



Mokka Digitization for the Next Generation Prototype of AHCAL

Pan Chuzhong, Peking University, P.R.China and FLC Group, DESY

Supervisor: Shaojun Lu, FLC Group, DESY

September 5, 2012

Abstract

In this report, we talk about the Mokka digitization for the next generation prototype of AHCAL. We also talk about designing proper filter to get rid of the noise generated from hardware. Digitization is implemented in C++ code using the standard *.slcio* file format while filter designing is implemented based on *Matlab*.

Contents

1	Introduction	3
2	Raw Physics Data Analysis	3
2.1	3D Position statistics	4
2.2	Time and Energy Statistics	6
2.3	MIP Calibration and MIP cut	8
3	Noise-Adding	11
4	Filter	11
4.1	Frequency-Domain-Filter	12
4.1.1	Basic Concept:Discrete Fourier Transform	12
4.1.2	Implementation of 1-D Frequency-Domain-Filter	15
4.1.3	Implementation of 2-D Frequency-Domain-Filter	18
4.2	Other Filter	20
4.2.1	Z-Transform filter	20
4.2.2	Radon-Transform filter	22

1 Introduction

The International Linear Collider(ILC) is proposed and under construction for the study of $e^+ e^-$ interactions with precise measurement at very high energies (500 GeV, upgrade to 1 TeV) where the production of multi-jet events with new heavy particles is expected. Optimal energy and mass resolution of all components of the final state will be achieved when the final state particle flow is reconstructed with highest possible detector granularity. Whereas all charged particles can be measured with sufficient precision in a large volume tracking system, the neutral particles have to be reconstructed and identified in a calorimeter system: an electromagnetic calorimeter (ECAL, silicon pixels or scintillator strips with W or Fe) followed by a hadronic calorimeter (HCAL, scintillator tiles + SiPM or gas detector as active medium) with highest possible granularity.

The physics prototype of the Analogue Hadronic CALorimeter(AHCAL) built by the CALorimeter for the LInear Collider with Electronics (CALICE) collaboration is a highly granular scintillator-steel sandwich-like calorimeter and a large scale application for Silicon Photomultipliers. This prototype has proved itself a high-efficiency and powerful calorimeter and some successful testbeam experiments have been conducted at DESY, CERN and FNAL. The prototype is shown below in Figure 1

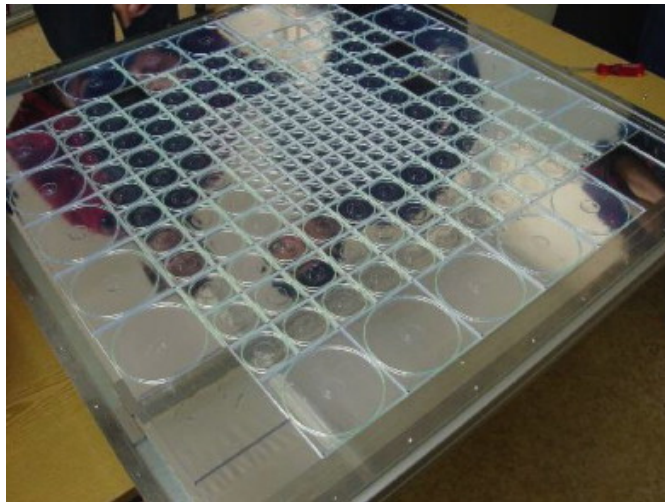


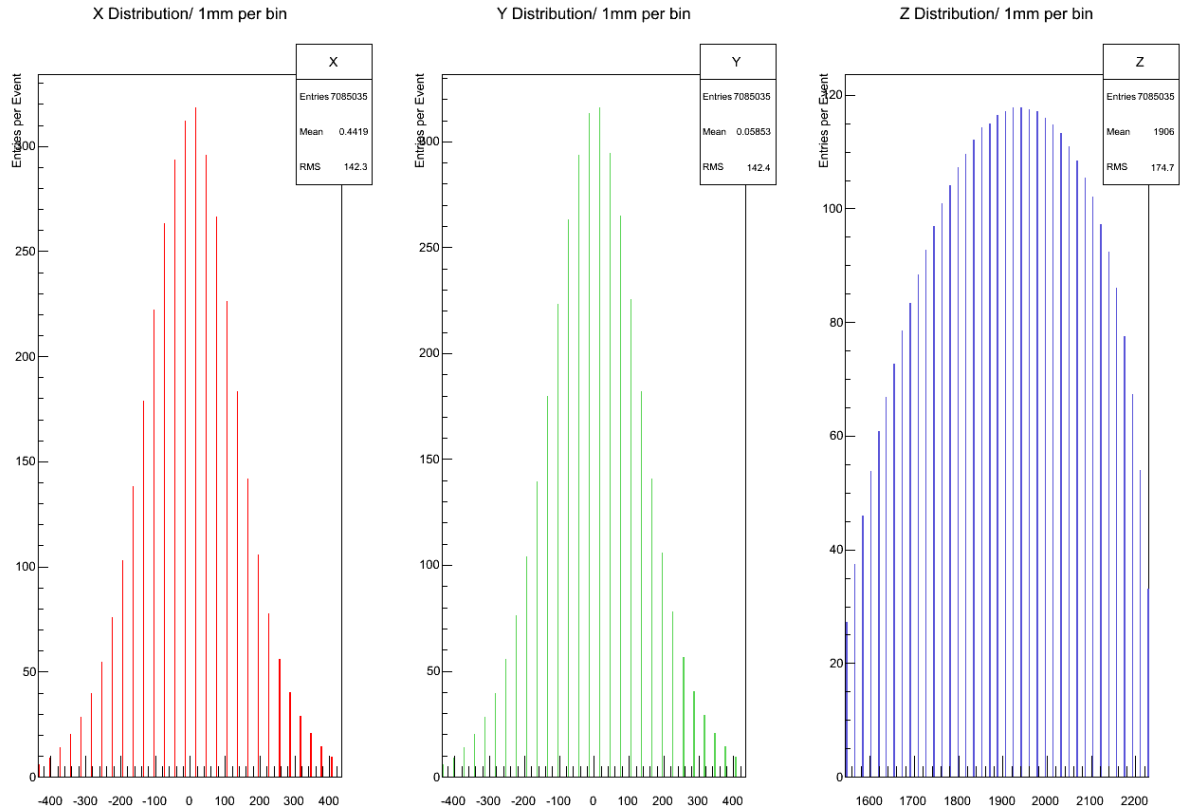
Figure 1: AHCAL Module

2 Raw Physics Data Analysis

At first, we should get to know what does the Mokka data look like. So, let's start with data analysis. We generated the raw physics data of Π^- s of 180GeV for 2000 events. The data file is in the standard *slcio* format.

2.1 3D Position statistics

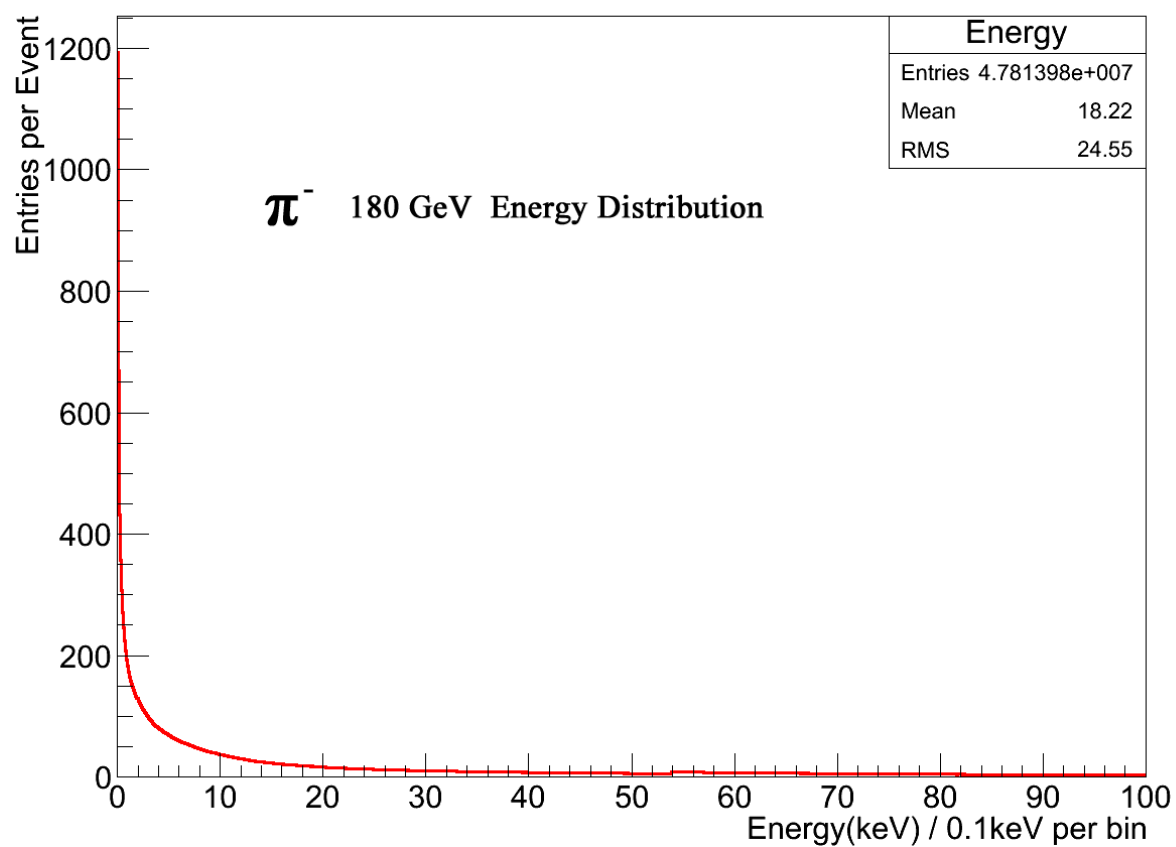
Firstly, we do some statistics over the 3D position of each and every object of the class SimCalirometer.



It's quite obvious that we have a standard gaussian distribution for the X and Y axis whose average are both around 0. But for the Z axis, it's quite different. The most possible position of the particle is at about 1751.66mm on the Z axis.

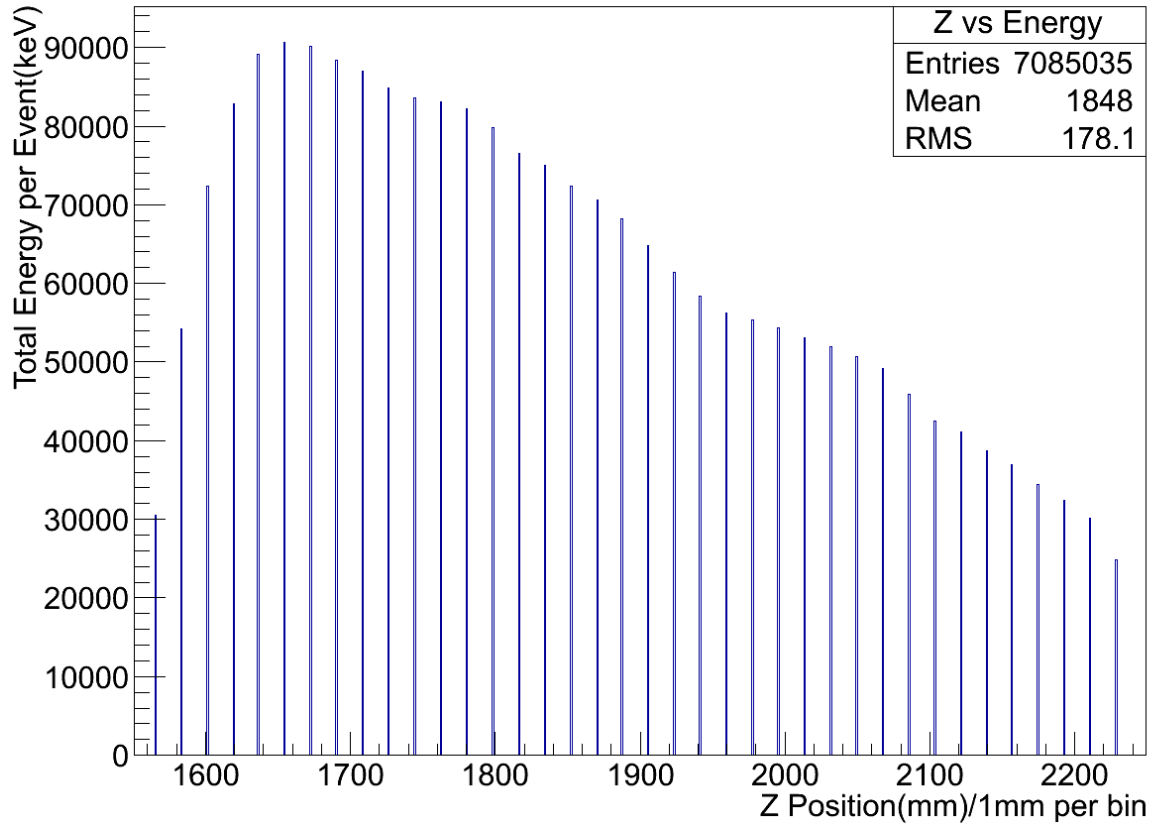
Next, we do statistics on the distribution of both Z position and energy. The first picture below is the distribution of Z position, or layers. The following picture is the energy distribution.

Energy



And also the distribution of Z position, or layers.

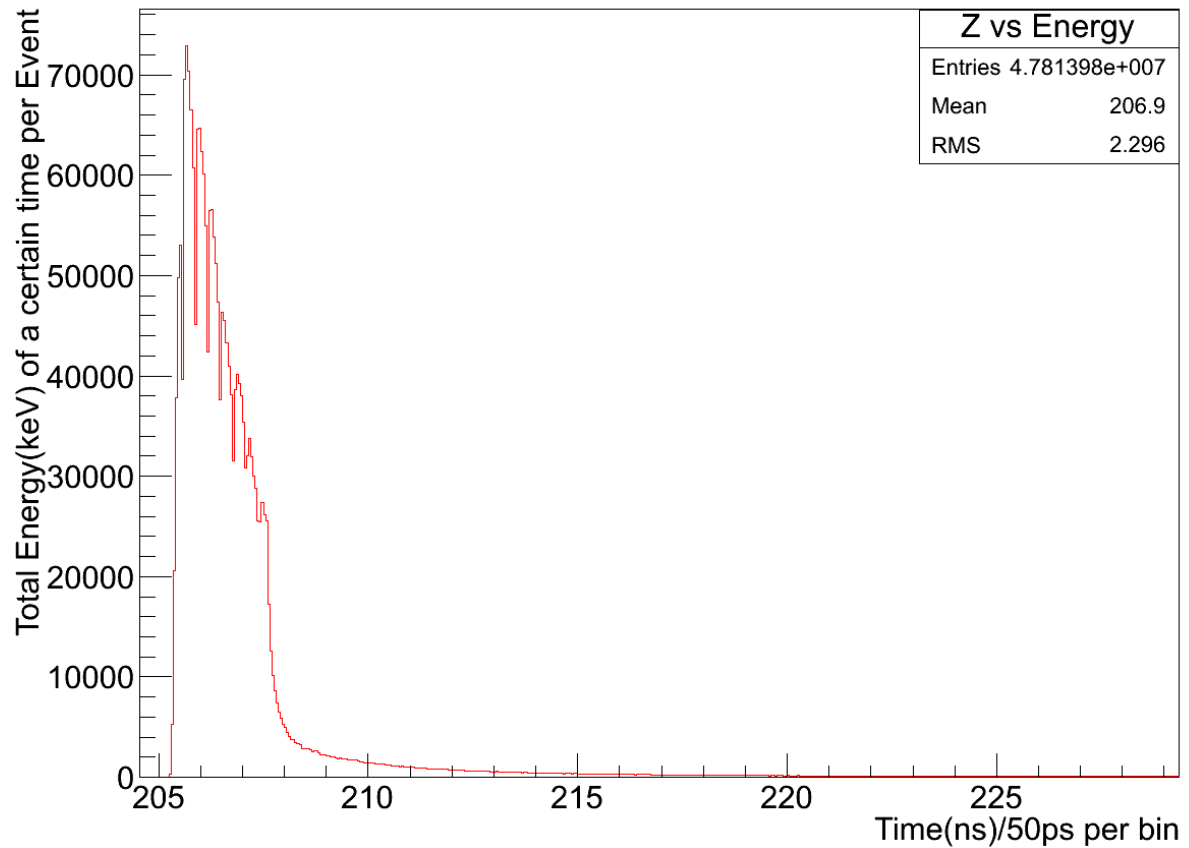
Energy Distribution to Z Position



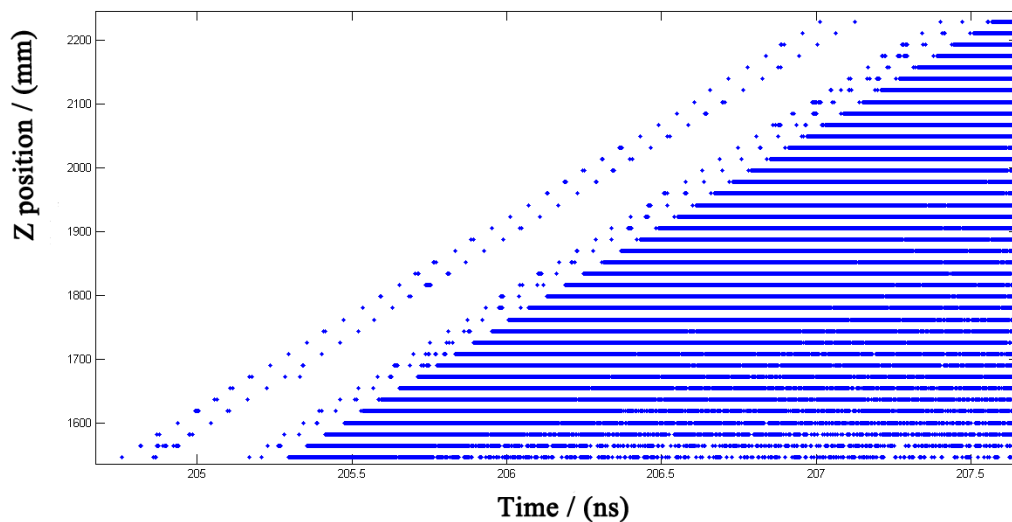
2.2 Time and Energy Statistics

Since the main signal we are dealing with is all about time and energy. Now we keep our focus on them. We do statistics on the energy distribution with time. As shown below, it seems that most energy was deposited in the detector at approximately the same time.

Energy Distribution to Time

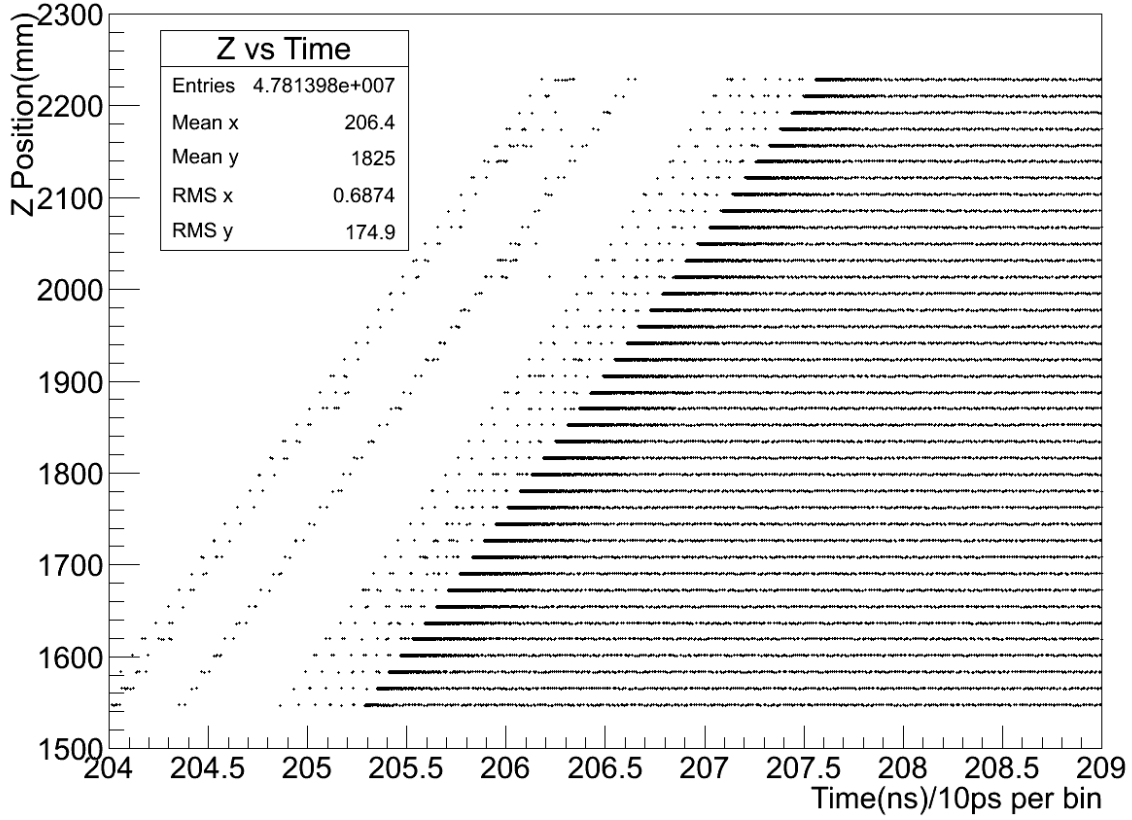


We try to find out the relation between the Z-Position and time. Noticing that in one event, there are many hits with different time and energy. So we go all over the hits. The result is shown below, the picture is generated by Matlab:



But if we do statics with a 2D histogram using ROOT, we will find out a similar result.

Energy Distribution to Time

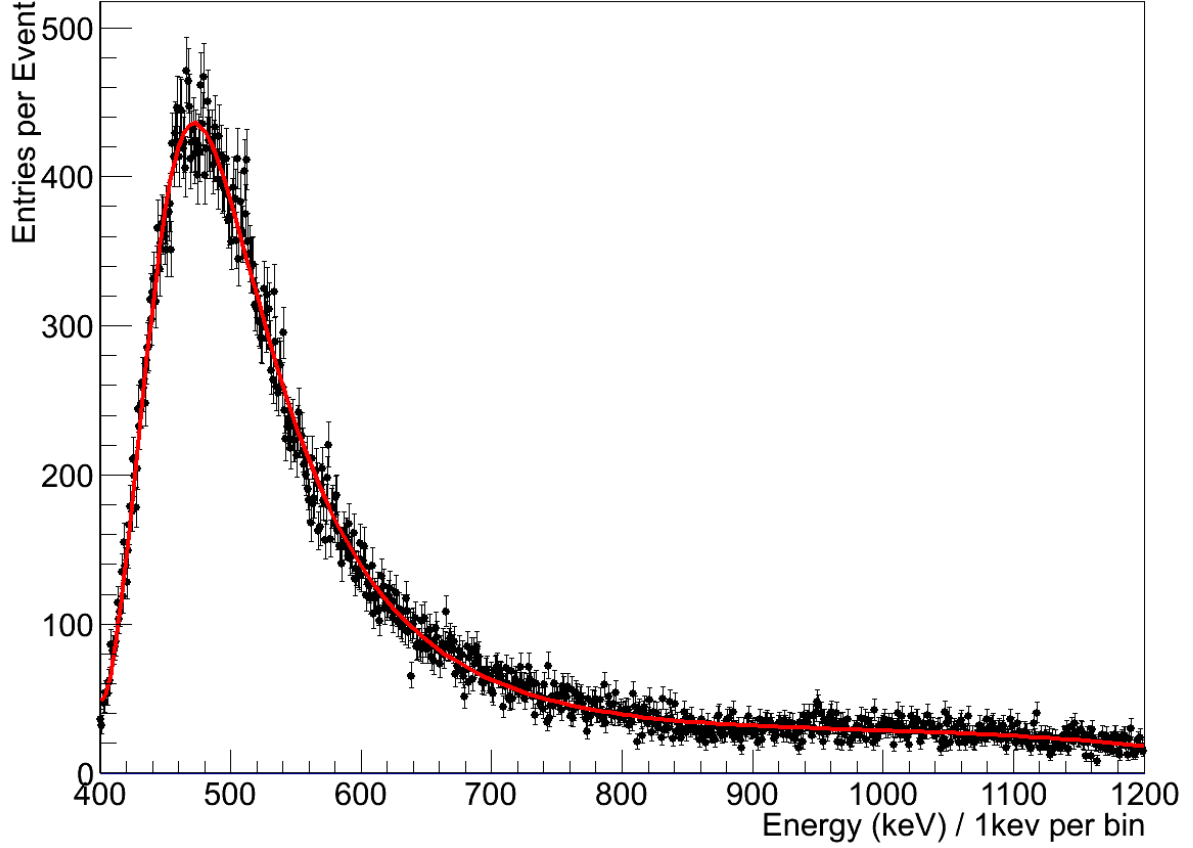


2.3 MIP Calibration and MIP cut

Muons acting as minimum ionising particles(MIPs) define the minimal energy depositions which can be measured by the AHCAL physics prototype. The response of a single SiPM to the scintillation light generated by the energy deposition of a muon in the corresponding scintillator tile can be described with a Landau convoluted with a Gaussian fit. The most probable value of this fit defines the MIP energy.

We generated 2000 events of μ^+ s with 180GeV energy. The energy distribution and fit function(Using RooFit) are shown below:

A RooPlot of "Energy (keV)"



The fitting parameters are shown below: The parameter we get is show below:

$$\text{Landau Means} = 479.512(\text{keV})$$

$$\text{Gaussian Means} = 0.22$$

$$\text{Landau } \sigma = 31.0166$$

$$\text{Gaussian } \sigma = 0.248757$$

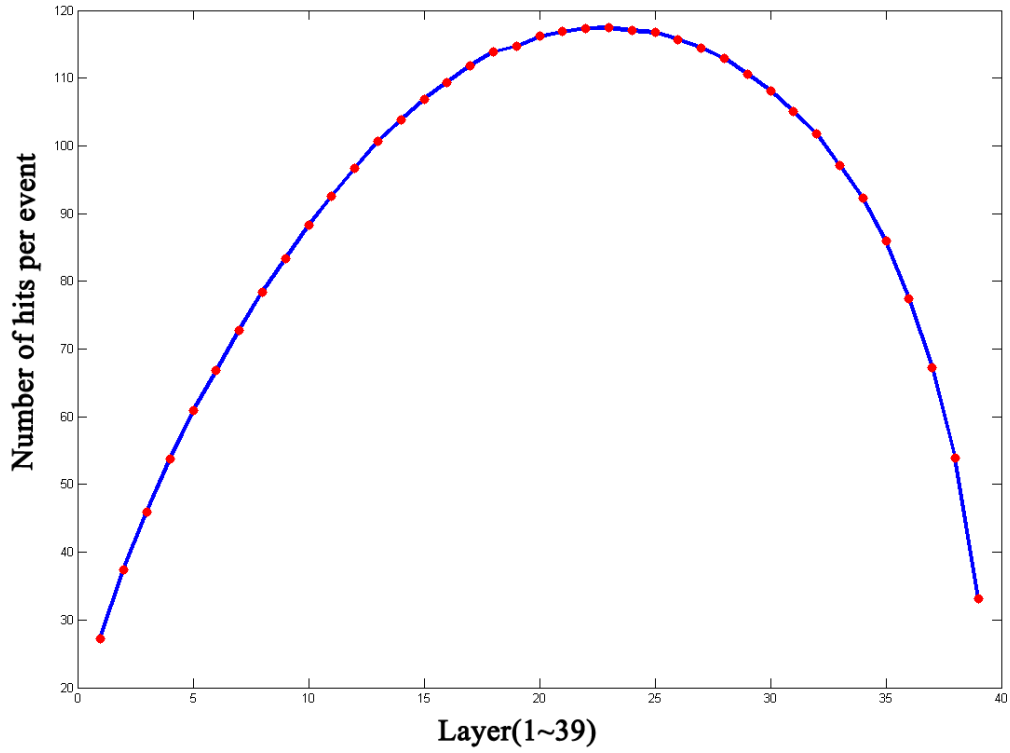
$$\text{signal} = 68347.4$$

$$\text{background} = 12214.8$$

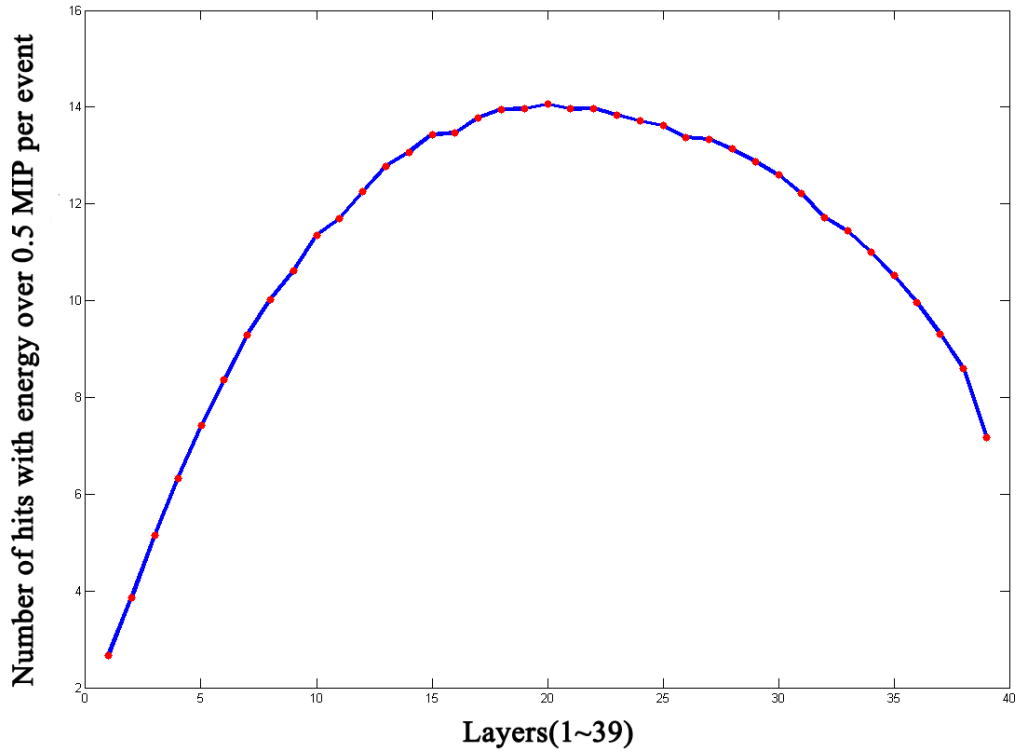
After getting the MIP value. We can now get back to the analysis of π^- s and apply some typical MIP cut, which is very useful in getting rid of the hardware noise, to find out how the cut will influence the actual physical result because now we are using the raw physical data. To do this, we focus on the last layer, which is always the most heavily influenced layer while doing MIP cut. We use the 0.5MIP threshold as an example.

The total number of hits in the 2000 events is **7085035**(all energy included), that is about **3542.5** per event. The number of hits in the 39th layer in 2000 event is **66142**,

that is about **33.07** per event. So, there is about 0.93% of the total hits in the last layer. The result is shown below:



After applying the 0.5MIP cut, the result is that the number of hits with energy higher than 0.5MIP is **867659**, that is **433.8295** per event. As for the last layer, there are about **14326** hits, which is approximately 1.65% of all required hits and 21.66% of all hits in the last layer.



3 Noise-Adding

Adding noise is always easier than getting rid of noise. So what we need to do is that according to the noise distribution reported by the hardware group, we add noise to the raw physical data with the exactly the same distribution. In this way, we can simulate the result in real testbeam experiment. We have a tiny problem in getting through the API document of the *slcio* format, but we've already solved it. So, we firstly use the most common random way to add some noise to the original signal. So this cause the random noise. Of course, we should still do the corresponding polynomial correction which was already measured by the hardware group. But I believe that creating a series of method to get rid of the random noise will be definitely much more important. So, this lead to the most important topic in my work, that is, the filter designing and implementation.

4 Filter

Filtering is a technique for modifying or enhancing signal. For example,you can filter an image to emphasize certain features or remove other features. Filter can be classified generally into two kinds. The Spatial-Domain-Filter and the Frequency-Domain-Filter. The Spatial-Domain-Filter is more often used in image processing. It is designed to be used as a neighborhood operation of an image (The value of any given pixel in the

output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel) mostly for smoothing, sharpening, edge enhancement and so on. In our case, the Frequency-Domain-Filter is much more important and effective. To demonstrate what is a Frequency-Domain-Filter, we should keep in mind some basic concept and theory.

4.1 Frequency-Domain-Filter

4.1.1 Basic Concept: Discrete Fourier Transform

Let's start from complex number. As for the algebraic equation:

$$z^n = 1 \quad (1)$$

It's known that it has n complex roots, which are located in the complex unit circle, called n th root of unity. If for any $k < n$, a n th root of unity is not the k th root of unity, this n th root of unity is called a Primitive n th root of unity. From the algebraic equation itself, we can easily deduce:

Lemma 1 (Primitive n th root of unity)

Suppose ω is a Primitive n th root of unity, k is an integer, then:

$$\sum_{j=0}^{n-1} \omega^j = \begin{cases} n, & \text{if } \frac{k}{n} \text{ is an integer} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

This lemma can be easily proved using equation 4.1.1. So here we skip over it. Using this lemma, we give the definition of Discrete Fourier Transform, which is often abbreviated as DFT.

Definition 1 (DFT)

Suppose $\mathbf{x} = [x_0, x_1, \dots, x_{n-1}]$ is a n -dimensional vector of real number, and that $\omega_j = e^{-\frac{2\pi i}{n}j}$. The DFT vector y of vector x is defined as:

$$y_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \omega_j^k \quad (3)$$

If we define the Fourier Transform Matrix as below($\omega = \omega_1$):

$$\mathbf{F}_n = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \dots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \dots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \omega^0 & \omega^3 & \omega^6 & \dots & \omega^{3(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)^2} \end{bmatrix} \quad (4)$$

We can easily notice that except for the first line, all the other lines of the matrix have a sum of 0, which is also true to the rows of the matrix. So, we can deduce the inverse matrix of the transform-matrix as:

$$\mathbf{F}_n^{-1} = \frac{1}{\sqrt{n}} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \dots & \omega^0 \\ \omega^0 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \omega^0 & \omega^{-2} & \omega^{-4} & \dots & \omega^{-2(n-1)} \\ \omega^0 & \omega^{-3} & \omega^{-6} & \dots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ \omega^0 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)^2} \end{bmatrix} \quad (5)$$

It is quite obvious, according to the definition of DFT, that $\mathbf{x} = \mathbf{F}_n^{-1}\mathbf{y}$ and that $\mathbf{y} = \mathbf{F}_n\mathbf{x}$. So, it is now quite obvious that: $\mathbf{F}_n^{-1} = \bar{\mathbf{F}}_n$ which means that \mathbf{F}_n is a Hermitian matrix. There is one thing that needs to be pointed out that, the fourier transform matrix in Matlab is a little bit different from the standard definition given here. It doesn't have the coefficient $\frac{1}{\sqrt{n}}$, so instead of using command "fft", we should use "ifft" for inverse fourier transform.

Generally speaking, the DFT vector is a vector of complex number. But so long as the original vector $\{x_i\}$ contains only real number, we can still find some regular pattern in the DFT vector $\{y_j\}$ that:

Lemma 2

Suppose y_j is the DFT vector of real vector x_i , then:

1. y_0 is a real number
2. $y_{n-k} = \bar{y}_k$ ($k = 1, \dots, n-1$)

Let alone the application of DFT, we can't avoid mentioning one of the most famous algorithm of the 20th century, namely the Fast-Fourier-Transform, which dramatically reduce the computation time of DFT from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \ln n)$. As the implementation is so famous and is always applicable in nearly any program language, we skip over it.

So far, we are just one more step away from the core idea of Frequency-Domain-Filter, that is, the DFT vector shows us the Frequency-Domain-Distribution of the original vector. Let's show how it works. The general Fourier-Transform of a function $f(x)$ is defined as below:

Definition 2

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t} dt \quad (6)$$

Because of the obvious fact that for anytime, we can only get finite number of data, which means it is a reasonable suppose that the function $f(x)$ describing the data is only none-zero in a certain area(the data area) from c to d . We can simplify the integration equation above as:

$$F(\omega) = \int_c^d f(t)e^{-i\omega t} dt \quad (7)$$

It is trivial that we can never get data with the density of real number. So we have to get back to the original definition of integration in order to use our data to do the computation. Let us suppose our data is uniformly distributed within the area $[c, d]$, then a approximation of the integration using our data can be written as:

$$F(\omega) = \sum_{i=0}^{n-1} \frac{d-c}{n} f(t_i) e^{-i\omega t_i} \quad (8)$$

The t_k in the above equation is defined as $t_i = c + \frac{d-c}{n}i$, which clearly hints us to associate this to the definition of DFT 1. Using the definition, we can easily reform the above equation as:

$$F(\omega_m) = \sum_{i=0}^{n-1} \frac{d-c}{n} f(t_i) e^{-i\omega_m t_i} \quad (9)$$

Suppose that here, $\omega_m = \frac{2\pi m}{d-c}$, we can easily deduce that:

$$\begin{aligned} F(\omega_m) &= \frac{d-c}{n} \sum_{k=0}^{n-1} x_k e^{-i\frac{2\pi m}{d-c}(c + \frac{d-c}{n}k)} \\ &= \frac{d-c}{n} e^{-i\frac{2\pi mc}{d-c}} \sum_{k=0}^{n-1} x_k e^{-i\frac{2\pi km}{n}} \end{aligned} \quad (10)$$

It is so obvious that the right part of the equation above now contains a standard form of fourier transform. Considering the Matlab characteristic of "fft", we rewrite the equation as:

$$F(\omega_m) = \frac{d-c}{n} e^{-i\frac{2\pi mc}{d-c}} FFT(\mathbf{x})_m \quad (\omega_m = \frac{2\pi m}{d-c}) \quad (11)$$

In this way, taking the fact that noise is often with high frequency and sometimes even known frequency into consideration, we can now easily get rid of noise by simply setting some part of $F(\omega)$ to 0 and then use the reverse transform to get the signal without noise we want. In other words, we are now ready to design some Frequency-Domain-Filter.

4.1.2 Implementation of 1-D Frequency-Domain-Filter

We now give an example of our Frequency-Domain-Filter. Firstly, we create a Gaussian-Distributed signal, which is most commonly seen 2:

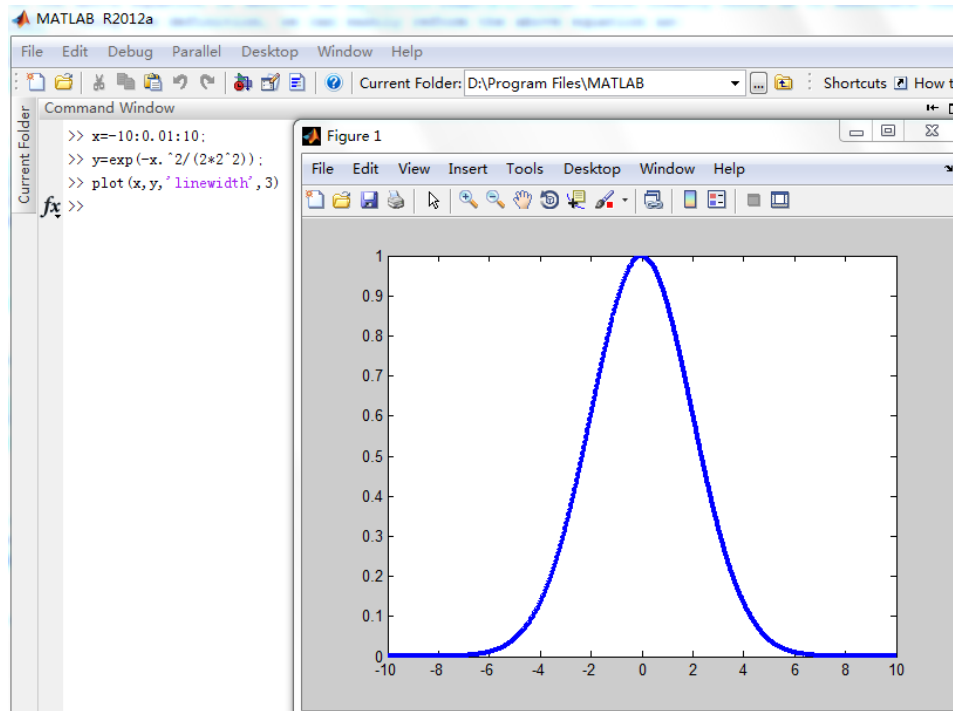
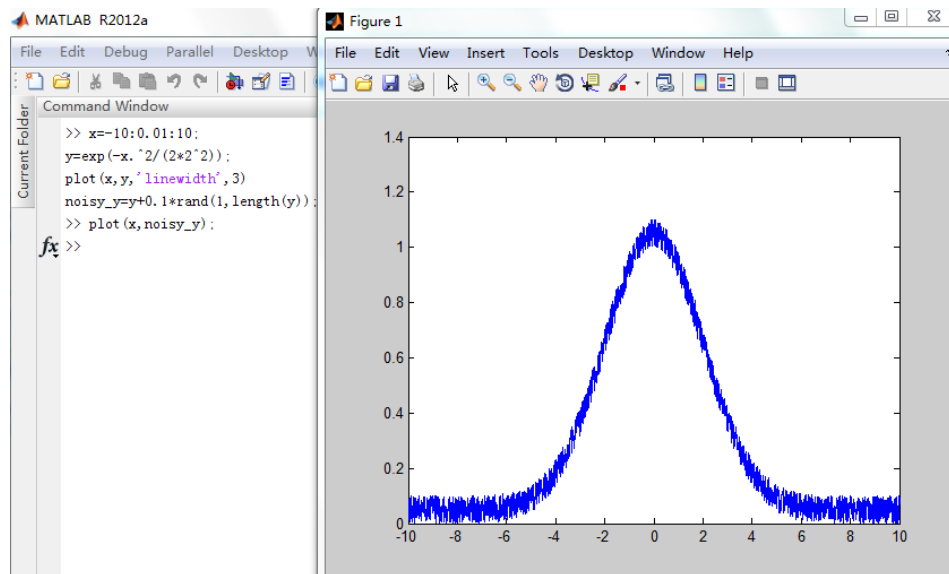


Figure 2: Original Signal

Secondly, we add some noise to it using the code shown in 4.1.2.

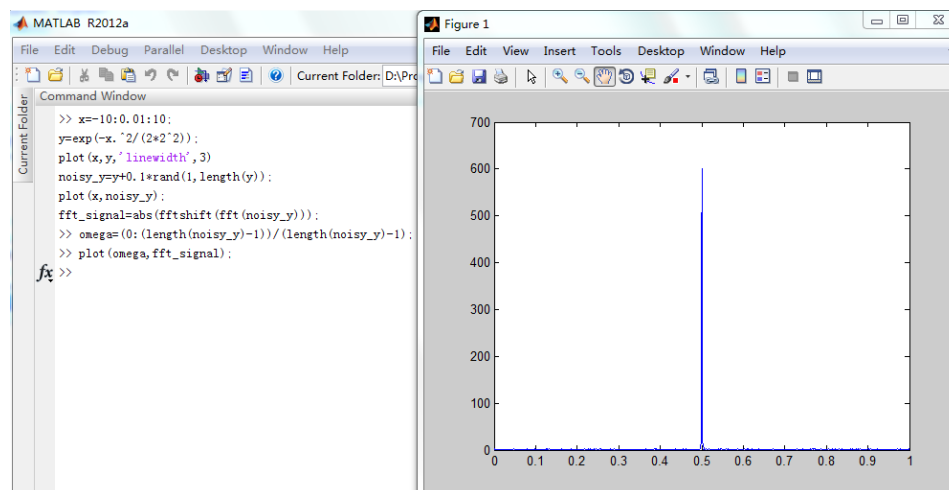


Add random noise with ratio 0.1.

Thirdly, we need to create the filter, using the following Matlab code let's do it step by step:

```
1 fft_signal=abs(fftshift(fft(noisy_y))); %creating the DFT vector of
   noisy_y, function fftshift is used to correct the shift generated
   during the recursive using of the function fft. Recursive using
   is the key to the acceleration of fft.
2 plot(fft_signal);
```

After using the code above and zooming in the picture a little, we get the Frequency-Domain distribution of the noisy signal 4.1.2:



Frequency-Domain distribution

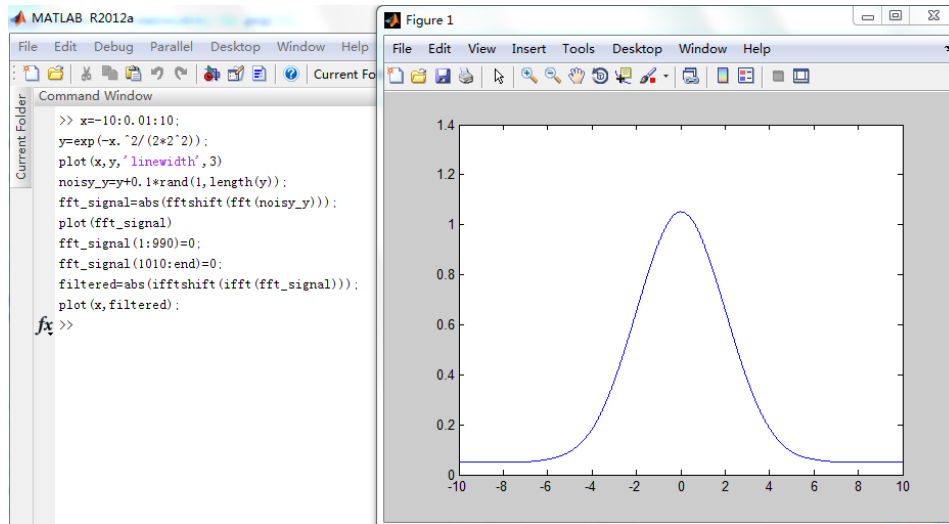
Finally, we can get rid of the high frequency part of the signal using the following code:


```

1 %990 is a reasonable guess of the noise frequency.
2 fft_signal(1:990)=0;
3 fft_signal(1010:end)=0;
4 filtered=abs(iffshift(iff(fft_signal)));%Applying reverse
  transform.
5 plot(filtered);

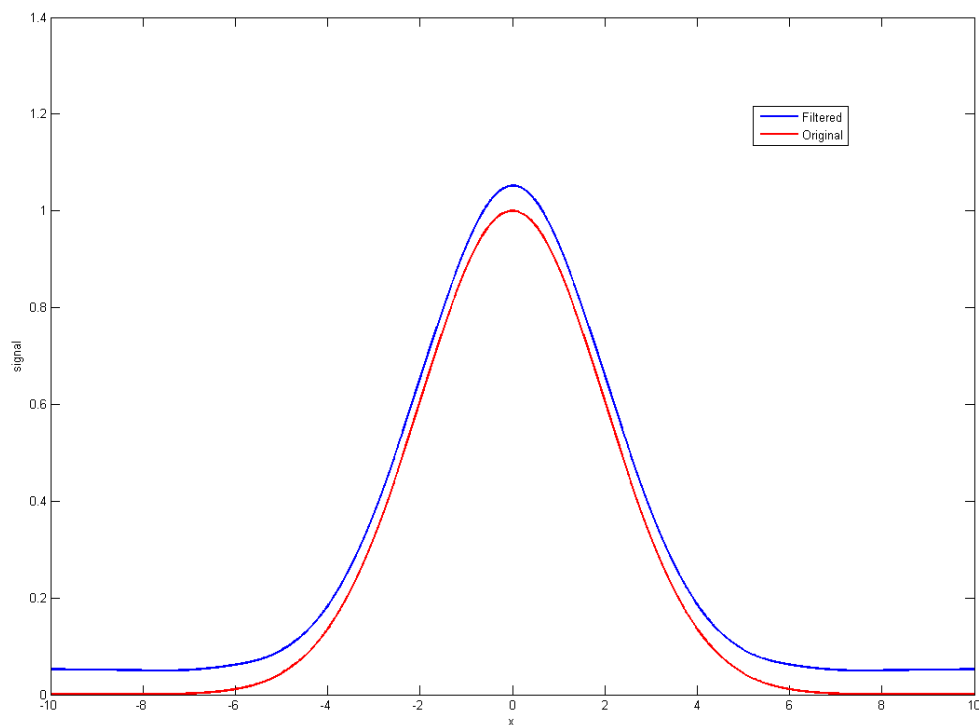
```

Now, we get the filtered signal 4.1.2:



Filtered Signal

Comparing the original and filtered signal ???. They're almost the same in shape. The only difference is in the amplitude, which is easy to explain: the random noise numbers are all positive therefore adding pedestal signal to the original signal. We can declare that the filter is a great success.



Compare

In addition to the discuss above, Matlab has provide a very useful function **freqz** to do a lot of work together. For example:

```
1 x=-10:0.01:10;
2 y=exp(-x.^2/(2*2^2));
3 plot(x,y,'linewidth',3)
4 noisy_y=y+0.1*rand(1,length(y));
5 [A,w]=freqz(noisy_y,1,length(noisy_y),'whole');
6 A(11:1991)=0;
7 filtered=abs(iff(A));
```

The code above is a shorter and more useful way to design the filter.

4.1.3 Implementation of 2-D Frequency-Domain-Filter

The 2-D filter is very similar to the 1-D one. We just need to change from function **fft** to function **fft2**, or change from **freqz** to **freqz2**. We provide an example here.

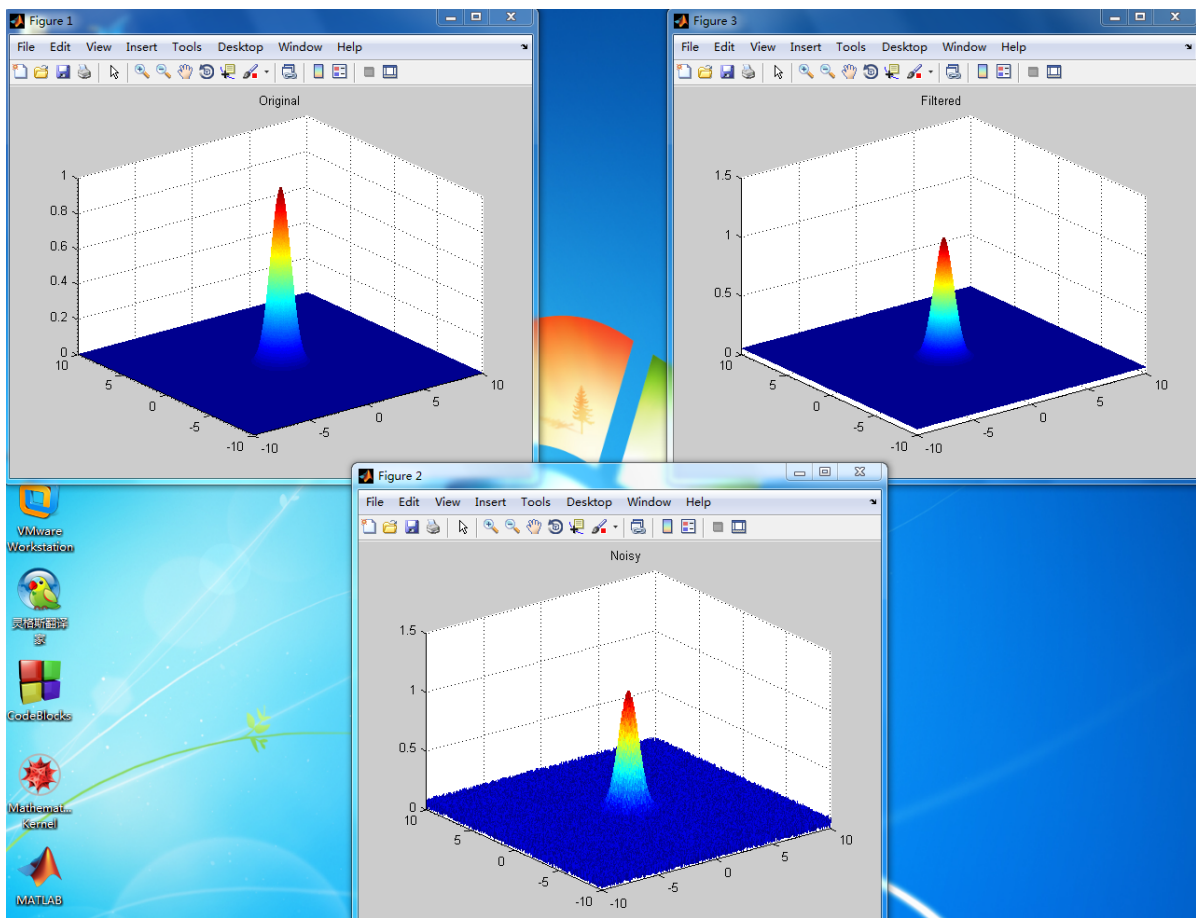
```
1 x=-10:0.02:10;%x-axis range
2 y=-10:0.02:10;%y-axis range
3 z= repmat(-x.^2,length(x),1);
4 z=z+ repmat(-y.^2.',1,length(y));
5 z=exp(z);
```

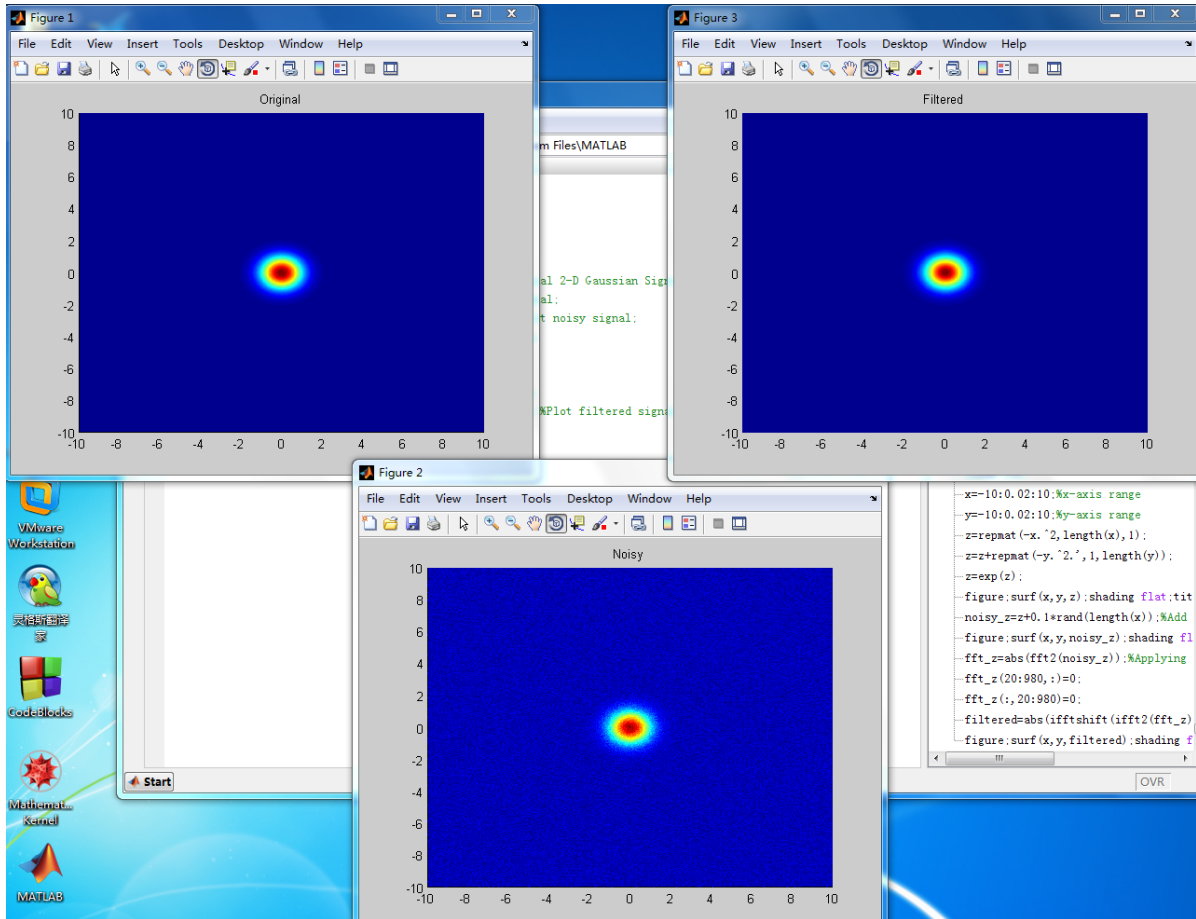
```

6 figure;surf(x,y,z);shading flat;title('Original');%Original 2-D
  Gaussian Signal
7 noisy_z=z+0.1*rand(length(x));%Add noise to Original Signal;
8 figure;surf(x,y,noisy_z);shading flat;title('Noisy');%Plot noisy
  signal;
9 fft_z=abs(fft2(noisy_z));%Applying 2-D Fourier Transform.
10 fft_z(20:980,:)=0;
11 fft_z(:,20:980)=0;
12 filtered=abs(ifftshift(ifft2(fft_z)));%Inverse Transform
13 figure;surf(x,y,filtered);shading flat;title('Filtered');%Plot
  filtered signal.

```

After running the code above, we can get 3 pictures 4.1.3 with original, noisy and filtered signal respectively.





2-D example viewed from 2 different angles.

4.2 Other Filter

The Frequency-Domain-Filter is the most important kind of filter. But it doesn't mean that the other kind of filter is negligible. On the contrary, there are a lot of other kind of filters whose potential are still not seen yet. The most common ones are Z-Transform based filter, Radon-Transform based filter and Hough-Transform based filter. Since they are not that important, we simply give a short introduction to each of them.

4.2.1 Z-Transform filter

The basic idea now known as the Z-transform was known to Laplace, and re-introduced in 1947 by W. Hurewicz as a tractable way to solve linear, constant-coefficient difference equations. It was later dubbed "the z-transform" by Ragazzini and Zadeh in the sampled-data control group at Columbia University in 1952. Z-transform, like many integral transforms, can be defined as either a one-sided or two-sided transform.

Definition 3 (Z-Transform)

$$X(z) = \mathcal{Z}\{x[n]\} = \sum_{n=-\infty}^{+\infty} x[n]z^{-n} \quad (12)$$

Of course, in order to implement a proper filter, we also need an inverse transform. Which is known as:

Definition 4 (Inverse Z-Transform)

$$x[n] = \mathcal{Z}^{-1}\{X(z)\} = \frac{1}{2\pi i} \oint_C X(z)z^{n-1}dz \quad (13)$$

C is the unit circle on the complex plane.

The Z-Transform based filter is implemented by multiplying a fractional polynomial to the $X(z)$ function. In general, the Z-Transform $Y(z)$ of a discrete-time filters output $y(n)$ is related to the Z-Transform $X(z)$ of the input by

$$Y(z) = H(z)X(z) = \frac{b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}}{a(1) + a(2)z^{-1} + \dots + a(m+1)z^{-m}}X(z) \quad (14)$$

In fact, this kind of filter is so widely used that the **filter** function itself is based on this kind of filter. The feature of this kind of filter can be summarised as follows:

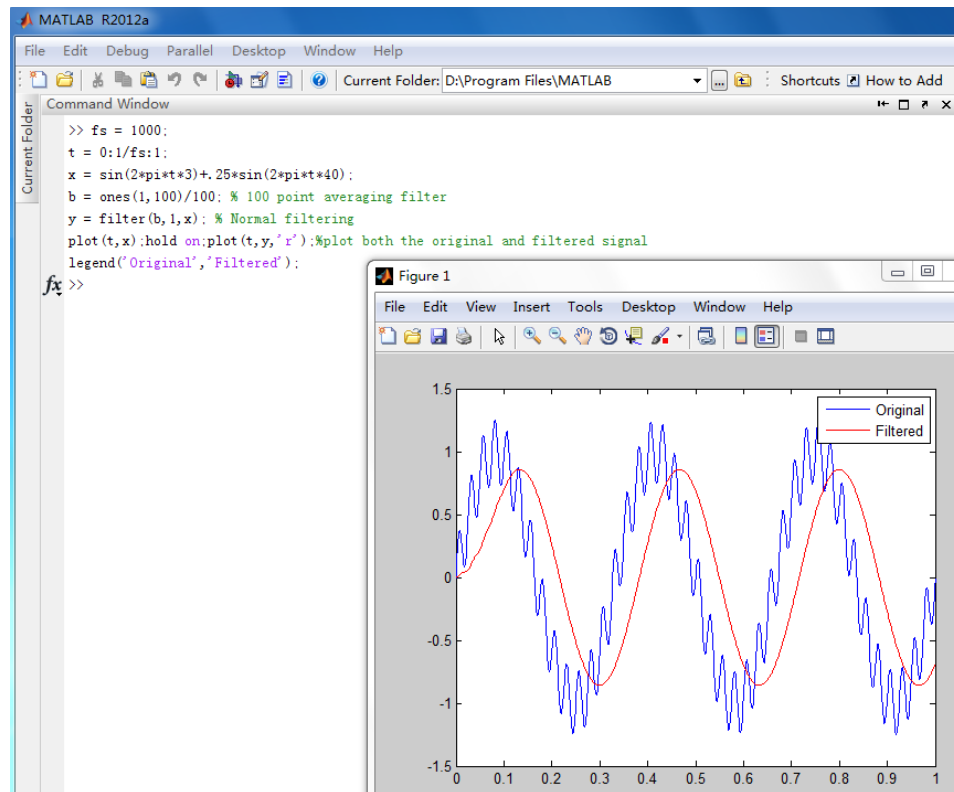
1. When $n=0$ (that is, b is a scalar), the filter is an Infinite Impulse Response(IIR), all-pole, recursive, or autoregressive (AR) filter.
2. When $m = 0$ (that is, a is a scalar), the filter is a Finite Impulse Response(FIR), all-zero, nonrecursive, or moving-average (MA) filter.
3. If both n and m are greater than zero, the filter is an IIR, pole-zero, recursive, or autoregressive moving-average (ARMA) filter.

We also provide a example here to illustrate how the filter works. The example is from [6]

```

1 fs = 1000;
2 t = 0:1/fs:1;
3 x = sin(2*pi*t*3) + .25*sin(2*pi*t*40); %2 different frequency signal,
    the one with smaller amplitude is noise.
4 b = ones(1,100)/100; % 100 point averaging filter
5 y = filter(b,1,x); % Normal filtering
6 plot(t,x); hold on; plot(t,y,'r'); %plot both the original and filtered
    signal
7 legend('Original','Filtered');
```

The result is shown in the picture 4.2.1. Despite a little phase shift, the filter result is perfect.



Z-Transform based filter

4.2.2 Radon-Transform filter

In mathematics, the Radon transform in two dimensions, named after the Austrian mathematician Johann Radon, is the integral transform consisting of the integral of a function over straight lines. The Radon transform is widely applicable to tomography, the creation of an image from the scattering data associated to cross-sectional scans of an object. But it can also be used to get rid of line-shaped noise since any line on the original 2-D signal is shown as a single point on the transformed signal. However, this kind of filter is only effective on 2-D signal, so it is not so useful in our case. We simply give an example to explain how the transformation works. The example is from [7]

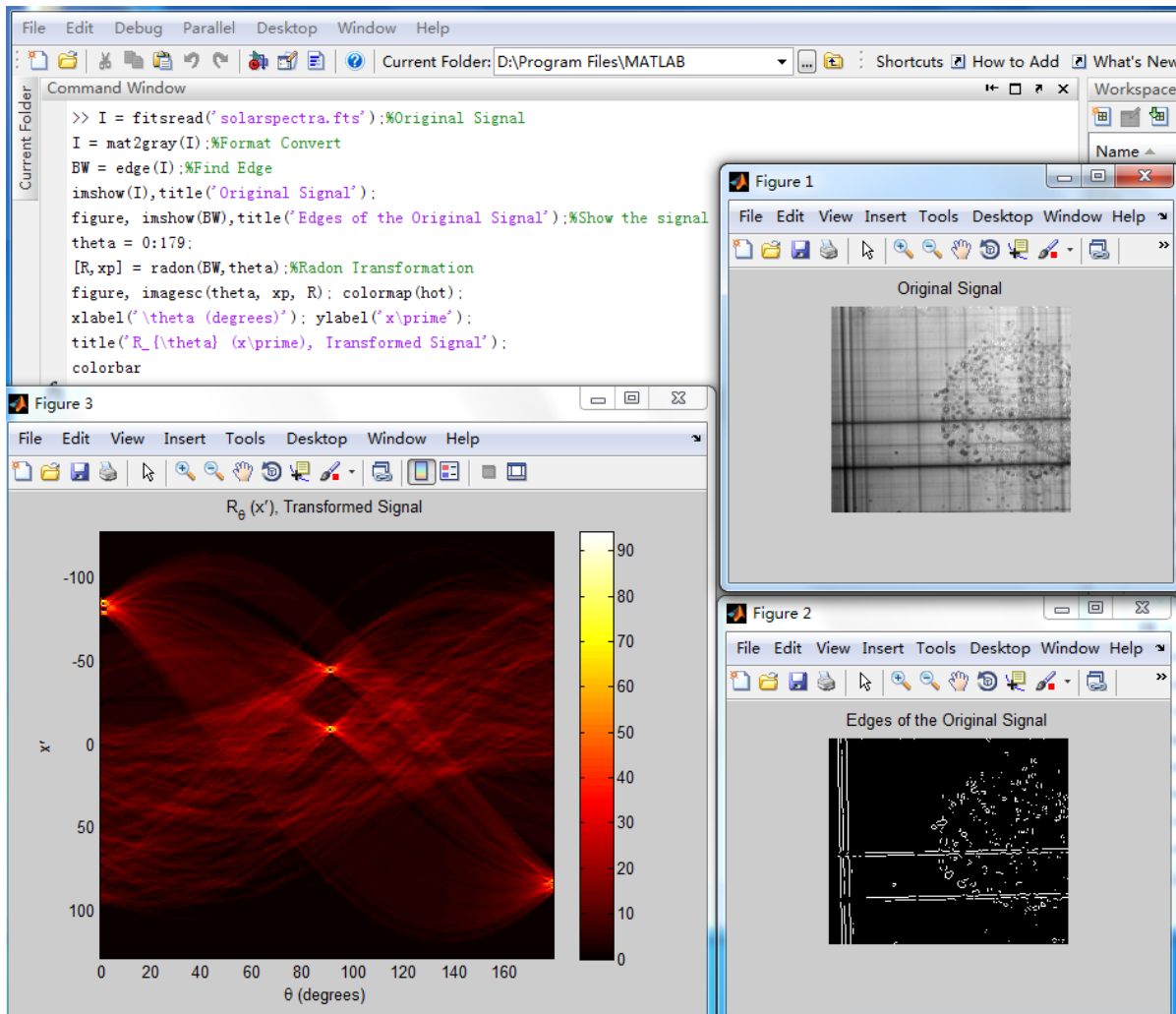
```
1 I = fitsread('solarspectra.fts');%Original Signal
2 I = mat2gray(I);%Format Convert
3 BW = edge(I);%Find Edge
4 imshow(I),title('Original_Signal');
5 figure, imshow(BW),title('Edges_of_the_Original_Signal');%Show the
   signal
6 theta = 0:179;
7 [R,xp] = radon(BW,theta);%Radon Transformation
8 figure, imagesc(theta, xp, R); colormap(hot);
```

```

9 xlabel('\theta (degrees)'); ylabel('x\prime');
10 title('R_{\theta}(x\prime), Transformed Signal');
11 colorbar

```

The result is shown in the picture 4.2.2.



Radon Transformation

References

- [1] Silicon Photomultipliers: Properties and Application in a Highly Granular Calorimeter *Nils Feege, Hamburger University*
- [2] <http://www-flc.desy.de/hcal/> CALICE Homepage, DESY
- [3] http://en.wikipedia.org/wiki/Discrete_Fourier_transform Discrete Fourier transform
- [4] <http://en.wikipedia.org/wiki/Z-transform> Z transform
- [5] http://en.wikipedia.org/wiki/Radon_transform Radon transform
- [6] <http://www.mathworks.cn/help/toolbox/images/> Mathworks, Matlab Image Processing Toolbox User's Guide
- [7] <http://www.mathworks.cn/help/toolbox/signal/> Mathworks, Matlab Signal Processing Toolbox User's Guide