Apsera Simulation

Konaark Berwal, Parth Satra and Nehaal Raj, students from BITS Pilani, Pilani campus, worked on a simulation of the digital signal processing unit of the Apsera project taken up by the Raman Research Institute. The project was started on 26th May and finished on 9th June. The simulation was built on Python, and all the underlying concepts required for the project were taught to us by our mentors Prof Girish B S and Prof Vani . This report covers all the concepts we have learnt and applied, a documentation of the code and reasoning behind the decisions taken while writing the code, and the difficulties faced when developing this simulation.

a) <u>Prerequisite Knowledge</u>

Joining RRI as sophomores was a challenge for all the BITS students as some prerequisite knowledge required for this project was not covered in our sophomore year, and it was the first time applying this knowledge in a practical scenario. Therefore, the first week involved reading and understanding the background required for astronomical signal processing. Prof Girish B S and Prof Vani undertook many lectures to help the students understand the content required and also provided research papers to complement the lectures and clear the concepts. First, an overview was given about how signals from an antenna are extracted and processed. A general understanding was established about how EM waves from space are converted into analog signals using RF circuitry and how the signals are amplified and passed through a sample and hold circuit before being processed through an ADC. Although this section was beyond the scope of this project, Prof Girish B S insisted that we had a good grasp on how signals are processed before inputting into an ADC.

One of the first tasks students were given was to deeply understand flash ADCs, their working and caveats. Since the simulation later will be implemented on the Red Pitaya board, it was required to know the working behind the components that would be used on the board during the hardware implementation of this project covered later. It was made clear after reading multiple documents on ADCs sent by Prof. Girish B S, and after lectures that the ideality normally considered when solving pen-and-paper questions in college cannot be considered here, and utmost attention must be paid to all the shortcomings that an ADC has. Various caveats of an ADC, such as quantization error, DNL, INL, etc., are later used to explain the errors from ideality seen in the simulation. After a digital signal was synthesized, windowing was implemented on the signal. Again, some time was spent on why signals are windowed and on the different windowing functions. The window used was an 18-bit Hann window to maintain simplicity. The implications of the window are discussed later. The next block involved in the simulation was the fast fourier transform. Prof Girish B S, again made it very apparent that simply calling a python function would not work in the real world. Some time was spent on understanding the FFT algorithm, the butterfly stages and the bit growth per stage. The 'Divide and Conquer' or 'M*N' algorithm was used to implement a 16k point FFT (divided into 16 * 1024 samples). This was done to simulate the similarity between the hardware and software implementation as performing a 16k point FFT has its limitations on the Red Pitaya, which are detailed later.

After FFT implementation, the students were introduced to correlation. A lecture by Prof Girish B S was taken to understand the mathematics behind correlation and power spectrum of a signal. By correlating our FFT output, the signal's power spectrum was obtained. Two types of correlation were implemented- Autocorrelation and cross correlation. Autocorrelation was done to give the power density spectrum of the signal and cross-correlation was done to give the cross power spectrum. These values are then accumulated 4k times to average our tone frequency peaks and remove any noise present.

b) <u>Establishing required constraints on Python to simulate FPGA like environment</u>

As mentioned above, this simulation exists as a roadmap and predicts shortcomings that might be faced while implementing the model on a Red-Pitaya 125-14 FPGA. The primary difference between Python and FPGA architecture is the speed and parallel processing offered, and the code is built with this processing in mind. Another difference is the lack of easy-to-use floating-point numbers in an FPGA.

The precision was handled at each stage by choosing a standard to run the entire simulation in, the options being entirely integral calculations (2^{14}-1, etc.) or entirely decimal (0.4356, 0.999, etc.). An integer can be easily converted to binary, and a decimal such as above can be seen as some binary number after the decimal point. The decision was made to follow the integer format [2’s complement due to usage in FPGA] due to the ease of performing arithmetic operations and visible bit growth at each stage if normalisation was not taken into consideration.

For testing purposes, a sine wave of 1dBm power was taken and passed through a 2 dBm full-scale ADC.

The precision was handled by choosing the corresponding maximum and minimum 2's complement integral values at each stage. Following are the hardware constraints simulated:

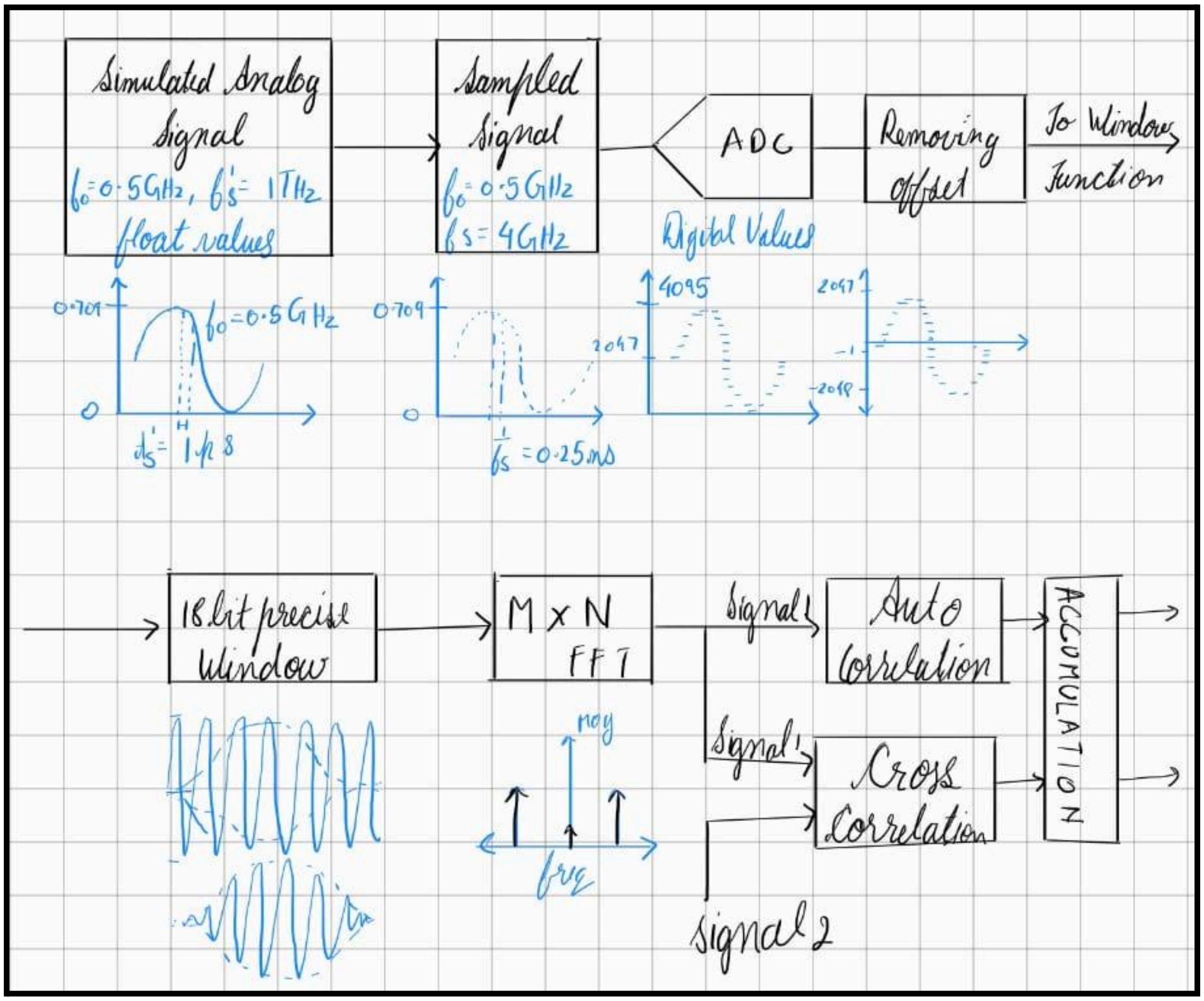
1) RF ADC output [12 bits] – [$V_{in} \rightarrow$ 12 bits]
   The sampling rate needs to be at least 4 GSPS for the RF signals we are expected to receive, and the ADC chosen as the best fit for our standards is a 12 bit one.
2) Window Function [18 bits] - [12 + 18 = 30]
   The 2’s complement multiplier in the DSP48 slice has 2 inputs of 25 bits and 18 bits, respectively. 18 was chosen over 27 bits, as the calculated decreased noise is not worth the increased computation and bit growth.
3) FFT input [12 bits] - [30 - 18 =12]
   The bit growth observed after windowing is 30 bits. Optimal truncation was calculated to be 18 bit.
4) FFT output [32 bits] - [12+20 = 32]
   The FFT output increases in size to approximately 32 bits due to bit growth at each butterfly stage.
5) Correlation input [18 bits] - [32 - 14 = 18]
   Inspection of maximum possible FFT output leads us to 7 truncated zeros from MSB and lose 7 bits LSB precision to get a total of 18 bits.

6) Correlation output [36 bits] - [18*2 = 36]

Squaring of complex coefficients leads to doubling of bits

7) Accumulator output [48 bits] - [36 + 12 = 48]

The 36 bit input is accumulated 4096 times, which adds [14 - 2 =] 12 bits. It was accumulated 4k times, as DSP48 has a 48 bit accumulator.



Flow of data across the system

Figure 1

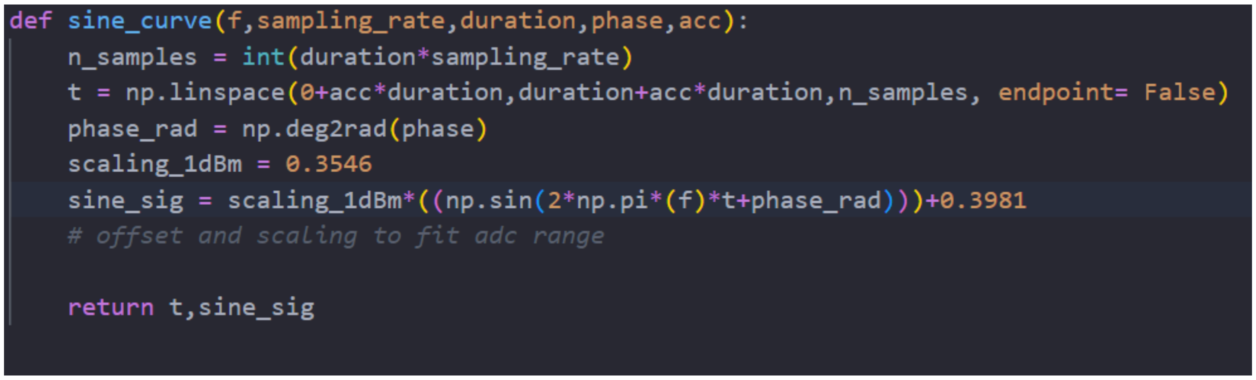
c) Code Implementation

https://github.com/Cryptos-Logos/RRI_BITS_PS1/tree/main/Final_apsera

i) Input generation

- As mentioned above, a sine signal was taken as input for simplification and for easy verification of the simulated values. A sine signal corresponding to 1dBm,i.e, 0.3546 V-peak was used.

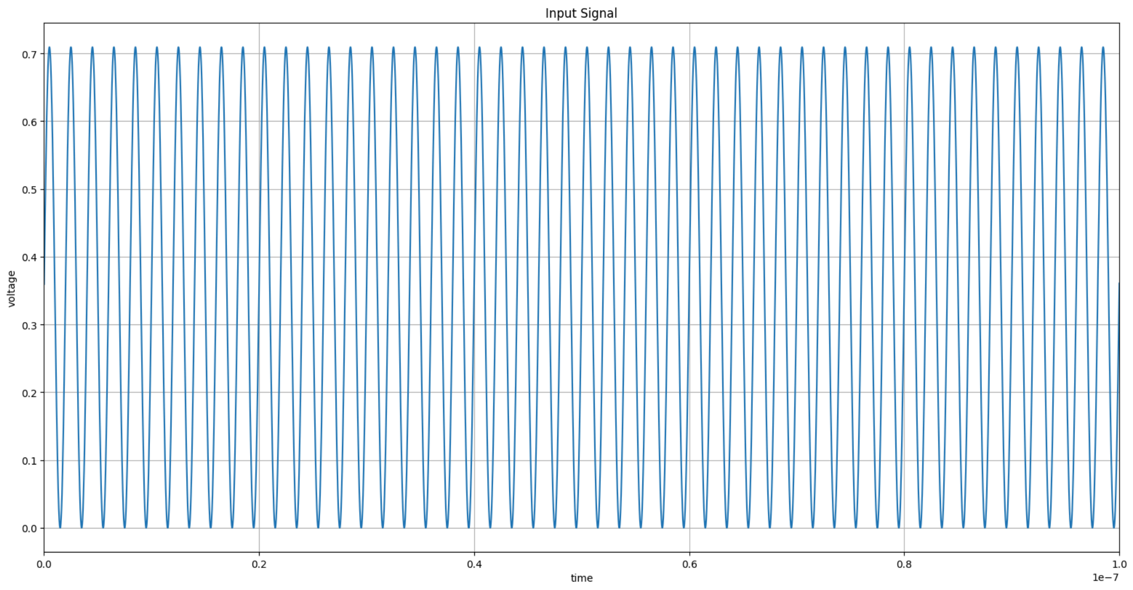
- The ADC code implemented was unipolar; therefore, the signal is offset such that it is within the ADC scale. (ADC full scale = 2 dBm == 0.3981 V peak)
- The signal was also generated for a time interval such that it contained 16k points when sampled with the ADC sampling rate of 4 GHz.
  (duration = fft_points/adc_sampling_rate)
- The signal was given a sampling rate of 12e9 to try to represent an analog signal.



```python
def sine_curve(f,sampling_rate,duration,phase,acc):
    n_samples = int(duration*sampling_rate)
    t = np.linspace(0+acc*duration,duration+acc*duration,n_samples, endpoint= False)
    phase_rad = np.deg2rad(phase)
    scaling_1dBm = 0.3546
    sine_sig = scaling_1dBm*((np.sin(2*np.pi*(f)*t+phase_rad)))+0.3981
    # offset and scaling to fit adc range

    return t,sine_sig
```

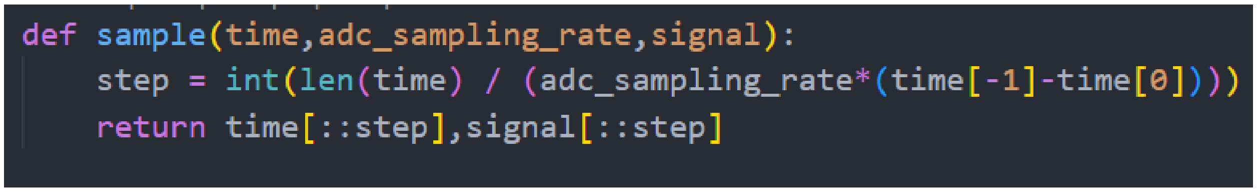
Function to generate ideal sine wave
Figure 2



Generated sine wave
Figure 3

ii) Sample and ADC

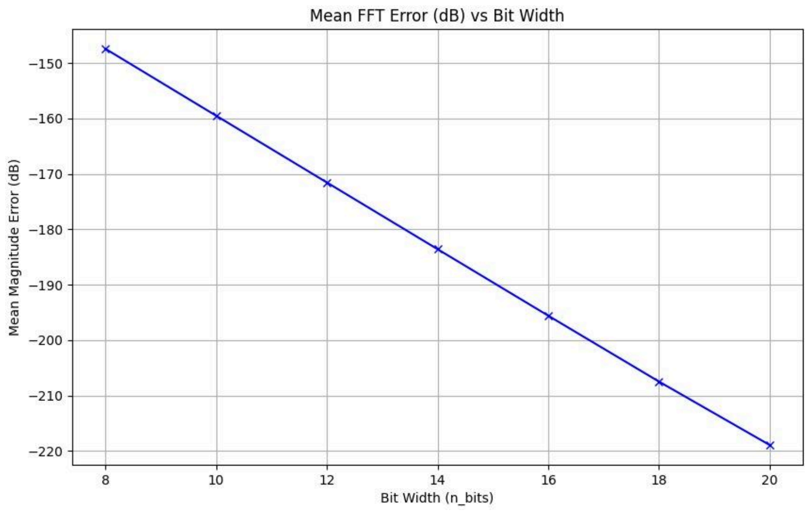
- Before feeding the signal into the ADC, the signal was to be sampled at 4 GHz. The signal therefore was fed into a sample function which sampled the signal and output the time accordingly.



```python
def sample(time,adc_sampling_rate,signal):
    step = int(len(time) / (adc_sampling_rate*(time[-1]-time[0])))
    return time[::step],signal[::step]
```

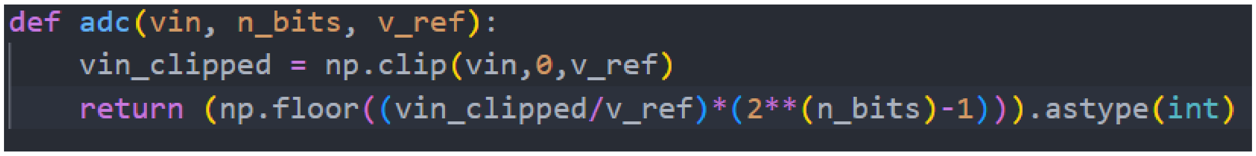
Function to sample the ideal wave
Figure 4

- A comparison of ADCs with different resolutions with full precision was also done, which showed a predicted graph of decreasing error with an increase in resolution.



Mean FFT error VS ADC resolution
Figure 5

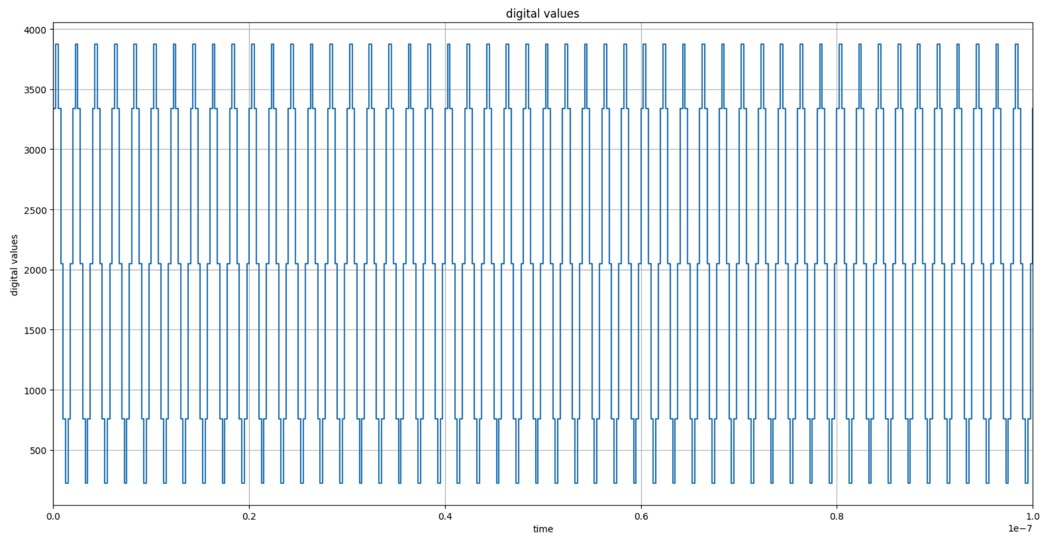
- The signal is then passed to the ADC with a full scale of 2 dBm. The ADC used in APSERA is a 12-bit ADC.
  The basic working principle of a flash ADC involves comparing the input voltage with reference voltage, that's divided by a resistive ladder, using a comparator. Initially attempts were made to design an ADC using the principle of comparing the input voltage with the voltage corresponding to m*LSB (where 1 LSB = V_full_scale/2^n, and m is any integer). This method, although yielding a correct result, was very time-consuming. Therefore, the implementation was changed, and the input voltage was divided by the full scale voltage, then multiplied by the max integer value. Therefore, our sine wave was digitised to values from 0 to 4095.



```python
def adc(vin, n_bits, v_ref):
    vin_clipped = np.clip(vin,0,v_ref)
    return (np.floor((vin_clipped/v_ref)*(2**(n_bits)-1))).astype(int)
```

Function to digitise the sampled signal

Figure 6



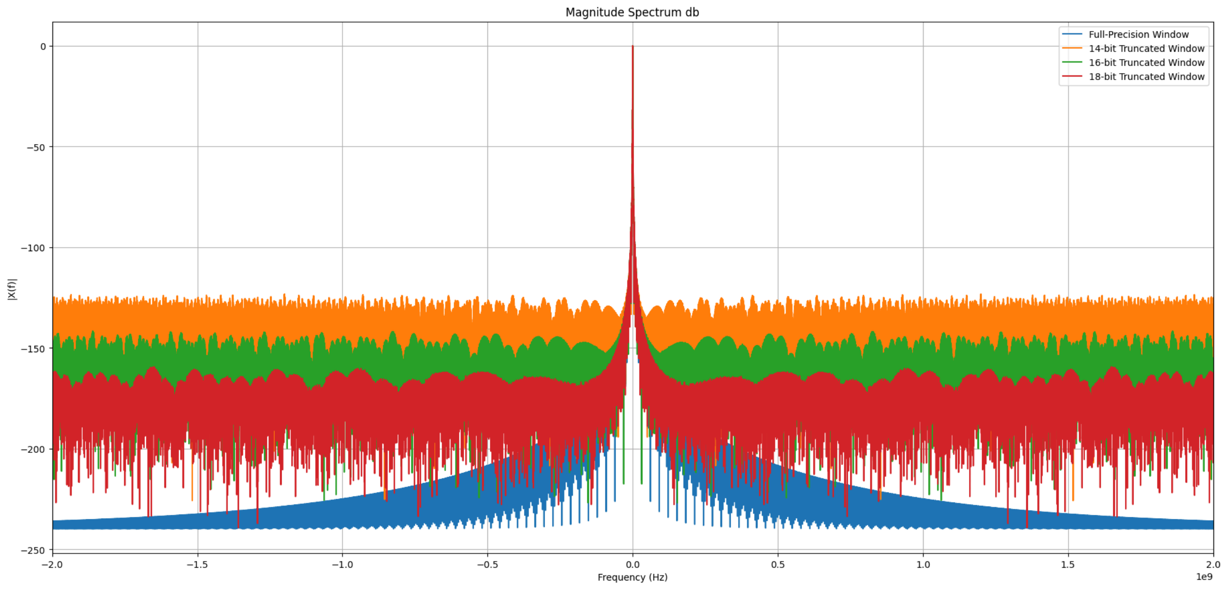
Digitised sine wave
Figure 7

- Characterisation of the ADC is avoided in this section and will be done during hardware implementation.
- **NOTE: In the GitHub repository linked to this report, some files have digital values normalised for easier comparison with full precision, while some do not have normalised values, i.e., purely integers to verify bit growth.**

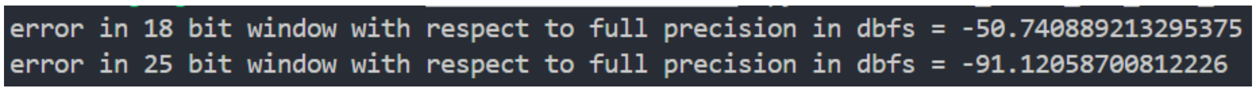
iii) Pre-Windowing
- ADC output ranges from 0 - 4095. Since each value in the FPGA is represented by a 2's complement number. These values will be interpreted as the positive values for a 13 bit 2's complement ADC and lead to a DC offset being shown in FFT. To combat this, the values are subtracted by 2048, which changes the range from 2047 to -2048 [range of 12 bit 2's complement].
- This leads to another small issue; previously the zero crossing of the sine wave was represented by 2047, and subtracting 2048 causes the zero to map to -1, which is indicated by a small peak at 0 Hz in the FFT.

iv) Windowing
- The signal was windowed to minimize the side lobes that occur after FFT.
- The DSP48 multiplier in the FPGA has 2 inputs, one of 18 bits and another of 25 bits.

- As evident by the graph, the precision provided by an 18 bit window is quite close to a full precision window. Therefore, there is no requirement to use more than 18 bits to generate the window signal.



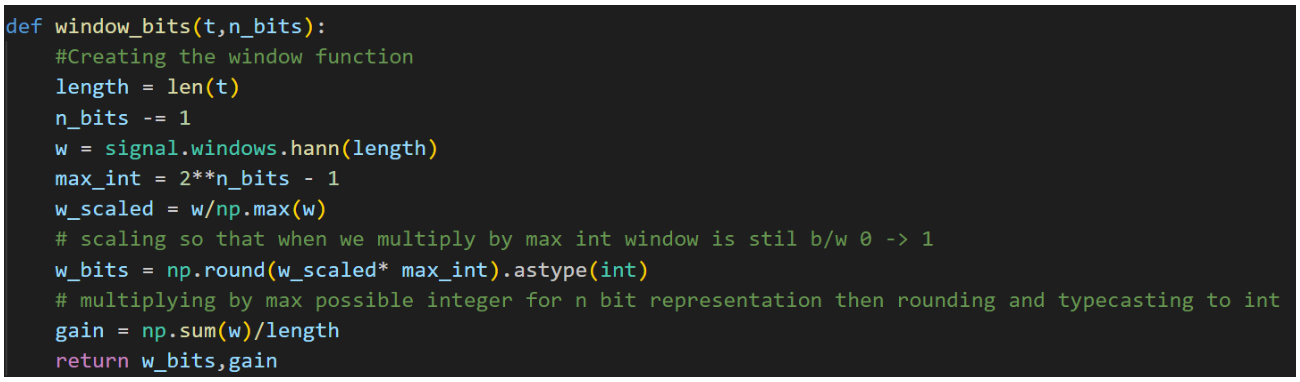
<u>Normalised FFT of window Functions with different precision</u>
<u>Figure 8</u>



```
error in 18 bit window with respect to full precision in dbfs = -50.740889213295375
error in 25 bit window with respect to full precision in dbfs = -91.12058700812226
```

<u>Mean error for 18 and 25 bit precise window</u>
<u>Figure 9</u>

- The Hanning window was chosen due to its simplicity and ease of implementation on the FPGA.
- The window function generated was from the SciPy library, and its usage is according to the code below. **A normalised window function has also been used in situations to compare precision**.

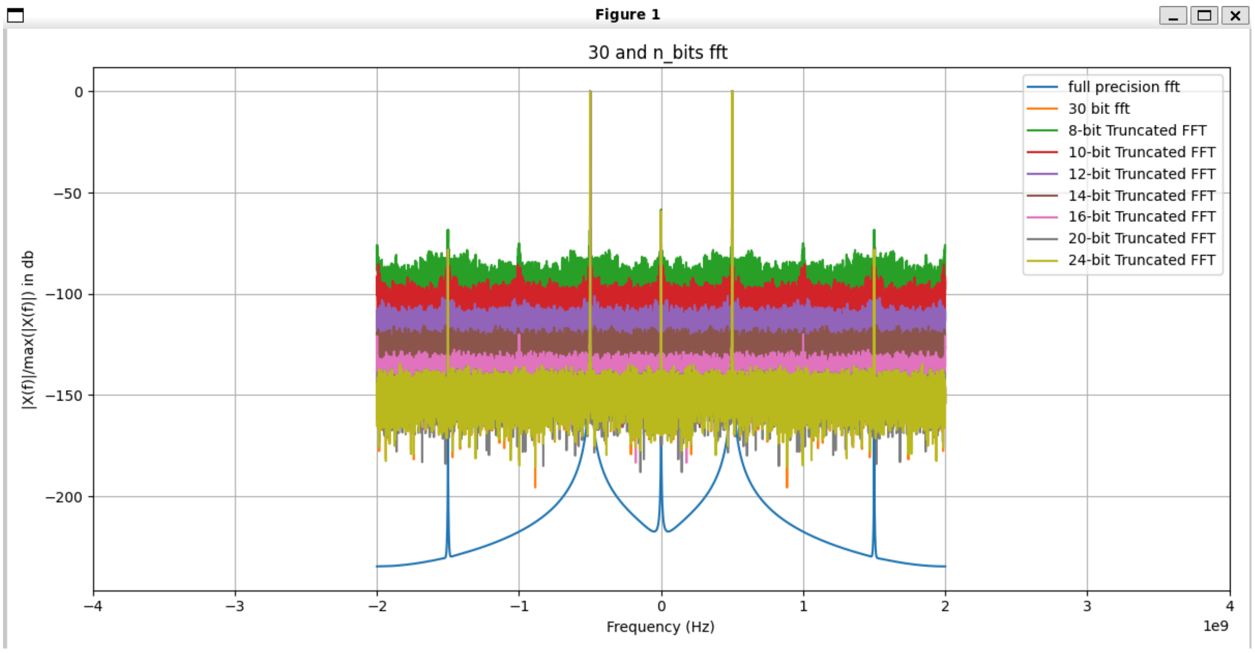


```
def window_bits(t,n_bits):
    #Creating the window function
    length = len(t)
    n_bits -= 1
    w = signal.windows.hann(length)
    max_int = 2**n_bits - 1
    w_scaled = w/np.max(w)
    # scaling so that when we multiply by max int window is stil b/w 0 -> 1
    w_bits = np.round(w_scaled* max_int).astype(int)
    # multiplying by max possible integer for n bit representation then rounding and typecasting to int
    gain = np.sum(w)/length
    return w_bits,gain
```

Function to generate window signal
Figure 10

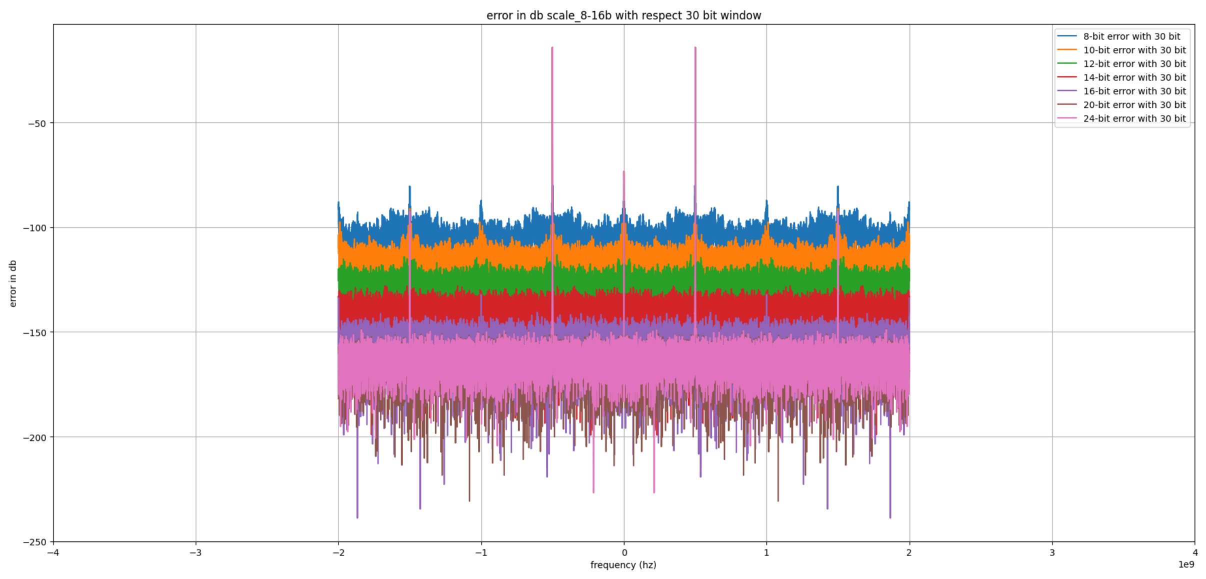
iv) Truncation after windowing

- The multiplication of the 12 bits ADC output and the 18 bit window resulted in a 30 bit representation of the windowed signal.



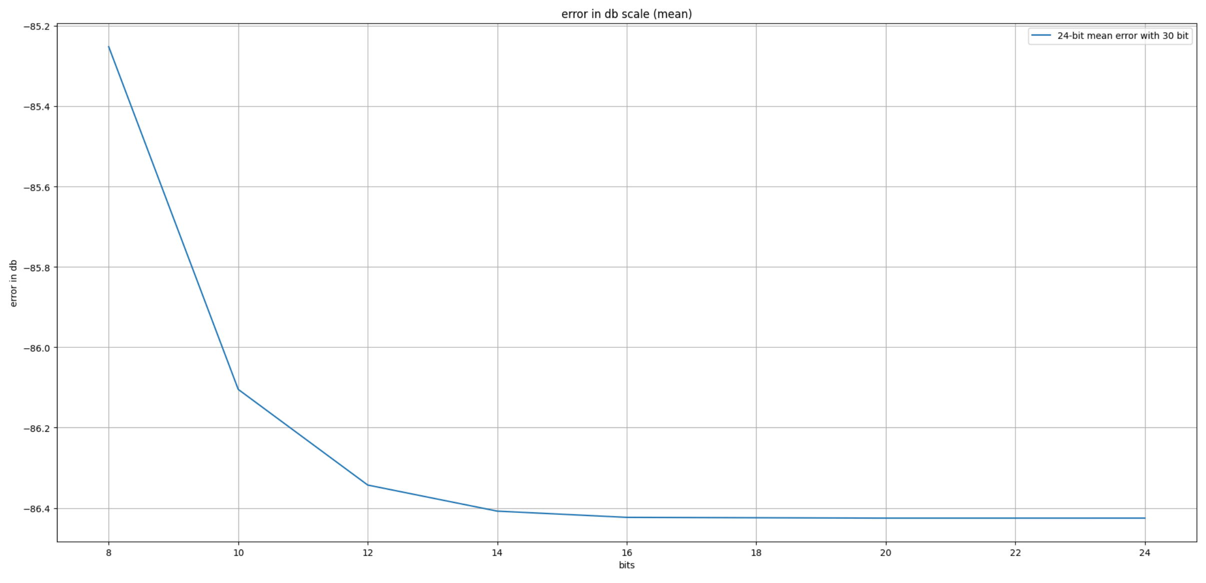
Plotting normalised FFT for windows with different truncation
Figure 11

- This windowed signal was truncated to a 12 bit signal. This was done to remain faithful to hardware constraints. With the M*N FFT having a bit growth of 21 bits, after the FFT stage we would need 51 bits to represent the signal. Which cannot be accommodated on the DSP48.
- To arrive at the ideal truncation to 12 bits, an error plot was graphed of all the windowed signals ranging from a truncation of 8 to 24 bits.



Error of FFT with different precision input with respect to 30 bit precise input
Figure 12

- From this graph it was evident that after 12 bits the error between the 30 bit windowed signals and that of the 12 bit and beyond windowed signals is negligible.

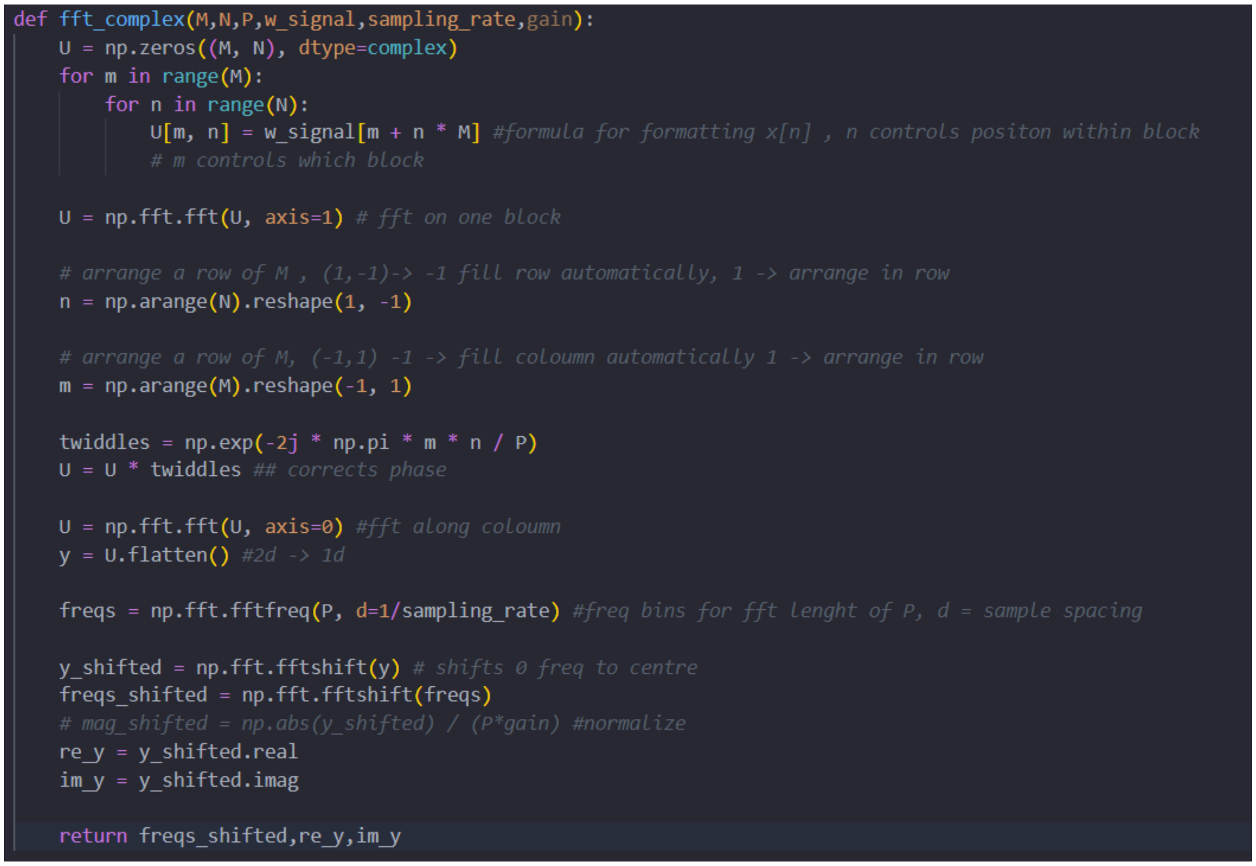


Mean error in FFT for different levels of precision
Figure 13

- This outcome leads us also to understand that multiplying such signals only offers as much data as the original signal had

vi) Fast Fourier Transform

- The truncated 12 bit windowed signal is then passed through an FFT engine to obtain the signal's frequency contents.
- Because of hardware constraints, a pipelined 2D FFT size 16 × 1024 was implemented and simulated to process efficiently a 16,384-point (P = 16 × 1024) FFT of sampled input signals. The value of M was chosen as 16 to match the system's timing constraints.



```python
def fft_complex(M,N,P,w_signal,sampling_rate,gain):
    U = np.zeros((M, N), dtype=complex)
    for m in range(M):
        for n in range(N):
            U[m, n] = w_signal[m + n * M] #formula for formatting x[n] , n controls positon within block
            # m controls which block

    U = np.fft.fft(U, axis=1) # fft on one block

    # arrange a row of M , (1,-1)-> -1 fill row automatically, 1 -> arrange in row
    n = np.arange(N).reshape(1, -1)

    # arrange a row of M, (-1,1) -1 -> fill coloumn automatically 1 -> arrange in row
    m = np.arange(M).reshape(-1, 1)

    twiddles = np.exp(-2j * np.pi * m * n / P)
    U = U * twiddles ## corrects phase

    U = np.fft.fft(U, axis=0) #fft along coloumn
    y = U.flatten() #2d -> 1d

    freqs = np.fft.fftfreq(P, d=1/sampling_rate) #freq bins for fft lenght of P, d = sample spacing

    y_shifted = np.fft.fftshift(y) # shifts 0 freq to centre
    freqs_shifted = np.fft.fftshift(freqs)
    # mag_shifted = np.abs(y_shifted) / (P*gain) #normalize
    re_y = y_shifted.real
    im_y = y_shifted.imag

    return freqs_shifted,re_y,im_y
```

Function to perform M*N Pipelined FFT
Figure 14

- With a 4 GHz sampling rate and a 250 MHz processing clock, the FFT was divided into 16 parallel blocks to align one FFT block per clock cycle. The technique split the computation into two phases in order to take advantage of pipelining and avoid computational complexity at the expense of accuracy and scalability.
- The input signal was reshaped initially to a matrix of size (M, N), where M = 16 is the number of blocks, and N = 1024 is the number of samples per block. A row-wise FFT was then performed on each of the 16 blocks independently with NumPy's np.fft.fft function.
- After this, twiddle factor multiplication was conducted via a matrix of complex exponentials that compensated for the phase misalignment caused in reshaping the matrix. This ensured that the output was that of a complete 1D FFT computation. Next, a

column-wise FFT was performed to finalize the transformation. The end result was then flattened into a one-dimensional array and post-processed.

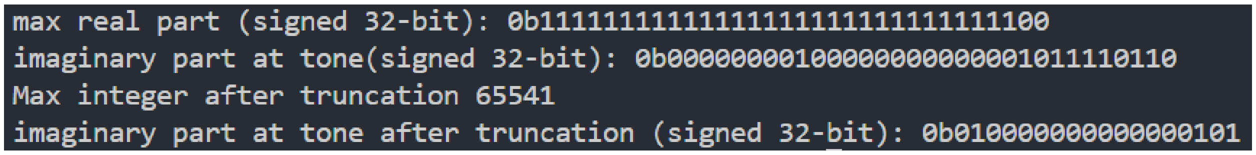
- This involved:
    - a) Calculation of frequency bins using np.fft.fftfreq
    - b) Spectrum centering with fftshift
    - c) Isolation of real and imaginary parts for spectral analysis

- The output of the FFT can be normalised by an adjustable gain factor, which can be adjusted according to system specifications like ADC resolution or scaling of the signal. This pipelined, modular FFT architecture is especially beneficial for high-throughput signal processing systems. It offers a scalable and hardware-efficient design for use in real-time spectrum analysis, communication systems (e.g., OFDM) and radar signal processing.

- During development, the internal working of the FFT was studied in greater depth using the radix-2 butterfly algorithm. This structure, which forms the core of efficient FFT computation, breaks down the DFT into $\log_2(N)$ stages of simpler 2-point computations involving additions and subtractions. At each stage, a butterfly unit combines pairs of inputs using weighted twiddle factors. This recursive decomposition significantly reduces the computational complexity from $O(N^2)$ to $O(N \log N)$. A key consideration in the FFT pipeline was the analysis of bit growth across stages. Given a 16,384-point FFT, the theoretical bit growth is around 1.5 bits per stage. Since the input to the FFT is a 12-bit number (truncated from 30 bits to 12 bits), the final output can grow to as much as 33 bits theoretically in the worst case, as we had 14 stages ($14 \times 1.5 = 21$). This necessitates careful word-length management, especially in fixed-point or hardware implementations, to avoid overflow and preserve numerical accuracy.

vii) <u>Truncation after FFT</u>

- After the FFT, the result had a theoretical approximate resolution of 32 bits. The process after FFT involved correlation, and this meant a bit growth from 32 to 64 bits. Furthermore, accumulation to average out the signal's power spectrum would lead to further bit growth. This obviously cannot be accommodated by the hardware.
- This meant that the 32 bit output must be truncated such that the constraints of the hardware were met.
- It was known that the FFT output without any normalisation would not grow more than a certain amount; therefore, some bits from the MSB side can be removed without affecting the signal's peaks.
  Since the signal used was a simple sine signal, the frequency contents and how the FFT must look was known.
- Therefore, to know how many bits was to be removed from the MSB, the maximum value of the imaginary part of the FFT (which would correspond to the signal's tone amplitude) and the maximum value of the real part of the FFT (which would correspond to the signal's noise) were represented in 2's complement form.



```
max real part (signed 32-bit): 0b11111111111111111111111111111100
imaginary part at tone(signed 32-bit): 0b00000000100000000000001011110110
Max integer after truncation 65541
imaginary part at tone after truncation (signed 32-bit): 0b010000000000000101
```

<u>32-bit output after the MxN FFT</u>
<u>Figure 15</u>

- From this it was evident that the first 7 bits were just an extension of the sign bit and therefore can be discarded without losing any data.
- This brought down the 32 bit number to 25, which was still not sufficient, as the correlation and 4k-accumulation process would still exceed the required bit width.
- Calculation determined that truncating the FFT output to 18 bits would lead to the final output being 48 bits wide. (18 bits - 36 bits (after correlation) - 48 bit (after accumulation)) (48 (DSP width) − 12 (accumulation growth) = 36, 18 x 2 = 36)

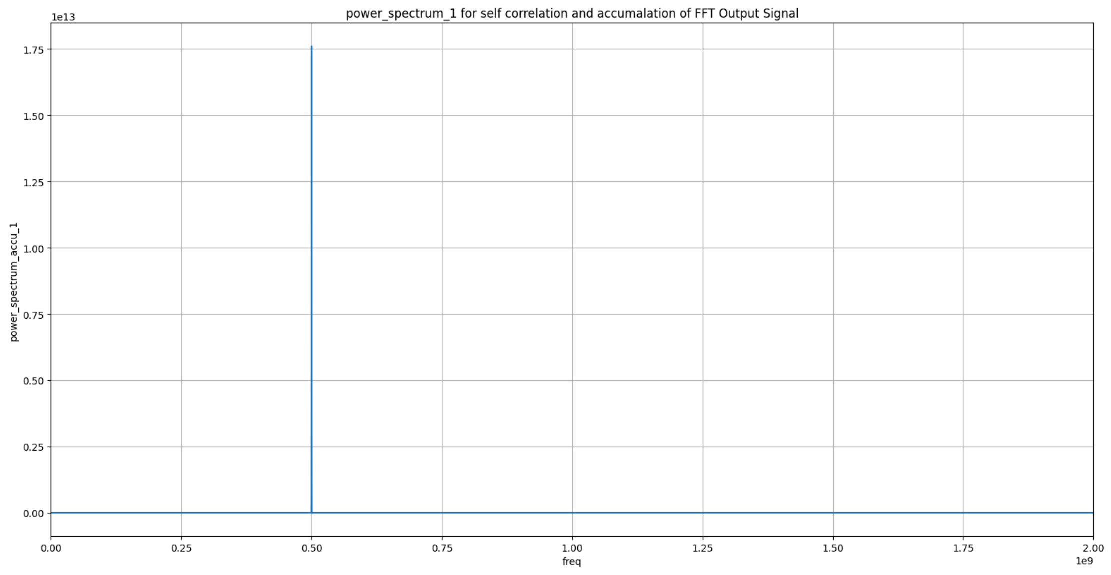
- A slight overshoot was seen from the expected value of 65536 after truncation. The expectation of 65536 as peak bin value after truncation is because peak bin value without normalisation is given by N.A/2 (where A = max value of n-bits , N = number of FFT points). Therefore with A = 2047 , the max value for FFT would be 16,769,024; compensating for window gain of 0.5 the result is 8,384,512 for full scale . Our imaginary value which is our FFT peak is equal to 8,389,366, the error could be due to the DC component present in the signal spilling into the FFT which cannot be avoided, and due to rounding errors that could happen in an effort to keep the code in integers only. Therefore when we truncate the result to 18 bits from 25 we are effectively dividing the output by 256 which gives us an integer of 65541. This is a very small error from expected for full scale and therefore can effectively be ignored.
- This meticulous truncation and scaling ensured compatibility with hardware limits while preserving signal integrity throughout the digital processing chain.

viii) <u>Correlation</u>
- With the truncated FFT output, two kinds of spectra were calculated:
  a) Auto-correlation spectra for the two channels were obtained by power computation at every frequency bin. This provided information regarding the distribution of energy of each signal in the frequency domain.

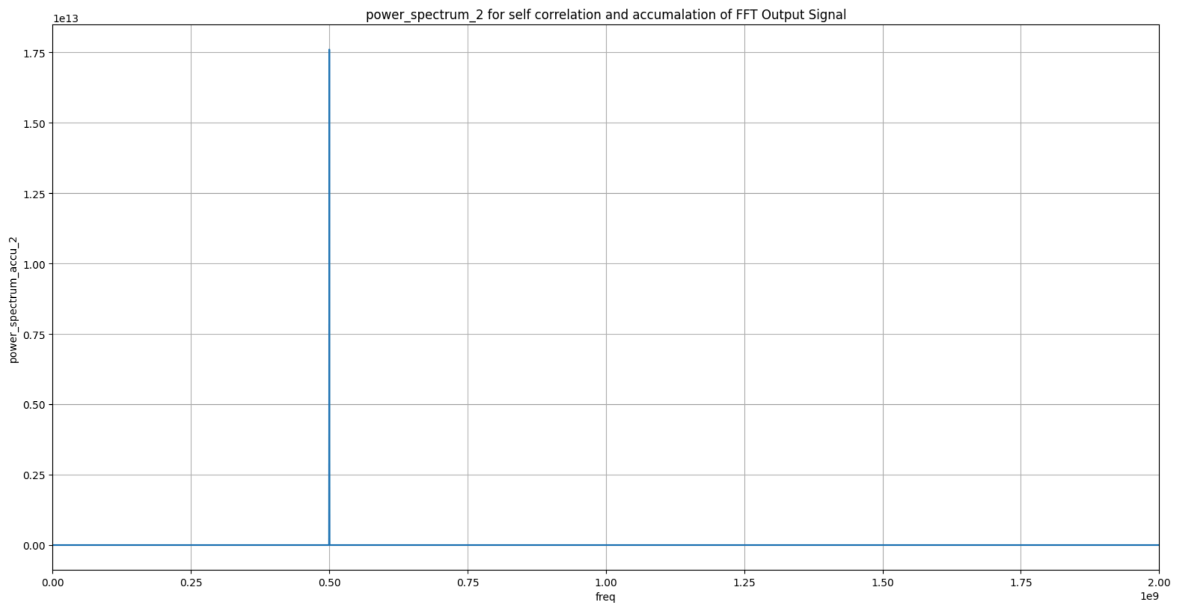
  b) Cross-correlation spectra were calculated in the frequency domain by multiplying the FFT output of one signal by the complex conjugate of the other. This retained magnitude and relative phase information between the two signals in every frequency bin.

- To enhance the signal-to-noise ratio and achieve statistically stable estimates, this whole process was iterated over 4,096 independent 16,384-point data chunks. For each iteration, the auto-correlation and cross-correlation values were calculated and then accumulated over iterations. This step of cumulation simply averaged out random noise and fluctuations but enhanced coherent features of the data.

- Once all iterations were performed, final cumulated spectra were utilised to pull out important information:
  1) The auto-correlation spectra showed the power distribution of frequencies for every input signal and periodicity, if any.
  2) The cross-correlation magnitude was indicative of the strength of similarity between the two signals at every frequency.
  3) The cross-correlation phase was indicative of relative phase shift between the signals for every frequency component.
- As seen from the graph the output for 2 sine signals differing only in phase by 90 degrees have the same magnitude for self correlation and cross correlation at the tone frequency.
- The phase plot shows the expected line of phase $\pi/2$ at tone frequency but we see unexpected lines at different frequencies. This can be attributed to the quantization error in the ADC or various other errors that could have occurred during rounding.

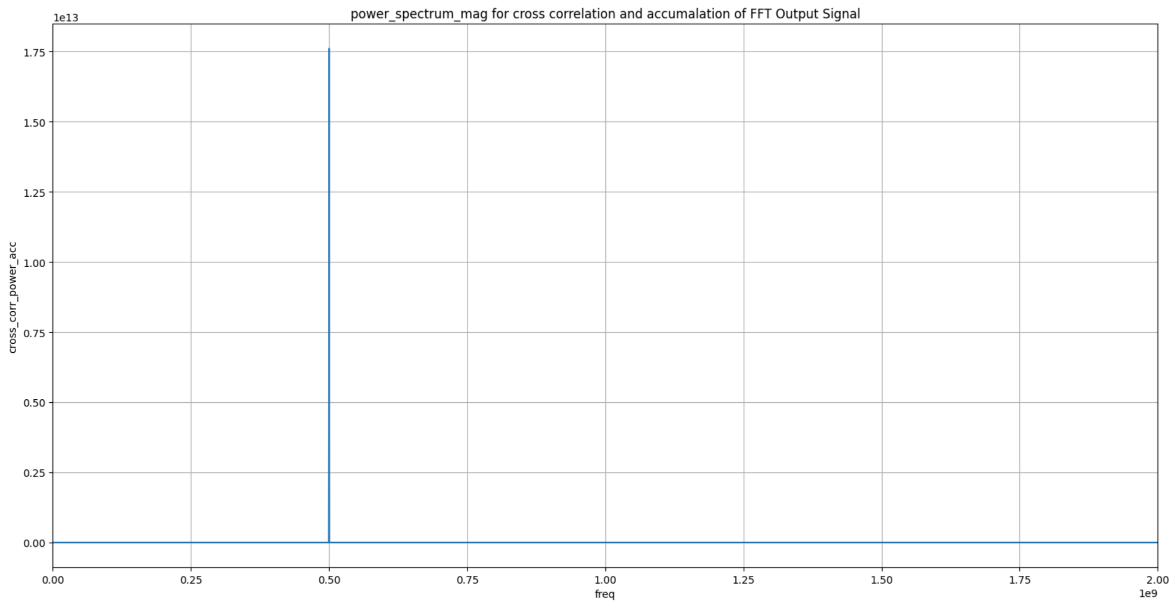


Auto correlation magnitude spectrum of antenna 1 (16k accumulated 4096 times)
Figure 16



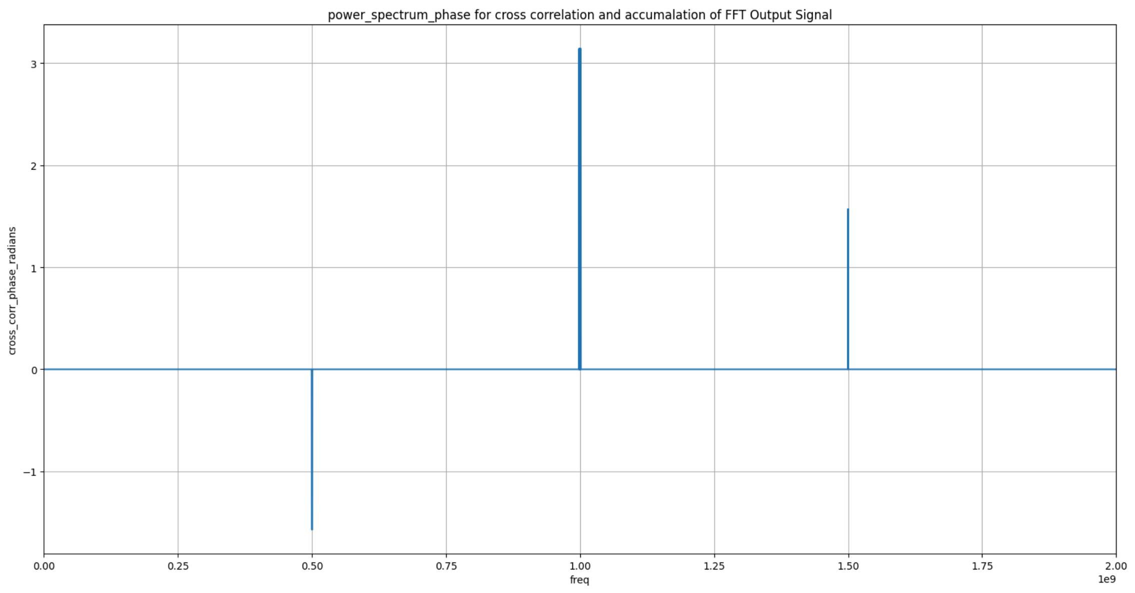
Auto correlation magnitude spectrum of antenna 2 (16k accumulated 4096 times)

Figure 17



Cross correlation magnitude spectrum (16k accumulated 4096 times)

Figure 18



Cross correlation phase spectrum (16k accumulated 4096 times)

Figure 19

d) Conclusion

- The Apsera simulation project provided an intensive, hands-on experience in digital signal processing (DSP) under realistic hardware constraints. Over a two-week period, successful modeling of the signal processing chain from ADC sampling to FFT computation and correlation was achieved, while enforcing precision and architectural limitations relevant to FPGA implementation, specifically on the Red Pitaya platform.
- Key challenges included bridging gaps in prerequisite knowledge, managing fixed-point arithmetic, and emulating hardware-like behavior in Python. Through rigorous mentorship, lectures, and independent study, a robust understanding of ADC behavior, windowing functions, FFT algorithms, and correlation techniques was developed.
- A key strength of this simulation lies in its attention to detail regarding bit growth and truncation across the signal chain—decisions that are crucial for successful deployment on constrained FPGA environments. The trade-offs made in choosing bit-widths, applying windowing, and managing accumulation were all geared toward balancing performance with resource limitations of DSP hardware.
- Ultimately, this project not only deepened the understanding of practical DSP and hardware-aware programming but also helped in preparation for real-world challenges in scientific instrumentation. The simulation acts as a reliable predictor of system behavior and a blueprint for efficient hardware implementation.