NIUS Report
E15 - Digital pulse shaping of HPGe detector for neutron damage study.

Vishal Sai Vetrivel
Student Code - P64-2022
UM DAE CEBS, Mumbai

Camp 19.2

Contents

1 Introduction **2**
1.1 Types of detectors . . . 2
1.2 Pulses and Pulse Shaping . . . 3
1.2.1 Shaping Algorithms . . . 3
1.2.2 Trapezoidal Shaping in Python . . . 4

2 Neutron Damage **6**
2.1 Introduction . . . 6
2.2 Theory . . . 6
2.3 The Algorithm . . . 8
2.3.1 Properties of the Algorithm . . . 13
2.3.2 Further Research . . . 13
2.4 Conclusion . . . 13

A Miscellaneous Gallery **14**

B Code **15**
B.1 Trapezoidal Shaping in Python . . . 15
B.2 2D Histogram of Rise Time and Energy in Root . . . 16
B.3 2D Histogram of Pulse Height and Energy in Root . . . 17
B.4 Correction of a Damaged Spectrum in Root . . . 18

Chapter 1

Introduction

1.1 Types of detectors

Gamma Ray Detectors are divided into two types, scintillators and solid state detectors. In a scintillation based detector the sensitive part is a luminescent material that is viewed by some photo-multiplying system. This leads to excellent time resolution but since the collection time is very less this leads to very poor energy efficiency when compared to semiconductor based solid state detectors.

When it comes to semiconductor based detectors the sensitive part is a large crystal of any semiconductor. A semiconductor is used as the band gap is usually just the right range to make useful measurements as a conductor would pick up too many stray signals and an insulator would be too insensitive. Germanium detectors are used most often as they have a higher atomic number and hence higher collision probabilities when compared to silicon and they are also easier to produce large and pure crystals of.

In this type of detector the rays are detected when they collide with the semiconducting crystal and produce electron-hole pairs through either photoelectric effect, Compton effect or even pair production where if the energy of the incoming ray is above 1022 keV. The photoelectric case is ideal as all the energy of the incoming ray is converted to electrons. Compton effect is less ideal as not all of the energy is immediately absorbed and some of it might even escape the detector. This leads to the generation of a Compton Edge in the spectrum. This is usually prevented by making larger detectors. In pair production part of the photon's energy is used to generate an electron and positron pair. The positron annihilates with another electron leading to the indirect production of an electron hole pair. This is also not ideal as not all the energy of the ray is absorbed.

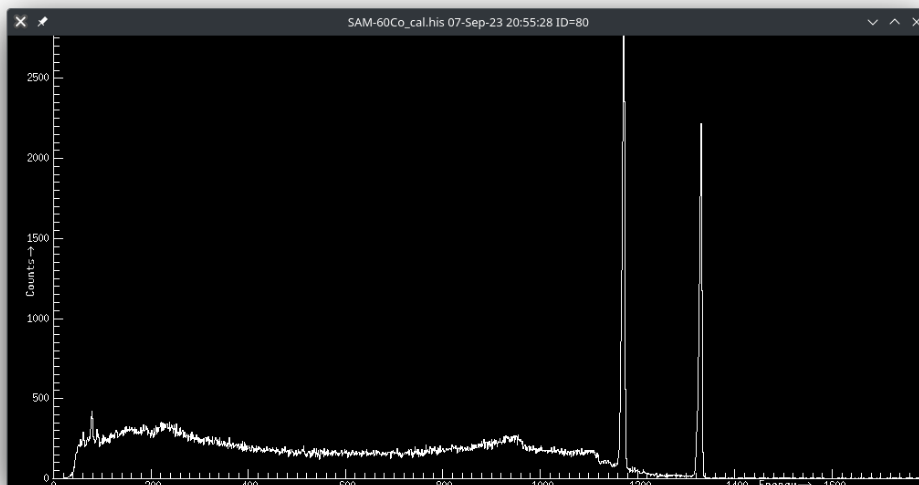


Figure 1.1: A labeled spectrum of a 60Co source.

1.2 Pulses and Pulse Shaping

Pulses are obtained from a detector when a gamma ray is incident on the detector. These pulses are the result of charge collection on the electrodes of a detector. Since a detector is in theory similar to a large capacitor it resembles a charge and subsequent discharge.

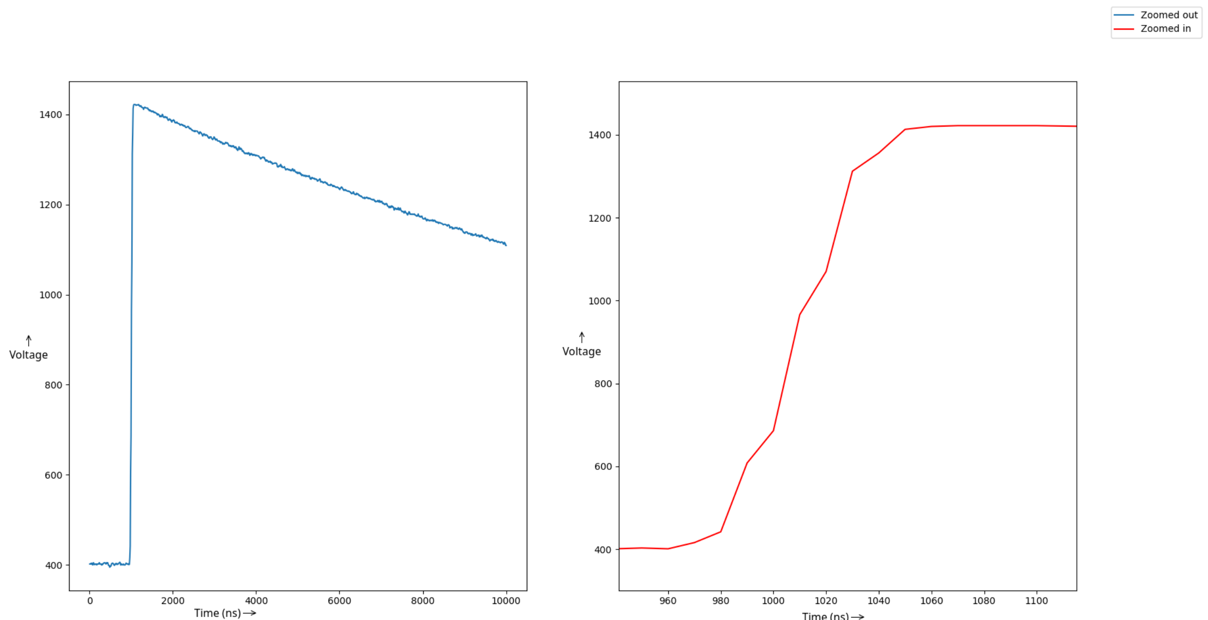


Figure 1.2: An example of a pulse from a HPGe detector, taken from the peak of a damaged spectrum.

The rising edge is usually incredibly fast lasting a few 10s of nanoseconds. This rise time however is significant in correcting the spectrum as seen through this project.

These pulses however have very little use in their current form. They are usually put through a shaping algorithm which turns this pulse into a more useful form. There are many choices for a shaping algorithm but the most commonly used is a trapezoidal filter.

1.2.1 Shaping Algorithms

Cusp Shaping

The theoretically best pulse shape for minimizing noise is a cusp shape. However implementing it in practice is extremely hard if not impossible and it also has the issue that the peak is a point and is subsequently hard to locate.

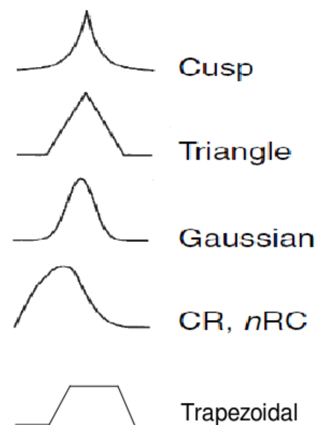


Figure 1.3: The various pulse shapes that can be obtained using filters.[3]

Triangular Shaping

The next best pulse shape for maximizing signal to noise ratio is a symmetrical triangle shape. However, it is almost impossible to achieve this shape through only passive circuits but if active circuits are utilized then a shape approaching a triangle is possible. This also has the disadvantage that the peak is difficult to locate.

Gaussian or CR-(RC)^n Shaping

This filter utilizes a single CR differentiation followed by several stages of RC integration to create a resulting pulse shape that approaches a Gaussian function. In practice with 4 stages of integration the difference between the resultant pulse and a true Gaussian becomes negligible [2].

Trapezoidal Shaping

The trapezoidal shape is the next best thing after triangular shaping and is quite easy to do with a digital pulse trace. This is the kind of shaping used in the detector data used in this project and it is the type of shaping that was implemented in python to confirm the same.

It is linear and it has three main parameters.

- Trapezoid Rise Time: This has the important requirement that it must be larger than the rise time of the pulse itself. If not the trapezoid becomes very rough and almost random.
- Trapezoid Flat Top: The size of the flat top. This can be almost anything but if the counts are high then a smaller value is preferred.
- Decay Time Constant: This value has to match the pulse as closely as possible. If it isn't matched then the trapezoid will not have a flat top and will slope to one side.

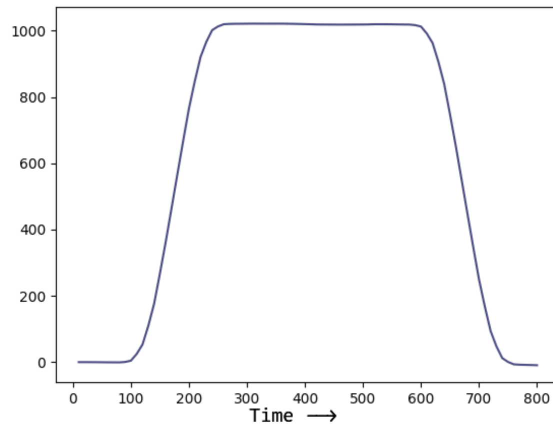


Figure 1.4: A pulse from the spectrum of a damaged detector after being processed by a Trapezoid Filter.

1.2.2 Trapezoidal Shaping in Python

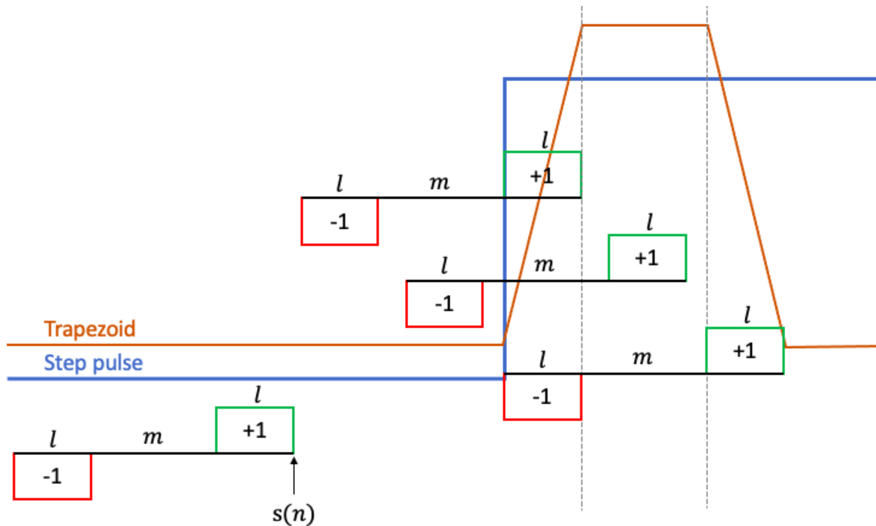


Figure 1.5: An Illustration of the working of the trapezoid filter.[1]

The height of the trapezoid is proportional to the energy of the pulse.

The trapezoidal filter on a very high level is the sum of two integration windows separated by a parameter M. M is the decay time constant and it must be matched. The filter can be expressed as follows:

$$S(n) = \sum_{i=n-l}^{n} f(i) - \sum_{i=n-2l-m}^{n-m-l} f(i) \tag{1.1}$$

Here S(n) is the trapezoidal output and f(i) is the input function. f(i) is a decaying pulse but in the limiting case it can be considered to be a step function if the rise time is sufficiently small and the decay time is sufficiently large.

However, since we'll be using a computer to perform these computations a recursive solution that reuses the calculations as much as possible is preferred. If we take the condition of a decaying signal and make use of previous computations to achieve a recursive formula we arrive at the formulas:

v(n) is the digitized signal with the condition that $v(n) = 0 \; \forall \; n < 0$

$$d^{m,l} = v(n) - v(n-m-l) - v(n-l) + v(n-m-2l) \tag{1.2}$$

$$p(n) = p(n-1) + d^{m,l}(n), n \geq 0 \tag{1.3}$$

$$s(n) = s(n-1) + p(n) + d^{m,l}(n), n \geq 0 \tag{1.4}$$

where

l is the Trapezoid Rise Time.

m is the Trapezoid Flat Top.

M is the Signal Decay Time

It is to note that the trapezoid must be scaled by a factor of $\frac{1}{M \times l}$ for it to match the height of the pulse.

The final code to calculate and plot the spectrum when given a file with raw traces of the pulses can be found in the appendix.

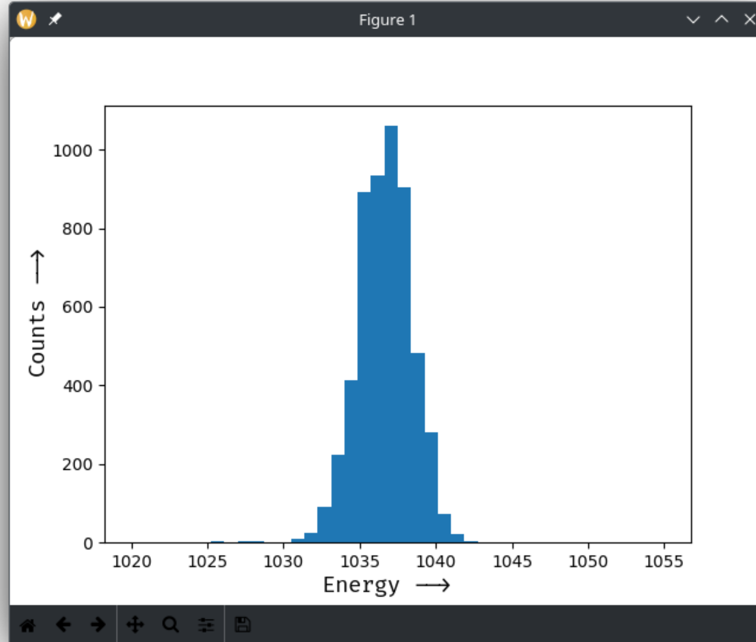


Figure 1.6: A healthy spectrum generated from the trace data from the detectors using the hand written python trapezoid filter. Since this is similar to the output from the detectors themselves it is safe to assume that the hardware filter is most probably a trapezoidal filter as well
Note: The energy is uncalibrated and in arbitrary units.

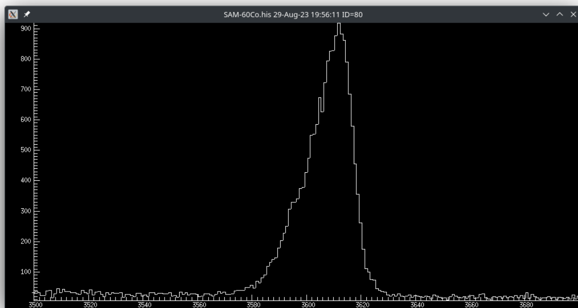
Chapter 2

Neutron Damage

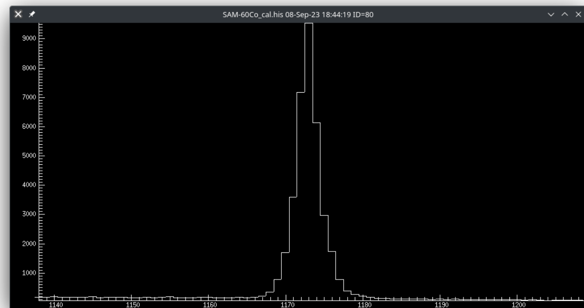
2.1 Introduction

The detectors discussed and used in this project are HpGe detectors. High purity Germanium detectors as explained before consist of a large high purity crystal of germanium. The crystal is semi-conducting and behaves like a large capacitor. The gamma rays hit the crystal and create a hole-electron pair through various processes. However, the process that creates the gamma rays also produces neutrons. The detector is shielded against the charged particles like protons and electrons by a thick plate of metal however the uncharged neutrons are able to penetrate this shielding.

They then accumulate inside the lattice of the germanium crystal. They then proceed to trap charges in these spots. This causes the pulse to take longer to be fully absorbed. This causes an apparent decrease in the energy of the pulses as some of the charges are not collecting in time and/or not attributed to the correct pulse. This causes the shape of the spectrum to change. As a consequence the resolution of the detector also suffers.



(a) A gamma ray spectrum from a 60Co source recorded by a damaged detector



(b) A gamma ray spectrum from a 60Co source recorded by a healthy detector

Figure 2.1: A comparison of two spectra.
Note that the scale of the spectrum from the damaged detector is shifted to the left by quite a bit and the heights and widths are also drastically different.

As we can see in this figure the damaged peaks do not have a proper Gaussian shape and develop a tail to the left of the curve. This is not present in a healthy detector.

The goal of this project is to find an algorithm to correct this defect on the software side.

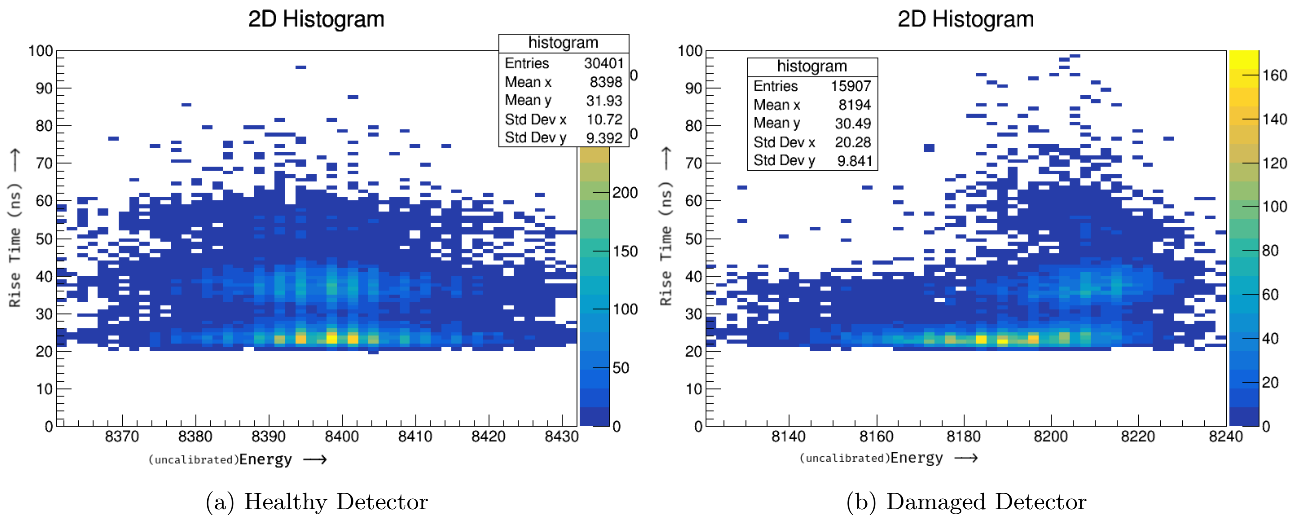
2.2 Theory

The goal is to correct the damaged spectrum of the detector. More specifically the goal is to find an algorithm that removes the damaged left tail from a spectrum peak without affecting the Gaussian part of the peak.

It is suspected that the rise time of the individual pulses from the detector would provide enough information to be able to distinguish the pulses that experiences trapping. This is due to that fact that the rise time would be slower for pulses that had some of their charge trapped due to neutron damage.

However, this rise time variation is also found in healthy detectors. While there are differences in the way the pulse time varies in healthy and damaged detectors it is still makes analysis and correction of the spectra challenging.

To this extent the trace data was obtained from the HpGe detector for various sources and was analyzed through root scripts. The data was used to generate 2D histograms between energy and rise time.



(a) Healthy Detector

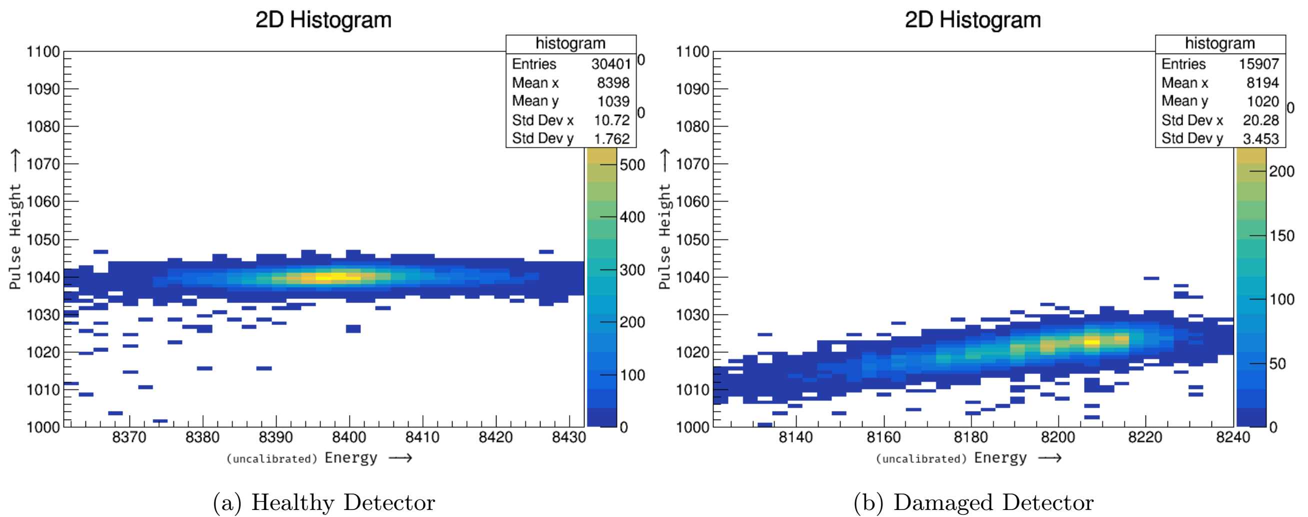
(b) Damaged Detector

Figure 2.2: Here we see that there is a distinct difference between the healthy and damaged detector.

There is a drastic difference between the healthy and damaged detector data when we plot a 2D histogram plot of the rise time (Y Axis) and energy (X Axis). The healthy data is more uniformly distributed whereas the damaged data is heavily skewed to the higher energies.

However, this information is only obvious when there is a statistical amount of pulses and is not suitable for individual pulse handling. This means that this approach cannot be used to distinguish between damaged pulses and healthy pulses on an individual basis.

The relation with the amplitude was also confirmed in this method.



(a) Healthy Detector

(b) Damaged Detector

Figure 2.3: It appears that there is a difference here as well.

NOTE: While it seems like there is a difference here the actual deviance from the linear relation between pulse height and the energy of the pulse, the actual difference between the slopes of the histogram when zoomed out is almost zero. The difference seen in this plot is mostly caused by the different X scales. The actual difference between the two plots here is how spread out the peak is, as in the damaged case the peak is much more spread out.

2.3 The Algorithm

1. Generate a 2D histogram with energy on the X axis and pulse rise time on the Y axis.
   (a) The pulse rise time is calculated by finding the time taken for the pulse to go from 10% of its total height to 50% of its total height.
   (b) To get a more continuous result the pulse curve is also interpolated using a spline.

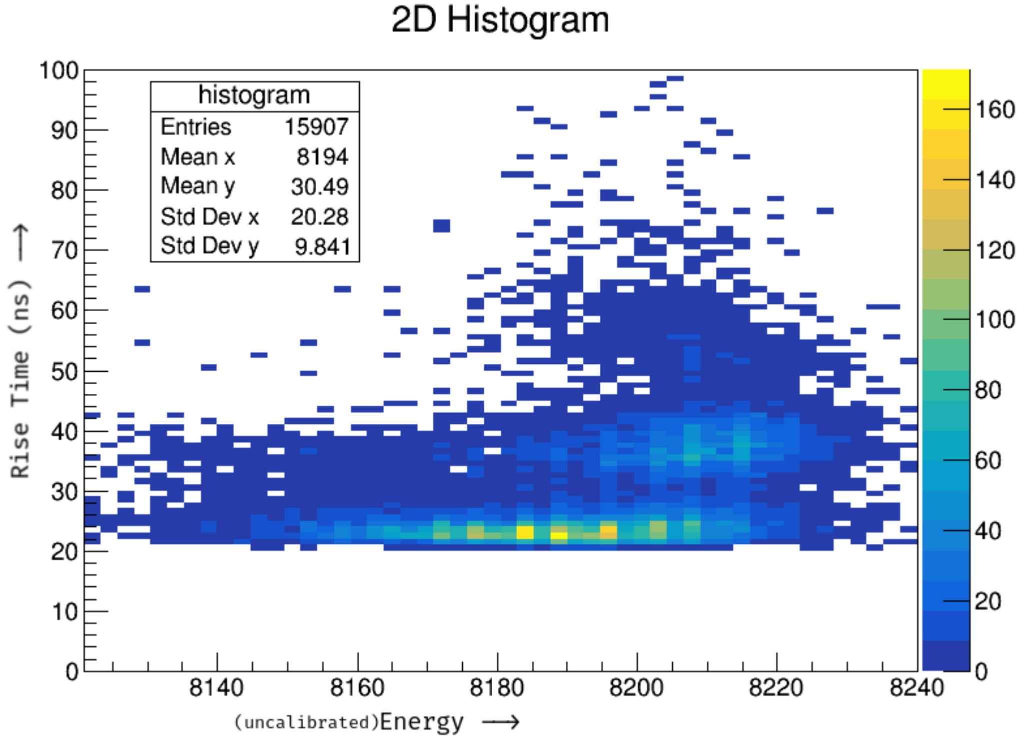


Figure 2.4: A 2D histogram of Energy vs Rise Time

2. For each energy bin take a sample of the rise time distribution around itself. The size of the sample (in number of bins) is defined in the code as `windowSize`. This is visualized in Fig. 2.5. Here we are taking the rise time distributions of several energies around our desired energy and summing them to get an average distribution of rise time around a certain energy. We do this to smooth out the data and get rid of any statistical variations.

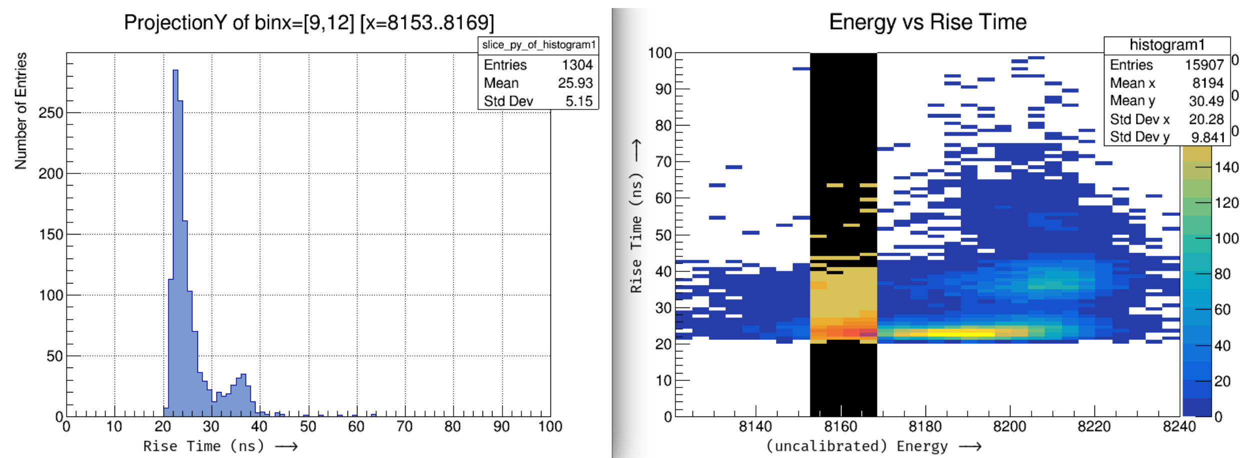


Figure 2.5: The distribution of rise time around a particular energy.

3. Two regions defined by the `window_a/b/c/d` variables are integrated separately and divided to give a ratio.

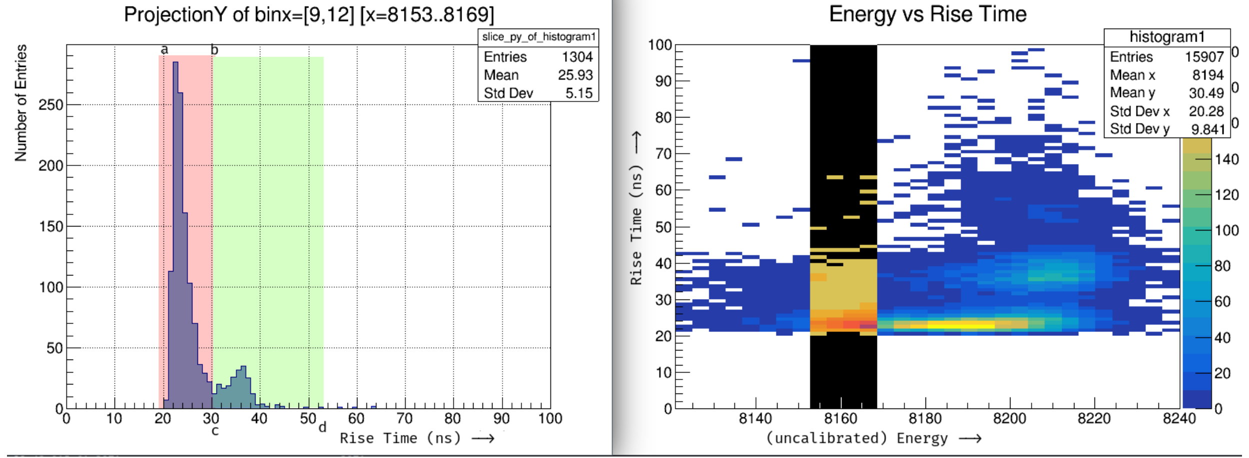


Figure 2.6: The windows of integration are shown here.

$$\text{Original Spectrum} = f(x) \tag{2.1}$$

$$g(x) = \frac{\int_a^b f(x)dx}{\int_c^d f(x)dx} \tag{2.2}$$

$$\text{Correction function} = c(x) = max(g(x)) - g(x) = 1 - \hat{g}(x)^1 \tag{2.3}$$

This ratio is a smooth function over the range of the spectrum and will called the correction function after the series of trivial arithmetic transformations shown above. This was done to make the function zero instead of peaking at the damaged lower tail of the spectrum. If the parameters are chosen carefully this quantity peaks at the lower tail of a damaged spectrum and is nearly constant if the spectrum is healthy.

Technically what we need is the ratio of the heights of the peaks however that is a tough value to calculate and is prone to statistical variations. So we fix two regions where the peaks can be found and simply divide the areas of the spectrum in those regions. This allows us to calculate the value g(x) in deterministic time and gets rid of most statistical variations. However, care must be taken in choosing the windows as the correlation between the ratio of the peaks and ratio of areas depends on the specific choice of windows of integration.

[1] $\hat{g}(x)$ is normalized by dividing all the elements by the largest value. We can do this because g(x) is just an array in implemented algorithm.

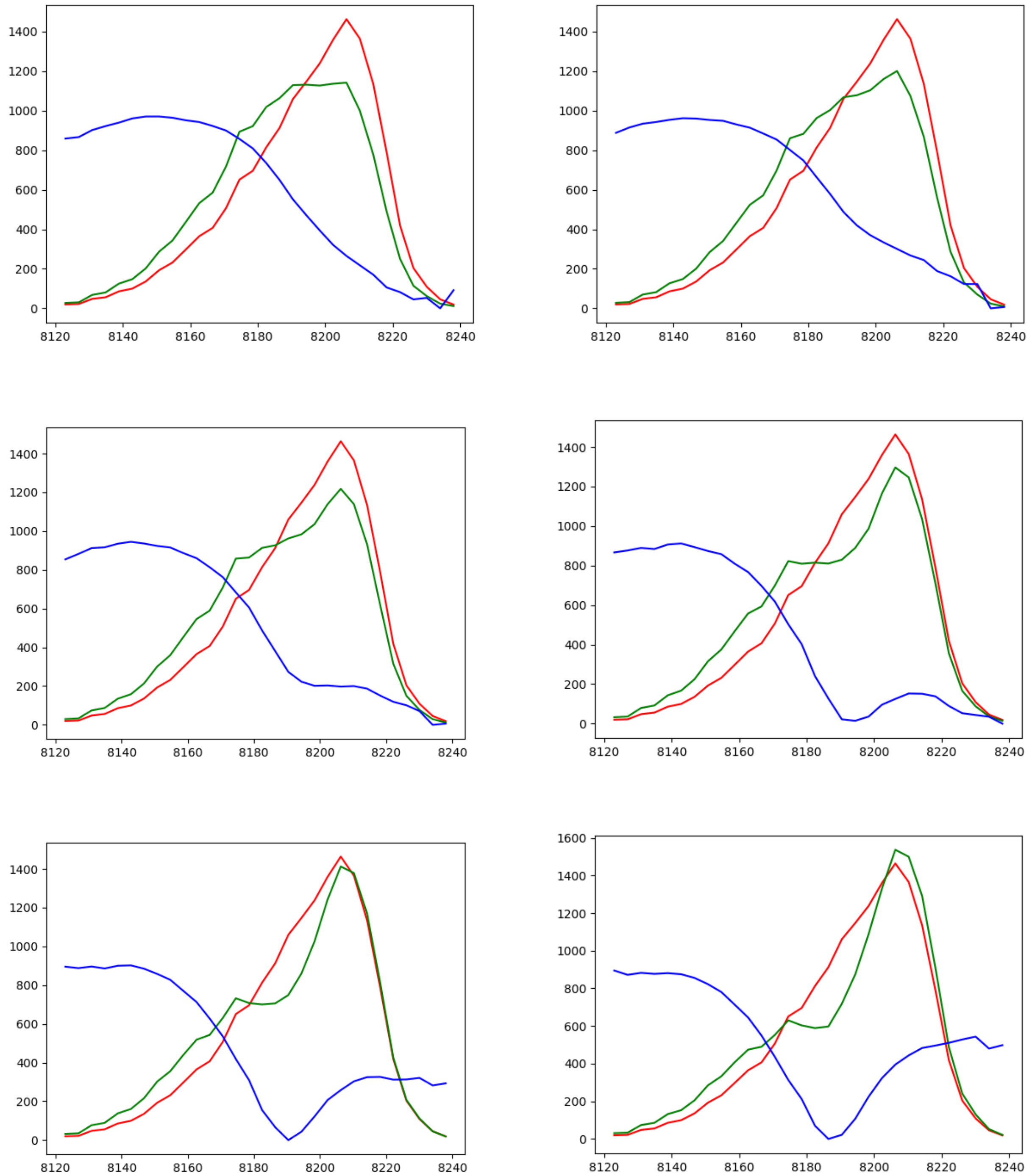


Figure 2.7: The various rise time definitions (start and end percent) give different correction functions. The most useful definition seems to be 10% to 50% which peaks at the lower tail of the damaged spectrum.

Note: The correction function (blue) is not to scale as it is normalized to 0 to 1 and would be invisible on this plot unless scaled appropriately. The X axis is in uncalibrated energy units and the Y axis is counts.

The blue curve represents the correction function.
The red curve represents the original spectrum.
The green curve represents the corrected spectrum using the corresponding correction function.

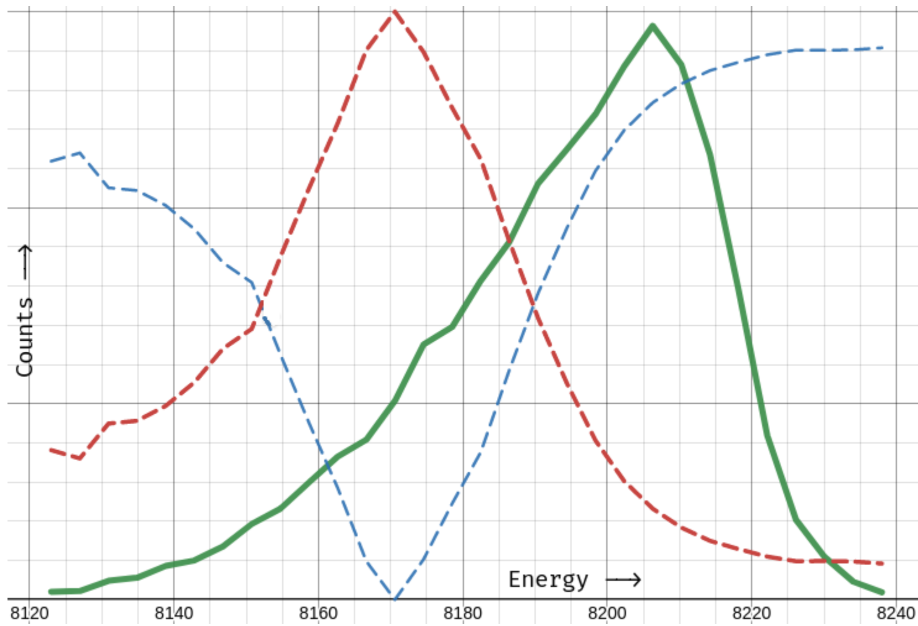


Figure 2.8: The correction function overlaid on the damaged spectrum. The red line represents the normalized values without inversion and the blue is normalized and inverted. The dotted lines are not to scale.

4. The correction function is then multiplied with the original spectrum using the following formula. This creates a graph with the desired shape but reduced counts/area.

$$\text{Corrected Spectrum} = f(x) \cdot c(x) \tag{2.4}$$

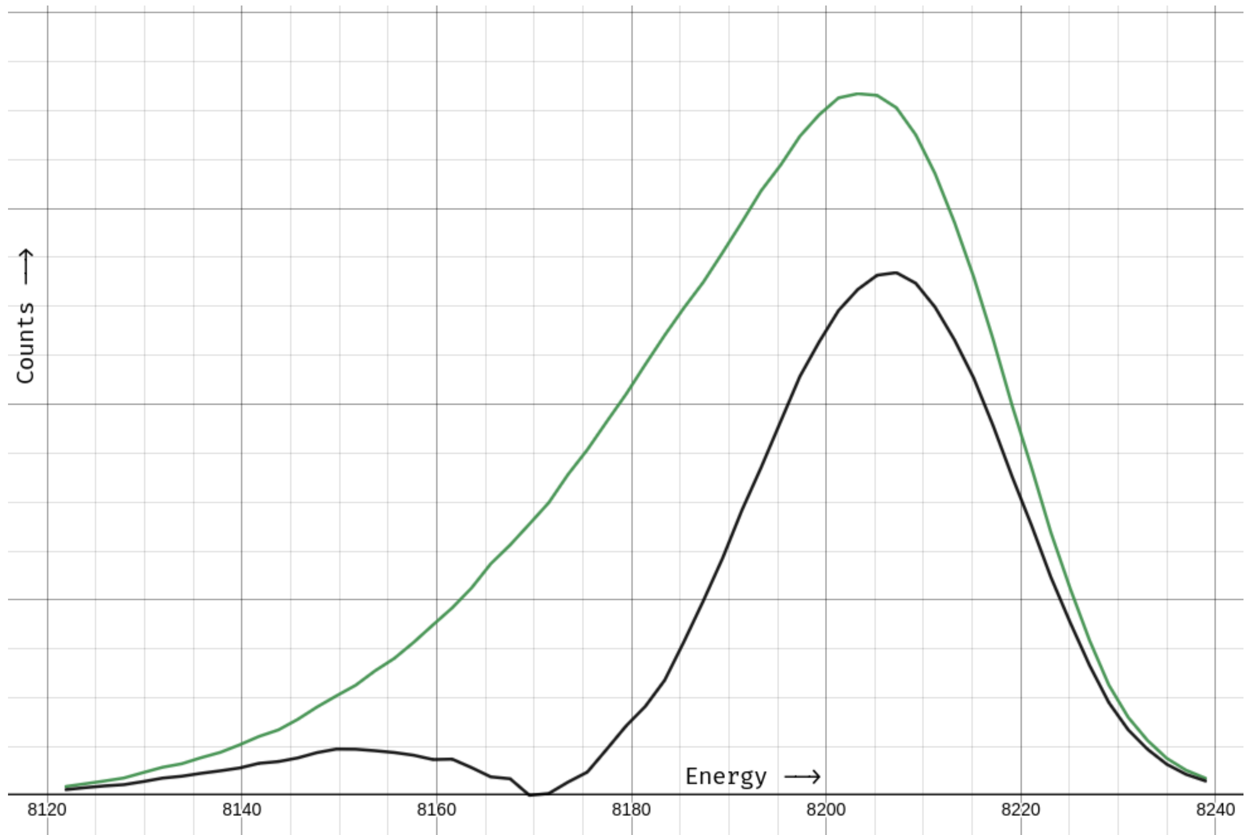


Figure 2.9: The Corrected Spectrum is shown in black. This correction has reduced height and area under the curve.

5. To deal with the loss in area under the curve, which is a serious issue as we lose resolution due to it, we modify the correction formula as follows.

To ensure the areas under the curve are the same we can add a constant to the correction function and assert the areas are equal to calculate it.

$$\text{Corrected Spectrum} = f(x) \cdot (c(x) + k) \tag{2.5}$$

where

$$\int_a^b f(x)\ dx = \int_a^b f(x) \cdot (c(x) + k)\ dx \tag{2.6}$$

$$\int_a^b f(x)\ dx - \int_a^b k \cdot f(x)\ dx = \int_a^b f(x) \cdot c(x)\ dx \tag{2.7}$$

$$(1 - k) \cdot \int_a^b f(x)\ dx = \int_a^b f(x) \cdot c(x)\ dx \tag{2.8}$$

$$(1 - k) = \frac{\int_a^b f(x) \cdot c(x)\ dx}{\cdot \int_a^b f(x)\ dx} \tag{2.9}$$

$$k = 1 - \frac{\int_a^b f(x) \cdot c(x)\ dx}{\int_a^b f(x)\ dx} \tag{2.10}$$

If we use this algorithm we finally get the following spectrum.

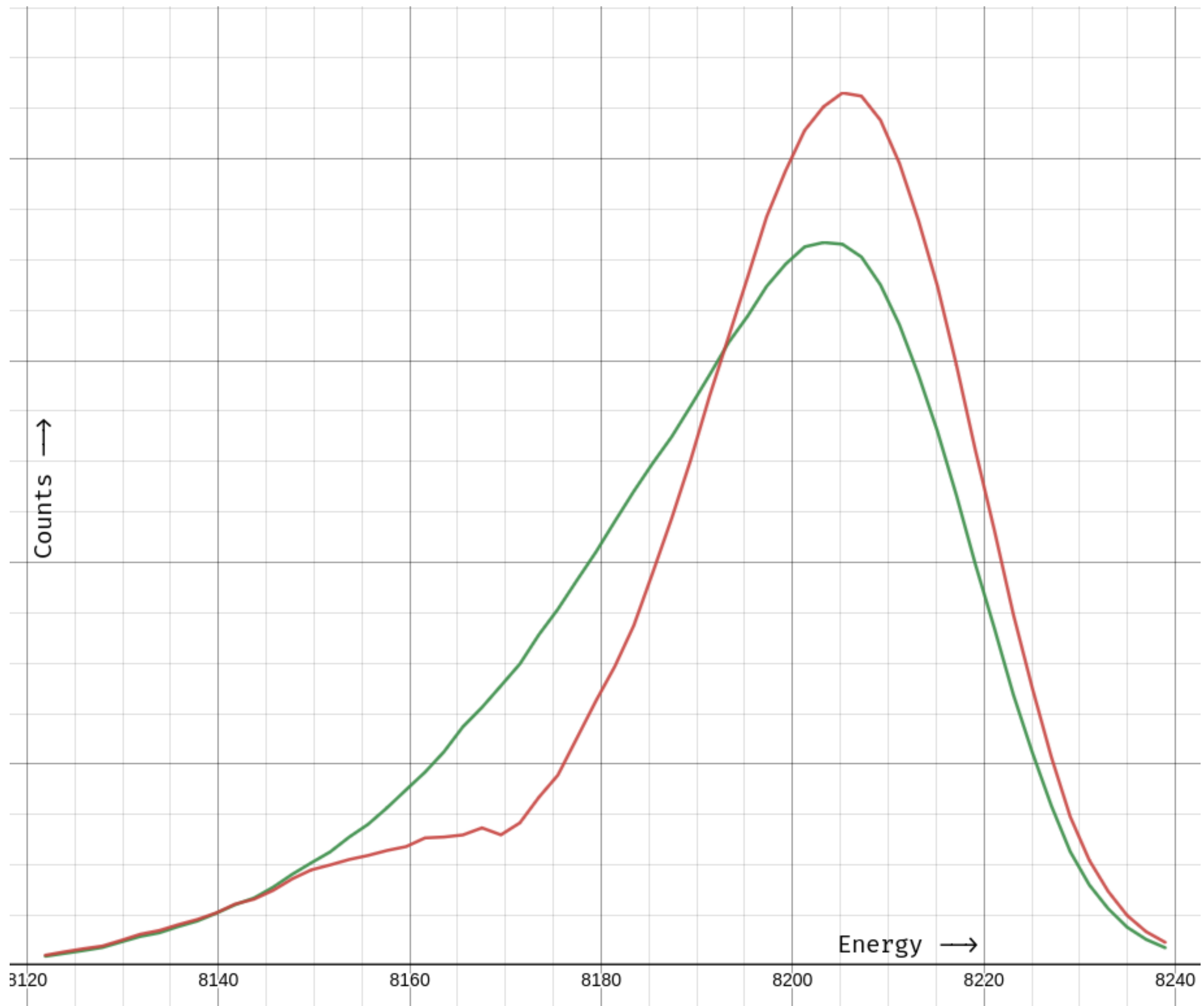


Figure 2.10: The Corrected Spectrum is shown in red. This correction has the same area under the curve as the original spectrum and consequently a higher and sharper peak.

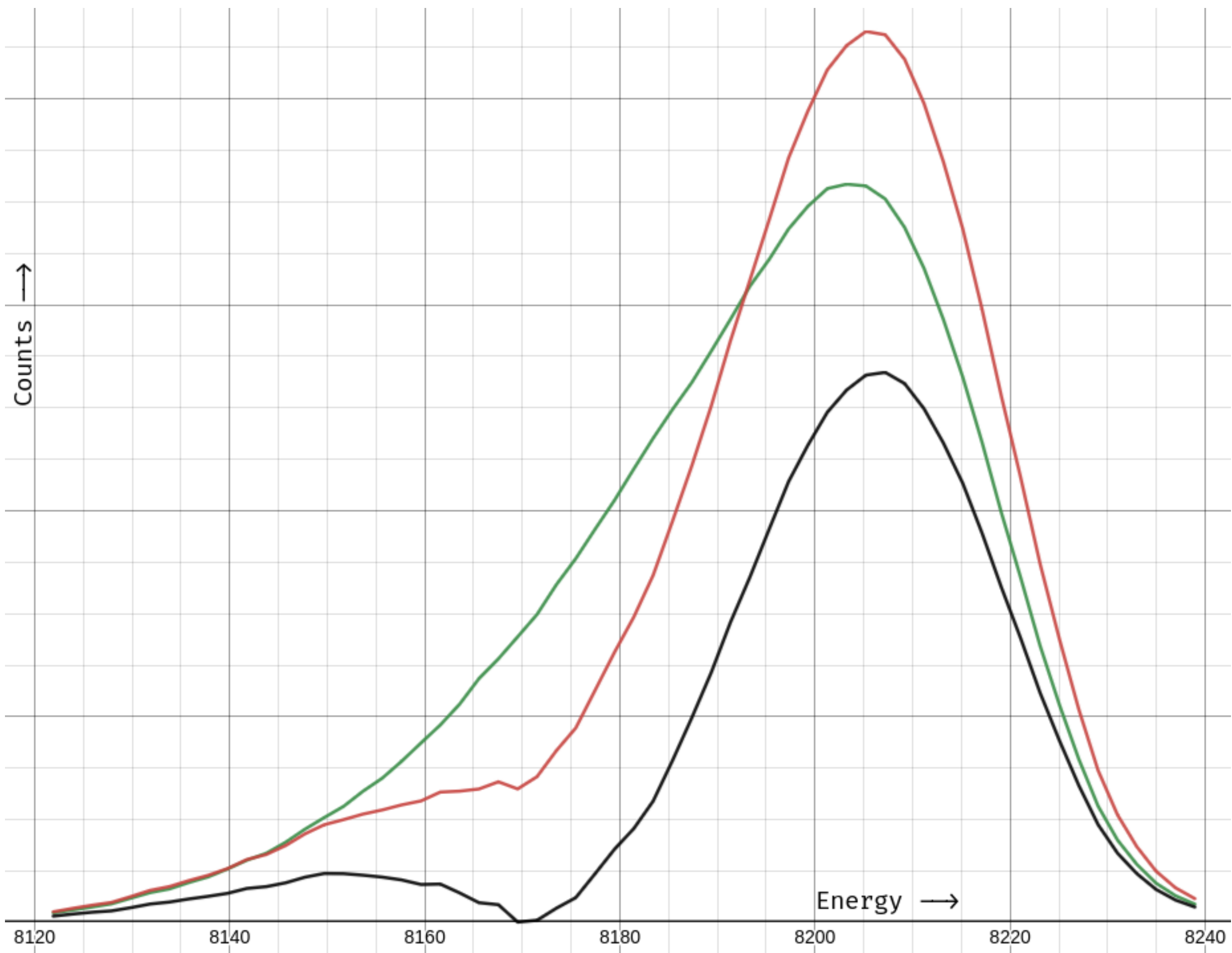


Figure 2.11: The Corrected Spectrum with compensation is shown in red and without compensation is shown in black. We can see that there is a significant improvement.

2.3.1 Properties of the Algorithm

This algorithm is capable of detecting the damaged parts of the spectrum and selectively correcting only those parts. It does not make any significant change to healthy spectra. This is a big advantage as it means it can be applied to the entire spectrum without any kind of peak detection. It is also reasonably performant although the implementation has room for improvement. For instance currently the algorithm loops over the spectrum three times. This could be a place for improvement.

2.3.2 Further Research

It seems that the biggest factor in resolution of the corrected spectrum is the area correction factor (k). If k is taken as a function of energy instead of a constant then it might be possible to increase the resolution even more, as currently on a conceptual level the algorithm simply redistributes the lost pulses onto the whole spectrum. A more sophisticated approach could use a Gaussian k factor to redistribute the pulses around the peak instead. This however comes with its own challenges as characterizing this function will be harder than it was for a constant.

2.4 Conclusion

Through this project I have learnt the working of gamma ray detectors, various pulse shaping methods, difficulties in analyzing data from detectors and the application of computer algorithms in correcting defects or errors in real world data.

Appendix A

Miscellaneous Gallery

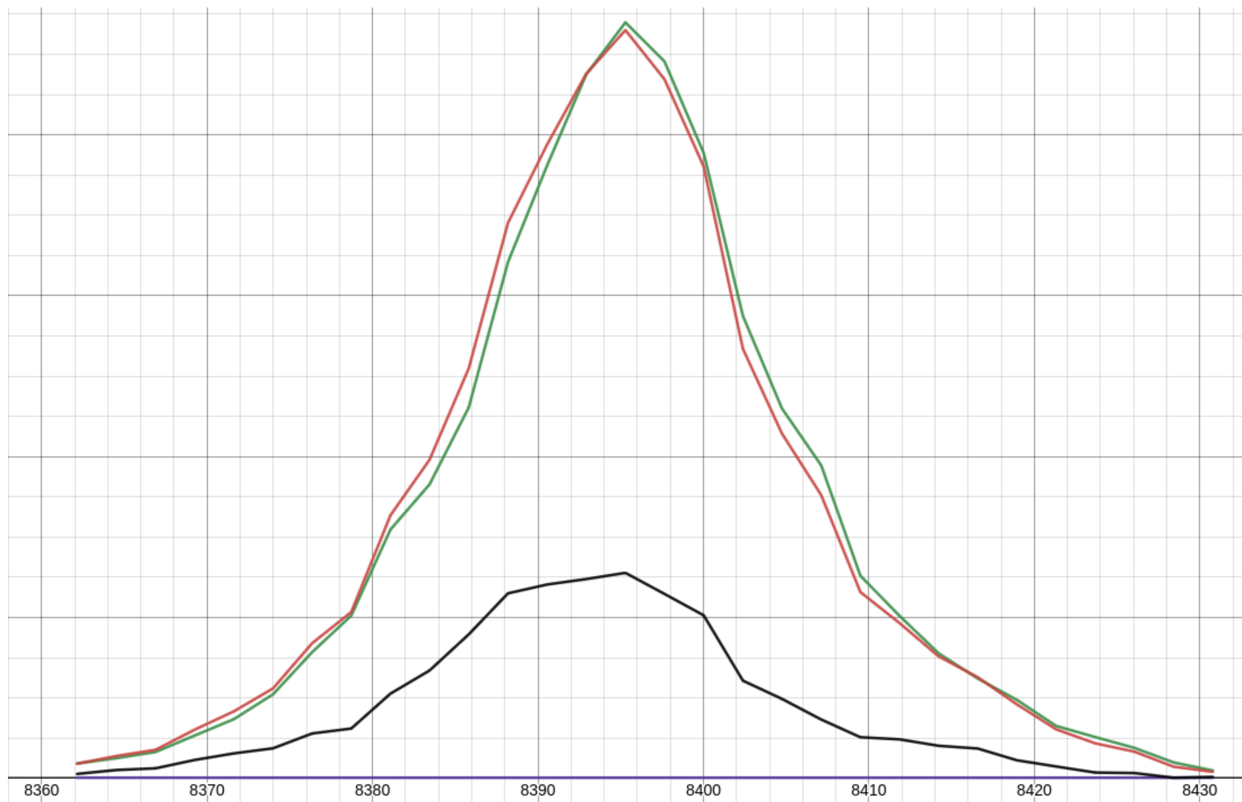


Figure A.1: Here we see what happens if we apply the correction algorithm to a healthy spectrum. The purple curve is the correction function and the red is the corrected spectrum. The black is the spectrum before area correction is applied.

Appendix B

Code

B.1 Trapezoidal Shaping in Python

```
import matplotlib.pyplot as plt
import queue

x = []
fy = []

l = 10
m = 50
M = 50000


def get_q(arr, index):
    if index >= 0 and index < arr.qsize():
        return arr.queue[index]
    else:
        return 0

def get_energy(y, m, l, M):
    filter_window = queue.Queue(m + 2 * l + 1)
    k = m + 2 * l
    n = 0
    last_r = 0
    last_p = 0

    energy = 0

    x = 1

    for y0 in y:
        put(filter_window, y0)
        k = filter_window.qsize() - 1
        d = get_q(filter_window, k) - get_q(filter_window, k - m - l) - get_q(filter_window, k
    - l) + get_q(filter_window, k - m - 2 * l)
        p = last_p + d
        res = last_r + p + d * M

        energy = max(res, energy)

        last_p = p
        last_r = res

        x += 1

        n += 1
    return (energy / (M * l))

def put(q, item):
    if q.full():
        q.get()
    q.put(item)


start = 0
end = 1000

```

```python
with open('total_1332_Area_Vishal_Proj.txt', 'r+', 1) as file:
    for _ in range(0, start * 1000):
        file.readline()

    for i in range(start, end):
        pos = file.tell()
        baseline = float(file.readline().strip().split(" ")[1])
        file.seek(pos)

        reader = (float(file.readline().strip().split(" ")[1]) - baseline for _ in range(0,
    1000))

        e = get_energy(iter(reader), m, l, M)
        fy.append(e)

plt.hist(fy, bins=20, range=(1005, 1025))
plt.show()
```

Note: All the root programs are light modifications of scripts provided to me.

B.2 2D Histogram of Rise Time and Energy in Root

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include "TH2F.h"
#include "TCanvas.h"

int matrix_2_peak() {
    std::ifstream inputFile("60Co_whole_Peak_ch15.txt");
    std::vector<float> times;
    std::vector<float> amplitudes;
    std::vector<float> energies;

    float t, y, E;
    while (inputFile >> t >> y >> E) {
        times.push_back(t);
        amplitudes.push_back(y);
        energies.push_back(E);
    }

    const int numPulses = energies.size() / 28;
    const int numBinsX = 50; // Number of bins for energy axis
    const float minEnergy = *std::min_element(energies.begin(), energies.end());
    const float maxEnergy = *std::max_element(energies.begin(), energies.end());
    const int numBinsY = 100; // Number of bins for rise time axis
    const float minRiseTime = 0.0;
    const float maxRiseTime = 100.0;

    TH2F* histogram = new TH2F("histogram", "2D Histogram", numBinsX, minEnergy, maxEnergy,
    numBinsY, minRiseTime, maxRiseTime);

    for (int i = 0; i < numPulses; i++) {
        // Obtain the energy of the pulse at the beginning
        float pulseEnergy = energies[i * 28 + 1];

        // Find the minimum amplitude as the baseline
        float baseline = *std::min_element(amplitudes.begin() + i * 28, amplitudes.begin() + (i
     + 1) * 28);


        // Obtain the maximum amplitude of the pulse
        float maxAmplitude = *std::max_element(amplitudes.begin() + i * 28, amplitudes.begin()
    + (i + 1) * 28);
        float pulseRiseTime_50 = 0.0; // time of the pulse at 50
        float pulseRiseTime_10 = 0.0;  // time of pulse at 10
        float pulseRiseTime = 0.0;     // Pulse rise time


        double amps[28];
        double tims[28];
        double start_time;
        double end_time;

        double high_percent = 0.5;
        double low_percent = 0.1;
```

```cpp

        for (int j = 0; j < 28; j++) {
            float amplitude = amplitudes[i * 28 + j];
            amps[j] = (double)amplitude;
            tims[j] = (double)times[i * 28 + j];


            if (amplitude < baseline + high_percent * (maxAmplitude-baseline)) {
                end_time = tims[j] + 15;
            }

            if (amplitude < baseline + low_percent * (maxAmplitude-baseline)) {
                start_time = tims[j] - 15;
            }
        }

        TSpline3 spline = TSpline3("Pulse", tims, amps, 28);

        int lod = 500; // Level of detail. The higher the number the more accurate the spline interpolation becomes.
        double increment = (end_time - start_time) / lod;
        double sample_x = start_time;

        for (int j = 0; j < lod; j++) {
            double amplitude = spline.Eval(sample_x);


            {if (amplitude < baseline + high_percent * (maxAmplitude-baseline)) {
                pulseRiseTime_50 = sample_x;
            }}

            {if (amplitude < baseline + low_percent * (maxAmplitude-baseline)) {
                pulseRiseTime_10 = sample_x;
            }}
            pulseRiseTime = pulseRiseTime_50 - pulseRiseTime_10;


            sample_x += increment;

        }

        histogram->Fill(pulseEnergy, pulseRiseTime);
    }

    TCanvas* canvas = new TCanvas("canvas", "2D Histogram", 800, 600);
    histogram->Draw("colz");

    canvas->SaveAs("histogram.png");

    return 0;
}
```

B.3 2D Histogram of Pulse Height and Energy in Root

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include "TH2F.h"
#include "TCanvas.h"

int matrix_2_whole_peak_amplitude() {
    std::ifstream inputFile("60Co_whole_Peak_Ch15.txt");
    std::vector<float> times;
    std::vector<float> amplitudes;
    std::vector<float> energies;

    float t, y, E;
    while (inputFile >> t >> y >> E) {
        times.push_back(t);
        amplitudes.push_back(y);
        energies.push_back(E);
    }

    const int numPulses = energies.size() / 28;
    const int numBinsX = 35; // Number of bins for energy axis
```

```
    const float minEnergy = *std::min_element(energies.begin(), energies.end());
    const float maxEnergy = *std::max_element(energies.begin(), energies.end());
    const int numBinsY = 100; // Number of bins for rise time axis
    const float minRiseTime = 1000.0;
    const float maxRiseTime = 1100.0;

    TH2F* histogram = new TH2F("histogram", "2D Histogram", numBinsX, minEnergy, maxEnergy,
    numBinsY, minRiseTime, maxRiseTime);

    for (int i = 0; i < numPulses; i++) {
        // Obtain the energy of the pulse at the beginning
        float pulseEnergy = energies[i * 28 + 1];

        // Find the minimum amplitude as the baseline
        float baseline = *std::min_element(amplitudes.begin() + i * 28, amplitudes.begin() + (i
     + 1) * 28);


        // Obtain the maximum amplitude of the pulse
        float maxAmplitude = *std::max_element(amplitudes.begin() + i * 28, amplitudes.begin()
    + (i + 1) * 28);

        float pulseRiseTime_50 = 0.0; // time of the pulse at 50
        float pulseRiseTime_10 = 0.0;  // time of pulse at 10
        float pulseRiseTime = 0.0;     // Pulse rise time


        for (int j = 0; j < 28; j++) {
            float amplitude = amplitudes[i * 28 + j];

            {if (amplitude < baseline + 0.5 * (maxAmplitude-baseline)) {
                pulseRiseTime_50 = times[i * 28 + j] - times[i * 28];
            }}

            {if (amplitude < baseline + 0.1 * (maxAmplitude-baseline)) {
                pulseRiseTime_10 = times[i * 28 + j] - times[i * 28];
            }}
            pulseRiseTime = pulseRiseTime_50 - pulseRiseTime_10;


        }
        histogram->Fill(pulseEnergy, maxAmplitude - baseline);
    }

    TCanvas* canvas = new TCanvas("canvas", "2D Histogram", 800, 600);
    histogram->Draw("colz");

    canvas->SaveAs("histogram.png");

    return 0;
}
```

B.4 Correction of a Damaged Spectrum in Root

This program takes any spectrum as input and outputs a corrected spectrum to the standard out. This is due to the fact that the spectrum is converted to an XY plot by the correction and the heights are no longer exactly the same and can't be interpreted as counts.

```
#include <iostream>
#include <fstream>
#include <vector>
#include "TH2F.h"
#include "TCanvas.h"
#include "TRandom.h"

int correction() {
    std::ifstream inputFile("60Co_whole_Peak_ch15.txt");
    std::vector<float> times;
    std::vector<float> amplitudes;
    std::vector<float> energies;

    float t, y, E;
    while (inputFile >> t >> y >> E) {
        times.push_back(t);
        amplitudes.push_back(y);
        energies.push_back(E);
```

```
    }

    const int numPulses = energies.size() / 28;
    const int numBinsX = 60; // Number of bins for energy axis
    const float minEnergy = *std::min_element(energies.begin(), energies.end());
    const float maxEnergy = *std::max_element(energies.begin(), energies.end());
    const int numBinsY = 100; // Number of bins for rise time axis
    const float minRiseTime = 0.0;
    const float maxRiseTime = 100.0;

    TH2F* histogram = new TH2F("histogram1", "2D Histogram", numBinsX, minEnergy, maxEnergy, numBinsY, minRiseTime, maxRiseTime);

    TH1F* hist = new TH1F("histogram2", "1D Histogram", 40, minEnergy, maxEnergy);

    double last_rise_time = 0;

    for (int i = 0; i < numPulses; i++) {
        // Obtain the energy of the pulse at the beginning
        float pulseEnergy = energies[i * 28 + 1];
        // Find the minimum amplitude as the baseline
        float baseline = *std::min_element(amplitudes.begin() + i * 28, amplitudes.begin() + (i + 1) * 28);


        // Obtain the maximum amplitude of the pulse
        float maxAmplitude = *std::max_element(amplitudes.begin() + i * 28, amplitudes.begin() + (i + 1) * 28);
        float pulseRiseTime_50 = 0.0; // time of the pulse at 50
        float pulseRiseTime_10 = 0.0;  // time of pulse at 10
        float pulseRiseTime = 0.0;     // Pulse rise time


        double amps[28];
        double tims[28];
        double start_time;
        double end_time;

        double high_percent = 0.5;
        double low_percent = 0.1;

        for (int j = 0; j < 28; j++) {
            float amplitude = amplitudes[i * 28 + j];
            amps[j] = (double)amplitude;
            tims[j] = (double)times[i * 28 + j];


            if (amplitude < baseline + high_percent * (maxAmplitude-baseline)) {
                end_time = tims[j] + 15;
            }

            if (amplitude < baseline + low_percent * (maxAmplitude-baseline)) {
                start_time = tims[j] - 15;
            }
        }

        TSpline3 spline = TSpline3("Pulse", tims, amps, 28);

        int lod = 500;
        double increment = (end_time - start_time) / lod;
        double sample_x = start_time;

        for (int j = 0; j < lod; j++) {
            double amplitude = spline.Eval(sample_x);


            {if (amplitude < baseline + high_percent * (maxAmplitude-baseline)) {
                pulseRiseTime_50 = sample_x;
            }}

            {if (amplitude < baseline + low_percent * (maxAmplitude-baseline)) {
                pulseRiseTime_10 = sample_x;
            }}
            pulseRiseTime = pulseRiseTime_50 - pulseRiseTime_10;


            sample_x += increment;
```

```
        }

        last_rise_time = pulseRiseTime;
        histogram->Fill(pulseEnergy, pulseRiseTime);
        hist->Fill(pulseEnergy);
    }

    TAxis *xaxis = histogram->GetXaxis();

    TH1F* histo = new TH1F("histogram3", "1D Histogram", 25, minEnergy, maxEnergy);

    TRandom* rand = new TRandom();

    double max_val = 0;
    double min_val = 1000;

    double window_a = 20;
    double window_b = 40;
    double window_c = 30;
    double window_d = 50;

    int windowSize = 10;

    for (int bin = 1; bin <= numBinsX; bin++)
    {
        int start_bin = max(1, bin - windowSize / 2);
        int end_bin = min(numBinsX, bin + windowSize / 2);

        TH1D* slice = histogram->ProjectionY("slice", start_bin, end_bin);
        Double_t val = slice->Integral(window_a,window_b) / slice->Integral(window_c,window_d);

        max_val = max(val, max_val);
        min_val = min(min_val, val);
    }

    double last_x = 0;
    double last_y = 0;
    double last_y1 = 0;

    double integral2 = 0;

    double integral = 0;

    for (int bin = 1; bin <= numBinsX; bin++)
    {
        int start_bin = max(1, bin - windowSize / 2);
        int end_bin = min(numBinsX, bin + windowSize / 2);

        TH1D* slice = histogram->ProjectionY("slice", start_bin, end_bin);
        Double_t val = slice->Integral(window_a,window_b) / slice->Integral(window_c,window_d);

        val /= max_val;

        Double_t binCenter = xaxis->GetBinCenter(bin);

        double y = (1 - val) * (slice->Integral());
        if (bin == 1) {
            last_x = binCenter;
            last_y = y;

            last_y1 = (slice->Integral());
        } else {
            integral += 0.5 * abs(binCenter - last_x) * (last_y + y);
            integral2 += 0.5 * abs(binCenter - last_x) * (last_y1 + (slice->Integral()));


            last_x = binCenter;
            last_y = y;
            last_y1 = (slice->Integral());
        }

    }

    double c = 1 - integral / integral2;

    cout << "Energy (binCenter)" << "," << "Uncorrected Spectrum" << "," << "Correction
    Function"  << "," << "Corrected Spectrum (Without count compensation)" << "," << "Corrected
```

```
    Spectrum (With count compensation)" << "\n";
169     for (int bin = 1; bin <= numBinsX; bin++)
170     {
171         int start_bin = max(1, bin - windowSize / 2);
172         int end_bin = min(numBinsX, bin + windowSize / 2);
173 
174         TH1D* slice = histogram->ProjectionY("slice", start_bin, end_bin);
175         Double_t val = slice->Integral(window_a,window_b) / slice->Integral(window_c,window_d);
176 
177         val /= max_val;
178 
179         Double_t binCenter = xaxis->GetBinCenter(bin);
180 
181         double y = (1 - val) * (slice->Integral());
182         if (bin == 1) {
183             last_x = binCenter;
184             last_y = y;
185 
186             last_y1 = (slice->Integral());
187         } else {
188             integral += 0.5 * abs(binCenter - last_x) * (last_y + y);
189             integral2 += 0.5 * abs(binCenter - last_x) * (last_y1 + (slice->Integral()));
190 
191 
192             last_x = binCenter;
193             last_y = y;
194             last_y1 = (slice->Integral());
195         }
196 
197         cout << binCenter << "," << slice->Integral() << "," << (1 - val) * 5000  << "," << (1
    - val) * (slice->Integral()) << "," << ((1 - val) + c) * (slice->Integral()) << "\n";
198 
199         for (int l = 0; l < histogram->ProjectionY("s", bin, bin)->Integral(); l ++) {
200 
201             double smp = rand->Uniform();
202 
203             if (smp < 1 - val)
204                 histo->Fill(binCenter);
205         }
206     }
207 
208 
209     TCanvas* canvas = new TCanvas("canvas", "2D Histogram", 800, 600);
210     histo->Draw("PFC");
211     hist->Draw("same");
212 
213     return 0;
214 }
```

Bibliography

[1] GoLuckyRyan. Trapezoid filter revisit, May 2022. `https://nukephysik101.wordpress.com/2022/05/11/trapezoid-filter-revisit/`.

[2] G.F. Knoll. *Radiation Detection and Measurement*. Wiley, 2010.

[3] Kavita Pathak and Dr. Sudhir Agrawal. Vhdl simulation of cusp-like filter for high resolution radiation spectroscopy. 2015.