. In ULP mode analysis each pulse is multiplied with $e^{\pm j2\pi f_o nT}$, where $f_o = \pm 62.5$ kHz is the transmitted frequency offset from the 430 MHz carrier frequency utilized in the ULP mode on alternate pulses, to demodulate the ULP pulses back to 430 MHz. In the CLP mode each pulse has a unique sequence of binary code imposed on it which requires a demodulation step accomplished by multiplying each data row with the complex conjugate of the transmitted samples contained in row 0.

The ISR spectrogram obtained by the accelerated procedure is corrected for its convolutional distortions by using the CLEAN algorithm as discussed in Section 5.1—namely, we use CLEAN to deconvolve the plasma line features in the spectrograms making use of the appropriate point spread functions of the ULP and CLP modes. CLEANed spectrograms can be fitted to a Gaussian model or reduced using an argmax based approach to identify the plasma line frequency for each sampled altitude. Both methods work well and produce near identical results but the argmax method is faster and thus preferable, in particular after the spline smoothing operation that is implemented as the final step.

Regarding future work, we will explore the feasibility of accelerating the plasma frequency estimation technique even further and apply the method to process all

the Arecibo USRP ULP and CLP data collected since July 2016. We will also apply the methods developed here to deconvolve and fit the ion-line component of the broadband ULP and CLP spectrograms derived from USRP data. Spectral fits of both the ion-line and plasma lines can be based on the theory outlined in Chapter 6. Theoretical ISR spectrum in non-magnetized plasmas can be derived by approximating the Gordeyev integrals required by the theory using Dawson's function tabulated in MATLAB and numpy. However for broadband and zoomed-in frequency calculations the Gordeyev integrals need to be computed using the Chirp-z algorithm as described by *Kudeki and Milla* [2011]. We plan to develop a Chirp-z transform based broadband ISR spectrum model to be fitted to both the ion-line and plasma-line features obtained in ULP and CLP experiments. By fitting the ion-lines and plasma-lines simultaneously we would have a most comprehensive means of detecting the ionospheric state parameters describing the electron and ion populations and their dynamic and thermodynamic states.

REFERENCES

Altschuler, D. R., The National Astronomy and Ionosphere Center's (NAIC) Arecibo Observatory in Puerto Rico, in *Single-Dish Radio Astronomy: Techniques and Applications*, vol. 278, pp. 1–24, 2002.

Djuth, F. T., M. P. Sulzer, and J. H. Elder, Application of the coded long-pulse technique to plasma line studies of the ionosphere, *Geophysical Research Letters*, *21*(24), 2725–2728, 1994.

Högbom, J., Aperture synthesis with a non-regular distribution of interferometer baselines, *Astronomy and Astrophysics Supplement Series*, *15*, 417, 1974.

Isham, B., C. Tepley, M. Sulzer, Q. Zhou, M. Kelley, J. Friedman, and S. González, Upper atmospheric observations at the Arecibo Observatory: Examples obtained using new capabilities, *Journal of Geophysical Research: Space Physics*, *105*(A8), 18,609–18,637, 2000.

Kelley, M., *The Earth's Ionosphere: Plasma Physics and Electrodynamics*, International Geophysics, Elsevier Science, 2009.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation, *Parallel Computing*, *38*(3), 157–174, 2012.

Kudeki, E., and M. Milla, Incoherent scatter spectrum theory for modes propagating perpendicular to the geomagnetic field, *Journal of Geophysical Research: Space Physics*, *111*(A6), 2006.

Kudeki, E., and M. A. Milla, Incoherent scatter spectral theories–part I: A general framework and results for small magnetic aspect angles, *IEEE Transactions on Geoscience and Remote Sensing*, *49*(1), 315–328, 2011.

Li, Y., S. Franke, and C. Liu, Numerical implementation of an adaptive fast-field program for sound propagation in layered media using the chirp z transform, *The Journal of the Acoustical Society of America*, *89*(5), 2068–2075, 1991.

Rich, J., W. De Blok, T. Cornwell, E. Brinks, F. Walter, I. Bagetakos, and R. Kennicutt Jr, Multi-scale clean: A comparison of its performance against classical clean on galaxies using things, *The Astronomical Journal*, *136*(6), 2897, 2008.

Sulzer, M. P., A radar technique for high range resolution incoherent scatter autocorrelation function measurements utilizing the full average power of klystron radars, *Radio Science*, *21*(06), 1033–1040, 1986.

Vierinen, J., B. Gustavsson, D. Hysell, M. Sulzer, P. Perillat, and E. Kudeki, Radar observations of thermal plasma oscillations in the ionosphere, *Geophysical Research Letters*, *44*(11), 5301, 2017.

Yngvesson, K., and F. Perkins, Radar Thomson scatter studies of photoelectrons in the ionosphere and Landau damping, *Journal of Geophysical Research*, *73*(1), 97–110, 1968.

APPENDIX A

ISR SPECTROGRAM GENERATION CODE

The following code consists of both the uncoded-long pulse and coded-long pulse operational spectrogram generation. The notebook uses both Python and Pycuda to process the ISR raw data from Arecibo Observatory.

A.1 ULP Spectrogram Generation

```
%pylab inline
import glob,os
dpath = '/mnt/remote2_rdata3_radar/'
dirlist = sorted(glob.glob1(dpath,"2016-09-24T??-??-??"))
filespath = os.path.join(dpath,dirlist[13])
flist = sorted(glob.glob1(filespath,"*.h5"))

def read_file_data(fname,verbose=False):
   import h5py
   fp =h5py.File(fname,"r")
   data = fp.get('rf_data')
   if verbose:
      print (data.dtype)
   ndata = data['r'].squeeze() + 1j * data['i'].squeeze()
   return ndata.astype("complex64")
#Modified code
def Detect_IPP_Tx(ndata0,verbose=False):
   import numpy as np
   decim=100
   pwr100 = (abs(ndata0)**2).reshape(ndata0.shape[0]//decim,decim
      ).sum(1)
   derv100 = pwr100[1:]-pwr100[:-1]
   thresh1 = 2e8
   thresh2 = -1e8
   curr_test = 0
```

```
#Adjust the threshold value
if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or np
    .nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
   thresh1 = 2e6
   thresh2 = -1e6

if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or np
    .nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
   TxUp = []
   IPPs_start = []
   if verbose:
      print ("No Pulses detected")
   return TxUp, IPPs_start

curr_test += np.nonzero(derv100[curr_test:]>thresh1)[0][0]
TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
while TxUp<50:
   if verbose:
      print ("false pulse")
   curr_test += TxUp
   # Its not a Tx pulse (maybe interference)
   if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or
        np.nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
      TxUp = []
      IPPs_start = []
      print ("No pulses detected")
      return TxUp, IPPs_start

   curr_test += np.nonzero(derv100[curr_test:]>thresh1)[0][0]
   TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
   if verbose:
      print (curr_test, TxUp)
IPPs_start = [curr_test* decim] # Start of first Tx
TxWs = [TxUp * decim]
curr_test += TxUp

while curr_test < len(derv100)-1:
   find_result = np.nonzero(derv100[curr_test:]>thresh1)[0]
   if len(find_result)==0:
      if verbose:
         print ("end of file")
      break
```

```
   curr_test += find_result[0]
   TxUp = np.nonzero(derv100[curr_test:] < thresh2)[0][0]
   while TxUp<50:
      if verbose:
         print ("false pulse",curr_test)
      curr_test += TxUp
      # Its not a Tx pulse (maybe interference)

      find_result = np.nonzero(derv100[curr_test:]>thresh1)[0]
      if len(find_result)==0:
         if verbose:
            print ("no more pulses found")
         break
      curr_test += find_result[0]
      TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
   if TxUp>50:
      IPPs_start += [curr_test* decim]
      TxWs += [TxUp * decim]
   curr_test += TxUp
IPPs_start = np.array(IPPs_start)
TxWs = np.array(TxWs)
if len(IPPs_start)<=1:
   TxUp = []
   IPPs_start = []
   print ("only one pulse detected")
   return TxUp, IPPs_start

if verbose:
   print ("Samples in Pulse:",TxUp * decim)
   print ("Samples in IPPs:",(IPPs_start[1]-IPPs_start[0]) *
      decim)

# Refining the Tx start

startTx = IPPs_start[0]
startscan = max(0,startTx-100) # in case pulse is close to the
    start of the file
maxTxval = max(abs(ndata0[startscan:startscan+400]))
findresult = np.nonzero(abs(ndata0[startscan:startscan+400])>
   maxTxval/2)[0]
if len(findresult)>0:
   newstartTx = startscan + findresult[0]
   correction = newstartTx - startTx
```

```
   else:
      correction = 0

   IPPs_start += correction

   return TxWs,IPPs_start

def get_delta_f_ph(data):
   x = np.fft.fftshift(abs(np.fft.fft(data))**2)
   omega = argmax(x)

   return omega - len(data)//2

import pycuda.autoinit
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
import skcuda.fft as cufft
from pycuda.compiler import SourceModule


nFFT = 16384
batch = 7864

plan2 = cufft.Plan(shape = (nFFT,), in_dtype=np.complex64,
    out_dtype=np.complex64,
              batch = batch, stream = None, mode = 1,
              inembed = np.array([nFFT],dtype = int),
              istride = 1,
              idist =nFFT,
              onembed = np.array([nFFT],dtype = int),
              ostride =1,
              odist = nFFT)

#Rearrange the one dimension array into a 20000*11000 matrix
kernel = SourceModule("""
#include <stdio.h>
#include <complex.h>
__global__ void rearrange(float2 *a, float2 *dest)
{
//int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
int batch = 7864;
int nFFT = 12500;
```

```
if(Cols < nFFT)
{
   for (int i = 0; i < batch; i++)
   dest[Cols + nFFT * i] = a[Cols + i * 25];
}

}
""")
rearrange = kernel.get_function('rearrange')

#Getting the magnitude of the matrix and add them up
kernel1 = SourceModule("""
#include <stdio.h>
#include<complex.h>
__global__ void power(float2 *a, float *dest)
{

//Thread index
int nFFT = 16384;
int batch = 7864;
const int Rows = blockIdx.y * blockDim.y + threadIdx.y;
const int Cols = blockIdx.x * blockDim.x + threadIdx.x;
int a_index = Cols + nFFT * Rows;
float reala,imaga;
if (Rows < batch && Cols < nFFT)
{
   reala = a[a_index].x;
   imaga = a[a_index].y;
   __syncthreads();
   dest[a_index] += reala * reala + imaga * imaga;

}
}
""")
power = kernel1.get_function('power')

##Multiply the samples with corresponding code
kernel2 = SourceModule("""
__global__ void shifting(float2 *a, float2 *cod)
{
int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
float real;
```

```
float imag;
int nFFT = 12500;
int batch = 7864;
if (Rows < batch && Cols < nFFT){
   real = a[Cols + Rows * nFFT].x * cod[Cols].x - a[Cols + Rows *
       nFFT].y * cod[Cols].y;
   imag = a[Cols + Rows * nFFT].y * cod[Cols].x + a[Cols + Rows *
       nFFT].x * cod[Cols].y;
   __syncthreads();
   a[Cols + Rows * nFFT].x = real;
   a[Cols + Rows * nFFT].y = imag;
}



}
""")
shifting = kernel2.get_function('shifting')

#store the FFT value
dest_temp = gpuarray.empty([batch,nFFT-3884],np.complex64)
#store the power value
dest_gpu = gpuarray.zeros([batch,nFFT],np.float32)
gpu_zero_padding = gpuarray.zeros([batch,nFFT],np.complex64)
dest_temp_gpu = gpuarray.empty((batch,nFFT),np.complex64)
cpu_dest = np.zeros([batch,nFFT],float32)

np.seterr(divide='ignore')
os.nice(20)
n=0
flag = False
t1 = time.time()
filespath = os.path.join(dpath,dirlist[13])
flist = sorted(glob.glob1(filespath,"*.h5"))
N = 12500
j = 2439
#flag = False
while (j<2449):
#for j in range(2473,2473+10):
   print ("processing the file",j)
   sys.stdout.flush()
   ndatai = read_file_data(os.path.join(filespath,flist[j]))
   TxWidth, IPPs_start = Detect_IPP_Tx(ndatai,False)
```

```
if (len(TxWidth) ==0 or len(IPPs_start) ==0):
   j+=1
#Determine if it's ULP
elif (all(IPPs_start[1:]-IPPs_start[:-1]==500000) and all(
   TxWidth==12500)):
   x_gpu = gpuarray.to_gpu(ndatai)
   t = np.arange(N)
   test_pulse = ndatai[IPPs_start[0]:IPPs_start[0] + 12500]
   omega = get_delta_f_ph(test_pulse)
   modsignal1 = np.exp(-1j*t*62.5/2*2*pi/N)
   modsignal2 = np.exp(1j*t*62.5/2*2*pi/N)
   modsignal1_gpu = gpuarray.to_gpu(modsignal1).astype(
      complex64)
   modsignal2_gpu = gpuarray.to_gpu(modsignal2).astype(
      complex64)

   if omega > 0:
      for numIPPs in range(0,len(IPPs_start)-1,2):
         #shifting
         rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
            numIPPs] + 25*batch + 12500 - 25],dest_temp,grid
            = (13,1),block = (1024,1,1))
         shifting(dest_temp,modsignal1_gpu,grid = (391,246),
            block = (32,32,1))
         #zero-padding
         gpu_zero_padding[:,:dest_temp.shape[1]] = dest_temp
         #Generating spectrum
         cufft.fft(gpu_zero_padding,dest_temp_gpu,plan2)
         power(dest_temp_gpu,dest_gpu,grid = (512,246),block =
              (32,32,1))
      for numIPPs in range(1,len(IPPs_start)-1,2):
         #shifting
         rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
            numIPPs] + 25*batch + 12500 - 25],dest_temp,grid
            = (13,1),block = (1024,1,1))

         shifting(dest_temp,modsignal2_gpu,grid = (391,246),
            block = (32,32,1))
         #zero-padding
         gpu_zero_padding[:,:dest_temp.shape[1]] = dest_temp
         #Generating spectrum
         cufft.fft(gpu_zero_padding,dest_temp_gpu,plan2)
```

```
            power(dest_temp_gpu,dest_gpu,grid = (512,246),block =
                (32,32,1))
      if omega < 0:
         for numIPPs in range(0,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
               numIPPs] + 25*batch + 12500 - 25],dest_temp,grid
               = (13,1),block = (1024,1,1))
            #rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
               numIPPs] + 10*batch + 12500 - 10],dest_temp,grid
               = (13,1),block = (1024,1,1))

            shifting(dest_temp,modsignal2_gpu,grid = (391,246),
               block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] = dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,plan2)
            power(dest_temp_gpu,dest_gpu,grid = (512,246),block =
                (32,32,1))
         for numIPPs in range(1,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
               numIPPs] + 25*batch + 12500 - 25],dest_temp,grid
               = (13,1),block = (1024,1,1))
            #rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[
               numIPPs] + 25*batch + 12500 - 25],dest_temp,grid
               = (13,1),block = (1024,1,1))

            shifting(dest_temp,modsignal1_gpu,grid = (391,246),
               block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] = dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,plan2)
            power(dest_temp_gpu,dest_gpu,grid = (512,246),block =
                (32,32,1))

   j = j+1

t2 = time.time()
cpu_dest = dest_gpu.get()
spectrum = np.fft.fftshift(cpu_dest,axes=1)
```

```
print ("the total time is",(t2-t1))

x = linspace(417.5,442.5,16384)
fnoise = spectrum[7500:,:].mean(0)
spectrum2 = spectrum[500:,:]/fnoise
```

A.2 CLP Spectrogram Generation

```
%pylab inline
import matplotlib.pyplot as plt
import glob,os
dpath = '/mnt/remote2_rdata3_radar/'
dirlist = sorted(glob.glob1(dpath,"2016-09-24T??-??-??"))

filespath = os.path.join(dpath,dirlist[13])
flist = sorted(glob.glob1(filespath,"*.h5"))
import time, calendar
def read_file_data(fname,verbose=False):
   import h5py
   fp =h5py.File(fname,"r")
   data = fp.get('rf_data')
   if verbose:
      print (data.dtype)
   ndata = data['r'].squeeze() + 1j * data['i'].squeeze()
   return ndata.astype("complex64")
#Modified code
def Detect_IPP_Tx(ndata0,verbose=False):
   import numpy as np
   decim=100
   pwr100 = (abs(ndata0)**2).reshape(ndata0.shape[0]//decim,decim
      ).sum(1)
   derv100 = pwr100[1:]-pwr100[:-1]
   thresh1 = 2e8
   thresh2 = -1e8
   curr_test = 0

   #Adjust the threshold value
   if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or np
      .nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
      thresh1 = 2e6
      thresh2 = -1e6
   if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or np
```

```
    .nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
   TxUp = []
   IPPs_start = []
   if verbose:
      print ("No Pulses detected")
   return TxUp, IPPs_start

curr_test += np.nonzero(derv100[curr_test:]>thresh1)[0][0]
TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
while TxUp<50:
   if verbose:
      print ("false pulse")
   curr_test += TxUp
   # Its not a Tx pulse (maybe interference)
   if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or
       np.nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
      TxUp = []
      IPPs_start = []
      print ("No pulses detected")
      return TxUp, IPPs_start

   curr_test += np.nonzero(derv100[curr_test:]>thresh1)[0][0]
   TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
   if verbose:
      print (curr_test, TxUp)
IPPs_start = [curr_test* decim] # Start of first Tx
TxWs = [TxUp * decim]
curr_test += TxUp

while curr_test < len(derv100)-1:
   find_result = np.nonzero(derv100[curr_test:]>thresh1)[0]
   if len(find_result)==0:
      if verbose:
         print ("end of file")
      break
   curr_test += find_result[0]
   TxUp = np.nonzero(derv100[curr_test:] < thresh2)[0][0]
   while TxUp<50:
      if verbose:
         print ("false pulse",curr_test)
      curr_test += TxUp
      # Its not a Tx pulse (maybe interference)
```

```
            find_result = np.nonzero(derv100[curr_test:]>thresh1)[0]
            if len(find_result)==0:
               if verbose:
                  print ("no more pulses found")
               break
            curr_test += find_result[0]
            TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]
         if TxUp>50:
            IPPs_start += [curr_test* decim]
            TxWs += [TxUp * decim]
         curr_test += TxUp
      IPPs_start = np.array(IPPs_start)
      TxWs = np.array(TxWs)
      if len(IPPs_start)<=1:
         TxUp = []
         IPPs_start = []
         print ("only one pulse detected")
         return TxUp, IPPs_start

      if verbose:
         print ("Samples in Pulse:",TxUp * decim)
         print ("Samples in IPPs:",(IPPs_start[1]-IPPs_start[0]) *
             decim)


      # Refining the Tx start

      startTx = IPPs_start[0]
      startscan = max(0,startTx-100) # in case pulse is close to the
          start of the file
      maxTxval = max(abs(ndata0[startscan:startscan+400]))
      findresult = np.nonzero(abs(ndata0[startscan:startscan+400])>
         maxTxval/2)[0]
      if len(findresult)>0:
         newstartTx = startscan + findresult[0]
         correction = newstartTx - startTx
      else:
         correction = 0

      IPPs_start += correction

      return TxWs,IPPs_start
   import pycuda.autoinit
```

```
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
import skcuda.fft as cufft
from pycuda.compiler import SourceModule
nFFT = 16384
batch = 7864
plan2 = cufft.Plan(shape = (nFFT,), in_dtype=np.complex64,
    out_dtype=np.complex64,
              batch = batch, stream = None, mode = 1,
              inembed = np.array([nFFT],dtype = int),
              istride = 1,
              idist =nFFT,
              onembed = np.array([nFFT],dtype = int),
              ostride =1,
              odist = nFFT)
#Rearrange the one dimension array into a 20000*11000 matrix
kernel = SourceModule("""
#include <stdio.h>
#include <complex.h>
__global__ void rearrange(float2 *a, float2 *dest)
{
//int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
int batch = 7864;
int nFFT = 11000;
if(Cols < nFFT)
{
   for (int i = 0; i < batch; i++)
   dest[Cols + nFFT * i] = a[Cols + i * 25];
}

}
""")
rearrange = kernel.get_function('rearrange')
#Getting the magnitude of the matrix and add them up
kernel1 = SourceModule("""
#include <stdio.h>
#include<complex.h>
__global__ void power(float2 *a, float *dest)
{

//Thread index
const int Rows = blockIdx.y * blockDim.y + threadIdx.y;
```

```
const int Cols = blockIdx.x * blockDim.x + threadIdx.x;
int batch = 7864;
int nFFT = 16384;
int a_index = Cols + nFFT * Rows;
float reala,imaga;
if (Rows < batch && Cols < nFFT)
{
   reala = a[a_index].x;
   imaga = a[a_index].y;
   __syncthreads();
   dest[a_index] += reala * reala + imaga * imaga;

}
}
""")
power = kernel1.get_function('power')
##Multiply the samples with corresponding code
kernel2 = SourceModule("""
__global__ void decode(float2 *a, float2 *cod)
{
int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
float real;
float imag;
int nFFT = 11000;
int batch = 7864;
if (Rows < batch && Cols < nFFT){
   real = a[Cols + Rows * nFFT].x * cod[Cols].x - a[Cols + Rows *
       nFFT].y * cod[Cols].y;
   imag = a[Cols + Rows * nFFT].y * cod[Cols].x + a[Cols + Rows *
       nFFT].x * cod[Cols].y;
   __syncthreads();
   a[Cols + Rows * nFFT].x = real;
   a[Cols + Rows * nFFT].y = imag;
}


}
""")
decode = kernel2.get_function('decode')
#store the FFT value
dest_temp = gpuarray.empty([batch,nFFT-5384],np.complex64)
#store the power value
```

```
dest_gpu = gpuarray.zeros([batch,nFFT],np.float32)
dest_temp_gpu = gpuarray.empty((batch,nFFT),np.complex64)
gpu_zero_padding = gpuarray.zeros([batch,nFFT],np.complex64)
cpu_dest = np.zeros([batch,nFFT],float32)
np.seterr(divide='ignore')
os.nice(20)
n=0
#flag = False
#IPPindex = 0
t1 = time.time()
filespath = os.path.join(dpath,dirlist[13])
flist = sorted(glob.glob1(filespath,"*.h5"))

#flag = False

j=2429
while j<2439:
   print ("processing the iteration",j)
   sys.stdout.flush()

   ndatai = read_file_data(os.path.join(filespath,flist[j]))
   ##Corner Case
   TxWidth, IPPs_start = Detect_IPP_Tx(ndatai,False)
   if (len(TxWidth) ==0 or len(IPPs_start) ==0):
      j+=1
   #Determine if it's ULP
   elif (all(IPPs_start[1:]-IPPs_start[:-1]==250000) and all(
      TxWidth==11000)):

      x_gpu = gpuarray.to_gpu(ndatai) ##pulse with rfi

      for numIPPs in range(len(IPPs_start)-1):

         codi = np.conj(ndatai[IPPs_start[numIPPs]:IPPs_start[
            numIPPs] + 11000])
         cod_gpu = gpuarray.to_gpu(codi)

         rearrange(x_gpu[IPPs_start[numIPPs]:IPPs_start[numIPPs]
            + 25*batch + 11000 - 25],dest_temp,grid = (11,1),
            block = (1024,1,1))

         decode(dest_temp,cod_gpu,grid = (344,246),block =
            (32,32,1))
```

```
            gpu_zero_padding[:,:dest_temp.shape[1]] = dest_temp

            cufft.fft(gpu_zero_padding,dest_temp_gpu,plan2)
            power(dest_temp_gpu,dest_gpu,grid = (512,246),block =
                (32,32,1))

    j = j+1

cpu_dest = dest_gpu.get()
spectrum = np.fft.fftshift(cpu_dest,axes=1)
t2 = time.time()
print ("the total time is",(t2-t1))
fnoise = spectrum[7500:,:].mean(0)
spectrum2 = spectrum[500:,:]/fnoise
```

APPENDIX B

CHIRP-Z GORDEYEV INTEGRAL CALCULATION

```
def chirpz(g,n,dt,dw,wo):
   """transforms g(t) into G(w)
   g(t) is n-point array and output G(w) is (n/2)-points starting
       at wo
   dt and dw, sampling intervals of g(t) and G(w), and wo are
   prescribed externally in an idependent manner
   --- see Li, Franke, Liu [1991]"""
   g[0]=0.5*g[0] # first interval is over dt/2, and hence ...
   W = exp(-1j*dw*dt*arange(n)**2/2.)
   S = exp(-1j*wo*dt*arange(n)) # frequency shift by wo
   x = g*W*S; y = conj(W)
   #x = g*W; y = conj(W)
   x2 = np.zeros(len(x)); y2 = np.empty(len(y))
   y2 = y[::-1]
   x3 = np.concatenate((x,x2)); y3 = np.concatenate((y,y2))

   #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)
   xi = fft.fft(x3); yi = fft.fft(y3); G = dt*W*(fft.ifft(xi*yi)
      [:len(x)])
   #return G[0:n/2]
   return G
# Ionospheric State
Ne=18.0e11 #Electron density (1/m^3)
fp=sqrt(Ne*80.6)
Ne = (8e6)**2/80.6;NO = Ne*0.998
# Ion Composition
NO=0.998*Ne
Te = TO = 2000
# Physical Paramters (MKS):
me=9.1093826e-31 # Electron mass in kg
```

```
mO = 1836.152*16.*me
mO=1836.152*16.*me # Ion mass
qe=1.60217653e-19 # C (Electron charge)
K=1.3806505e-23 # Boltzmann cobstant 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
aspect=45.*pi/180. # Aspect angle (rad) with 0 perp to Bs
Ce = sqrt(K*Te/me);CO = sqrt(K*TO/mO);
# Debye Lengths
debe = sqrt(eps0*K*Te/(Ne*qe**2));debO=sqrt(eps0*K*TO/(NO*qe**2))
Tmax=10*1.0e-6 #total integration time for electron Gordeyev
    integral
N = 22000*100/2
dt=Tmax/N
fo=0.0e6
fmax=12e6# Hz units (I choose this)
df=(fmax-fo)/(N/2) # in Hz units - only N/2 elements are returned
     from chirpz
wo=2*pi*fo
dw=2*pi*df
w=wo+arange(N/2)*dw ##??????????
fradar=430.0e6 # Radar Frequency (Hz)
lam=c/fradar/2.
kB=2*pi/lam # Bragg wavenumber kB = 2*ko
#Electron Gordeyev integral (Brownian)
t=arange(N)*dt
acfe = exp(-(kB*Ce*t)**2/2.)
Ge=chirpz(acfe,N,dt,dw,wo) # Electron Gordeyev Integral
figure(1)
plot(t/1.e-6,acfe); xlabel('Time Lag (us)'); ylabel('Electron ACF
    ')
figure(2)
plot(w/2./pi/1e6,real(Ge)); xlabel('Doppler Frequency (MHz)');
    ylabel('Re[Electron Gordeyev]')
plot(w/2./pi/1e6,imag(Ge))
# Oxygen Gordeyev integral (Brownian)
dtO=dt*100
t=arange(N)*dtO #adjust dt such that full range of acfi is
    covered by range t
acfO = exp(-(kB*CO*t)**2/2.)
GO=chirpz(acfO,N,dtO,dw,wo) # Ion Gordeyev Integral
figure(1)
```

```
plot(t/1.e-6,acfO); xlabel('Time Lag (ms)'); ylabel('O+ ACF')
figure(2)
plot(w/2./pi/1e6,real(GO)); xlabel('Doppler Frequency (MHz)');
   ylabel('Re[O+ Gordeyev]')
plot(w/2./pi/1e6,imag(GO))
# Total ISR Spectrum
yO=(1-1j*w*GO)/(kB**2*debO**2) # oxygen admittance
ye=(1-1j*w*Ge)/(kB**2*debe**2) # electron admittance
spec=real(Ne*2*Ge)*abs((1+yO)/(1+ye+yO))**2+ \
     real(NO*2*GO)*abs((ye)/(1+ye+yO))**2
plot(w/2./pi/1e6,spec);
xlabel('Doppler Frequency (MHz)'); ylabel('ISR Spectrum')
xlim(0,0.1)
loglog(w/2./pi/1e6,spec);
xlabel('Doppler Frequency (MHz)'); ylabel('ISR Spectrum')
#ylim(1.0e-16,1.0e-2)
plot(w/2./pi/1e6,10*log10(spec));
xlabel('Doppler Frequency (MHz)'); ylabel('ISR Spectrum')
#xlim(0,0.1)
```

APPENDIX C

PLASMA LINE DERIVATION CODE

```
import glob
import sys,os
import pycuda.autoinit
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray
import skcuda.fft as cufft
from pycuda.compiler import SourceModule
dpath = '/mnt/remote2_rdata3_radar/'
dirlist = sorted(glob.glob1(dpath,'*'))
syear_arr,smonth_arr,sday_arr,shh_arr,smm_arr,sss_arr =
    [],[],[],[],[],[]
for i,fname in enumerate(dirlist[:-10]):
   syear_arr += [fname[:4]]
   smonth_arr += [fname[5:7]]
   sday_arr += [fname[8:10]]
   shh_arr += [fname[11:13]]
   smm_arr += [fname[14:16]]
   sss_arr += [fname[17:19]]
for i,syear in enumerate(syear_arr):
   print (i,syear_arr[i],smonth_arr[i],sday_arr[i],shh_arr[i],
       smm_arr[i],sss_arr[i])
def process_spec(yyyy,mm,dd,hh,mmin,ss):
   from time import gmtime,strftime
   from calendar import timegm
   from glob import glob1
   import numpy as np
   import sys,os
   nyear = int(yyyy)
   nmonth = int(mm)
   nday = int(dd)
   nhh = int(hh)
   nmm = int(mmin)
   nss = int(ss)
```

```
#ndoy = gmtime(timegm((nyear,nmonth,nday,0,0,0))).tm_yday

##Create folders to saving
specfolder_year = '/home/yang158/notebook/FFT_on_ULP/CLEAN/
   func/processed/%.4d/'%nyear
specfolder = '/home/yang158/notebook/FFT_on_ULP/CLEAN/func/
   processed/%.4d/%.4d_%.2d_%.2dT%.2d_%.2d/'%(nyear,nyear,
   nmonth,nday,nhh,nmm)

if not os.path.exists(specfolder_year):
   os.makedirs(specfolder_year)
if not os.path.exists(specfolder):
   os.makedirs(specfolder)
dpath = '/mnt/remote2_rdata3_radar/'
dirlist = sorted(glob.glob1(dpath,'*'))

##Reading data in each file
def read_file_data(fname,verbose=False):
   import h5py
   fp =h5py.File(fname,"r")
   data = fp.get('rf_data')
   if verbose:
      print (data.dtype)
   ndata = data['r'].squeeze() + 1j * data['i'].squeeze()
   return ndata.astype("complex64")
#Modified code
def Detect_IPP_Tx(ndata0,verbose=False):
   decim=100
   pwr100 = (abs(ndata0)**2).reshape(ndata0.shape[0]//decim,
      decim).sum(1)
   derv100 = pwr100[1:]-pwr100[:-1]
   thresh1 = 2e8
   thresh2 = -1e8
   curr_test = 0
   TxWs = []
   IPPs_start = []

   #Adjust the threshold value
   if (np.nonzero(derv100[curr_test:]>thresh1)[0].size == 0 or
       np.nonzero(derv100[curr_test:]<thresh2)[0].size ==0):
      thresh1 = 2e6
      thresh2 = -1e6
```

```
try:
   curr_test += np.nonzero(derv100[curr_test:]>thresh1)
      [0][0]
   TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]

except:
   return TxWs, IPPs_start

while TxUp<50:
   if verbose:
      print ("false pulse")
   curr_test += TxUp
   # Its not a Tx pulse (maybe interference)
   try:

      curr_test += np.nonzero(derv100[curr_test:]>thresh1)
         [0][0]
      TxUp = np.nonzero(derv100[curr_test:]<thresh2)[0][0]

   except:
      return TxWs, IPPs_start

   if verbose:
      print (curr_test, TxUp)
IPPs_start = [curr_test* decim] # Start of first Tx
TxWs = [TxUp * decim]
curr_test += TxUp

while curr_test < len(derv100)-1:
   find_result = np.nonzero(derv100[curr_test:]>thresh1)[0]
   if len(find_result)==0:
      if verbose:
         print ("end of file")
      break
   curr_test += find_result[0]
   try:
      TxUp = np.nonzero(derv100[curr_test:] < thresh2)
         [0][0]
   except:
      return TxWs,IPPs_start
   while TxUp<50:
      if verbose:
         print ("false pulse",curr_test)
```

```
      curr_test += TxUp
      # Its not a Tx pulse (maybe interference)

      find_result = np.nonzero(derv100[curr_test:]>thresh1)
          [0]
      if len(find_result)==0:
         if verbose:
            print ("no more pulses found")
         break
      curr_test += find_result[0]
      try:
         TxUp = np.nonzero(derv100[curr_test:]<thresh2)
             [0][0]
      except:
         return TxWs,IPPs_start

   if TxUp>50:
      IPPs_start += [curr_test* decim]
      TxWs += [TxUp * decim]
   curr_test += TxUp
IPPs_start = np.array(IPPs_start)
TxWs = np.array(TxWs)

if verbose:
   print ("Samples in Pulse:",TxUp * decim)
   print ("Samples in IPPs:",(IPPs_start[1]-IPPs_start[0])
       * decim)
# Refining the Tx start

startTx = IPPs_start[0]
startscan = max(0,startTx-100) # in case pulse is close to
   the start of the file
maxTxval = max(abs(ndata0[startscan:startscan+400]))
findresult = np.nonzero(abs(ndata0[startscan:startscan
   +400])>maxTxval/2)[0]
if len(findresult)>0:
   newstartTx = startscan + findresult[0]
   correction = newstartTx - startTx
else:
   correction = 0

IPPs_start += correction
```

```
    return TxWs,IPPs_start


def get_delta_f_ph(data):
    x = np.fft.fftshift(abs(np.fft.fft(data))**2)
    omega = argmax(x)

    return omega - len(data)//2


##Generate the PSF
def psf_func(txdata0):
    N = 12500
    t = arange(N)
    #TxWidth, IPPs_start = Detect_IPP_Tx(data,verbose=False)
    #txdata0 = data[IPPs_start[0]:IPPs_start[0] + TxWidth[0]]
    omega = get_delta_f_ph(txdata0)
    if omega<0:
        modsignal = np.exp(1j*t*62.5/2*2*pi/N)
    if omega>0:
        modsignal = np.exp(-1j*t*62.5/2*2*pi/N)
    basebandTx = txdata0 * modsignal
    dest_cpu = np.zeros([2*N,N+3884],np.complex64)
    dest_output = np.empty([2*N,N+3884],np.complex64)
    dest = np.empty(dest_cpu.shape).astype(complex64)

    for i in range(1,N):
        dest_cpu[i,0:i] = basebandTx[N-i:] ##leftshift
    for i in range(N):
        dest_cpu[i+N,:N-i] = basebandTx[:N-i]##rightshift

    nFFT = 16384
    batch = 2**13

    for i in range(dest_cpu.shape[0]):
        dest[i,:] = np.fft.fft(dest_cpu[i,:])
    #print (i)
    #sys.stdout.flush

    dest2 = abs(dest)**2
    psf = dest2.reshape(1000,25,16384).mean(axis=1)
    #psf = dest2.reshape(25,1000,16384).mean(axis=0)
    psf = np.fft.fftshift(psf,axes=1)
```

```
   ref = average(psf[:50,:500])
   psf_ref = np.copy(psf)
   for i in range(psf.shape[0]):
      reference = amax(psf[i,:])
   #for j in range(6250,8250):
      for j in range(8192,8292+300):
         if 10*log10(psf[i,j])<10*log10(reference)-10:
            psf_ref[i,j:] = 0
            psf_ref[i,:2*8192-j] = 0

            break
   return psf_ref

nFFT = 16384
batch = 16384
#Rearrange the one dimension array into a 20000*11000 matrix
plan2 = cufft.Plan(shape = (nFFT,), in_dtype=np.complex64,
   out_dtype=np.complex64,
            batch = batch, stream = None, mode = 1,
            inembed = np.array([nFFT],dtype = int),
            istride = 1,
            idist =nFFT,
            onembed = np.array([nFFT],dtype = int),
            ostride =1,
            odist = nFFT)
kernel = SourceModule("""
#include <stdio.h>
#include <complex.h>
__global__ void rearrange(float2 *a, float2 *dest)
{
//int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
int batch = 16384;
int nFFT = 12500;
if(Cols < nFFT)
{
   for (int i = 0; i < batch; i++)
   dest[Cols + nFFT * i] = a[Cols + i * 1];
}

}
""")
rearrange = kernel.get_function('rearrange')
```

```
#Getting the magnitude of the matrix and add them up
kernel1 = SourceModule("""
#include <stdio.h>
#include<complex.h>
__global__ void power(float2 *a, float *dest)
{

//Thread index
int nFFT = 16384;
int batch = 16384;
const int Rows = blockIdx.y * blockDim.y + threadIdx.y;
const int Cols = blockIdx.x * blockDim.x + threadIdx.x;
int a_index = Cols + nFFT * Rows;
float reala,imaga;
if (Rows < batch && Cols < nFFT)
{
   reala = a[a_index].x;
   imaga = a[a_index].y;
   __syncthreads();
   dest[a_index] += reala * reala + imaga * imaga;

}
}
""")
power = kernel1.get_function('power')

##Multiply the samples with corresponding code
kernel2 = SourceModule("""
__global__ void shifting(float2 *a, float2 *cod)
{
int Rows = blockIdx.y*blockDim.y + threadIdx.y;
int Cols = blockIdx.x*blockDim.x + threadIdx.x;
float real;
float imag;
int nFFT = 12500;
int batch = 16384;
if (Rows < batch && Cols < nFFT){
   real = a[Cols + Rows * nFFT].x * cod[Cols].x - a[Cols +
      Rows * nFFT].y * cod[Cols].y;
   imag = a[Cols + Rows * nFFT].y * cod[Cols].x + a[Cols +
      Rows * nFFT].x * cod[Cols].y;
   __syncthreads();
   a[Cols + Rows * nFFT].x = real;
```

```
    a[Cols + Rows * nFFT].y = imag;
}


}
""")
shifting = kernel2.get_function('shifting')

def decimation(spec):
    spectrum = spec[:196600,:].reshape(7864,25,16384).mean(axis
        =1)
    spectrum = np.fft.fftshift(spectrum,axes=1)[533:,:]

    return spectrum

def band_pass_filter(spec):
    fnoise = spec[7000:,:].mean(0)
    spec = spec/fnoise

    return spec

def pulse_process(numFile,length): ##10 sec ULP processing
    counter = 0
    nFFT = 16384
    batch = 16384
    ## Store data in CPU
    cpu_dest = np.zeros([12*batch,nFFT],float32)
    ## Store data after shifting
    dest_temp = gpuarray.empty([batch,nFFT-3884],np.complex64)
    ## Store the zero-padding data
    gpu_zero_padding = gpuarray.zeros([batch,nFFT],np.complex64
        )
    #Store FFT value
    dest_temp_gpu = gpuarray.empty((batch,nFFT),np.complex64)
    #store the power value
    dest_gpu = gpuarray.zeros([batch,nFFT],np.float32)

    N = 12500
    t = arange(N)

    while (counter < 9 and numFile < length):
        ndata = read_file_data(os.path.join(filespath,flist[
            numFile]))
```

```
TxWidth, IPPs_start = Detect_IPP_Tx(ndata,False)
numFile += 1
#if (len(TxWidth_0) != 50 or len(IPPs_start_0) != 50):
   #numFile += 1
if (len(TxWidth) == 50 and len(IPPs_start) == 50 and all
    (IPPs_start[1:]-IPPs_start[:-1]==500000) and all(
    TxWidth==12500)):
   counter += 1
   x_gpu = gpuarray.to_gpu(ndata)
   test_pulse = ndata[IPPs_start[0]:IPPs_start[0] +
       12500]
   omega = get_delta_f_ph(test_pulse)
   modsignal1 = np.exp(-1j*t*62.5/2*2*pi/N)
   modsignal2 = np.exp(1j*t*62.5/2*2*pi/N)
   modsignal1_gpu = gpuarray.to_gpu(modsignal1).astype(
       complex64)
   modsignal2_gpu = gpuarray.to_gpu(modsignal2).astype(
       complex64)

   for i in range(12):
      if omega > 0:
         for numIPPs in range(0,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs] + i *
               batch:IPPs_start[numIPPs]+ i*batch + 1*
               batch + 12500 - 1],dest_temp,grid =
               (13,1),block = (1024,1,1))
            shifting(dest_temp,modsignal1_gpu,grid =
               (391,512),block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] =
               dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,
               plan2)
            power(dest_temp_gpu,dest_gpu,grid =
               (512,512),block = (32,32,1))
         for numIPPs in range(1,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs] + i *
               batch:IPPs_start[numIPPs]+ i*batch + 1*
               batch + 12500 - 1],dest_temp,grid =
               (13,1),block = (1024,1,1))
```

```
            shifting(dest_temp,modsignal2_gpu,grid =
                (391,512),block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] =
                dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,
                plan2)
            power(dest_temp_gpu,dest_gpu,grid =
                (512,512),block = (32,32,1))
    if omega < 0:
        for numIPPs in range(0,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs] + i *
                batch:IPPs_start[numIPPs]+ i*batch + 1*
                batch + 12500 - 1],dest_temp,grid =
                (13,1),block = (1024,1,1))
            shifting(dest_temp,modsignal2_gpu,grid =
                (391,512),block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] =
                dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,
                plan2)
            power(dest_temp_gpu,dest_gpu,grid =
                (512,512),block = (32,32,1))
        for numIPPs in range(1,len(IPPs_start)-1,2):
            #shifting
            rearrange(x_gpu[IPPs_start[numIPPs] + i *
                batch:IPPs_start[numIPPs]+ i*batch + 1*
                batch + 12500 - 1],dest_temp,grid =
                (13,1),block = (1024,1,1))
            shifting(dest_temp,modsignal1_gpu,grid =
                (391,512),block = (32,32,1))
            #zero-padding
            gpu_zero_padding[:,:dest_temp.shape[1]] =
                dest_temp
            #Generating spectrum
            cufft.fft(gpu_zero_padding,dest_temp_gpu,
                plan2)
            power(dest_temp_gpu,dest_gpu,grid =
                (512,512),block = (32,32,1))
```

```
            cpu_dest[i*batch:(i+1)*batch,:] += dest_gpu.get()
            dest_gpu *= 0

    spectrum = decimation(cpu_dest)
    spec = band_pass_filter(spectrum)
    ##Free GPU memeory
    dest_temp.gpudata.free()
    gpu_zero_padding.gpudata.free()
    dest_gpu.gpudata.free()
    dest_temp_gpu.gpudata.free()
    x_gpu.gpudata.free()
    modsignal1_gpu.gpudata.free()
    modsignal2_gpu.gpudata.free()

    return spec



def overlap(dirty,psf,mx,my):
    radius_x = int(psf.shape[1]/2)
    radius_y = int(psf.shape[0]/2)
    dxbegin = 0
    dxend = dirty.shape[1]
    pxbegin = radius_x - mx
    pxend = pxbegin + dirty.shape[1]
    if my - radius_y < 0:
        dybegin = 0
        dyend = my + radius_y
        pybegin= radius_y - my
        pyend = 2 * radius_y
    elif my + radius_y > dirty.shape[0]:
        dybegin = my - radius_y
        dyend = dirty.shape[0]
        pybegin = 0
        pyend = radius_y - my + dirty.shape[0]

    #elif my-psf.shape[1]/2>=0 and my+psf.shape[1]/2<dirty.
        shape[1]:
    elif my + radius_y <= dirty.shape[0] and my - radius_y >=
        0:
        dybegin = my - radius_y
        dyend = my + radius_y
```

```
        pybegin = 0
        pyend = 2 * radius_y


    return (int(dxbegin), int(dxend), int(dybegin), int(dyend))
        , (int(pxbegin), int(pxend), int(pybegin), int(pyend))

def hogbom(dirty,psf,gain):
    #frac = 0
    res = np.array(dirty)
    cleaned = np.zeros(dirty.shape)
    point = np.ones(psf.shape)*0

    sigma_x = 20
    sigma_y = 10
    x = linspace(-100,100,201)
    y = linspace(-100,100,201)
    x,y = np.meshgrid(x,y)
    z = 1/sqrt(2*pi*sigma_x**2)*exp(-x**2/2/sigma_x**2) * 1./
        sqrt(2*pi*sigma_y**2)*exp(-y**2/2/sigma_y**2)
    z = z/amax(z)

    point[psf.shape[0]//2-100:psf.shape[0]//2+101,psf.shape
        [1]//2-100:psf.shape[1]//2+101] = z

    for i in range(30000):
        my, mx=numpy.unravel_index(argmax(res), res.shape)
        mval= res[my, mx] * gain
        d_index, p_index = overlap(res,psf,mx,my)
        res[d_index[2]:d_index[3],d_index[0]:d_index[1]] -= psf[
            p_index[2]:p_index[3],p_index[0]:p_index[1]] * gain
        #res[my,:] = 0
        #res[d_index[2]:d_index[3],d_index[0]:d_index[1]] -= psf
            [p_index[2]:p_index[3],p_index[0]:p_index[1]] * mval
        cleaned[d_index[2]:d_index[3],d_index[0]:d_index[1]] +=
            mval*point[p_index[2]:p_index[3],p_index[0]:p_index
            [1]]
        #frac=sum(dirty)/sum(cleaned)

        ##Condition
        #if amax(res)<0.02:
        if amax(res)<0.002 or amax(res)<(dirty[4000:,:]).max():
        #if frac < 0.02
```

```
            break
    restored = cleaned + res
    fnoise = restored[4000:,:].mean()
    ref_spec = restored - fnoise
    #return cleaned, i,res
    return ref_spec



def spline_fit(spec1,spec2):
    from scipy.interpolate import UnivariateSpline as spline

    x = linspace(79.95,694.35,4096)
    index=4096
    y,y2 = [],[]
    for i in range(index):
        y.append(argmax(spec1[i,:]))
        y2.append(argmax(cleaned_spec2[i,:]))

    y = 1.0*25*(np.array(y)+3184)/16384-12.5
    y2 = 1.0*25*(np.array(y2)+9104)/16384-12.5

    result = spline(x[:index],y)
    result.set_smoothing_factor(10)

    result2 = spline(x[:index],y2)
    result2.set_smoothing_factor(10)

    return x,y,y2,result(x),result2(x)




##Main Program
filespath = dpath+syear_arr[i]+'-'+smonth_arr[i]+'-'+sday_arr[
    i]+'T'+shh_arr[i]+'-'+smm_arr[i]+'-'+sss_arr[i]
flist = sorted(glob.glob1(filespath,"*.h5"))
file_length = len(flist)
numFile = 0
base = []
while(numFile + 9 < file_length):

    ndatai = read_file_data(os.path.join(filespath,flist[
```

```
   numFile]))
#print(os.path.join(filespath,flist[numFile]))
#sys.stdout.flush()
TxWidth_0, IPPs_start_0 = Detect_IPP_Tx(ndatai,False)
print ("processing the file:",numFile)
sys.stdout.flush()

##Determine if it's ULP
if (len(TxWidth_0) != 50 or len(IPPs_start_0) != 50):
   numFile += 1

elif (all(IPPs_start_0[1:]-IPPs_start_0[:-1]==500000) and
   all(TxWidth_0==12500)):

   ##Determine if it's starting ULP

   numFile_ref = numFile + 8

   ndata_ref = read_file_data(os.path.join(filespath,flist[
      numFile_ref]))
   TxWidth_ref, IPPs_start_ref = Detect_IPP_Tx(ndata_ref,
      False)

   if (len(TxWidth_ref) != 50 or len(IPPs_start_ref) != 50)
      :
      numFile += 9

   elif (all(IPPs_start_ref[1:]-IPPs_start_ref
      [:-1]==500000) and all(TxWidth_ref==12500)):

      psf_data = ndatai[IPPs_start_0[0]:IPPs_start_0[0] +
         TxWidth_0[0]]
      psf = psf_func(psf_data)

      spectrum_10s = pulse_process(numFile,file_length)

      ##Deconvolution by using clean algorithm
      spec_dirty1 = spectrum_10s[:4096,3184:7280]
      spec_dirty2 = spectrum_10s[:4096,9104:13200]

      spec_dirty1_norm = spec_dirty1/amax(spec_dirty1)
      psf_norm = psf/amax(psf)
      spec_dirty2_norm = spec_dirty2/amax(spec_dirty2)
```

```
    cleaned_spec = hogbom(spec_dirty1_norm,psf_norm,0.05)
    cleaned_spec2 = hogbom(spec_dirty2_norm,psf_norm
        ,0.05)

    height,sp_upshift_frequency,sp_downshift_frequency,
        downshift_frequency,upshift_frequency=spline_fit(
        cleaned_spec,cleaned_spec2)

    s1 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[:2]
    s2 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[3:5]
    s3 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[6:8]
    s4 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[9:11]
    s5 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[12:14]
    s6 = time.strftime("%x,%X",time.gmtime(int(flist[
        numFile][3:-7])))[15:18]


    sstime = []
    sstime = [s1]+[s2]+[s3]+[s4]+[s5]+[s6]
    #print (sstime)


    outfname = time.strftime("%X",time.gmtime(int(flist[
        numFile][3:-7])))
    np.savez_compressed(specfolder+outfname,
       height = height,
       sp_upshift_frequency =sp_upshift_frequency,
       downshift_frequency = downshift_frequency,
       upshift_frequency = upshift_frequency,
       sp_downshift_frequency = sp_downshift_frequency,
       time = sstime
       )

    base.append(upshift_frequency)
    numFile += 9

else:
```

```
            numFile += 9
      else:
         numFile += 1


print ("done")
return base
```