Design and implementation of spectrum sensing software in sub-6GHz bands using energy detection

Διδάσκων: Αθανάσιος Κοράκης
Υπεύθυνος Καθηγητής: Βιργίλιος Πασσάς
Φοιτητής: Ηλίας Ηλιάδης
ΑΕΜ: 02523 email: ililiadis@uth.gr

Σεπτέμβρης 2022

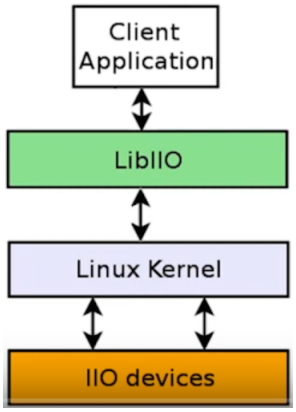
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Ηλεκτρονικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας

Contents

1 PludoSDR setup and Software/Drivers Install

Πρίν ξεκινήσουμε την υλοποίηση του project έπρεπε να εγκατασταθούν κάποιες απαραίτητες βιβλιοθήκες οι οποίες δημιουργούν την επικοινωνία μεταξύ των drivers του PlutoSDR και του Linux Industrial Input/Output (IIO) Subsystem (μέσω hardware interfacing απο την βιβλιοθήκη **libiio**). Επίσης, για την απλοποιήση της χρήσης διάφορων IIO drivers, έχει δημιουργηθεί απο την Analog Devices το python package **pyadi-iio**. Το συγκεκριμένο module παρέχει device-specific APIs τα οποία είναι υλοποιημένα πάνω σε ήδη υπάρχοντα libIIO python bindings. Οι βιβλιοθήκες που εγκαταστάθηκαν είναι οι εξής:

- **libiio:** Analog Device's "cross-platform" library for interfacing hardware.
- **libad9361-iio:** AD9361 is the specific RF chip inside the PlutoSDR.
- **pyadi-iio:** the Pluto's Python API, this is our end goal, but it depends on the previous two libraries.



Σχήμα 1: LibIIO interfacing

Επιπρόσθετα, η default αρχικοποίηση του PlutoSDR έχει περιορισμένο εύρος συχνοτήτων και ρυθμού δειγματοληψίας (sampling rate), ενώ το RF chip του Pluto υποστηρίζει μεγαλύτερο εύρος συχνοτήτων, πράγμα που χρειαζόμαστε καθώς θέλουμε να υποστηρίξουμε spectrum sensing σε όσο μεγαλύτερο εύρος συχνοτήτων γίνεται. Για να «ενεργοποιήσουμε» το μέγιστο εύρος συχνοτήτων εκτελούμε στο τερματικό του Pluto (μέσω ssh) τις παρακάτω εντολές:

 # fw_setenv attr_name compatible
 # fw_setenv attr_val ad9364
 # reboot

Με το νέο configuration θα μπορούμε να κάνουμε tune απο 70 MHz εώς 6 GHz.

Για εγκατάσταση των βιβλιοθηκών και την υλοποίηση του spectrum sensing software δημιουργήθηκε docker image.

2 Receive and process signals using PlutoSDR

Πρίν προχωρήσουμε στην επίδειξη της υλοποίησης του spectrum sensing software , θα εξηγήσουμε πως πραγματοποιείται η λήψη και η επεξεργασία ενός σήματος με την χρήση του PlutoSDR και της python.

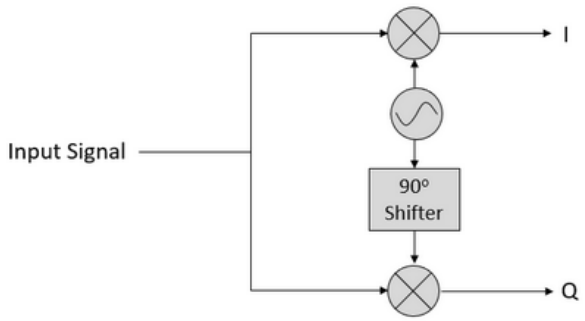
2.1 Receive signal

Για να λάβει ένας radio receiver ένα σήμα (π.χ ένα FM σήμα), χρησιμοποιείται quadrature sampling. Αυτο που λαμβάνεται απο την κεραία του receiver είναι ένα πραγματικό σήμα το οποίο χωρίζεται σε δύο σήματα, ένα ημίτονο και ένα συνημίτονο, τα οποία έχουν διαφορά φάσης 90 μοίρες μεταξύ τους. Συμβολίζοντας με I ("In phase") το πλάτος του συνημιτόνου και Q ("Quadrature") το πλάτος του ημιτόνου τα δυο σήματα συμβολίζονται ώς εξής:

$$I\cos(2\pi ft)$$

$$Q\sin(2\pi ft)$$

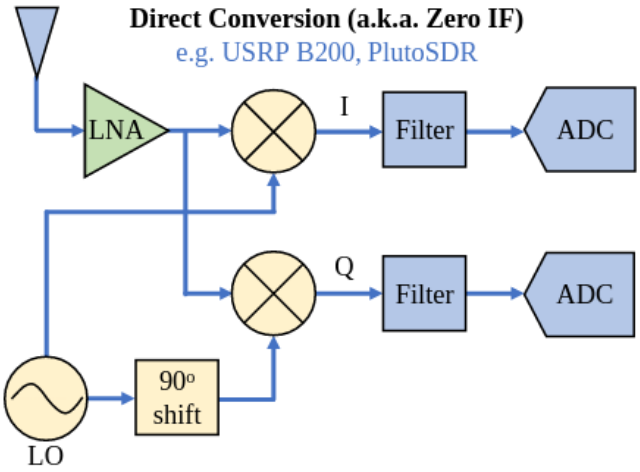
Η εν λόγω διαδικασία παρουσιάζεται σε επίπεδο αρχιτεκτονικής απο το παρακάτω διάγραμμα:



Σχήμα 2: split input signal into I and Q

Στην συνέχεια, πραγματοποιείται δειγματοληψία και των δύο σημάτων ξεχωριστά χρησιμοποιώντας δύο Analog to Digital Conventers (**ADCs**) με ρυθμό δειγματοληψίας (sample rate) **Fs**. Να σημειωθεί ότι τα περισσότερα SDR πραγματοποιούν filtering στην πλευρά του παραλήπτη κρατώντας μόνο τις τιμές εντός του διαστήματος [-Fs/2, Fs/2] ακριβώς πρίν το στάδιο της δειγματοληψίας. Μετά την δειγματοληψία των δυο σημάτων συνδιάζουμε I, Q ζευγάρια και τα αποθηκεύομε σε μορφή μιγαδικού αριθμού. Με άλλα λόγια, για ένα IQ sample, παράγεται ένας μιγαδικός αριθμός $\boldsymbol{I + jQ}$. Μετά την ολοκλήρωση της μετατροπής του αναλογικού σήματος σε ψηφιακό, έχουμε στην διάθεση μας ένα μονοδιάστατο διάνυσμα με τους μιγαδικούς αριθμούς που προκύπτουν απο τις I,Q παραμέτρους του σήματος.

Παρακάτω απεικονίζεται η ολοκληρωμένη αρχιτεκτονική του PlutoSDR στην πλευρά του παραλήπτη πρίν την επεξεργασία του σήματος:



Σχήμα 3: PlutoSDR receiver architecture

2.2 Process signal in frequency domain using python

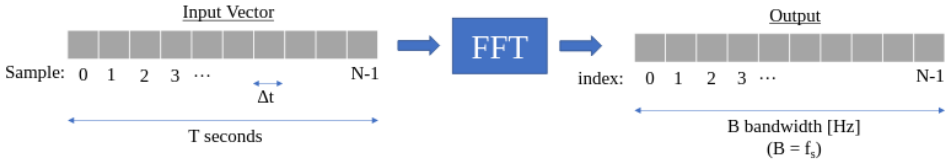
Έχοντας στην διάθεση μας το μονοδιάστατο διάνυσμα που προέκυψε απο το προηγούμενο κεφάλαιο, για να επεξεργαστούμε το σήμα θα πρέπει να το μεταφέρουμε απο το πεδίο του χρόνου στο πεδίο της συχνότητας. Η μετατροπή αυτη απο το πεδιο του χρόνου στο πεδίο της συχνότητας ονομάζεται μετασχιματισμός Fourier και περιγράφεται απο τον παρακάτω μαθηματικό τύπο:

$$X(f) = \int x(t) e^{-j2\pi ft} dt$$

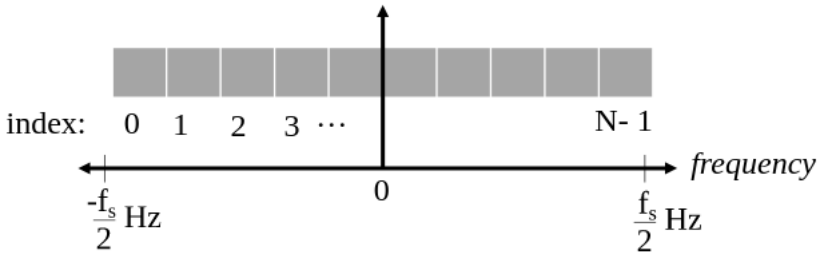
Η παραπάνω εξίσωση απευθύνεται σε συνεχή σήματα, τα οποία τα βλέπουμε μόνο σε μαθηματικά προβλήματα. Η διακριτή μορφή του Fourier (Discrete Fourier Transform : **DFT**) συσχετίζεται άμεσα με το πρόβλημα μας καθώς θέλουμε να μετατρέψουμε διακριτά σύνολα απο το πεδίο του χρόνου στο πεδίο της συχνότητας. Ο τύπος παρουσιάζεται παρακάτω:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi}{N} kn}$$

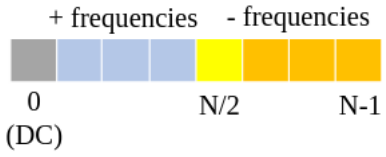
Στα πλαίσια της python , για τον υπολογισμο του DFT χρησιμοποιείται ο Fast Fourier Transform (**FFT**) ο οποίος είναι ένας αλγόριθμος για ταχύτερο υπολογισμό του DFT. Η εκτέλεση της συνάρτησης fft() της python παίρνοντας ώς είσοδο το διάνυσμα των μιγαδικών στο πεδίο του χρόνου απεικονίζεται παρακάτω:



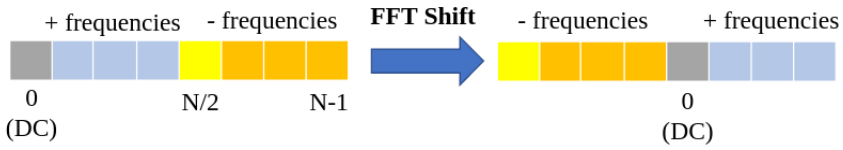
Το διάνυσμα που προκύπτει είναι η εκδοχή του διανύσματος εισόδου στο πεδίο της συχνότητας διατηρώντας το ίδιο μέγεθος. Αυτο που αξίζει να δώσουμε σημασία είναι ότι το διάνυσμα που προκύπτει μετά τον fft απεικονίζεται στο εύρος συχνοτήτων [-Fs/2, Fs/2] όπου Fs το sample rate με την ακόλουθη αντιστοιχία:



Το κάθε FFT bin (δηλαδή το κάθε στοιχείο του διανύσματος που προκύπτει) αντιστοιχίζεται στην συχνότητα Fs/N Hz όπου N το μέγεθος του διανύσματος. Επίσης, αυξάνοντας τον αριθμό των δειγμάτων ενός σήματος αποκτάμε καλύτερη ανάλυση του σήματος στο πεδίο της συχνότητας. Όταν υπολογίζουμε τον FFT στην python μέσω της συνάρτησης numpy.fft.fft(), για μαθηματικούς λόγους το αποτέλεσμα έχει την παρακάτω μορφή:



Για να κάνουμε την αντιστοιχία των στοιχείων του διανύσματος σε συχνότητες ίδια με αυτη που περιγράψαμε αρχικά, εφαρμόζουμε ένα FFT shift μέσω της συνάρτησης numpy.fft.fftshift(), η οποία αναδιατάσσει τα στοιχεία του διανύσματος ώστε να προκύψει το επιθυμητό αποτέλεσμα. Η διαδικασία απεικονίζεται παρακάτω:



Τέλος για την μετατροπή του σήματος στην τελική του μορφή (σε dB) θα πρέπει να υπολογίσουμε το Power Spectral Density (**PSD**) του διανύσματος στο πεδίο της συχνότητας. Ο κώδικας που υπολογίζει την εν λόγω ποσότητα απεικονίζεται παρακάτω:

```
# This function computes the fft of the given rx_buffer.
# Right after that it computes the Power Spectral Density (PSD)
# in dB through the fft sequence
def psd(rx_samples):
    fft_rx = np.fft.fft(rx_samples)
    scale = 2.0/(len(rx_samples) * len(rx_samples))
    psd = scale * (fft_rx.real**2 + fft_rx.imag**2)
    psd_log = 10.0*np.log10(psd)
    psd_shifted = np.fft.fftshift(psd_log)
    return(psd_shifted)
```

όπου rx_samples το διάνυσμα των μιγαδικών αριθμών $I + jQ$. Τέλος, να αναφέρουμε ότι πρίν τον υπολογισμό του FFT, γίνεται συνέλιξη του διανύσματος rx_samples με την συνάρτηση παραθύρου Blackman (numpy.blackman()) ούτως ώστε να αποφύγουμε απότομες διαφορές μεταξύ του πρώτου και του τελευταίου στοιχείου του διανύσματος.

```
rx_samples = rx_samples*np.blackman(len(rx_samples))
```

3 First software implementation to sense LoRa signal transmission

Αξιοποιώντας την παραπάνω διδικασία επεξεργασίας ενός σήματος καθώς και το python API του PlutoSDR υλοποιήθηκε το αρχικός κώδικας για λήψη και επεξεργασία σημάτων που εκπέμπονται απο έναν LoRa transmitter. Μετά το τέλος της επεξεργασίας του σήματος γίνεται plot για λόγους επαλήθευσης αποτελεσμάτων. Η αρχικοποίηση του PlutoSDR παρουσιάζεται παρακάτω

```
def lora(sdr):
    sample_rate = 1e6 # Hz
    center_freq = 866.1e6 # Hz
    num_samps = 100000 # number of samples per call to rx()

    sdr.sample_rate = int(sample_rate)

    # Config Rx
    sdr.rx_lo = int(center_freq)
    sdr.rx_rf_bandwidth = int(20e6)
    sdr.rx_buffer_size = num_samps
    sdr.gain_control_mode_chan0 = 'manual'
    sdr.rx_hardwaregain_chan0 = 64.0 # dB, increase to increase the
     receive gain, but be careful not to saturate the ADC
```

Πριν προχωρήσουμε στον αλγόριθμο , θα περιγράψουμε κάποιες σημαντικές μεταβλητές της αρχικοποίησης του SDR:

- **sample_rate :** ρυθμός δειγματοληψίας του σήματος, το PlutoSDR θα λάβει δείγματα του σήματος εντός του διαστήματος [-samplerate/2, sample_rate/2] (τα υπόλοιπα εκτός του διαστήματος γίνονται filter out απο το φίλτρο που υπάρχει πρίν απο τον ADC).
- **sdr.rx_lo :** αρχικοποιεί την συχνότητα συντονισμού η οποία αρχικοποιείται στην μεταβλητή **center_freq**.
- **sdr.rx_rf_bandwidth :** ρυθμίζει το εύρος συχνοτήτων των αναλογικών φλίλτρων στην πλευρά του RX (receiver) .
- **sdr.buffer_size :** αρχικοποιεί το μέγεθος του εσωτερικού buffer ο οποίος αποθηκεύει τα IQ samples σε μορφή μιγαδικών αριθμών. Το μέγεθος του buffer εκφράζεται σε πλήθος samples το οποίο αρχικοποιείται στην μεταβλητή **num_samps**.
- **sdr.gain_control_mode_chan0 :** χρησιμοποιείται για να ενεργοποιήσει το automatic gain control (AGC) το οποίο είναι ένα closed-loop feedback κύκλωμα το οποίο μεταβάλει το LNA gain ουτως ώστε να διατηρεί σταθερό το επίπεδο ενέργειας του σήματος που εξέρχεται απο τον LNA παρά τις πιθανές μεταβολές της ενέργειας του σήματος εισόδου. Με την τιμή 'manual' απενεργοποιούμε αυτο το κύκλωμα και θέτουμε εμείς το receive gain.
- **sdr.rx_hardwaregain_chan0 :** αρχικοποιούμε το receive gain της κεραίας του RX. Η μέγιστη τιμή είναι στα 71 dB ενώ για συχνότητες συντονισμού άνω των 4GHz η μέγιστη τιμή είναι στα 62 dB (επιλέγουμε υψηλό gain για καλύτερο sensitivity).

Παρακάτω παραθέτουμε τον αλγόριθμο του spectrum sensing, μετά την αρχικοποίηση του RX:

```
1    # Calculate noise floor by tuning SDR in a frequency with no
     transmission
2    sdr.rx_lo = int(780e6)
3    rx_samples = sdr.rx()
4
5    noise_floor = np.mean( psd(rx_samples)*np.blackman(len(psd(
     rx_samples))) )
6    print("**************************")
7    print("Noise floor: ", noise_floor)
8    print("**************************")
9    print("\n")
10   sdr.rx_lo = int(center_freq)
11
12   start_time = time.time()
13   while time.time() - start_time < 0.5:
14       rx_samples = sdr.rx()
15
16   psd_shifted = psd(rx_samples*np.blackman(len(rx_samples)))
17
18   fft_fr = np.fft.fftshift( np.fft.fftfreq(len(rx_samples), d=1/
     sample_rate) )
19
20   transmission_freq = 0
21   start = stop = 0
22
23   start = find_nearest(fft_fr , value = (-LORA_BW/2))
24   stop = find_nearest(fft_fr , value = (+LORA_BW/2))
25
26   transmission_freq = math.ceil(((fft_fr[stop] - fft_fr[start]) /
      2) + fft_fr[start]) + sdr.rx_lo
27   transmission_freq = round(transmission_freq / 1e6 , 1)
28
29   avg = np.mean(psd_shifted[start:stop])
30           toy
31   snr_dB = avg - noise_floor
32
33   if(snr_dB > 15):
34       print("transmission frequency: " + str(transmission_freq) +
      "MHz")
35       print("Average: " + str(round(avg, 2)) + " dB")
36       print("\n")
37
38   plt.figure(1)
39   plt.plot(fft_fr/1e6, psd_shifted)
40   plt.xlabel("Frequency [MHz]")
41   plt.ylabel("PSD")
42   plt.show()
```

Listing 1: spectrum sensing algorithm for LoRa transmission

Στην γραμμή 3, μέ την κλήση της συνάρτησης **sdr.rx()** ξεκινάει η λήψη δειγμάτων αποθηκεύοντας τα στον εσωτερικό buffer του sdr. Στην συνέχεια αφού γεμίσει ο buffer, στην γραμμή 5 υπολογίζεται ο μέσος όρος της ενέργειας του σήματος που προκύπτει μετά την κλήση της συνάρτησης psd() (η λειτουρία της περιγράφεται στο Κεφάλαιο **2.2**). Σε αυτό το σημείο αυτο που υπολογίζεται είναι το **noise floor** συντονίζοντας το sdr στην συχνότητα **780 MHz**, στην οποία δεν θα υπάρχει καμία μετάδοση. Στην γραμμή

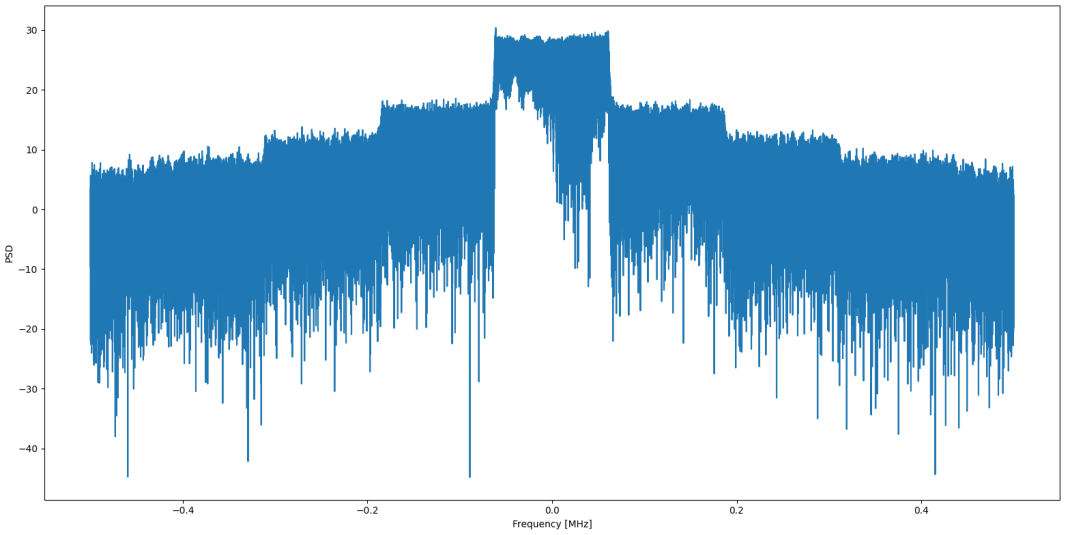
13, αφού συντονίσουμε το sdr στην επιθυμητή συχνότητα (Η μετάδοση LoRa πακέτων πραγματοποιείται στα **866.1 MHz**) λαμβάνει δείγματα αποθηκεύοντας τα στον buffer για 500 milliseconds. Μέσα σε αυτο το χρονικό περιθώριο ο buffer κάνει drop τις παλίες τιμές με το που γεμίσει και συνεχίζει να αποθηκεύει καινουργιες τιμές. Αυτό γίνεται για να έχουμε μια καλύτερη εικόνα για την σημπεριφορά και την σταθερότητα του επιπέδου ενέργειας του σήματος. Στις γραμμές 16-18 μετατρέπονται οι τιμές του σήματος σε dB (μέσω της συνάρτησης psd()) και αποθηκεύονται στην μεταβλητή **psd_shifted**. Επίσης, η συνάρτηση **np.fft.fftfreq()** επιστρέφει έναν πίνακα ο οποίος περιέχει τις συχνότητες στις οποίες «κάθεται» κάθε FFT bin του σήματος psd_shifted. Στις γραμμές 23-24 μέσω της συνάρτησης **find_nearest**, επιστρέφεται η θέση του στοιχείου του πίνακα **fft_fr** , του οποίου η τιμή είναι πολύ κοντά στην τιμή -LORA_BW/2 . Το ίδιο γίνεται και για την τιμή LORA_BW/2. Ο κώδικας της find_nearest παρουσιάζεται παρακάτω:

```
# This function returns the index of the element in array that
    contains
# the value which is closer to the given number
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return(idx)
```

Οι τιμές που επιστρέφονται αποθηκεύονται στις τιμές start , stop. Με αυτον τον τρόπο εστιάζουμε στα bins εντός του LORA_BW το οποίο είναι το εύρος συχνοτήτων που καλύπτει ένα LoRa σήμα και είναι ίσο με 125 kHz (για το συγκεκριμένο LoRa modulation που εξετάζουμε). Αυτο πραγματοποιείται με την εντολή **psd_shifted[start:stop]**. Στην συνέχεια στις γραμμές 26 - 42 υπολογίζεται η κεντρική συχνότητα μετάδοσης με βάση τις τιμές των συχνοτήτων στις θέσεις που υπολογίσαμε προηγουμένως. Στην συνέχεια υπολογίζουμε την μέσο επίπεδο ενέργειας του σήματος και υπολογίζουμε το SNR του. Η σχέση του SNR στην γραμμη 31 εκφράζεται απο την διαφορά της μέσης τιμής της ενέργειας του σήματος (σε dB) και του noise floor που υπολογίσαμε στην αρχή της εκτέλεσης (σε dB). Τέλος, άμα η τιμή του SNR είναι μεγαλύτερη του 15, θεωρούμε ότι υπάρχει μετάδοση και εκτυπώνουμε τις απαραίτητες πληροφορίες για το σήμα και το αναπαριστούμε γραφικά συναρτήσει της ενέργειας του και της συχνότητας. Τα αποτελέσματα της εκτέλεσης παρατίθονται παρακάτω:

```
root@hlias:~/spectrum_analyzer# python3 spectrum_sensing_software.py lora
**************************
Noise floor:  -10.610986802359259
**************************


transmission frequency: 866.1MHz
Average: 22.74 dB
```

Σχήμα 4: printed results after execution



Σχήμα 5: received LoRa signal with PlutoSDR tuned in 866.1 MHz

4 Final Implementation of spectrum sensing software

Με βάση την δομή του spectrum sensing αλγορίθμου που αναλύσαμε παραπάνω, δημιουργήθηκαν 2 επιπλέον συναρτήσεις fm και wifi, οι οποίες πραγματοποιούν spectrum sensing πάνω στο εύρος συχνοτήτων που καλύπτει το **FM** και το **Wi-Fi** πρωτόκολλο αντίστοιχα.

4.1 Structure of FM spectrum sensing algorithm

4.1.1 Initialization

Η διαδικασία αρχικοποίησης των μεταβλητών της RX πλευράς του SDR όπως και ο υπολογισμός του noise floor είναι ακριβώς ίδια με αυτη του LoRa πρωτοκόλλου, με την διαφορά ότι αλλάζουν οι συχνότητες συντονισμού:

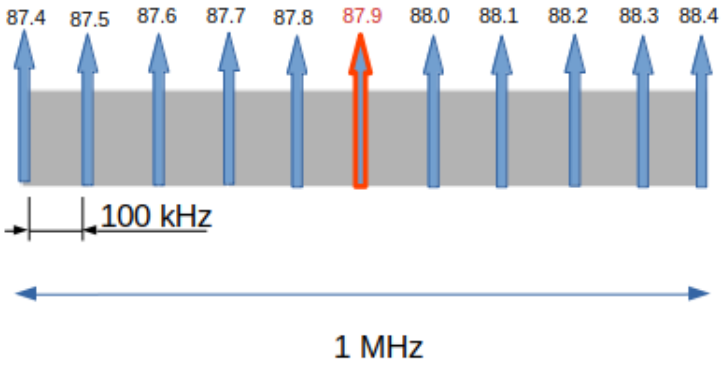
```
def fm_band(sdr, left_limit, right_limit):
    sample_rate = 1e6 # Hz
    center_freq = 87.9e6 # Hz
    num_samps = 100000 # number of samples per call to rx()

    # Initialize LP filter dictionary and lists for various
    # time measurements
    dict = {}
    single_transm_time = []
```

```
    whole_scan_time = []
    rx_lo_change_time = []
    rx_lo_change_freq = []
    sampling_time = []
    frequency_table = []
    fm_signals = []
    trans_freq_table = []
    counter = 0

    # Config Rx
    sdr.sample_rate = int(sample_rate)
    sdr.rx_lo = int(center_freq)
    sdr.rx_rf_bandwidth = int(20e6)
    sdr.rx_buffer_size = num_samps
    sdr.gain_control_mode_chan0 = 'manual'
    sdr.rx_hardwaregain_chan0 = 64.0 # dB, increase to increase the
     receive gain, but be careful not to saturate the ADC

    # Calculate noise_floor by tuning SDR in a frequency with no
    transmission
    sdr.rx_lo = int(82e6)

    rx_samples = sdr.rx()
    rx_samples = rx_samples*np.blackman(len(rx_samples))
    noise_table = psd(rx_samples)
    noise_floor = np.mean(noise_table)
    print("**************************")
    print("Noise floor: ", noise_floor)
    print("**************************")
    print("\n")
```

Οι παράμετροι **left_limit** και **right_limit** της συνάρτησης είναι οι ακραίες τιμές του εύρος συχνοτήτων που θέλει να εξετάσει ο χρήστης εντός του FM, το οποίο καλύπτει συχνότητες απο **87.5 MHz** εώς **108.0 MHz**. Επίσης στις γραμμές 10-17 οι λίστες που αρχικοποιούνται χρησιμοποιούνται για μετρήσεις χρόνων τους οποίους θα παρουσιάσουμε γραφικά σε παρακάτω υποκεφάλαιο. Στο επόμενο υποκεφάλαιο παρουσιάζεται η λογική και ο κώδικας του spectrum sensing.

4.1.2 Spectrum sensing algorithm

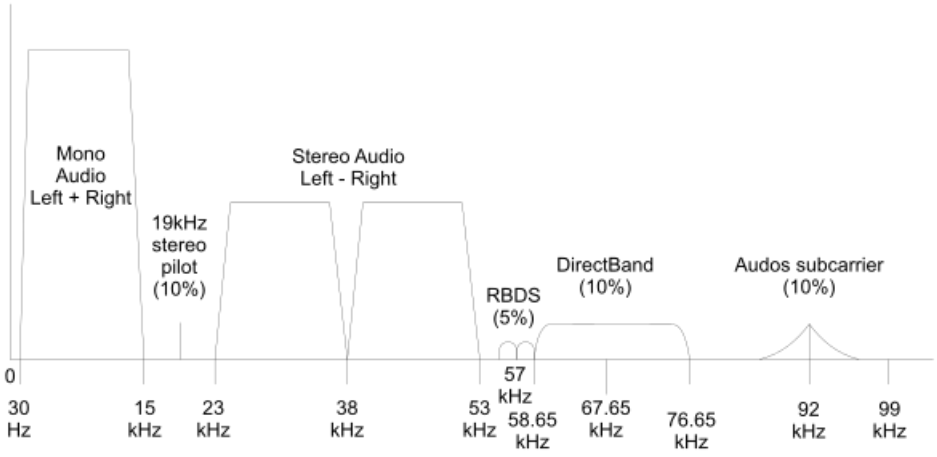
Επειδή στο FM κάθε πιθανή συχνότητα μετάδοσης απέχει 100 kHz απο τις γειτονικές της, αρχικοποιείται ένας πίνακας ο οποίος περιέχει τιμές συχνοτήτων εντός του εύρους sample_rate (αφου το sdr λαμβάνει δείγματα μόνο εντός του sample_rate) οι οποίες ισαπέχουν κατα 100 kHz μεταξύ τους. Δηλαδή, για sample rate = 1 MSps ο πίνακας που προκύπτει είναι ο εξής: f = [-500 kHz, -400 kHz, ... , 500 kHz] . Άμα προσθέσουμε στον πίνακα την αρχική κεντρική συχνότητα (center_freq = 87.9 MHz), προκύπτει το εξής αποτέλεσμα:



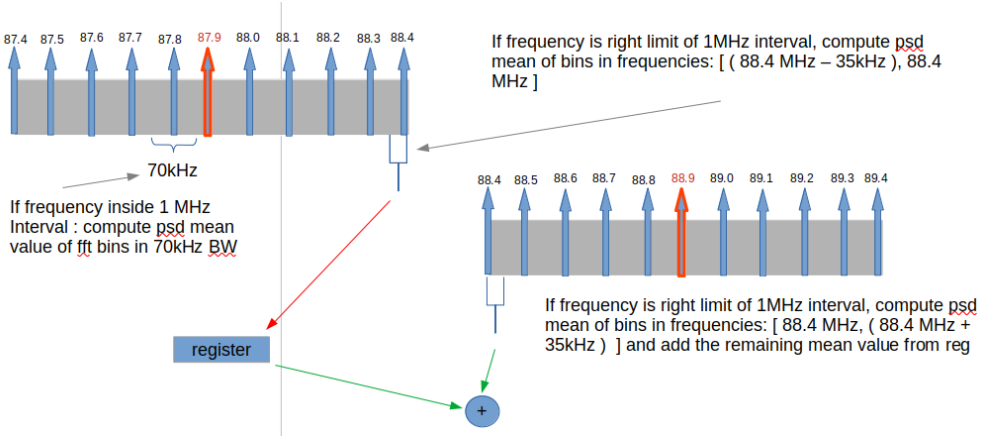
Επίσης αρχικοποιείται ένα dictionary στο οποίο αποθηκεύονται οι ακραίες τιμές του bandwidth που καλύπτει ένα FM σήμα για κάθε συχνότητα του πίνακα f. Με αυτόν τον τρόπο, μπορούμε να τοποθετήσουμε ένα low pass filter μεγέθους FM_BW ώστε να κρατήσουμε τα fft bins εντός του φίλτρου για να ελένξουμε άμα υπάρχει μετάδοση σε κάθε συχνότητα του πίνακα. Το κομμάτι του κώδικα που κάνει τις εν λόγω αρχικοποιήσεις είναι το εξής:

```
    # Number of center frequencies (equal spaced) within 1MHz
    num = int((sample_rate/100e3) + 1)

    #Create an equal spaced array of frequencies within len(
    sample_rate)
    f = np.linspace(-sdr.sample_rate/2, sdr.sample_rate/2, num)

    # Get array of frequency values of each FFT bin
    fft_fr = np.fft.fftfreq(len(rx_samples), d=1/sample_rate)
    fft_fr = np.fft.fftshift(fft_fr)

    # Fill dictionary with 70kHz low_pass filters for each
    frequency within 1MHz analysis
    for i in range(num):
        if (i == 0):
            stop = find_nearest(fft_fr, value = (f[i] + FM_BW/2))
            dict[i] = list()
            dict[i].append(stop)

        elif (i == num - 1):
            start = find_nearest(fft_fr, value = (f[i] - FM_BW/2))
            dict[0].insert(0,start)

        else:
            start = find_nearest(fft_fr, value = (f[i] - FM_BW/2))
            stop = find_nearest(fft_fr, value = (f[i] + FM_BW/2))
            dict[i] = list()
            dict[i].append(start)
            dict[i].append(stop)

    sdr.rx_lo = int(center_freq)
```

Η λογική του αλγορίθμου είναι να ελέγχει όλες τις συχνότητες του πίνακα **f + center_freq** (ο οποίος θα έχει σταθερό μέγεθος ισό με το sample_rate) ξεκινώντας με την τιμή center_freq = 87.9 MHz, και μόλις ολοκληρωθεί ο έλεγχος να αλλάζει η συχνότητα συντονισμού στην τιμή center_freq = center_freq + sample_rate ώστε να εξετάσουμε το αμέσως επόμενο εύρος συχνοτήτων. Η επαναληπτική διαδικασία σταματάει όταν η συχνότητα που εξετάζουμε είναι ίση με το δεξί όριο που έχει θέσει ο χρήστης στην αρχή της εκτέλεσης. Να σημειωθεί ότι για την εκτέλεση του αλγορίθμου έχει επιλεχθεί τιμή του **sample_rate = 1 MSps** και **fm signal bandwidth = 70 kHz**. Η τιμή του bandwidth καθορίστηκε απο την φύση του fm σήματος η οποία είναι η εξής:



Πρίν παρουσιάσουμε τον κώδικα του αλγορίθμου , παραθέτουμε ένα διάγραμμα το οποίο απεικονίζει την εκτέλεση του αλγορίθμου για τις πρώτες δυο επαναλήψεις:



Στην πρώτη επανάληψη, δεν έχει νόημα να εξετάσουμε την αριστερότερη συχνότητα του πίνακα, καθώς είναι εκτός ορίων του FM. Για τις ενδιάμεσες τιμές του πίνακα f + center_freq υπολογίζουμε το μέσο επίπεδο ενέργειας των FFT bins εντός του εύρους FM_BW = 70kHz. Στην τελευταία συχνότητα του πίνακα, υπολογίζεται το μέσο επίπεδο ενέργειας των FFT bins εντός του διαστήματος [f[i] - FM_BW/2) , f[i]]. Ο μέσος όρος αποθηκεύεται σε έναν register ο οποίος θα χρησιμοποιηθεί στην επόμενη επανάληψη. Στην δεύτερη επανάληψη (αφού έχουμε αλλάξει την συχνότητα συντονισμού σε center_freq + sample_rate), υπολογίζουμε το μέσο επίπεδο ενέργειας των FFT bins εντός του διαστήματος [f[i] , (f[i] + FM_BW/2)] και το προσθέτουμε με τον register που έχει αποθηκευμένη την μέση τιμή των υπόλοιπων bins απο την προηγούμενη επανάληψη. Με την πρόσθεση των δυο τιμών μπορούμε να ελέγξουμε πλέον αν η συγκεκριμένη συχνότητα είναι συχνότητα μετάδοσης. Η διαδικασία συνεχίζεται μέχρι να πετύχουμε το right_limit που έχει θέσει ο χρήστης. Η απόφαση για το άμα υπάρχει μετάδοση καθορίζεται με το SNR με τον ίδιο τρόπο που υπολογίζεται στην περίπτωση του LoRa πρωτοκόλλου. Παρακάτω παραθέτουμε τον κώδικα του αλγορίθμου:

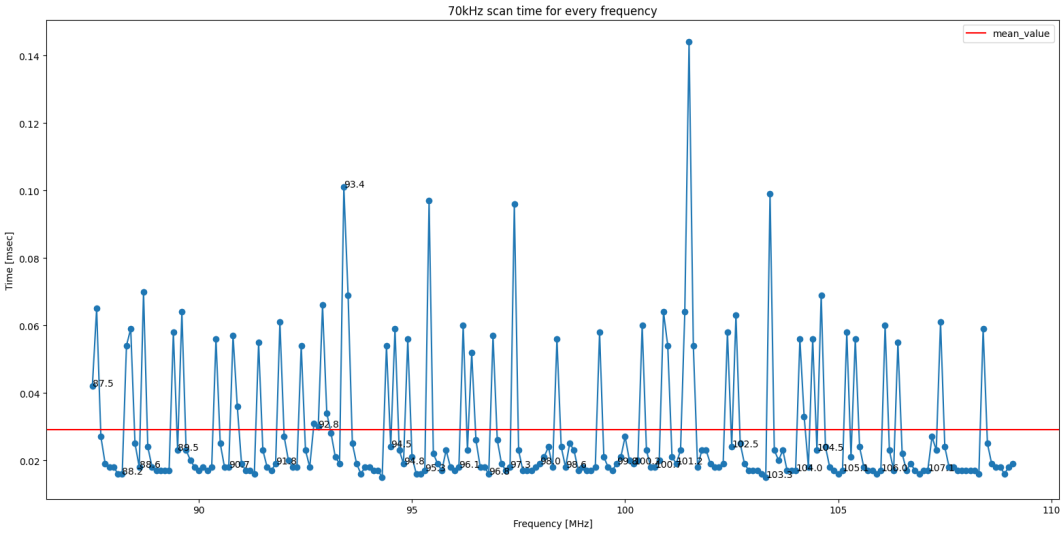
```
    reg_end = 0
    transmission_freq = 0
    reg_time = 0

    while True:

        rx_lo_change_freq.append(87.9 + counter)
        rcv_time = time.time()

        # Receive I, Q data for 500msec
        start = datetime.datetime.now()
        while time.time() - rcv_time < 0.5:
            rx_samples = sdr.rx()

        # Apply Blackman window in order to avoid sudden
        # transitions between the first and the last sample
        rx_samples = rx_samples*np.blackman(len(rx_samples))

        # Calculate fft and psd
        psd_shifted = psd(rx_samples)

        end_time = (datetime.datetime.now() - start)
        sampling_time.append( end_time.total_seconds()*1000 )

        store_time = list()

        # Conduct 70kHz analysis for every frequency
        # within 1MHz bandwidth
        for i in range(num):

            #left limit of frequency array
            if(i == 0):

                if(reg_end == 0):
                    continue

                start_1 = datetime.datetime.now()

                # Compute the psd mean value of samples within
                # [0:35e3] kHz. Add the psd mean value of the
```

```
                # remaining samples stored in the register reg_end
                # in order to complete the analysis
                # for this frequency point.
                stop = dict[i][1]
                avg = np.mean(psd_shifted[0:stop]) + reg_end

                start_1 = datetime.datetime.now() - start_1
                start_1 = start_1.total_seconds()*1000
                single_transm_time.append(start_1 + reg_time)
                store_time.append(single_transm_time[-1])

                # Get center frequency of the fft bins
                # within the 70kHz bandwidth
                transmission_freq = math.ceil((fft_fr[0])
                                              + sdr.rx_lo)
                transmission_freq = round(transmission_freq / 1e6
                                          , 1)
                frequency_table.append(transmission_freq)

            #right limit of frequency array
            elif(i == num - 1):
                reg_time = datetime.datetime.now()

                start = dict[0][0]

                # Compute the psd mean value for the elements
                # psd_shifted[(num_samps - 35e3):num_samps]
                # and store the mean value to the register reg_end.
                # In the next 1MHz interval, the register
                # will be added with the psd mean value of the
                # fft bins within [0:35e3] Hz starting from the
                # left limit of the new interval
                reg_end = np.mean( psd_shifted[start:num_samps] )

                reg_time = datetime.datetime.now() - reg_time
                reg_time = (reg_time).total_seconds()*1000
                continue

            # if 0 < i < num conduct 70kHz scan for this frequency
            else:

                start_1 = datetime.datetime.now()

                start = dict[i][0]
                stop = dict[i][1]

                avg = np.mean(psd_shifted[start:stop])

                start_1 = (datetime.datetime.now() - start_1)
                start_1 = start_1.total_seconds()*1000
                single_transm_time.append(start_1)
                store_time.append(single_transm_time[-1])

                # Get center frequency of the fft bins
                # within the 70kHz bandwidth

                transmission_freq = math.ceil(((fft_fr[stop] -
```

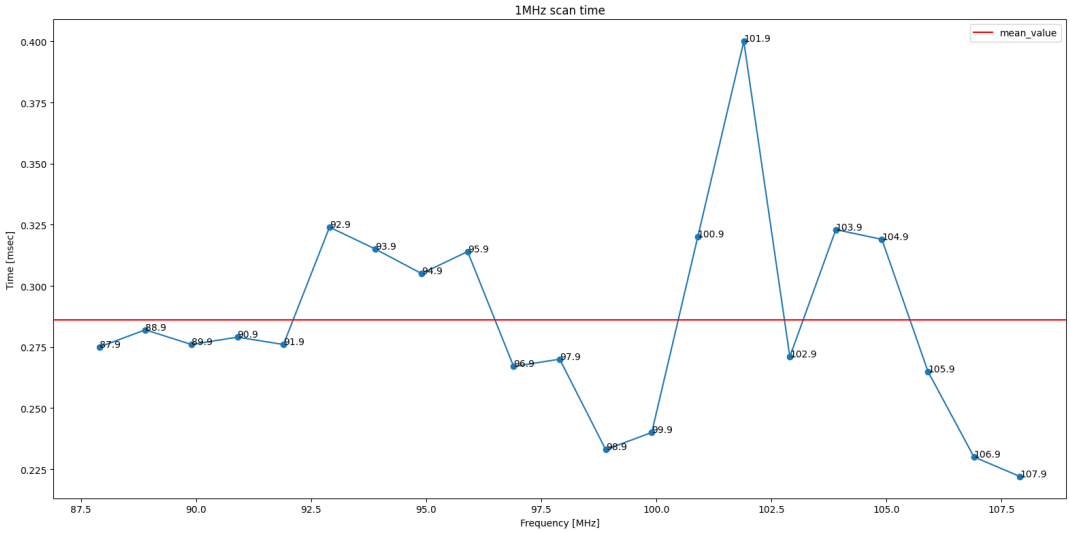
```
                                fft_fr[start]) / 2)  + fft_fr[start])

                transmission_freq += sdr.rx_lo

                transmission_freq = round(transmission_freq/1e6,1)
                frequency_table.append(transmission_freq)

            # Analyze only frequencies which are equal or greater
            # than the given left limit of the total interval
            if( transmission_freq < float(left_limit) ):
                continue

            # Right limit of total interval reached.
            # plot time tables before ending the execution
            elif( transmission_freq > float(right_limit) ):

                """printing time metrics
                            . . .
                            . . .
                  plot time graphs
                            . . .
                            . . .
                            . . . """
                sys.exit()

            # Compute Signal to Noise Ratio (SNR)
            # for a given signal. It is expressed in decibels

            snr_dB = avg - noise_floor

            # If SNR is greater than the given threshold,
            # we decide that the level of our signal in this
            # frequency is high enough to be considered
            # as a transmission signal.
            if(snr_dB > 15):

                trans_freq_table.append(transmission_freq)
                fm_signals.append(round(avg, 2))
                print("transmission frequency: "
                        + str(transmission_freq) + "MHz")
                print("snr_dB: ", snr_dB)
                print("Average: " + str(round(avg, 2)) + " dB")
                print("\n")

        whole_scan_time.append(round(np.sum(store_time), 3))

        # tune SDR to the next center frequency,
        # which is centered to the 1MHz interval.
        counter += int(sample_rate/1e6)
        next = (87.9 + counter)*1e6

        start = datetime.datetime.now()
        sdr.rx_lo = int(next)
        rx_lo_change_time.append( (datetime.datetime.now() - start)
    .total_seconds()*1000 )
```

4.1.3 Executions and metrics

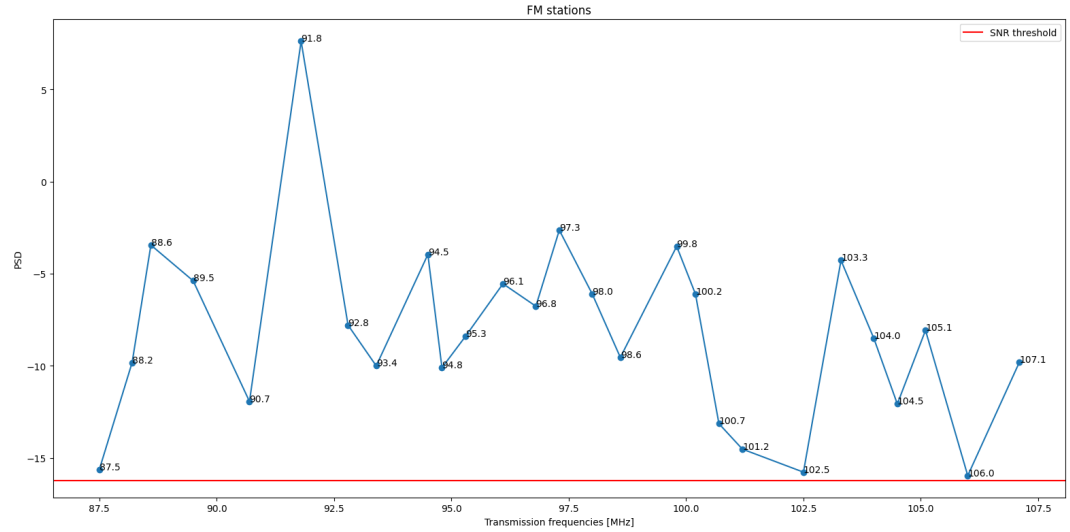
Παρακάτω παρατίθονται τα διαγράμματα που παράγονται πρίν απο τον τερματισμό του προγράμματος:



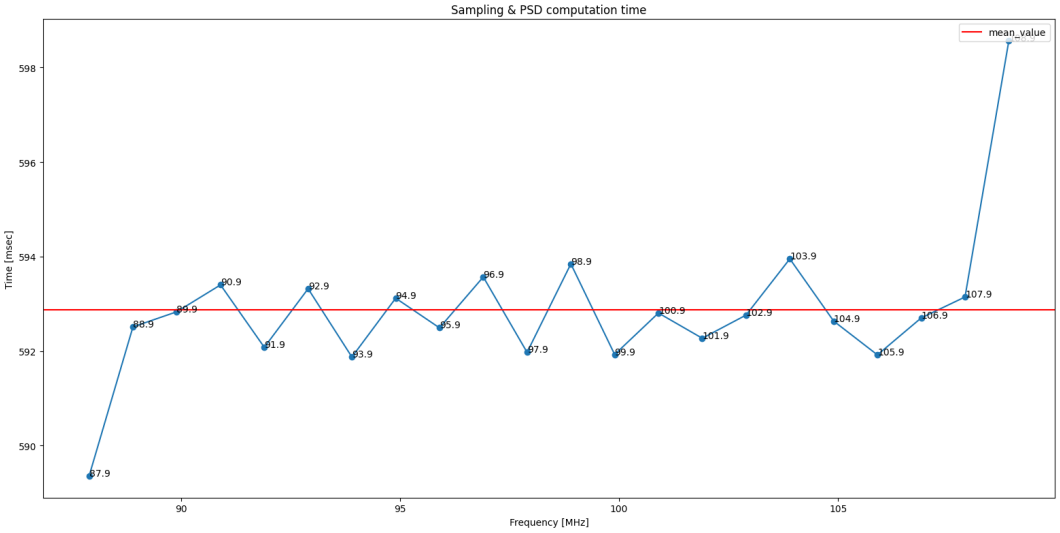
Σχήμα 6: 70 kHz scan time for every frequency that has been iterated



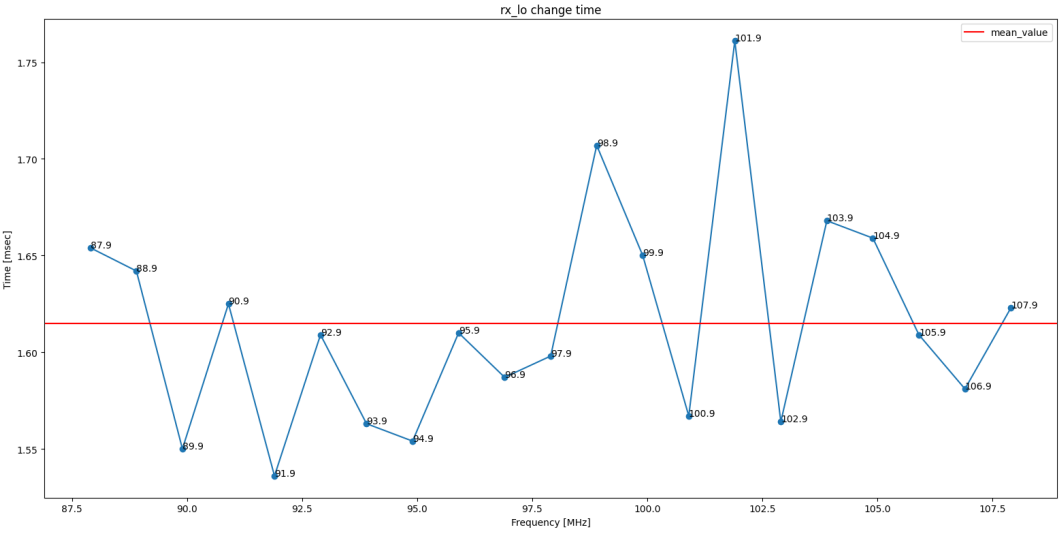
Σχήμα 7: How much time it takes to scan each 1 MHz interval. The indicated frequencies are the center of each interval



Σχήμα 8: FM stations and their respective average energy level, the bottom line is the SNR threshold

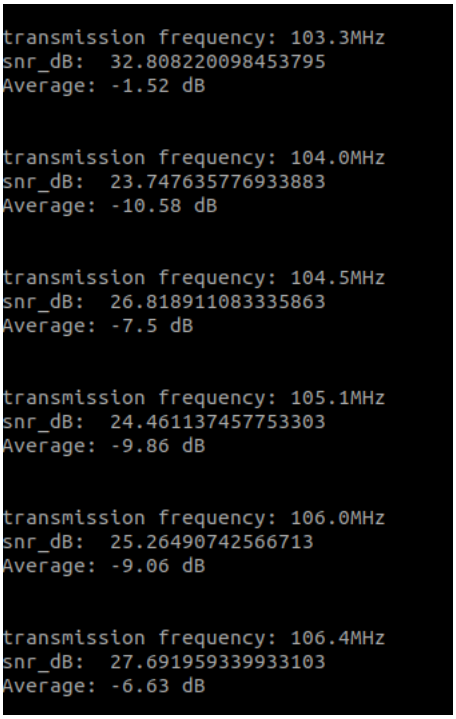


Σχήμα 9: ow much time it takes to receive samples for 500 msec and convert signal in dB



Σχήμα 10: How much time it takes to change SDR center frequency

Τέλος, παρακάτω απεικονίζεται ένα κομμάτι της εκτέλεσης του κώδικα κατά την διαδικασία επιλογής υποψήφιων ραδιοφωνικών σταθμών:



Παρατήρηση : Τα αποτελέσματα των υποψήφιων συχνοτήτων δεν είναι πάντα τα ίδια, καθώς η λήψη σημάτων απο την κεραία εξαρτάται απο διάφορες παραμέτρους (ποιότητα κεραίας, φυσικά εμπόδια, υψόμετρο κτλ).

4.2 Structure of Wi-Fi spectrum sensing algorithm

4.2.1 Initialization

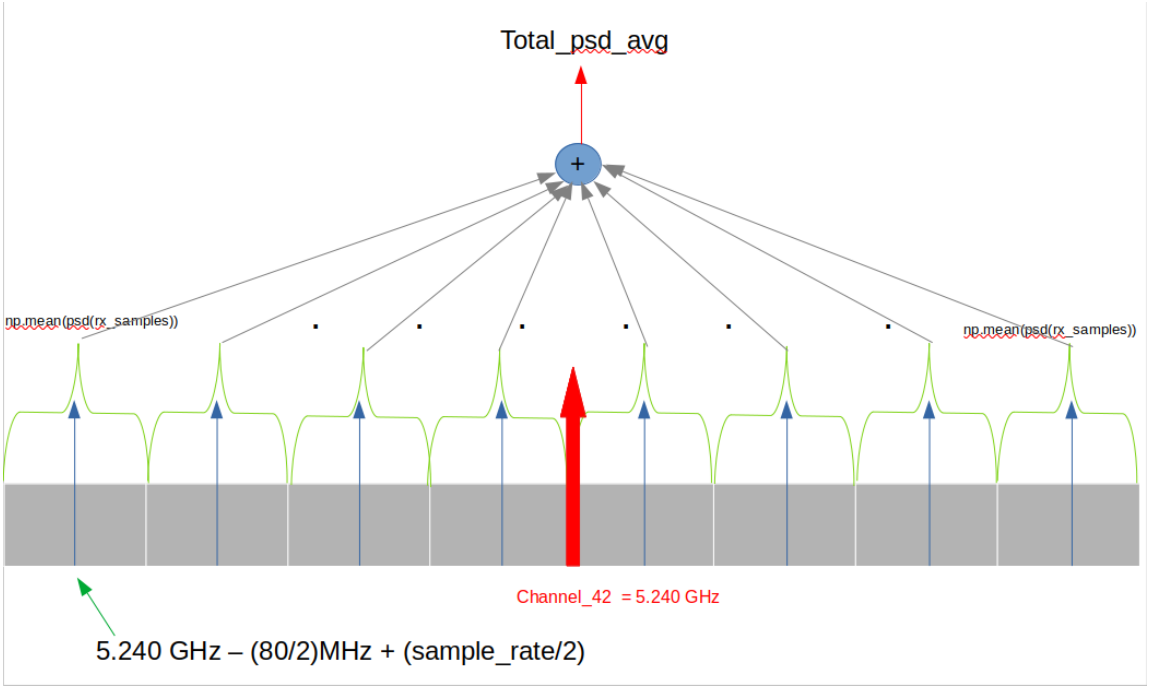
Η αρχικοποίηση της RX πλευράς του SDR είναι σχεδόν ίδια με αυτή του FM αλγορίθμου, με την διαφορά ότι αλλάζει η μεταβλητή sdr.rx_hardwaregain_chan0, καθώς για συχνότητες συντονισμού άνω των 4 GHz, το μέγιστο gain που υποστηρίζει το PlutoSDR είναι 62 dB. Επίσης, επειδή η διεξαγωγή πειραμάτων πραγματοποιήθηκε στην μπάντα των 5 GHz , αλλάζει και η τιμή του sample_rate καθώς το κάθε κανάλι έχει εύρος 80 MHz. Η τιμή που λαμβάνει το sample rate θα εξηγηθεί στο υποκεφάλαιο όπου θα περιγράψουμε το sensing. Η αρχικοποίηση απεικονίζεται παρακάτω:

```
def wifi_band(sdr):

    # This dictionary contains all the wi-fi channels that participated in wi-fi experiments and analysis.
    wifi_channels = {"channel_1" : 2.412e9 ,
                     "channel_2" : 2.417e9 ,
                     "channel_3" : 2.422e9 ,
                     "channel_4" : 2.427e9 ,
                     "channel_5" : 2.432e9 ,
                     "channel_6" : 2.437e9 ,
                     "channel_7" : 2.442e9 ,
                     "channel_8" : 2.447e9 ,
                     "channel_9" : 2.452e9 ,
                     "channel_10" : 2.457e9 ,
                     "channel_11" : 2.462e9 ,
                     "channel_12" : 2.467e9 ,
                     "channel_13" : 2.472e9 ,
                     "channel_42" : 5.210e9 ,
                     "channel_48" : 5.240e9 ,
                     "channel_106": 5.530e9 ,
                     "channel_46" : 5.220e9 ,
                     "no_signal_5g" : 5.400e9}

    sample_rate = 10e6 # Hz
    num_samps = 100000 # number of samples per call to rx()
    sdr.sample_rate = int(sample_rate)

    # Config Rx
    sdr.rx_rf_bandwidth = int(20e6)
    sdr.rx_buffer_size = num_samps
    sdr.gain_control_mode_chan0 = 'manual'
    sdr.rx_hardwaregain_chan0 = 50.0  # dB, increase to increase the receive gain, but be careful not to saturate the ADC

    sdr.rx_lo = int(5.400e9)

    # Calculate noise_floor by tuning SDR in a frequency with no transmission
    rx_samples = sdr.rx()

    noise_floor = np.mean(psd(rx_samples))
    print("**************************")
    print("Noise floor: ", noise_floor)
    print("**************************")
    print("\n")

    # Get array of frequency values of each FFT bin
    fft_fr = np.fft.fftshift( np.fft.fftfreq(len(rx_samples), d=1/sample_rate) )

    # Initialize values and lists for time measurements
    total_psd = 0
    time_axis = []
    avg_psd_axis = []
    sampling_time = []
    total_exec_time = []
    latency_time = 0
```

Το dictionary που αρχικοποιείται στην αρχή της συνάρτησης περιέχει όλες τις συχνότητες που εξετάστηκαν στο Wi-Fi. Επειδή όμως στην πράξη στην μπάντα των 2.4GHz υπήρχε πολύ μεγάλο interference σε κάθε κανάλι, τα τελικά πειράματα διεξάχθηκαν στην μπάντα των 5GHz.

4.2.2 Spectrum sensing algorithm

Το παρακάτω διάγραμμα απεικονίζει τον τρόπο με τον οποίο διεξάγεται το sensing για ένα κανάλι στην μπάντα των 5 GHz (συγκεκριμένα το κανάλι 42 με κεντρική συχνότα 5.240 GHz), το οποίο καλύπτει εύρος συχνοτήτων ίσο με 80 MHz. Το sample rate αρχικοποιείται στα 10 MSps .



Αρχικά, τεμαχίζουμε το εύρος των 80 MHz (γκρί γραμμή στο διάγραμμα) σε (80 MHz) / (sample_rate) καθώς το Pluto SDR δεν υποστηρίζει τόσο μεγάλο εύρος συχνοτήτων (μέγιστη τιμή sample rate = 56 MHz). Για το sensing επιλέξαμε αρχικά sample_rate = 10 MSps, συνεπώς το bandwidth του καναλιού χωρίζεται σε 8 υποσύνολα , καλύπτοντας εύρος sample_rate = 10 MHz το καθένα. Μετά πηγαίνουμε επαναληπτικά και λαμβάνουμε samples για όλο το εύρος του κάθε υποσυνόλου (10 MHz), ξεκινώντας με αρχική συχνότητα συντονισμού **center_freq = 5.240 GHz - (80/2)MHz + (sample_rate/2) MHz**, όπως φαίνεται και στο διάγραμμα. Σε κάθε υποσύνολο υπολογίζεται η μέση τιμή του επιπέδου ενέργειας των FFT bins και προστίθεται στην μεταβλητή total_psd_avg. Μολίς ολοκληρωθεί η ανάλυση του υποσυνόλου, συντονίζουμε το SDR σε νέα συχνότητα με τιμή center_freq = center_freq + (sample_rate/2) . Η επαναληπτική διαδικασία σταματάει όταν έχουμε καλύψει όλο το εύρος του καναλιού. Στο τέλος του αλγορίθμου, η μεταβλητή total_psd_avg θα περιέχει την συνολική μέση

τιμή όλων των FFT bins. Τέλος, επειδή στο τέλος του sensing θα έχουμε απλά μία μέση τιμή, Ο αλγόριθμος είναι εμφωλευμένος μεσα σε μία While η οποία είναι ενεργή για όσα δευτερόλεπτα θέσει ο χρήστης. Κάθε υπολογισμός του συνολικού μέσου όρου ενέγειας αποθηκεύεται σε μία λίστα. Με αυτόν τον τρόπο μετά το πέρας του συγκεκριμένου χρονικού διαστήματος, μπορούμε να παράγουμε ένα διάγραμμα το οποίο δείχνει την συνολική συμπεριφορά του καναλιού συναρτήσει του χρόνου. Ο κώδικας που υλοποιεί όλα τα παραπάνω είναι ο εξής:

```
# Iterate dictionary
    for key, value in wifi_channels.items():

        # If key is equal to the desired channel, start analysis
        if(key == "channel_42"):
            shift_value = 0
            temp = value + (-CH_48_BW / 2) + (sample_rate / 2)

            start = datetime.datetime.now()
            store_time = list()

            #Conduct 80MHz analysis for n seconds
            while (datetime.datetime.now() - start).total_seconds()
    < 60:
                total_psd = 0
                shift_value = 0
                start_exec = time.time()

                # Compute average psd of all the samples within
                # sample_rate range. Repeat until the whole wi-fi
                # channel bandwidth is analyzed
                while CH_48_BW - shift_value > 0:

                    center_freq =  temp + shift_value
                    sdr.rx_lo = int(center_freq)

                    start_time = time.time()

                    if(latency_time):
                        store_time.append(time.time()
                                          - latency_time)

                    sample_time = time.time()

                    rx_samples = sdr.rx()

                    latency_time = time.time()

                    sample_time = time.time() - sample_time
                    sampling_time.append(sample_time)

                    total_psd += np.mean(psd(rx_samples))

                    shift_value += sample_rate


                time_axis.append( (datetime.datetime.now()
                                   - start).total_seconds() )
                avg_psd_axis.append(total_psd / (
```

```
                                        CH_48_BW / sample_rate))
                total_exec_time.append(time.time() - start_exec)

                # After channel analysis, print
                # some critical time values for evaluation
                print("channel: " + str(key) + " signal_power: "
                 + str(total_psd / (CH_48_BW / sample_rate)) +'\n')

                print("simple latency: ", store_time[0])

                print("latency: ",
                        np.sum(store_time[0:int((CH_48_BW /
                                        sample_rate))]))

                print("Average sampling time: ",
                                np.mean(sampling_time))

                print("Total execution time: ",
                                np.mean(total_exec_time))

            # Plot channel behaviour for elapsed time of n seconds
            plt.figure(1)
            plt.plot(time_axis, avg_psd_axis)
            plt.xlabel("Elapsed_time sec")
            plt.ylabel("PSD")
            plt.show()
```

Πριν προχωρήσουμε στην επίδειξη των πειραμάτων που διεξάχθηκαν και των αντοίστιχων μετρικών τους, παρατηρήθηκε σημαντική αλλαγή στην απόδοση του αλγορίθμου όταν αλλάξαμε την τιμή του sample_rate απο 10 MSps σε 20 MSps. Ο πίνακας που ακολουθεί περιέχει κάποιες σημαντικές χρονικές μετρικές κατά την εκτέλεση του αλορίθμου (για ένα σκανάρισμα των 80 MHz) με sample rate = 10 MSps και 20 MSps:

metrics	10 Msps	20 Msps
simple_latency	7.2 msec	7.3 msec
total_latency	49.11 msec	21.4 msec
sample_time	19.06 msec	19.12 msec
80MHz_scan_time	209.09 msec	104.5 msec

- **simple_latency :** Ο κενός χρόνος μεταξύ δυο διαδοχικών sdr.rx() κατά τον οποίο διεξάγονται άλλοι υπολογισμοί.
- **total_latency :** το άθροισμα των κενών χρόνων.
- **sample_time :** χρόνος εκτέλεσης μίας εντολής sdr.rx().
- **80_MHz_sacn_time :** Συνολικός χρόνος ανάλυσης των 80 MHz.

Παρατηρούμε τα εξής:

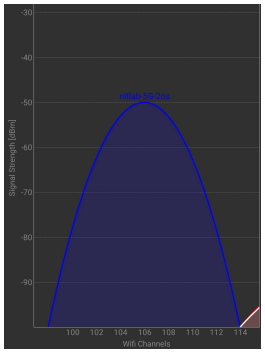
- Ο χρόνος που απαιτείται για να γεμίσει ο εσωτερικός buffer μπορεί να θεωρηθεί ίδιος (πολύ μικρή απόκλιση) με την διαφορά ότι στην περίπτωση των 20 MSps λαμβάνουμε διπλάσιο αριθμό δειγμάτων στον ίδιο χρόνο!
- Ο χρόνος όπου δεν «ακούει» το SDR στο κανάλι (total_latency) στην περίπτωση των 20 MSps είναι ο μισός απο την περίπτωση των 10 MSps.
- Στον μισά του χρόνου εκτέλεσης των 10 MSps, ο αλγόριθμος με sample_rate = 20 MSps έχει ολοκληρώσει την εκτέλεση του και μάλιστα έχοντας συλλέξει τον διπλάσιο αριθμό δειγμάτων.

Απο τα παραπάνω καταλαβαίνουμε ότι στην περίπτωση των 20 MSps έχουμε καλύτερη «ανάλυση» του σήματος όπως και καλύτερη είκονα της συμπεριφοράς του καναλιού σε συνάρτηση με τον χρόνο. Τα παραπάνω συμπεράσματα επαληθεύτηκαν με τα πειράματα που θα παρουσιάσουμε στο επόμενο υποκεφάλαιο.

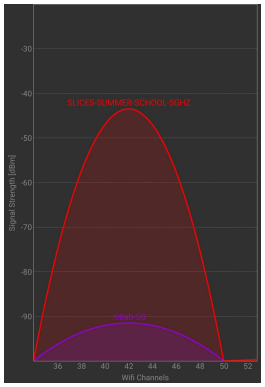
4.2.3 Executions and metrics

Τα πειράματα στην μπάντα των 5 GHz πραγματοποιήθηκαν στο NITlab στα κανάλια με τις εξής προδιαγραφές:

- **channel 42:** center frequency: 5.240 GHz, SSID: SLICES-SUMMER-SCHOOL-5GHZ
- **channel 106:** center frequency: 5.530 GHz, SSID: nitlab-5G-2os
- 5.400 GHz: 5G noise

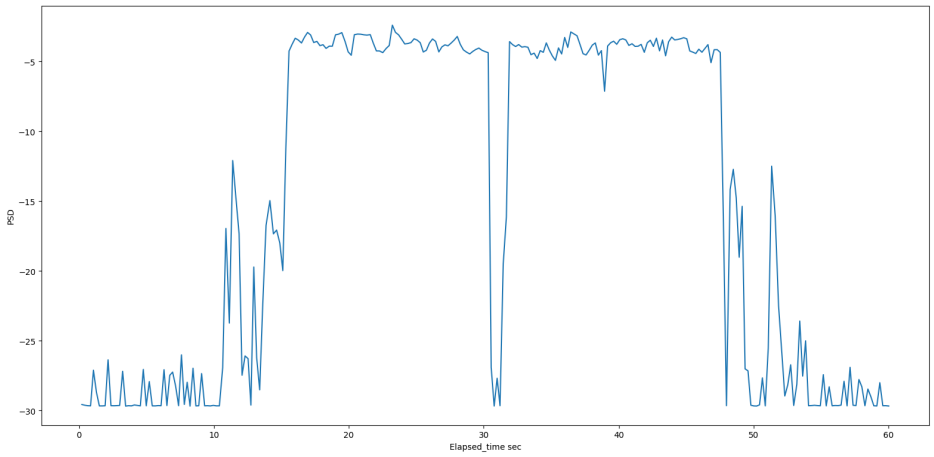


channel 106

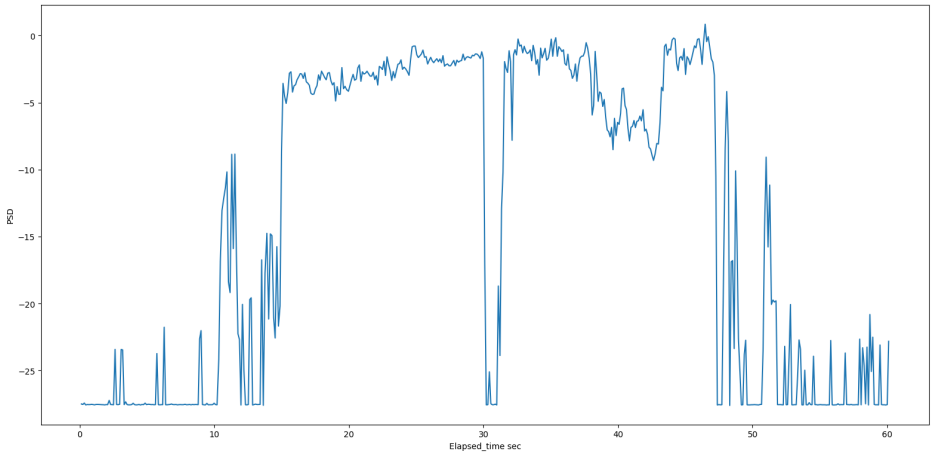


channel 42

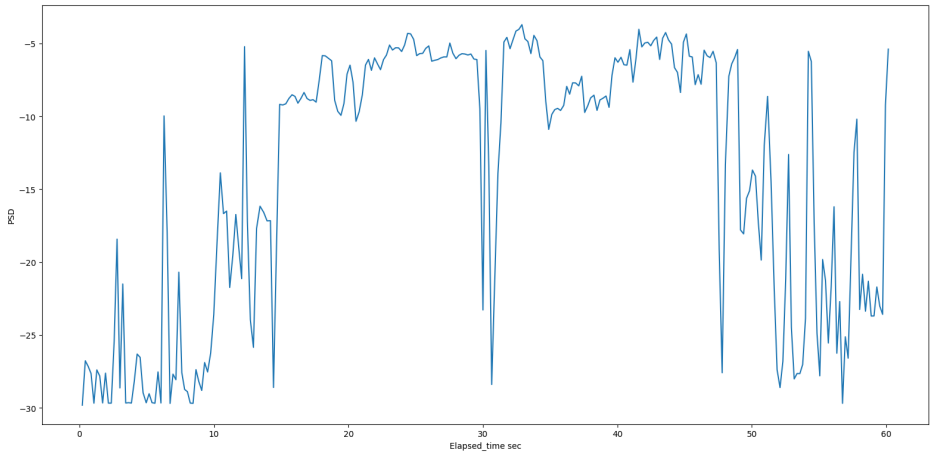
Αρχικά ελέγχθηκε η συμπεριφορά των καναλιών 42 και 106 κατα την εκτέλεση ενός speed test. Η εκτέλεση του κώδικα έγινε για 60 δευτερόλεπα με sample rate = 10 MSps και sample rate = 20 MSps:



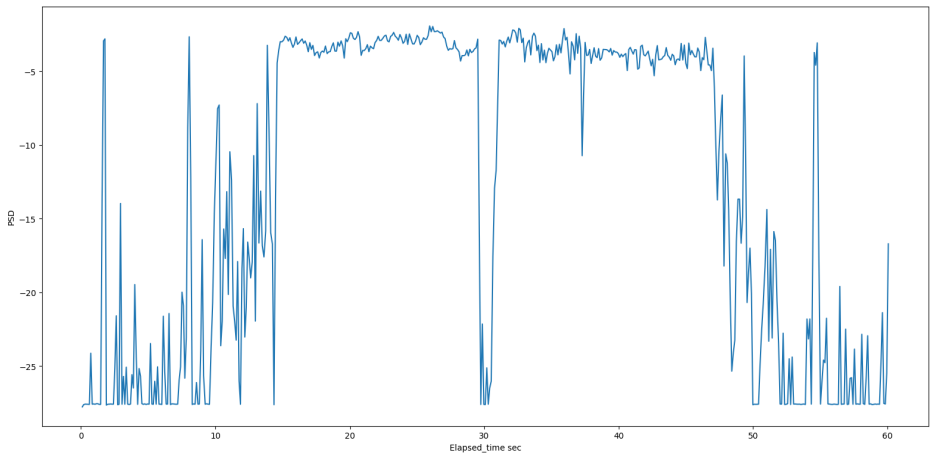
Σχήμα 11: channel 42: Performing speed test with sample rate = 10MSps



Σχήμα 12: channel 42: Performing speed test with sample rate = 20MSps



Σχήμα 13: channel 106: Performing speed test with sample rate = 10MSps

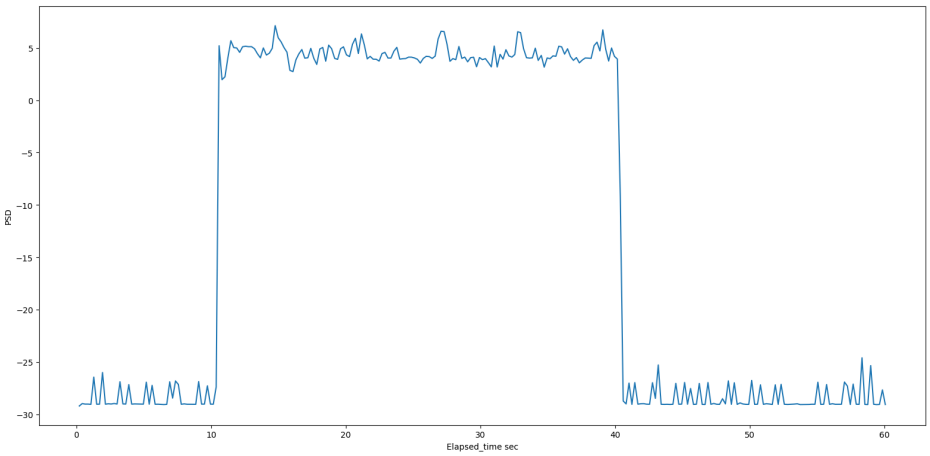


Σχήμα 14: channel 42: Performing speed test with sample rate = 20MSps

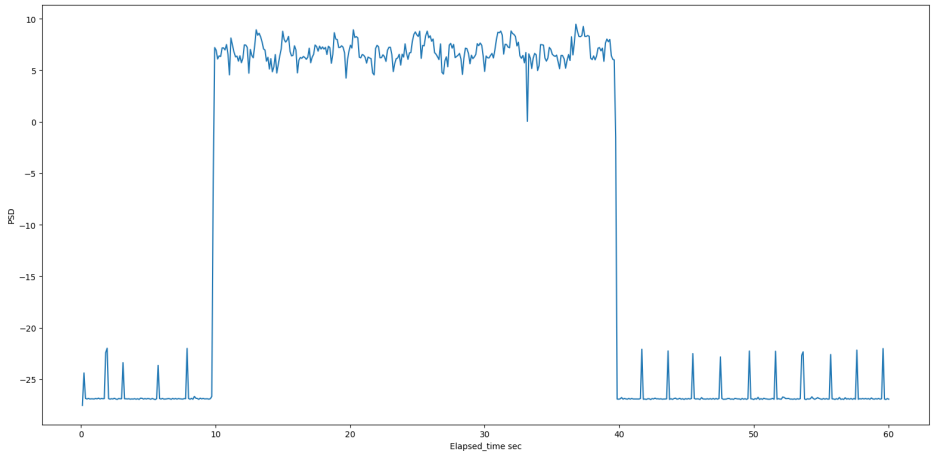
Στην συνέχεια ελέγχθηκε η συμπεριφορά των καναλιών 42 και 106 κατα την εκτέλεση μιας iperf μεταξύ δυο τερματικών συνδεδεμένα στο ίδιο κανάλι. Ο client στέλνει UDP traffic με rate = 100Mbps:

 # iperf -c server_ip_addr -u -i 1 -t 30 -b 100M

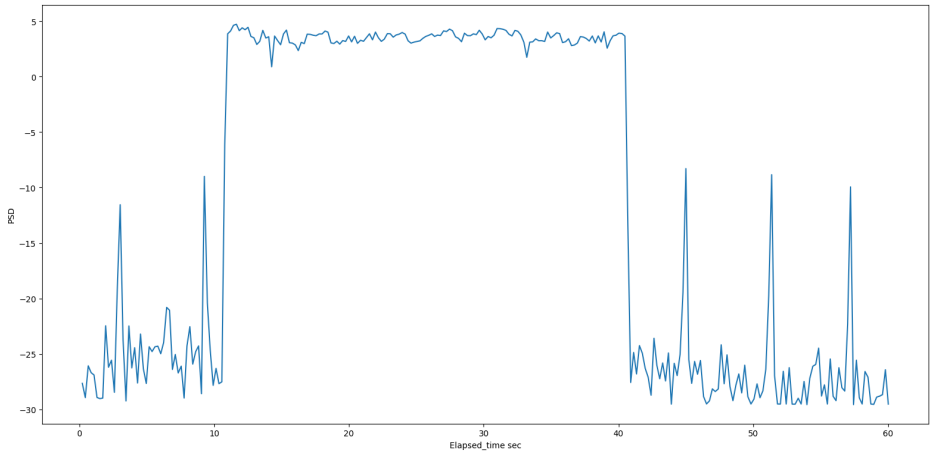
Η εκτέλεση του κώδικα έγινε για 60 δευτερόλεπα με sample rate = 10 MSps και sample rate = 20 MSps:



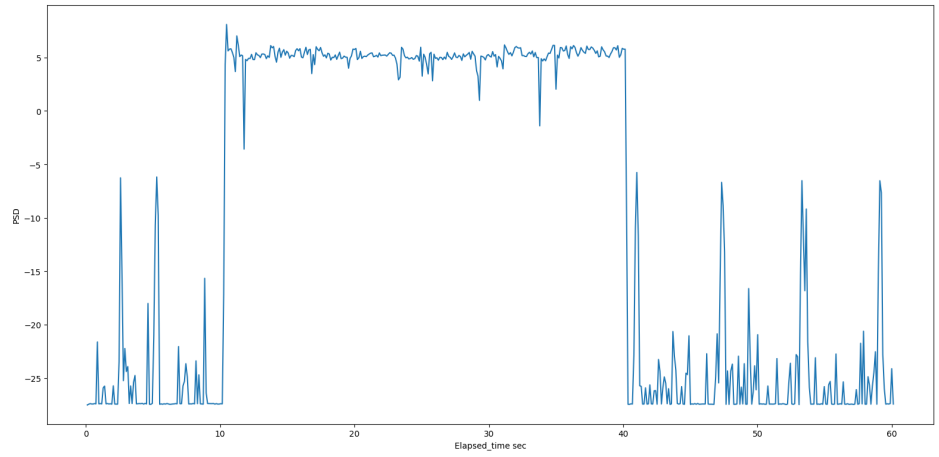
Σχήμα 15: channel 42: Performing iperf with sample rate = 10MSps



Σχήμα 16: channel 42: Performing iperf with sample rate = 20MSps

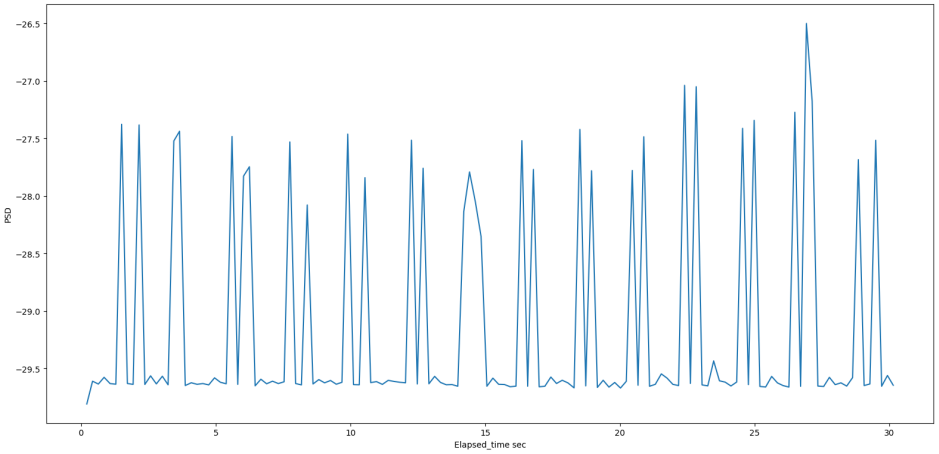


Σχήμα 17: channel 106: Performing iperf with sample rate = 10MSps

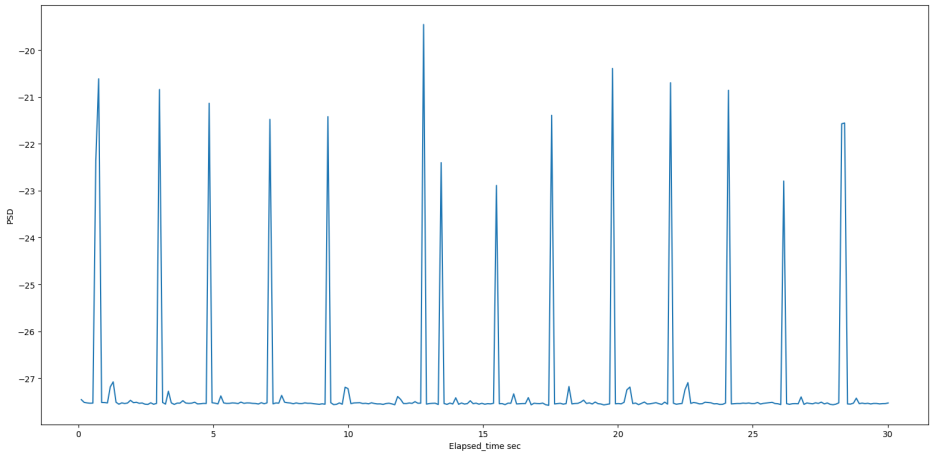


Σχήμα 18: channel 42: Performing iperf with sample rate = 20MSps

Τέλος, επειδή στο κανάλι 42 ηταν συνδεδεμένο μόνο το δικό μας τερματικό, δημιουργούσαμε μόνο εμείς traffic. Με αυτον τον τρόπο μπορέσαμε εύκολα να εντοπίσουμε beacon πακέτα «ακούγωντας» απλά στο κανάλι για 30 δευτερόλεπτα:

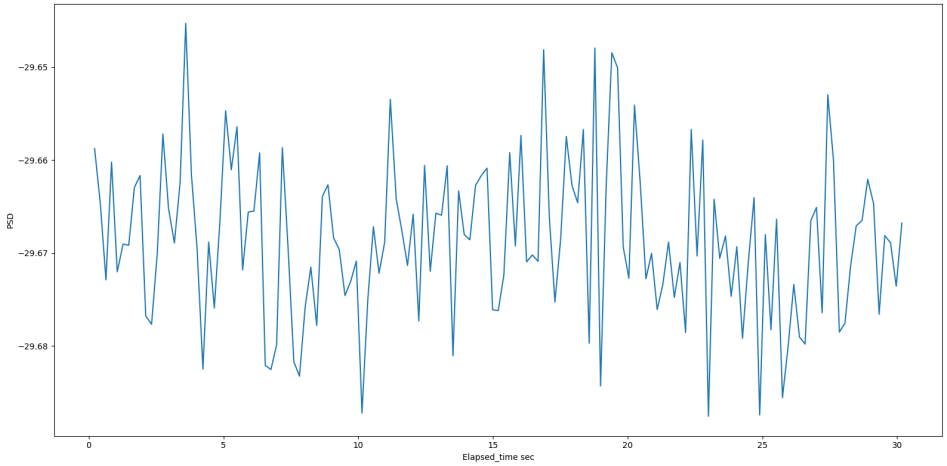


Σχήμα 19: channel 42: sensing beacon packets with sample rate = 10MSps

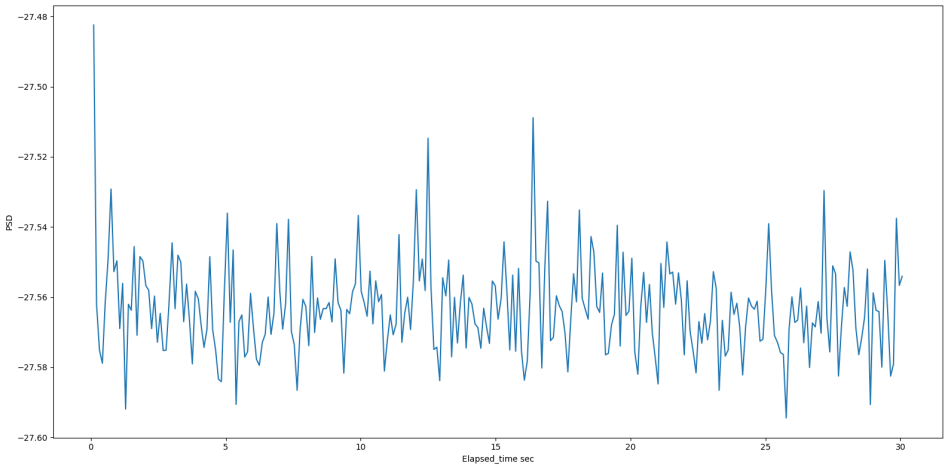


Σχήμα 20: channel 42: sensing beacon packets with sample rate = 20MSps

Η ύπαρξη των beacon πακέτων επιβεβαιώθηκε συντονίζοντας το SDR στην συχνότητα 5.400 GHz όπου υπήρχε μόνο θόρυβος:



Σχήμα 21: sensing noise with sample rate = 10MSps



Σχήμα 22: sensing noise with sample rate = 20MSps

Παρατήρηση: Μελετώντας τα διαγράμματα επιβεβαιώνονται οι παρατηρήσεις που αναφέρθηκαν στο τέλος του κεφαλαίου 4.2.2. Στις περιπτώσεις που τρέχουμε τον αλγόριθμο με sample rate = 20 MSps παρατηρούμε ότι έχουμε μια πιο καθαρή εικόνα για την συμπεριφορά του καναλιού σε σχέση με αυτη με sample rate = 10 MSps. Αυτο φαίνεται έντονα στα διαγράμματα με τα beacon πακέτα όπου στην περίπτωση του sensing με sample rate = 10 MSps το επίπεδο ενέργειας «κόβεται», καθώς ο αλγόριθμος ξοδεύει αρκετό χρόνο σε υπολογισμούς διακόπτοντας επανειλημμένα την λήψη δειγμάτων.

5 Execution commands

Πλοηγηθείτε στο home directory του docker container και μπείτε στον φάκελο spectrum_analyzer. Ο φάκελος περιέχει το πρόγραμμα spectrum_sensing_software.py. Η εκτέλεση του γίνεται με τις παρακάτω εντολές:

- **LoRa protocol:** *# python3 spectrum_sensing_software.py lora*
- **FM protocol:** *# python3 spectrum_sensing_software.py fm*
  Μετά απο αυτή την εντολή θα σας ζητηθεί raw input για να εισάγετε τις ακριανές τιμές του διαστήματος που θέλετε να εξετάσετε (οι τιμές θα πρέπει να είναι τύπου float και να είναι της μορφής: πχ 87.5 και 99.8)
- **Wi-Fi protocol:** *# python3 spectrum_sensing_software.py wifi*

References

[1] PySDR: A Guide to SDR and DSP using Python
https://pysdr.org/index.html

[2] https://wiki.gnuradio.org/index.php/PlutoSDR_Source