**APPLICATIONS OF FOURIER TRANSFORM**

**CODE FOR HF RADAR DATA PROCESSING**

AN INDUSTRIAL INTERNSHIP REPORT

*submitted by*

**MARLA AKHIL REDDY**

**(13BCE1032)**

*in partial fulfillment for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

in

**COMPUTER SCIENCE AND ENGINEERING**



JULY 2016



**School of Computing Sciences and Engineering**

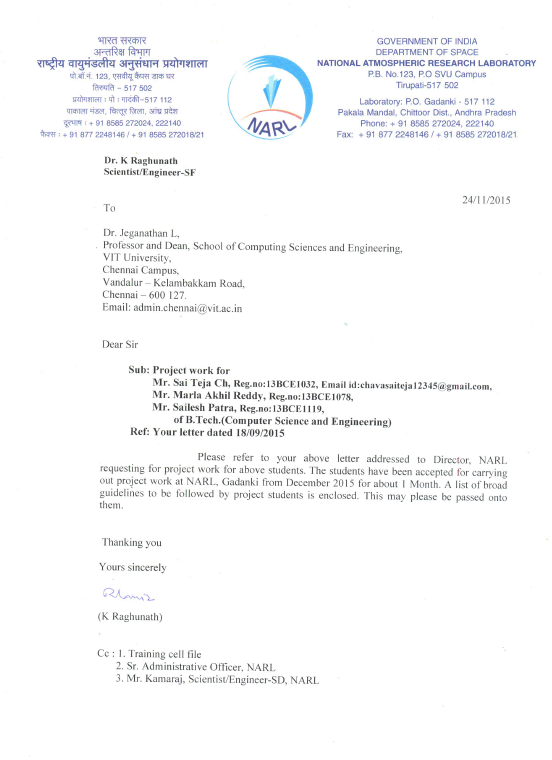
**DECLARATION BY THE CANDIDATE**

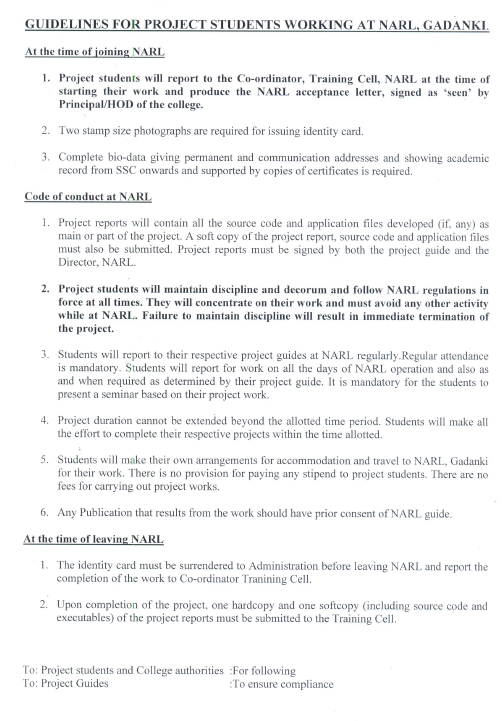
I hereby declare that the Industrial Internship Report entitled “**Applications of Fourier Transform Code For HF Radar Processing”** submitted by me to VIT University, Chennai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** is a record of bonafide industrial training undertaken by me under the supervision of **Dr. Amit P Kesarkar, Scientist-SE, NARL.** I further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Chennai Signature of the Candidate

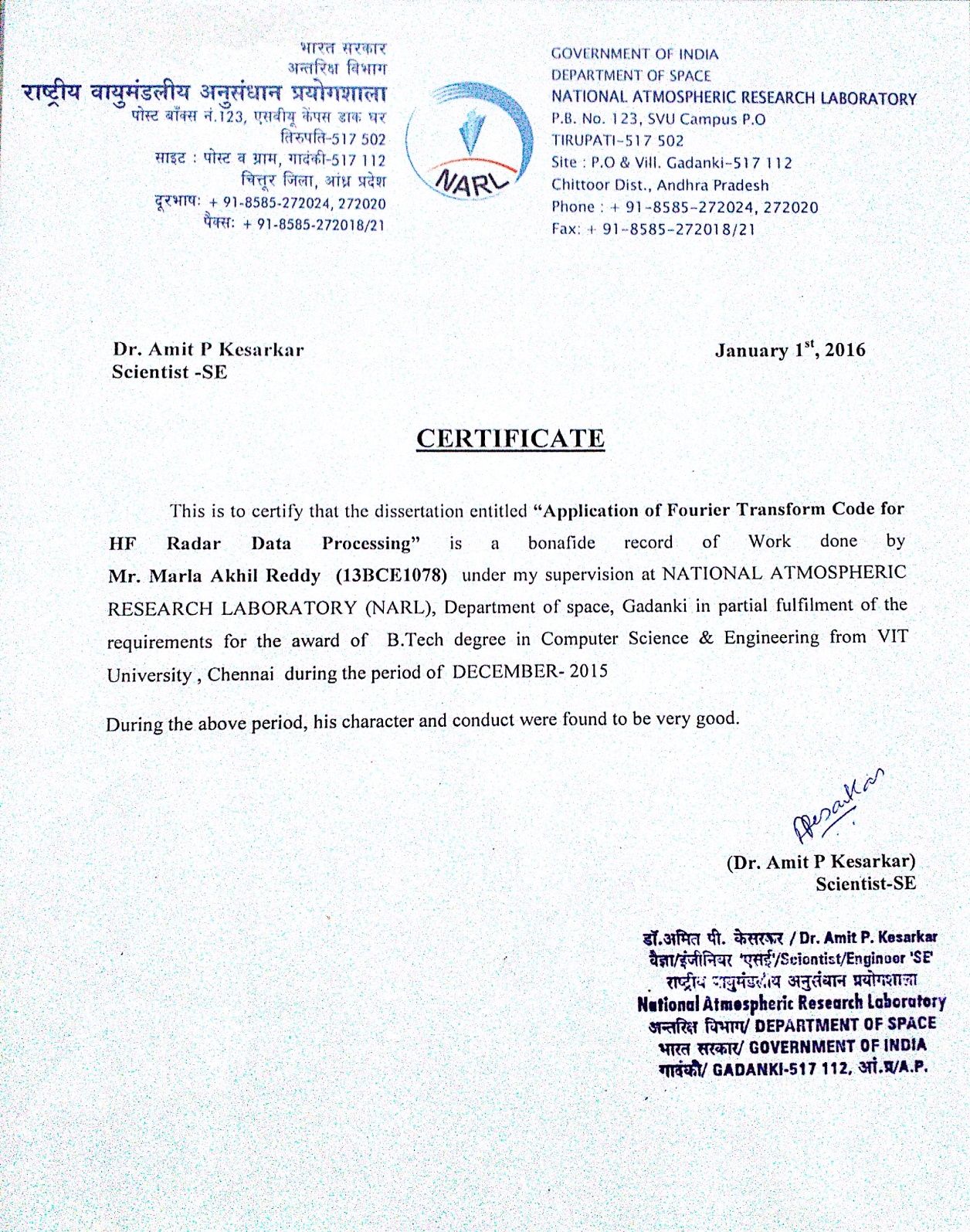
Date:

**LETTER FROM THE INDUSTRY**





**CERTIFICATE BY TRAINING IN- CHARGE AT THE INDUSTRY**





**School of Computing Sciences and Engineering**

**BONAFIDE CERTIFICATE**

This is to certify that the Industrial Internship Report entitled “**Applications of Fourier Transform Code for HF Radar Data Processing”** submitted by **MARLA AKHIL REDDY (13BCE1078)** to VIT University, Chennai in partial fulfillment of the requirement for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** is a record of bonafide internship undertaken by him fulfills the requirements as per the regulations of this institute and in my opinion meets the necessary standards for submission. The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

**Programme Chair**

**(B.Tech CSE)**

Date:

**Examiner (s) Signature**

**1. 2.**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter** | **Title** | **Page** |
|  | Declaration |  |
|  | Letter from the Industry |  |
|  | Certificate from the Industry |  |
|  | Bonafide |  |
|  | Acknowledgement |  |
|  |  |  |
|  | Table of Contents |  |
|  |  |  |
|  | Abstract |  |
|  | About the Project |  |
|  | About the company |  |
|  | Related Work |  |
|  | Proposed System |  |
|  | Implementation of System |  |
|  | Modules |  |
|  | Results |  |
|  | Conclusion |  |

**ABSTRACT**

|  |
| --- |
|  |
| To develop a program that is efficient enough to perform mathematical computations in real time so that analysis can be made in real time and appropriate action can be taken. NARL is atmospheric research laboratories where they predict atmospheric variations. They give the data to SHAR(Sriharikota) from which most of rocket launches are taking place these days. If you don't correctly predict the variations in atmosphere, rocket launches won't be successful. So data should be accurate, reliable, dependable. One of the major tasks is acquiring the data in real time. Acquiring means not just simply getting some garbage data. It should be accurate or at least good enough to make conclusions. This part of acquiring data has been done using different types of equipment deployed in NARL. Now comes the task of fetching some good results from that data by processing in real time. So the objective is to develop an efficient algorithm which satisfies all the above mentioned criteria and gives us satisfactory results from which useful predictions can be made regarding atmosphere from which many things can be controlled like prediction of rains, whether, wind speeds etc. Useful predictions like when the weather is suitable for launching the rockets. If you launch the rocket and it becomes unsuccessful huge amount of money spent on building that and efforts of the scientists will go in vain. |

1. **Introduction**

**ABOUT THE PROJECT**

NARL has many Radars from which data is collected in different forms and analyzed using different techniques. Among those one of them was HF radar. HF radar used to have some abut 100 antennas deployed in open area at certain altitude through which data is collected. The data is analyzed after different parameter values are captured using appropriate hardware. The technique to analyze data was FFT method i.e Fast Fourier Transform to convert the data obtained to valuable output.

FFT requires some complex mathematical operations. It requires huge processing power which normally is not available in desktops and laptops. So there is need for efficient algorithm which reduces the no of computations required so that run time can be brought down.

The problem was to analyze the data in real time. Since data is acquired in real time(to predict the temperature and atmospheric related factors) data should also be analyzed in real time. Since the data is huge, huge means I mean about 1,00,000,00 - 5,00,000,00 data points in a single second, there should be a efficient mechanism to analyze such huge data in real time. To analyze one set of data it used to take about half an hour to an hour through the program/software previously deployed.

The task is to overcome the problem of analyzing the data in real time by overcoming such large amount of time for analyzing single set of data. Our task is to bring down the run time as much as possible.

**ABOUT THE COMPANY**

National Atmospheric Research Laboratory (NARL) is an autonomous Research Institute funded by the [Department of Space](https://en.wikipedia.org/wiki/Department_of_Space) of the [Government of India](https://en.wikipedia.org/wiki/Government_of_India). NARL is engaged in fundamental and applied research in the field of [Atmospheric Sciences](https://en.wikipedia.org/wiki/Atmospheric_Sciences). The research institute was started in 1992 as National Mesosphere-Stratosphere-Troposphere (MST) Radar Facility (NMRF). Over the years many other facilities such as Mie/Rayleigh [Lidar](https://en.wikipedia.org/wiki/Lidar), Lower atmospheric [wind profiler](https://en.wikipedia.org/wiki/Wind_profiler), optical [rain gauge](https://en.wikipedia.org/wiki/Rain_gauge), [disdrometer](https://en.wikipedia.org/wiki/Disdrometer), automated [weather stations](https://en.wikipedia.org/wiki/Weather_stations) etc. were added. The NMRF was then expanded into a research institute and renamed as National Atmospheric Research Laboratory on 22 September 2005.

2.**Related work**

**Literature Study**

Under the guidance of scientists we have gone through many books relating to digital signal processing and antennas. Among the books "The Scientist and Engineers Guide To Digital Signal Processing" gave a proper insight of how DSP is done. We have also gone through many NPTEL videos on DSP and came to know that FFT is one of the techniques widely used these days in DSP.

There were some header files in C relating to FFT. We opened the header file and made a detailed study of how exactly FFT was implemented and what are functions readily available and what we need to modify in order to meet our needs. Since we are working on HF Radar, documentation relating to HF radar is thoroughly studied i.e how is the data collected by HF radar, in what form, format of header file, how many channels etc. Studied the complexities of various algorithms and advantages and disadvantages of those algorithms. FFT was the algorithm with least complexity available.

**PROPOSED SYSTEM**

1)FFT was the only Algorithm available that best suits the purpose.

2)Previously used MATLAB hasn't been able to meet the needs due to runtime. Complexity was in the order of O(n^4) which signifies a very large complexity.

3)It took large runtimes than desired. So there was a need to reduce the complexity so that it could run much faster and process huge amounts of data in real time so that appropriate actions can be taken.

4)Implemented the FFT Algorithm in C which is believed to do well compared to MATLAB since MATLAB works based on C i.e internally MATLAB needs to convert the code to C and then process.

5)Runtimes hasn't been improved much compared to previous implementation using MATLAB. Even though some efficiency had been obtained it was not to the proper level so that useful results can be obtained in real time. There was a need to further improve the performance.

6) Complexity was reduced, even though there was a need for something better technique apart from reducing complexity.

7)Why not go with parallelism?

8)Introduce parallelism into code when ever and where ever possible so that you can achieve performance much better.

9)Theoretically you can achieve the performance n times, where n is no of cores the program can use. Even though practically you can achieve that much better performance but still you can get performance far better than serial code execution.

9)Since the radar will be gathering data through 5 channels and computation that needs to be performed on the each channel remains the same, parallelism can be introduced to make efficient use of computer cores so that run times can be reduced.

10)Parallelize the computations on each channel by invoking a thread or two depending upon the need for each channel(Threads a concept in parallelism).

**Implementation of System**

1)Data points which are collected from HF\_rader are available in form of binary files ,which are generally of size varying from 500 MB to 1.2 TB

2)These Binary data is taken as input and converted to proper format using Header structure.Now we have binary data in proper required format.

3)Above System consists of 5 Channel which inturn seperated as Q and I Channels.So we need to apply fourier Transform on these data points from Q and I as Real and Imaginary respectively.For each channel we get a distict value of conjuagate result.

4)To perform this operation we call function from separate files called fft.c and aki\_header.c

5)As This operation is performed serially on each of channels,We can parallise this operation on each of channels seperately.We use function called channel runner,Which uses POSIX Threads to parallise the operation.

6)In the end we perform S value calculation for each of channel data by using formula :

S = ((complexi \* complexconjj) / (complexi \*complexconji) \* (complexj\*complexconjj))

7)We output text file consisting of Input and Output data points in human readable format.

HFfftcomplexdata.txt

HF1ffttestdata.txt

HF1testdata.txt

HFsvalue.txt

Now we can analyse results from these text files easily.

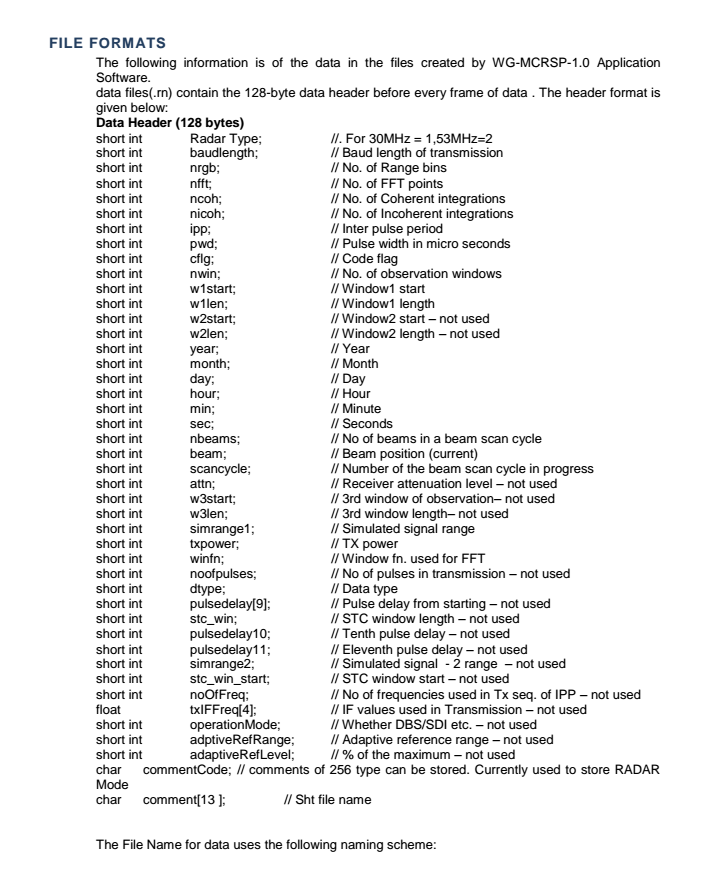
8)Total process time taken in above system is quite small compared to other serial system which perform above operations serially one after other channel.

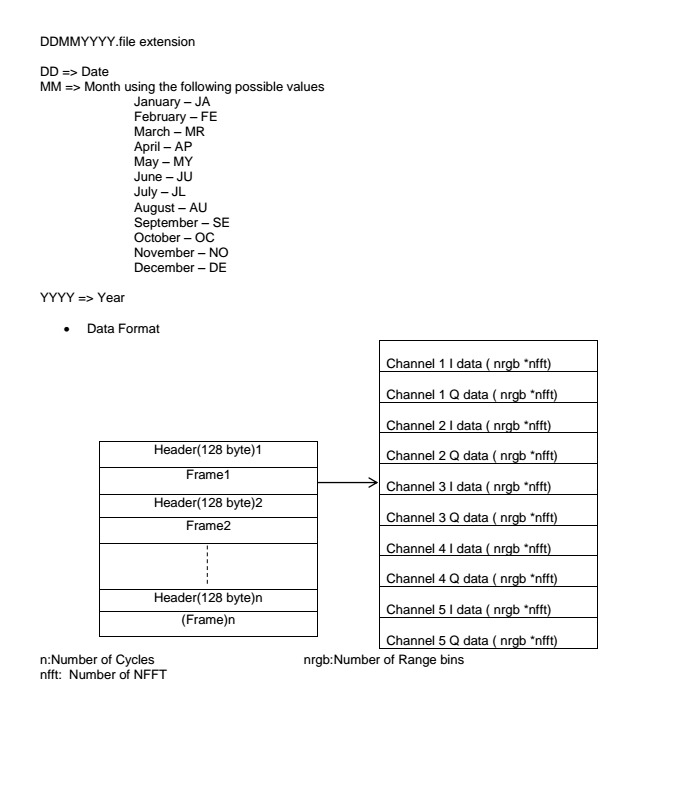
9)Working on high speed super computers will certainly improve performance of this code.

**Header**

Data is in the form of stream of bytes. Similar to network protocols which will have a defined header, here also we have a defined header.

Depending on the parameters in header we need to manipulate with the data i.e various parameters in the header give useful information about the data we operate on.





File is stored in particular format i.e date followed by month represented by two characters as you can see from the above image followed four digit year.

Data is in the form of frames. Header is of size 128 bytes. Each header is followed by frame for which it holds the data about. HF Radar has 5 channels. Each channel has 2 components i.e i component and j component. Since 5 channels so 5 \* 2 =10 values.

As you can see from above figure that each from holds the data of 5 channels i and j components.

**Fourier Analysis**

Fourier Analysis The principle of Fourier Analysis is to ‘test’ for the presence of each frequency component by multiplying the waveform, f(t), by a sine and cosine waveform of the same test frequency and average the results over one or more cycles of the test frequency. For example: 2 0 2 0 ( ) 2 ( ) sin( ) in phase component ( ) 2 ( ) cos( ) quadrature component an f t x n t bn f t x n t π π ω ω = = ∫ ∫ Figure 2 The magnitude of the harmonic can easily be determined as: 1 2 2 2 mn an bn () ( () ()) = + Figure 3 In the case of digital Fourier Analysis, the sample waveform is multiplied by numbers representing the sample of the test sine and cosine waveforms.

**FFT**

The fast Fourier transform (FFT) is a [discrete Fourier transform](http://mathworld.wolfram.com/DiscreteFourierTransform.html) [algorithm](http://mathworld.wolfram.com/Algorithm.html) which reduces the number of computations needed for N points from 2N^2 to 2NlgN, where [lg](http://mathworld.wolfram.com/Lg.html) is the base-2 [logarithm](http://mathworld.wolfram.com/Logarithm.html).

FFTs were first discussed by Cooley and Tukey (1965), although Gauss had actually described the critical factorization step as early as 1805 (Bergland 1969, Strang 1993). A [discrete Fourier transform](http://mathworld.wolfram.com/DiscreteFourierTransform.html) can be computed using an FFT by means of the [Danielson-Lanczos lemma](http://mathworld.wolfram.com/Danielson-LanczosLemma.html) if the number of points N is a [power](http://mathworld.wolfram.com/Power.html) of two. If the number of points N is not a [power](http://mathworld.wolfram.com/Power.html) of two, a transform can be performed on sets of points corresponding to the prime factors of N which is slightly degraded in speed. An efficient real Fourier transform algorithm or a fast [Hartley transform](http://mathworld.wolfram.com/HartleyTransform.html) (Bracewell 1999) gives a further increase in speed by approximately a factor of two. Base-4 and base-8 fast Fourier transforms use optimized code, and can be 20-30% faster than base-2 fast Fourier transforms. [prime](http://mathworld.wolfram.com/PrimeNumber.html) factorization is slow when the factors are large, but discrete Fourier transforms can be made fast for N=2, 3, 4, 5, 7, 8, 11, 13, and 16 using the [Winograd transform](http://mathworld.wolfram.com/WinogradTransform.html) [algorithm](http://mathworld.wolfram.com/Algorithm.html).

Fast Fourier transform algorithms generally fall into two classes: decimation in time, and decimation in frequency. The Cooley-Tukey FFT [algorithm](http://mathworld.wolfram.com/Algorithm.html) first rearranges the input elements in bit-reversed order, then builds the output transform (decimation in time). The basic idea is to break up a transform of length N into two transforms of length N/2 using the identity

|  |
| --- |
| sum_(n=0)^(N-1)a_ne^(-2piink/N)=sum_(n=0)^(N/2-1)a_(2n)e^(-2pii(2n)k/N)   +sum_(n=0)^(N/2-1)a_(2n+1)e^(-2pii(2n+1)k/N)   =sum_(n=0)^(N/2-1)a_n^(even)e^(-2piink/(N/2))   +e^(-2piik/N)sum_(n=0)^(N/2-1)a_n^(odd)e^(-2piink/(N/2)), |

sometimes called the [Danielson-Lanczos lemma](http://mathworld.wolfram.com/Danielson-LanczosLemma.html). The easiest way to visualize this procedure is perhaps via the [Fourier matrix](http://mathworld.wolfram.com/FourierMatrix.html).

The Sande-Tukey [algorithm](http://mathworld.wolfram.com/Algorithm.html) (Stoer and Bulirsch 1980) first transforms, then rearranges the output values (decimation in frequency).

**Difference between FFT and DFT**

FFT The FFT, or Fast Fourier Transform is a method of calculating harmonics not one at a time, but as a group, using a special algorithm. The FFT requires much less processing power than a DFT for the same number of harmonic results. An FFT however, requires that the number of samples being analyzed to be a binary number e.g. a power of two.

The disadvantage of the DFT technique is that it requires each harmonic to be calculated separately, which requires much more processing power. However, if that processing power is available, then the DFT provides very accurate answers

**Modules**

The above proposed system consists of 3 Important Modules.They are:

1.Main - HF\_Rader.c

2.Header - Aki\_Header.c

3.Tranform - Function FFT.c

**Main : HF\_Rader.c**

This Module forms the core of system.It takes data from a binary file and process data parallely .

Each of different Compoents of Module are explained in detail below :

1.Channel Input Structure:

**struct channel\_input**

**{**

**float \*array1;**

**float \*array2;**

**complex float \*channelval;**

**complex float \*channel1conj;**

**};**

array1 and array2 take each of real and imaginary points respectively.

Channelval values are calculated before conjugation for a channel.

Channelconj is conjugated value set of points for a channel.

2. Code forming Crux of parallisation:

**void\* channel\_runner(void\* arg)**

above function channel\_runner perform job of parallisation.

**size\_t m = CCOUNT;**

**status = transform(array1,array2,m);**

Above transform function is called from fft.c .This function performs fourier transform on array1 and array2 data points.

**complex float \*channel1;**

**channel1=(complex float \*) malloc(CCOUNT\*sizeof(complex float));**

**for (j=0 ; j<CCOUNT ; j++)**

**{**

**\*(channel1+j) = \*(array1+j) + \*(array2+j)\*I;**

**}**

Convertion to Complex FFT is performed in the above code snippet.

**fftshift(channel1, count);**

fftshift function from aki\_header.h is used for Performing FFT Shift.

**complex float \*channel1conj;**

**channel1conj=(complex float \*) malloc(CCOUNT\*sizeof(complex float));**

**channel1conj = conjugate(channel1,channel1conj,count);**

**limit\_ptr->channel1conj = channel1conj ;**

Conjugation is performed using conjugate function from fft.c

Finally freeing data as below:

free(channel1conj);

free(channel1);

free(limit\_ptr);

3.Structure of Header is given in following Code:

**struct Header**

**{**

**short int Radar\_Type; //. For 30MHz = 1,53MHz=2**

**short int baudlength; // Baud length of transmission**

**short int nrgb; // No. of Range bins**

**short int nfft; // No. of FFT points**

**short int ncoh; // No. of Coherent integrations**

**short int nicoh; // No. of Incoherent integrations**

**short int ipp; // Inter pulse period**

**short int pwd; // Pulse width in micro seconds**

**short int cflg; // Code flag**

**short int nwin; // No. of observation windows**

**short int w1start; // Window1 start**

**short int w1len; // Window1 length**

**short int w2start; // Window2 start â€“ not used**

**short int w2len; // Window2 length â€“ not used**

**short int year; // Year**

**short int month; // Month**

**short int day; // Day**

**short int hour; // Hour**

**short int min; // Minute**

**short int sec; // Seconds**

**short int nbeams; // No of beams in a beam scan cycle**

**short int beam; // Beam position (current)**

**short int scancycle; // Number of the beam scan cycle in progress**

**short int attn; // Receiver attenuation level â€“ not used**

**short int w3start; // 3rd window of observationâ€“ not used**

**short int w3len; // 3rd window lengthâ€“ not used**

**short int simrange1; // Simulated signal range**

**short int txpower; // TX power**

**short int winfn; // Window fn. used for FFT**

**short int noofpulses; // No of pulses in transmission â€“ not used**

**short int dtype; // Data type**

**short int pulsedelay[9]; // Pulse delay from starting â€“ not used**

**short int stc\_win; // STC window length â€“ not used**

**short int pulsedelay10; // Tenth pulse delay â€“ not used**

**short int pulsedelay11; // Eleventh pulse delay â€“ not used**

**short int simrange2; // Simulated signal - 2 range â€“ not used**

**short int stc\_win\_start; // STC window start â€“ not used**

**short int noOfFreq; // No of frequencies used in Tx seq. of IPP â€“ not used**

**float txIFFreq[4]; // IF values used in Transmission â€“ not used**

**short int operationMode; // Whether DBS/SDI etc. â€“ not used**

**short int adptiveRefRange; // Adaptive reference range â€“ not used**

**short int adaptiveRefLevel; // % of the maximum â€“ not used**

**char commentCode; // comments of 256 type can be stored. Currently used to store RADAR Mode**

**char comment[13 ]; // Sht file name**

**float data[512000];**

**};**

4.Main

**fp=fopen("15AU2015SASHT1.r1","rb");**

Taking large file as Input.

**while(1)**

**{**

**int n=0;**

**if(counter == 1)**

**{**

**break;**

**}**

Starting an infinite loop to take datapoints in channel wise pattern.

**//Channel 1 I data**

**float \*array1;**

**array1=(float \*) malloc(CCOUNT\*sizeof(float));**

**for (n ; n<CCOUNT ; n++)**

**{**

**\*(array1+n) = s.data[n];**

**}**

**arrayI[0] = array1;**

**//Channel 1 Q data**

**float \*array2;**

**array2=(float \*) malloc(CCOUNT\*sizeof(float));**

**for (n,j=0 ; n<2\*CCOUNT ; n++,j++)**

**{**

**\*(array2+j) = s.data[n];**

**}**

**arrayQ[0] = array2;**

Channel 1 I and Q data are taken into array1 and array2 respectivly.

5.Multi-Threading part:

**pthread\_t tids[CVAL];**

**Calculating Thread Id**

**for(i=0; i < CVAL ; i++)**

**{**

**args[i].array1 = arrayI[i];**

**args[i].array2 = arrayQ[i];**

**pthread\_attr\_t attr;**

**pthread\_attr\_init(&attr);**

**pthread\_create(&tids[i], &attr, channel\_runner, &args[i]);**

**}**

Creating attributes for each thread is performed above.

**for( i=0; i < CVAL ; i++)**

**{**

**pthread\_join(tids[i], NULL);**

**complexresult[i] = args[i].channelval;**

**complexCresult[i] = args[i].channel1conj;**

**}**

Waiting until threads are done its work.

Calculation of S value is done in below code Snippet:

Sval is the final result of of data in proper format as required by HF Rader.

**Sval = (complex float \*\*\*)malloc(r\*sizeof(complex float\*\*));**

**for (i = 0; i< r; i++)**

**{**

**Sval[i] = (complex float \*\*) malloc(c\*sizeof(complex float \*));**

**for (j = 0; j < c; j++)**

**{**

**Sval[i][j] = (complex float \*)malloc(CCOUNT\*sizeof(complex float));**

**}**

**}**

**int k;**

**for(i=0;i<CVAL;i++)**

**{**

**for(j=0;j<CVAL;j++)**

**{**

**if(i==j)**

**{**

**break;**

**}**

**for (k=0 ; k<CCOUNT ; k++)**

**{**

**\*(Sval[i][j] + k) = ((\*(complexresult[i] + k))\*(\*(complexCresult[j] + k)))/(csqrtf((\*(complexresult[i] + k))\*(\*(complexCresult[i] + k)))\*csqrtf((\*(complexresult[j] + k))\*(\*(complexCresult[j] + k))));**

**}**

**}**

**}**

**counter++;**

This is used to keep control on number of frames to process.

**Tranform Functions: FFT.c**

1.Tranform Function:

**int transform(float real[], float imag[], size\_t n) {**

**printf("Entered the transform...\n");**

**if (n == 0)**

**return 1;**

**else if ((n & (n - 1)) == 0) // Is power of 2**

**return transform\_radix2(real, imag, n);**

**else // More complicated algorithm for arbitrary sizes**

**return transform\_bluestein(real, imag, n);**

**}**

Computes the discrete Fourier transform (DFT) of the given complex vector, storing the result back into the vector.

The vector can have any length. This is a wrapper function. Returns 1 (true) if successful, 0 (false) otherwise (out of memory).

2.Inverse Transfrom:

**int inverse\_transform(float real[], float imag[], size\_t n) {**

**return transform(imag, real, n);**

**}**

Computes the inverse discrete Fourier transform (IDFT) of the given complex vector, storing the result back into the vector.

The vector can have any length. This is a wrapper function. This transform does not perform scaling, so the inverse is not a true inverse.

Returns 1 (true) if successful, 0 (false) otherwise (out of memory).

3.Transform radix2 function:

**int transform\_radix2(float real[], float imag[], size\_t n) {**

**// Variables**

**printf("Entered the transform\_radix2...\n");**

**int status = 0;**

**unsigned int levels;**

**float \*cos\_table, \*sin\_table;**

**size\_t size;**

**size\_t i;**

**// Compute levels = floor(log2(n))**

**{**

**size\_t temp = n;**

**levels = 0;**

**while (temp > 1) {**

**levels++;**

**temp >>= 1;**

**}**

**if (1u << levels != n)**

**return 0; // n is not a power of 2**

**}**

**// Trignometric tables**

**if (SIZE\_MAX / sizeof(float) < n / 2)**

**return 0;**

**size = (n / 2) \* sizeof(float);**

**cos\_table = malloc(size);**

**sin\_table = malloc(size);**

**if (cos\_table == NULL || sin\_table == NULL)**

**goto cleanup;**

**for (i = 0; i < n / 2; i++) {**

**cos\_table[i] = cos(2 \* M\_PI \* i / n);**

**sin\_table[i] = sin(2 \* M\_PI \* i / n);**

**}**

**// Bit-reversed addressing permutation**

**for (i = 0; i < n; i++) {**

**size\_t j = reverse\_bits(i, levels);**

**if (j > i) {**

**float temp = real[i];**

**real[i] = real[j];**

**real[j] = temp;**

**temp = imag[i];**

**imag[i] = imag[j];**

**imag[j] = temp;**

**}**

**}**

**// Cooley-Tukey decimation-in-time radix-2 FFT**

**for (size = 2; size <= n; size \*= 2) {**

**size\_t halfsize = size / 2;**

**size\_t tablestep = n / size;**

**for (i = 0; i < n; i += size) {**

**size\_t j;**

**size\_t k;**

**for (j = i, k = 0; j < i + halfsize; j++, k += tablestep)**

**{**

**float tpre = real[j+halfsize] \* cos\_table[k] + imag[j+halfsize] \* sin\_table[k];**

**float tpim = -real[j+halfsize] \* sin\_table[k] + imag[j+halfsize] \* cos\_table[k];**

**real[j + halfsize] = real[j] - tpre;**

**imag[j + halfsize] = imag[j] - tpim;**

**real[j] += tpre;**

**imag[j] += tpim;**

**}**

**}**

**if (size == n) // Prevent overflow in 'size \*= 2'**

**break;**

**}**

**status = 1;**

**cleanup:**

**free(cos\_table);**

**free(sin\_table);**

**return status;**

**}**

Computes the discrete Fourier transform (DFT) of the given complex vector, storing the result back into the vector.

The vector's length must be a power of 2. Uses the Cooley-Tukey decimation-in-time radix-2 algorithm.

Returns 1 (true) if successful, 0 (false) otherwise (n is not a power of 2, or out of memory).

4.Transform\_bluestein Function:

**int transform\_bluestein(float real[], float imag[], size\_t n) {**

**printf("Entered the transform\_bluestein...\n");**

**// Variables**

**int status = 0;**

**float \*cos\_table, \*sin\_table;**

**float \*areal, \*aimag;**

**float \*breal, \*bimag;**

**float \*creal, \*cimag;**

**size\_t m;**

**size\_t size\_n, size\_m;**

**size\_t i;**

**// Find a power-of-2 convolution length m such that m >= n \* 2 + 1**

**{**

**size\_t target;**

**if (n > (SIZE\_MAX - 1) / 2)**

**return 0;**

**target = n \* 2 + 1;**

**for (m = 1; m < target; m \*= 2) {**

**if (SIZE\_MAX / 2 < m)**

**return 0;**

**}**

**}**

**// Allocate memory**

**if (SIZE\_MAX / sizeof(float) < n || SIZE\_MAX / sizeof(float) < m)**

**return 0;**

**size\_n = n \* sizeof(float);**

**size\_m = m \* sizeof(float);**

**cos\_table = malloc(size\_n);**

**sin\_table = malloc(size\_n);**

**areal = calloc(m, sizeof(float));**

**aimag = calloc(m, sizeof(float));**

**breal = calloc(m, sizeof(float));**

**bimag = calloc(m, sizeof(float));**

**creal = malloc(size\_m);**

**cimag = malloc(size\_m);**

**if (cos\_table == NULL || sin\_table == NULL**

**|| areal == NULL || aimag == NULL**

**|| breal == NULL || bimag == NULL**

**|| creal == NULL || cimag == NULL)**

**goto cleanup;**

**// Trignometric tables**

**for (i = 0; i < n; i++) {**

**float temp = M\_PI \* (size\_t)((unsigned long long)i \* i % ((unsigned long long)n \* 2)) / n;**

**// Less accurate version if long long is unavailable: float temp = M\_PI \* i \* i / n;**

**cos\_table[i] = cos(temp);**

**sin\_table[i] = sin(temp);**

**}**

**// Temporary vectors and preprocessing**

**for (i = 0; i < n; i++) {**

**areal[i] = real[i] \* cos\_table[i] + imag[i] \* sin\_table[i];**

**aimag[i] = -real[i] \* sin\_table[i] + imag[i] \* cos\_table[i];**

**}**

**breal[0] = cos\_table[0];**

**bimag[0] = sin\_table[0];**

**for (i = 1; i < n; i++) {**

**breal[i] = breal[m - i] = cos\_table[i];**

**bimag[i] = bimag[m - i] = sin\_table[i];**

**}**

**// Convolution**

**if (!convolve\_complex(areal, aimag, breal, bimag, creal, cimag, m))**

**goto cleanup;**

**// Postprocessing**

**for (i = 0; i < n; i++) {**

**real[i] = creal[i] \* cos\_table[i] + cimag[i] \* sin\_table[i];**

**imag[i] = -creal[i] \* sin\_table[i] + cimag[i] \* cos\_table[i];**

**}**

**status = 1;**

**// Deallocation**

**cleanup:**

**free(cimag);**

**free(creal);**

**free(bimag);**

**free(breal);**

**free(aimag);**

**free(areal);**

**free(sin\_table);**

**free(cos\_table);**

**return status;**

**}**

Computes the discrete Fourier transform (DFT) of the given complex vector, storing the result back into the vector.

The vector can have any length. This requires the convolution function, which in turn requires the radix-2 FFT function.

Uses Bluestein's chirp z-transform algorithm. Returns 1 (true) if successful, 0 (false) otherwise (out of memory).

5.Function to perform Real Convolution:

**int convolve\_real(const float x[], const float y[], float out[], size\_t n) {**

**float \*ximag, \*yimag, \*zimag;**

**int status = 0;**

**ximag = calloc(n, sizeof(float));**

**yimag = calloc(n, sizeof(float));**

**zimag = calloc(n, sizeof(float));**

**if (ximag == NULL || yimag == NULL || zimag == NULL)**

**goto cleanup;**

**status = convolve\_complex(x, ximag, y, yimag, out, zimag, n);**

**cleanup:**

**free(zimag);**

**free(yimag);**

**free(ximag);**

**return status;**

**}**

**Computes the circular convolution of the given real vectors. Each vector's length must be the same.**

**Returns 1 (true) if successful, 0 (false) otherwise (out of memory).**

6. Function to perform Complex Convolution:

**int convolve\_complex(const float xreal[], const float ximag[], const float yreal[], const float yimag[], float outreal[], float outimag[], size\_t n) {**

**int status = 0;**

**size\_t size;**

**size\_t i;**

**float \*xr, \*xi, \*yr, \*yi;**

**if (SIZE\_MAX / sizeof(float) < n)**

**return 0;**

**size = n \* sizeof(float);**

**xr = memdup(xreal, size);**

**xi = memdup(ximag, size);**

**yr = memdup(yreal, size);**

**yi = memdup(yimag, size);**

**if (xr == NULL || xi == NULL || yr == NULL || yi == NULL)**

**goto cleanup;**

**if (!transform(xr, xi, n))**

**goto cleanup;**

**if (!transform(yr, yi, n))**

**goto cleanup;**

**for (i = 0; i < n; i++) {**

**float temp = xr[i] \* yr[i] - xi[i] \* yi[i];**

**xi[i] = xi[i] \* yr[i] + xr[i] \* yi[i];**

**xr[i] = temp;**

**}**

**if (!inverse\_transform(xr, xi, n))**

**goto cleanup;**

**for (i = 0; i < n; i++) { // Scaling (because this FFT implementation omits it)**

**outreal[i] = xr[i] / n;**

**outimag[i] = xi[i] / n;**

**}**

**status = 1;**

**cleanup:**

**free(yi);**

**free(yr);**

**free(xi);**

**free(xr);**

**return status;**

**}**

Computes the circular convolution of the given complex vectors. Each vector's length must be the same.

Returns 1 (true) if successful, 0 (false) otherwise (out of memory).

**3.Header: Aki\_Header.c**

1.FFTShift Function:

**void fftshift(complex float \*data, int count)**

**{**

**int k = 0;**

**int c = (int) floor((float)count/2);**

**// For odd and for even numbers of element.We are useing different algorithm Here.**

**if (count % 2 == 0)//This is for Even Count**

**{**

**for (k = 0; k < c; k++)**

**swap(&data[k], &data[k+c]);**

**}**

**else // This is for Odd Count**

**{**

**complex float tmp = data[0];**

**for (k = 0; k < c; k++)**

**{**

**data[k] = data[c + k + 1];**

**data[c + k + 1] = data[k + 1];**

**}**

**data[c] = tmp;**

**}**

**}**

This function is used to shift odd and even points accordingly for FFT.

It uses static function which cannot be accessed outside this file.

swap() function is used in fftshift to use accordingly.

2.IFFTShift Function:

**void ifftshift(complex float \*data, int count) {**

**int k = 0;**

**int c = (int) floor((float)count/2);**

**if (count % 2 == 0)**

**{**

**for (k = 0; k < c; k++)**

**swap(&data[k], &data[k+c]);**

**}**

**else**

**{**

**complex float tmp = data[count - 1];**

**for (k = c-1; k >= 0; k--)**

**{**

**data[c + k + 1] = data[k];**

**data[k] = data[c + k];**

**}**

**data[c] = tmp;**

**}**

**}**

This Function is used for Circular shift of Complex values.

3.Conjugate function:

**complex float \*conjugate(complex float \*in,complex float \*out,int n)**

**{**

**int i;**

**for(i=0;i<n;i++)**

**{**

**complex float z = in[i];**

**complex float z2 = conj(z);**

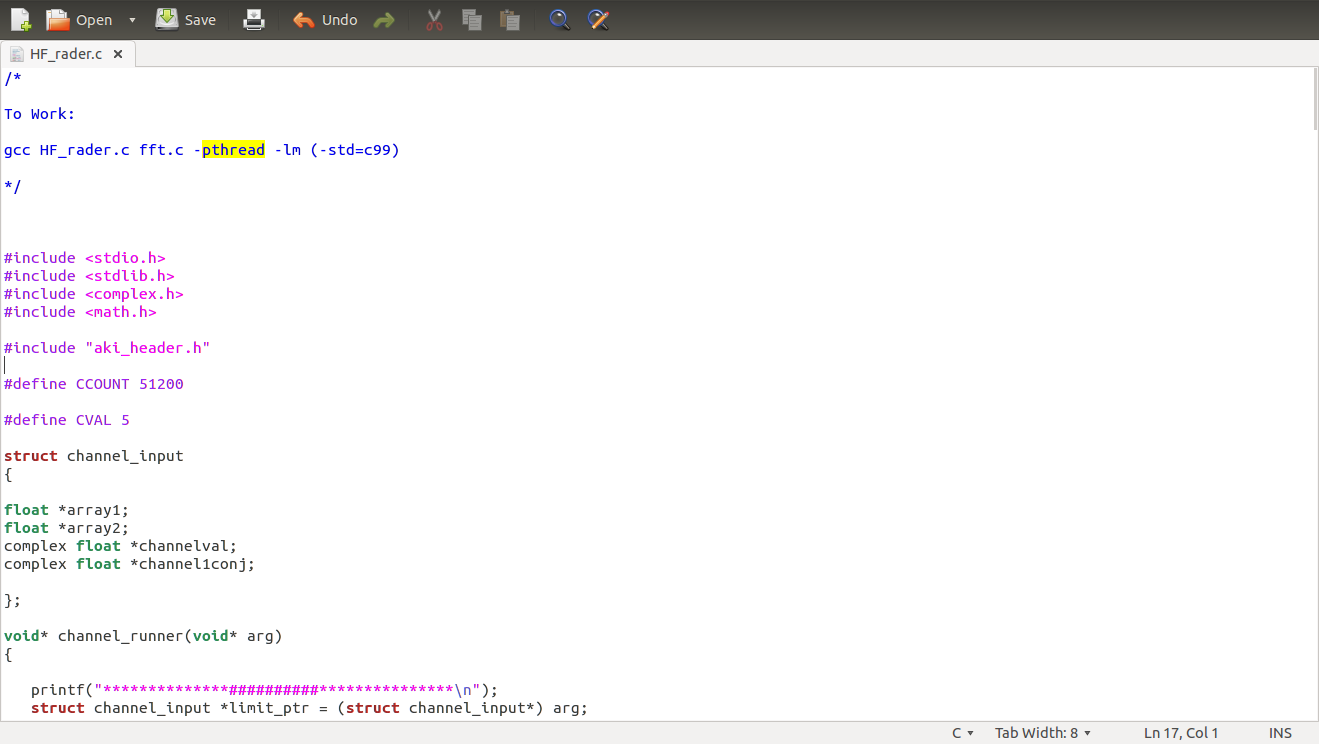
**out[i] = z2;**

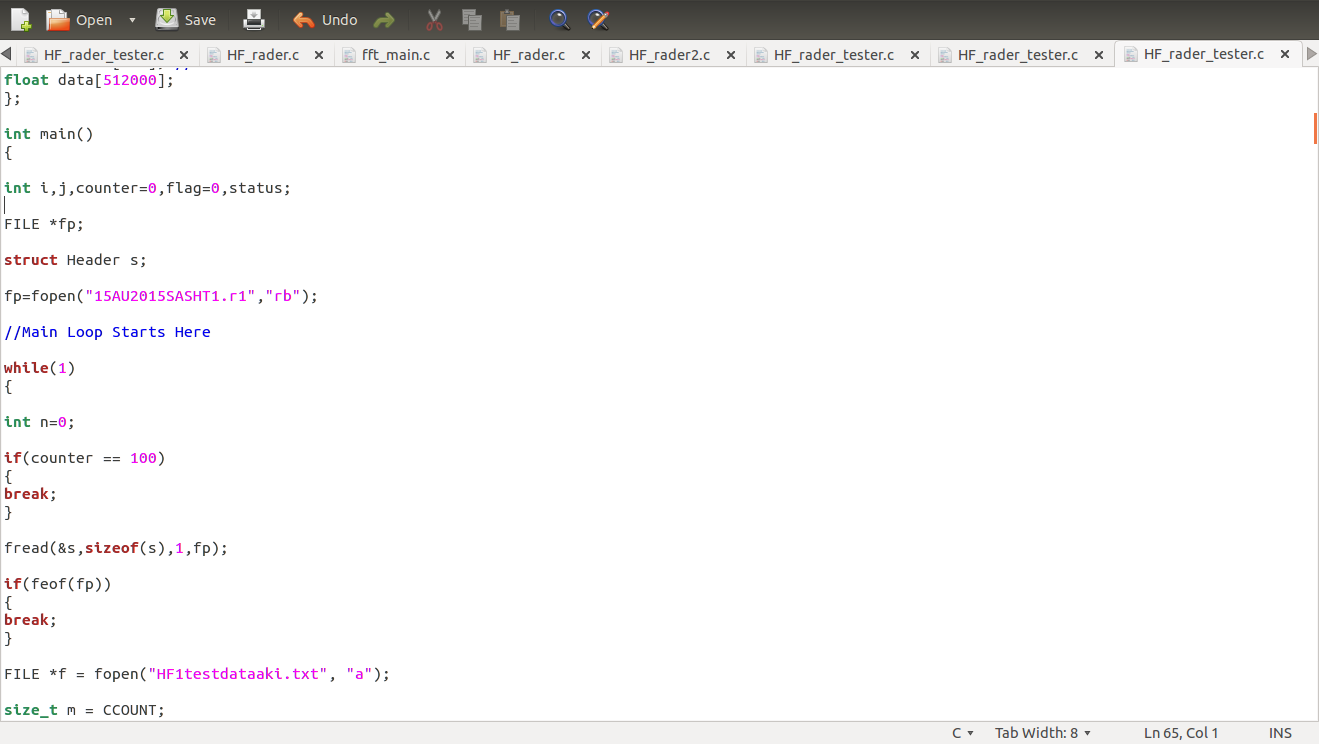
**}**

This Function is used for doing Conjugation for Each Channel Values.

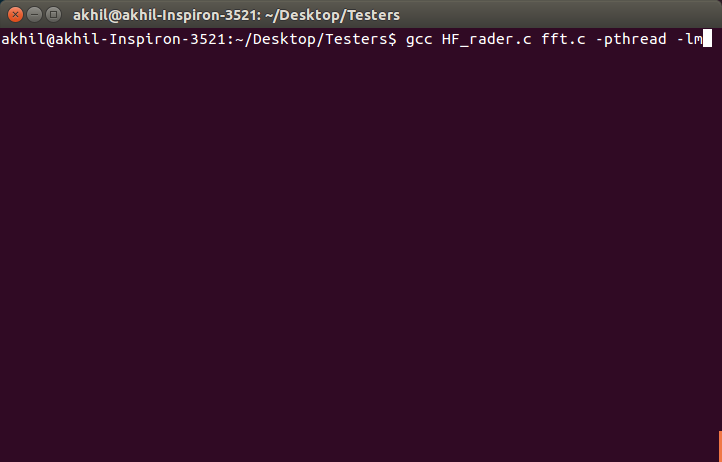
We are using conj() function defined in complex.h header but this function is mainly used to get Proper value as Out vector.

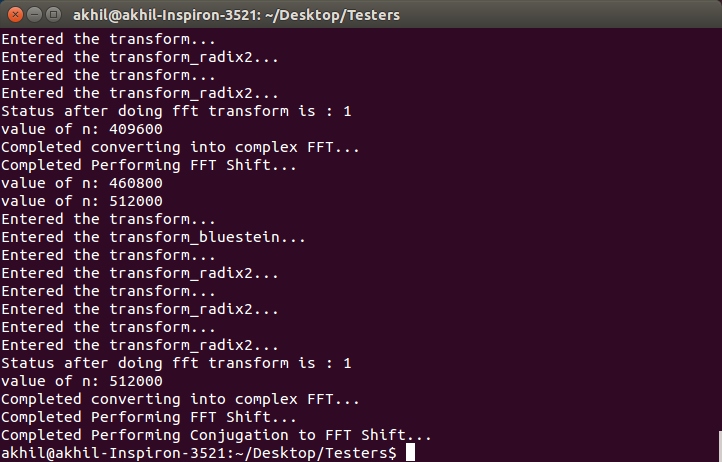
**Screen Shot of Results**

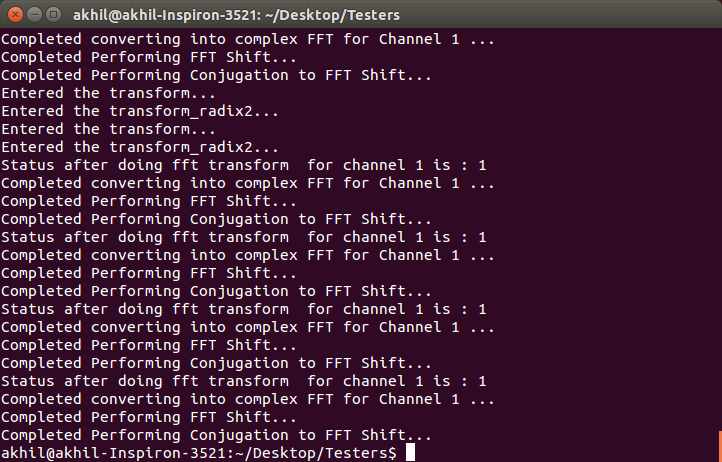
1.Code:



2.Running code on System:

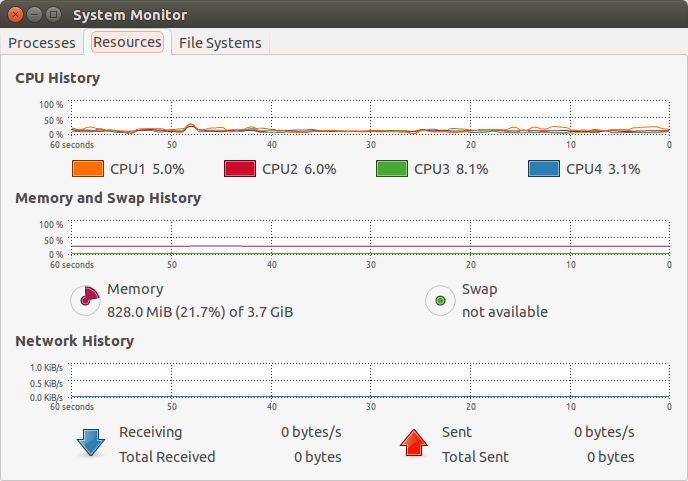




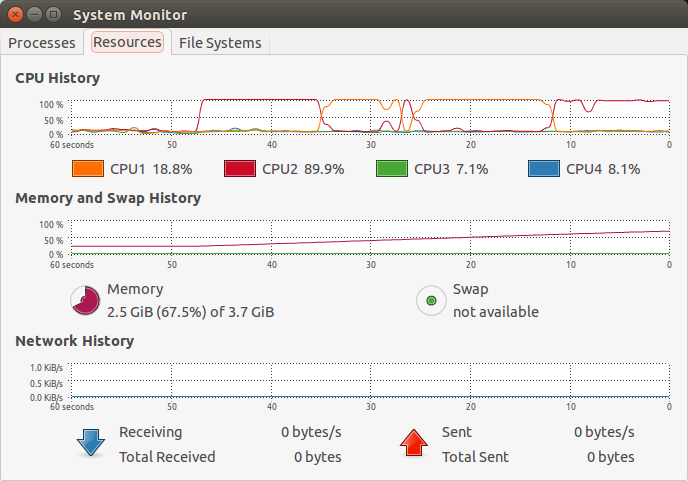


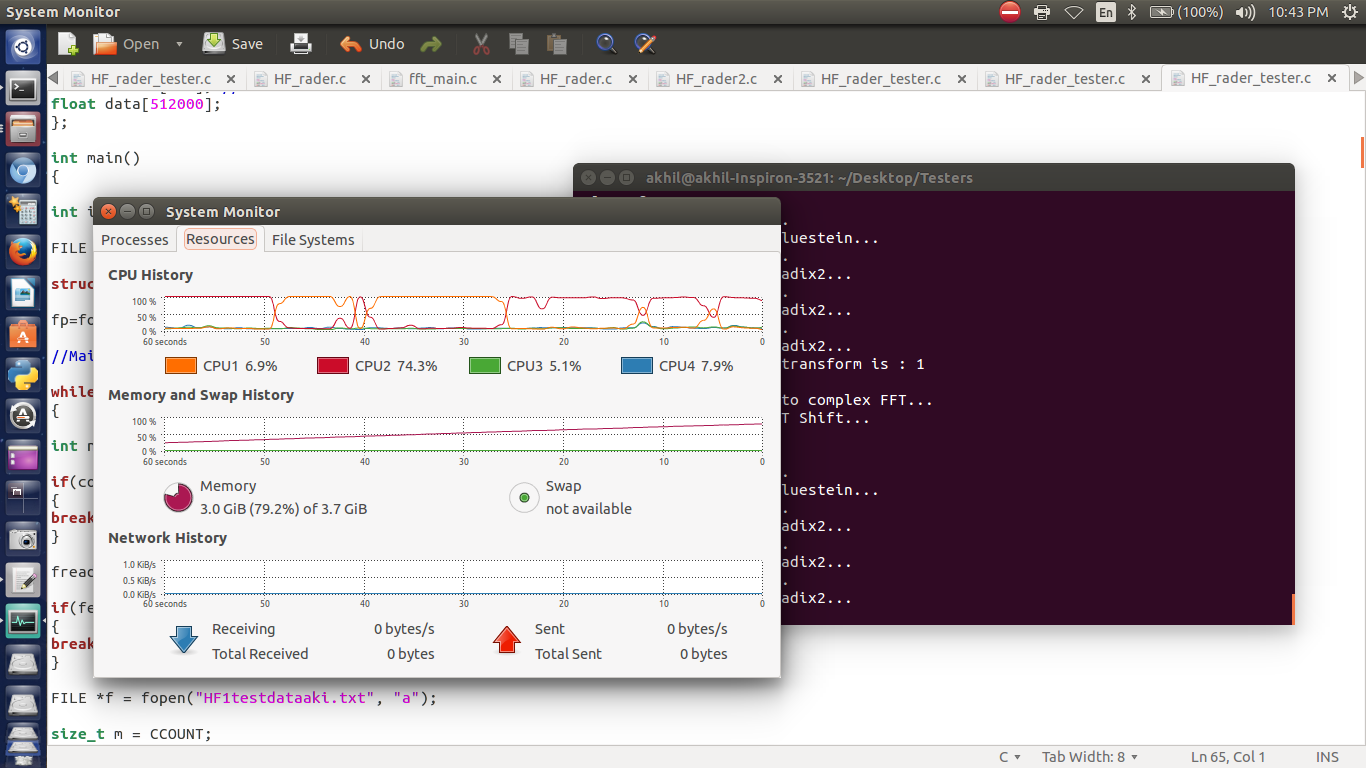
3.System CPU status Referance:

Normal State:

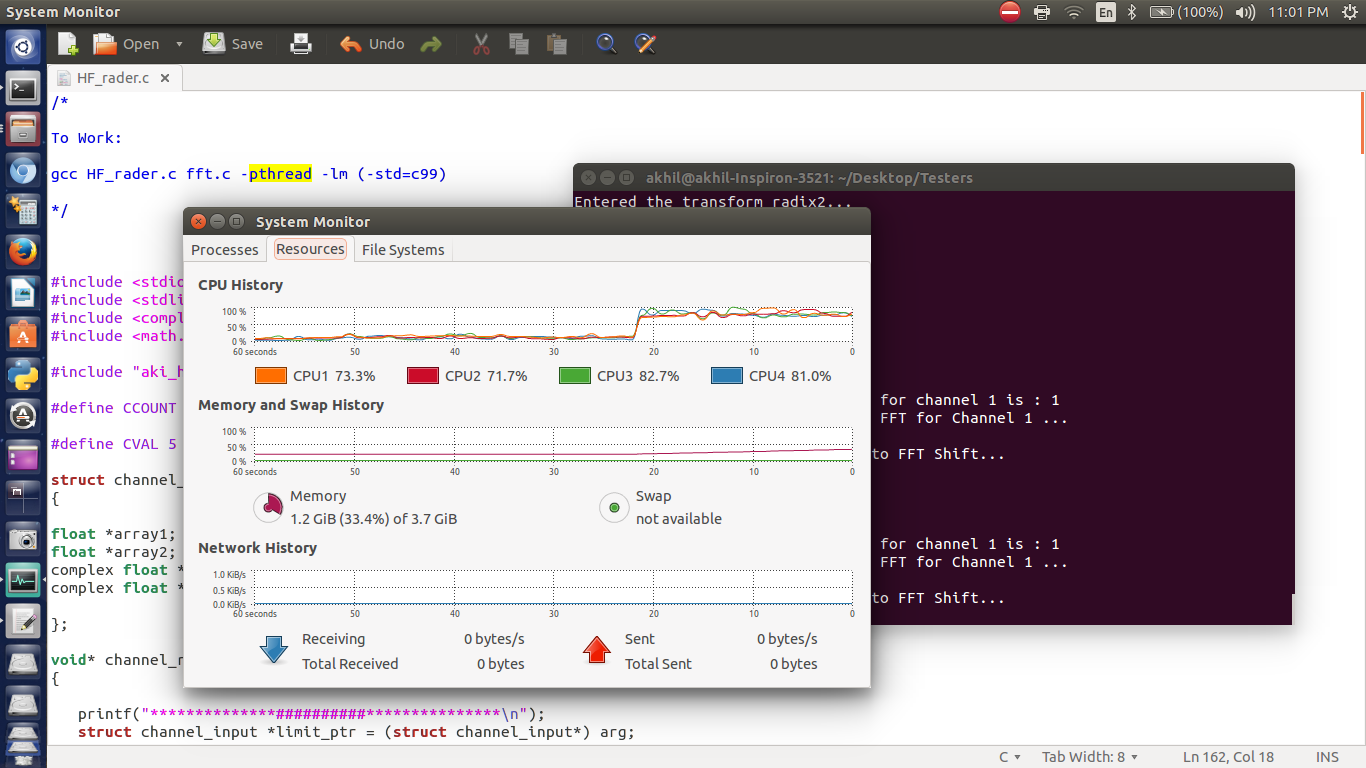


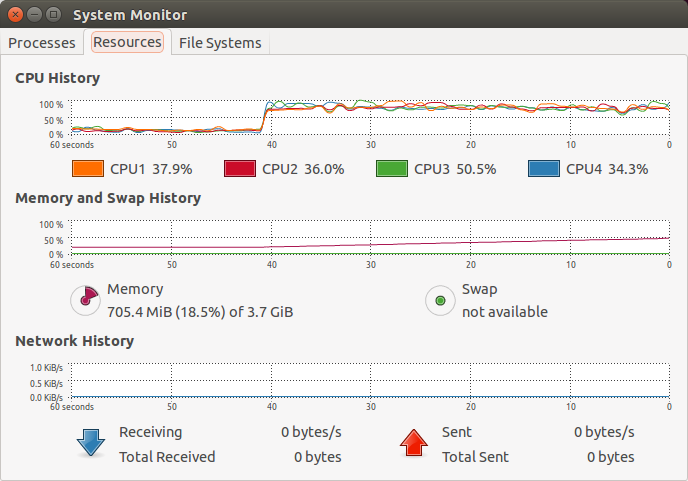
Running Serial Code:

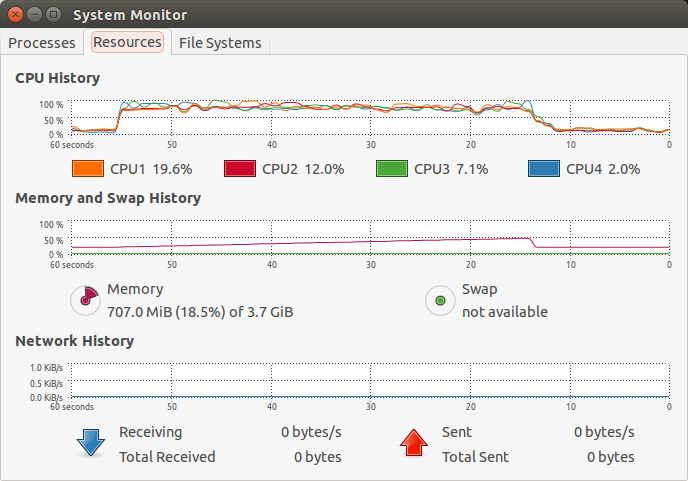
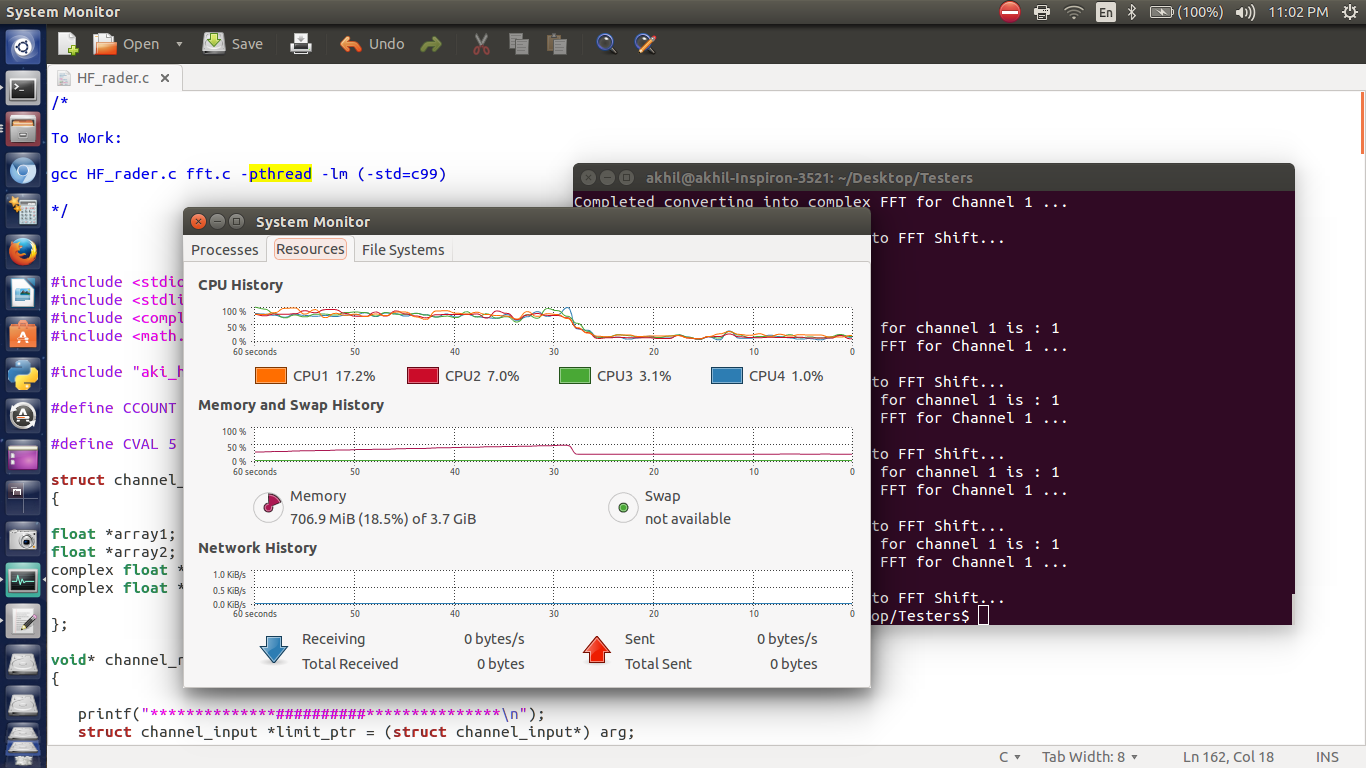




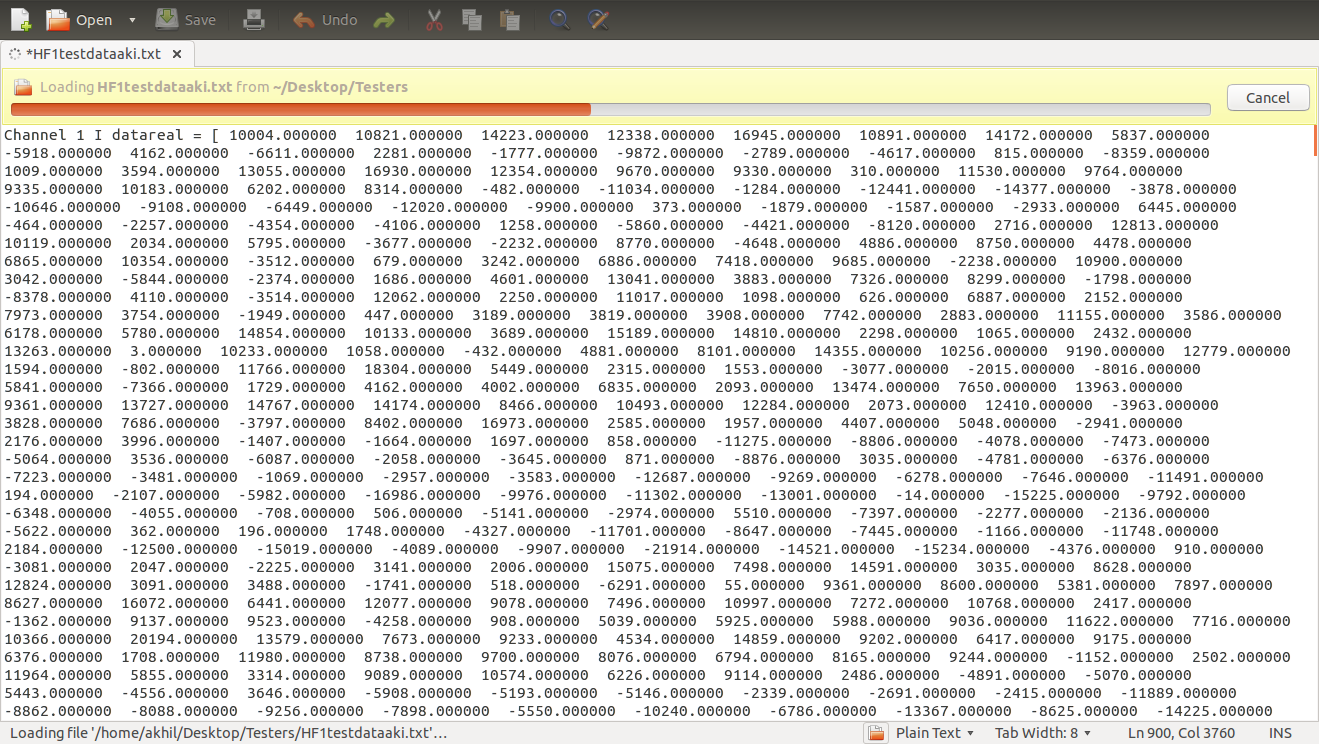
Running Parallel Code:

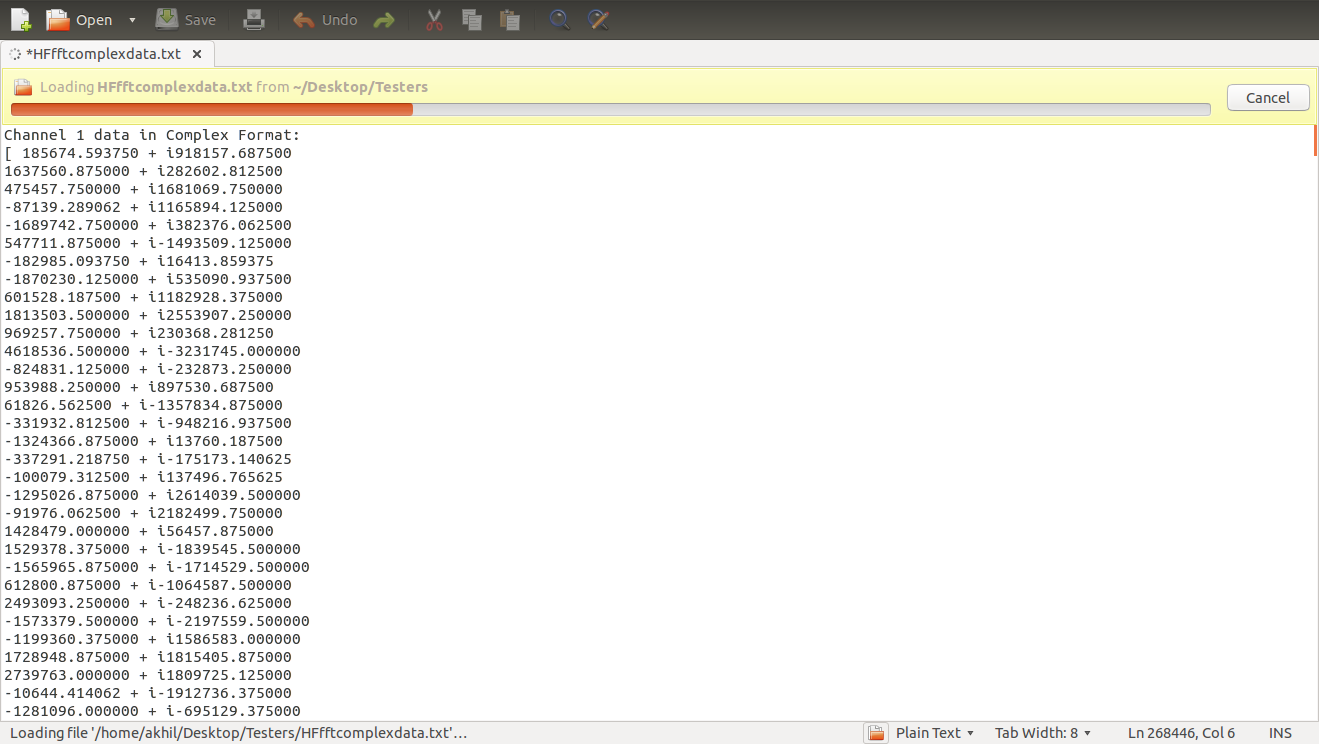


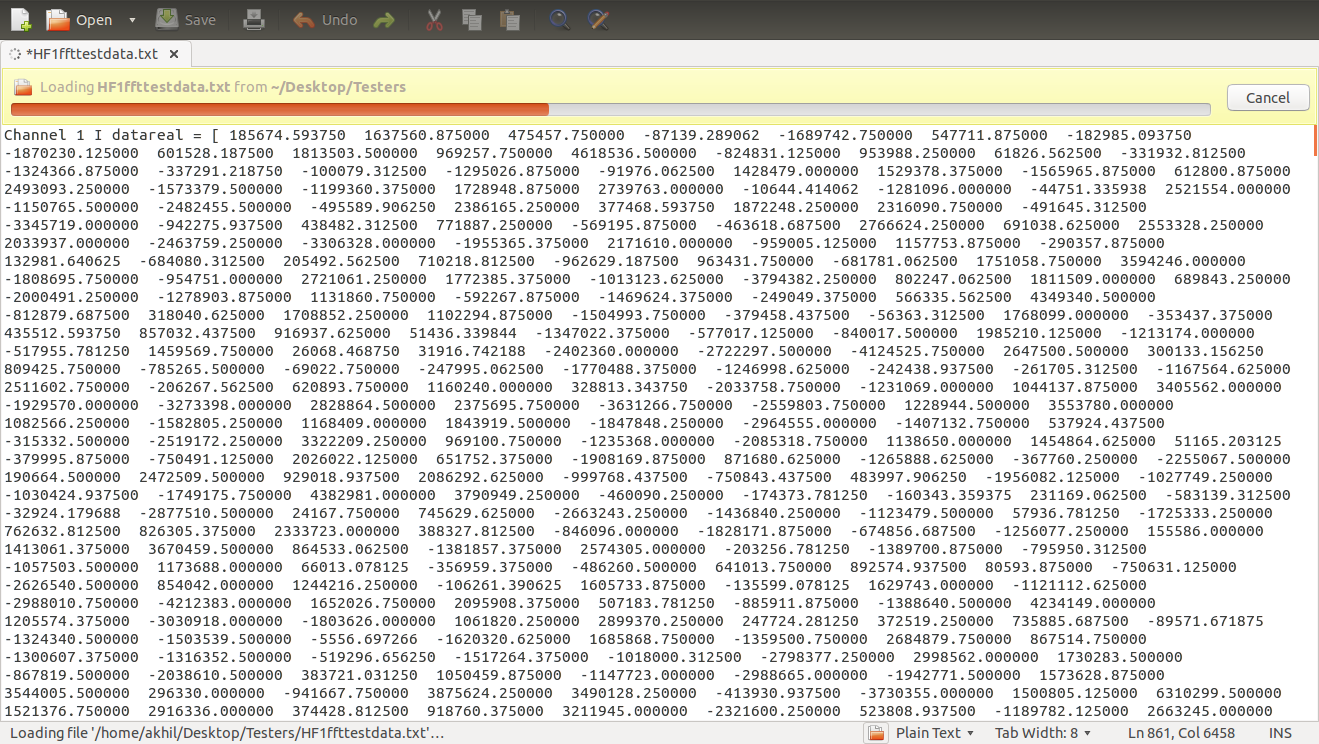




Text Documents with Output of Data Points after Processing:







**Conclusion**

This proposed system has some of important advantages than older system running parallely on MATLAB.

Methods used in our system to improve data processing speed are:

1.Implementaion of Code in “C” ,which is very close to assembly thus reducing complexity of System in converting code to “Assembly”.

2.Used Some of faster FFT transforms like bluestein and radix2 to performs Fourier Transform on data.

3.Using “POSIX Threads” to parallise code to perform computation on each of 5 Channels seperately.

Using these improvements,We could drastically improve performance of HF\_rader data processing System.