# Finding the Mass of $\mathbf{K}^0$ and $\Lambda^0$ with a Program Created to Analyze a Bubble Chamber Photograph

Matthew Hipp

*PHY 4822L, Spring 2023*

January 21, 2023

**Abstract**

Through the use of opencv and python, a program was created to read through an image of a bubble chamber in order to gain information about the paths of a negative pion and positive pion, the decay products of a neutral kaon. This program was also used to analyze the paths of a negative pion and a proton which came from the decay of a neutral lambda particle. Using the information of their decay products, the masses of the kaon and lambda particles were calculated to be 497.29 $\frac{MeV}{c^2}$ and 1078.71 $\frac{MeV}{c^2}$, respectively.

## Contents

1 Introduction 3

2 Background 3

3 Program Setup 7

4 Program Results 8

5 Analysis 9

6 Conclusion 9

Links and References 10

## 1 Introduction

Through the photos of a a bubble chamber, it is possible to calculate the mass of a neutral particle that decayed inside of the system. Figure 1 shows the source photograph that contains a pion-proton collision which results in the production of a neutral lambda particle and a neutral kaon particle. After some time, the lambda particle an the kaon particle decay into a proton and a negative pion and a positive and negative pion, respectively. In order to calculate the momentums, energies, and masses of these particles, a program was written to analyze the paths of the decay products and use them to calculate information about the particle they decayed from. This works by defining paths, start positions, and end positions on the source image with different color, and then reading that file through opencv to find pixels which share the colors we're looking for. Converting the positions found in opencv to vectors that are more easily understood allows us to perform vector operations on the path to get all of the information needed to calculate the momentum and energy. Figure 2 details the particle interaction that is begin analyzed. All of the images and scripts used for the program can be found in the repository [1] linked in the "links and references" section.

Following this section, the background section will go over the theory of how the code works. All of the mathematics required to understand the algorithms are located there. The setup section will give instructions on how what went in to setting up the environment for the program to run in. The results section talks about the programs output. In the analysis section, the absolute and relative errors and what may have induced them are discussed. Lastly, the conclusion gives a brief summary of the results and discusses future additions to the program.

## 2 Background

It is important to know how opencv reads images. As it turns out, opencv likes to read vertically and then horizontally, so a pixel given to you by opencv will be in the format [y, x]. In order to work more intuitively with the algorithms, a vector library was created. The function `get_vector` takes two opencv points, gets the distance between their respective x, y coordinates, and maps it to a vector formatted [x, y]. Note that when this article refers to "getting the vector", what is meant is that we are using this function to converting two opencv points into a trajectory with normal [x, y] coordinates. The rest of the functions in the vector library are normal things you might see when dealing with vectors, like finding the scalar product between two vectors or normalization.

The vector library is used extensively throughout the project, It is the basis for the algorithm which finds the sagitta. Before that, it is important to break down a particle's path through a magnetic field into 4 segments: the arc length, base, sagitta and the radius of curvature. These are pictured in figure 3. Because of their curved nature, in order to get all of the information about the paths, the

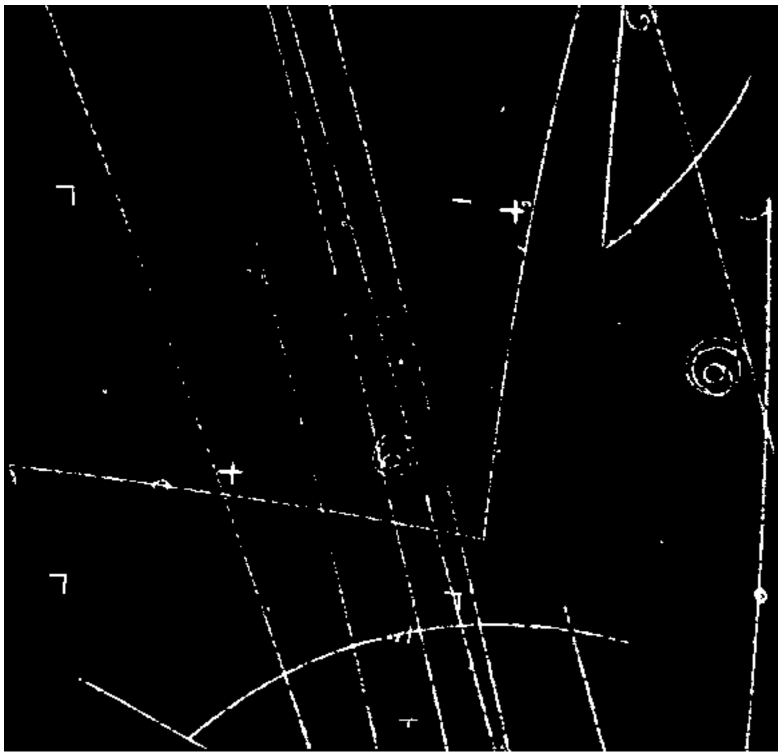

Figure 1: The source image used to draw all of the paths on [2]

starting position and the complete path were painted onto the source image for each particle. All of this information is stored in a class called by the name of `Path`.The arc length can be obtained simply by counting each pixel in the path the particle takes. The base vector is obtained by finding the vector between the first point and last point of the particle's path. The magnitude of this vector will give you the length of the base, but in this experiment, the vector was normalized since we only need the direction it is in. Finding the sagitta requires use of the base vector and the position vector of each pixel in the path. A pixel's position vector can be found by getting the vector between the starting position and the current position of the pixel. By subtracting the position vector from its projection onto the base direction vector, we can find a vector perpendicular to the base, whose magnitude gives us the distance between the path and the base. The maximum of this distance is the sagitta, as shown below:

$$\mathbf{d}_i = \mathbf{r}_i - (\mathbf{r}_i \cdot \hat{b})\hat{\mathbf{b}} \tag{1}$$

$$s = max\{||\mathbf{d}_i||\}_i^n \tag{2}$$

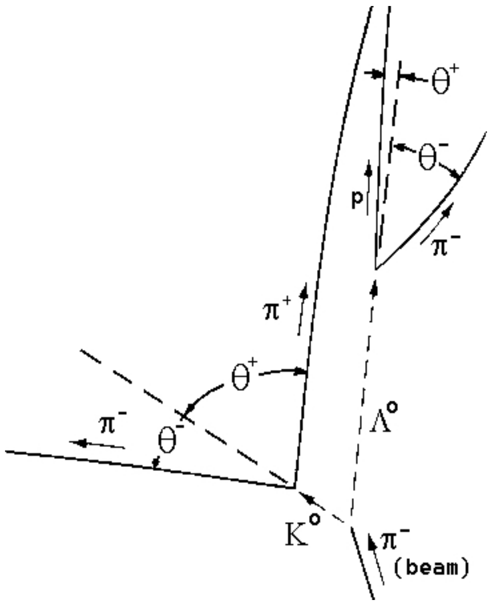

Figure 2: Image detailing the pion-proton collision and the decays of a neutral kaon and neutral lambda particle [2]

where $\mathbf{d}_i$ is the distance between pixel $i$ and the base, $\mathbf{r}_i$ is the position vector of pixel $i$, $\hat{\mathbf{b}}$ is the unit base vector, $n$ is the number of pixels in the arc, and $s$ is the sagitta. The sagitta and the arc length can be used to find the radius of curvature:

$$R = \frac{L^2 + 4s^2}{8s} \tag{3}$$

where $R$ is the radius of curvature, and L is the arc length. Since R is in pixels, a conversion factor, $f$, is needed to get from pixels to millimeters. This happens to be a per-computer value and getting this value is detailed in the next section. There is also a magnification factor, $g$, that must be accounted for. The true radius of curvature is found by dividing by the conversion factor and multiplying by the magnification factor: $g \cdot f \cdot R$.

Using this geometry, the momentum for a particle in a magnetic field can be found:

$$p = cRB \cdot 10^{-6} \tag{4}$$

where $p$ is the momentum in $\frac{MeV}{c}$, $c$ is the speed of light, $R$ is our radius of curvature, and $B$ is the magnitude of the magnetic field. The $10^{-6}$ term is there to make sure we get the desired units. It is important to consider what units you are using for c and account for this. R must be in meters and $B$

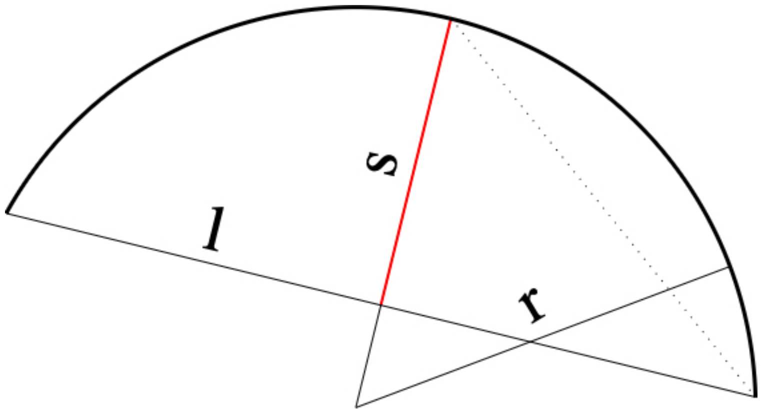

Figure 3: Visualization of the arc length (total length of the semicircle), sagitta (s), radius of curvature (r), and base (l) [3]

must be in Telsas, so $c$ must be converted to units of $\frac{m}{s}$ if it isn't already. In this experiment, $c$ ended up having units of $\frac{mm}{ps}$ which led to the momentum equation begin changed to:

$$p = cRB \cdot 10^3 \tag{5}$$

With the momentum and the mass, one can find the particle's relativistic energy:

$$E = \sqrt{p^2c^2 + m^2c^4} \tag{6}$$

Going forward, the kaon and lambda particles will be referred to as parent particles, and their decay products will be referred to as child particles. For the kaon, the child particles would be a positive pion and a negative pion. For the lambda particle, the child particles would be a negative pion and a proton. Equations one through six all provide information about the child particles. Using this information we can recover the momentum and energy of the parent particle. We start by defining the parent particle's trajectory. Since the parents are neutral particles, their trajectories will be linear, thus only requiring two positions to be fully defined. These positions are obtained through looking at an image which defines the starting and ending positions of the parent particles in two separate colors. Finding the vector between these two points gives the direction of travel of the parent particle, while it magnitude gives us the distance the parent particle traveled before decaying into its children. We can get the initial direction of the children by finding the vector between the starting position and the 30th pixel in the path. This pixel was chosen because it offered an approximate tangential vector to the curve. Intuitively, one might try to pick the first and second pixels to define the initial direction, but because we're dealing with paths formed by discrete pixels, this is disadvantageous. The second pixel often points the initial direction too much horizontally or vertically, which means the initial particle direction would not be as tangent as we'd like it to be.

Using the child's initial direction vector and the parent's direction vector, we can find the angle between the parent and the child using the `get_angle` function in the vector library. With this angle, we can find the projection of the child's momentum along the parent's trajectory. Summing these momentum projections gives us the momentum of the parent particle:

$$p_0 = p_1 \cos\theta_1 + p_2 \cos\theta_2 \tag{7}$$

where $p_0$ is the parent's momentum, $p_1$, $p_2$, $\theta_1$, and $\theta_2$ are the children's momenta and angles respectively.
Finding the energy of the parent particle is much simpler. It is just the sum of each child's energy:

$$E_0 = E_1 + E_2 \tag{8}$$

Using the energy and momentum of the parent particle, it's mass can now be found:

$$mc^2 = \sqrt{E_0^2 - p_0^2 c^2} \tag{9}$$

which will give us the parent particle's mass energy in $\frac{MeV}{c^2}$.

## 3 Program Setup

To create the images used in the analysis, a web photo editor called Pixlr E [4] was used. I used a source image that was 580x559 px but that size does not matter. What matters is that the same source image is used for each path that is drawn. I created a 6 separate images – four of which defined the starting position and the paths of the child particles. The remaining two contained the initial and final positions of the parent particles. Three separate colors were used. In [B, G, R] colorspace (which is the colorspace of opencv): [0, 0, 255] was used to draw the particle paths, [255, 0, 238] was used to define every starting position, and [255, 187, 0] was used to define the parent particle's ending position. These values, along with the image width and height, were stored in variables under the `# image constants` section of a python file by the name of "constants". This file is where all of the image, environment, and physical constants are kept. In regards to drawing the particle paths, it is good to stick to a few rules. First and foremost, the path should always be one pixel wide. Diagonals should be prioritized as the algorithm will includes them. The particle path in the image is often 2 pixels or 3 pixels wide. Trying to stick to the center of the source path's width is a good rule of thumb to minimize error. Lastly, use good judgment on which pixels to shade in order to minimize error. This step is where the most error will be introduced, but it can easily be minimized by intuitively shading the pixels. Lastly, wherever the parent particle ends should be where the child particles start.

The next step would be to figure out the conversion factor and magnification factor. For the conversion factor, find the horizontal and vertical pixel counts of the monitor that is running the program. Summing their squares and taking the square root gives the pixel count of the monitor's body diagonal. Using this, you can find the $\frac{px}{mm}$ conversion factor by dividing the monitor's body diagonal length in pixels by its length in millimeters. For example, the 2022 M2 Macbook Air has a screen resolution of 2560x1664px. The length of the body diagonal in pixels would then be: $\sqrt{2560^2 + 1664^2} = 3053.2763px$. Looking on the laptop's product specifications page, Apple tells us that the screen's body diagonal is 13.6 in, or 345.44 mm. Dividing the body diagonal length in pixels by its length in mm gives us a conversion factor of: $f = \frac{3053.3763}{345.44} = 8.8388$. For the magnification factor, take the pixel height of the image, convert it to millimeters, and then divide by 175 mm. This turned out to be $g = 559px \cdot \frac{1mm}{8.8388px} \cdot \frac{1}{175mm} = 0.3612$ for an image height of 559 px and a conversion factor of 8.8388. These values were once again put in the "constants.py" file under the section `# environment constants`. There is one other scale factor there which I had to set to 0.15, but this will be discussed later on in the article. The rest of the constants in this file are physical constants. This includes all of the accepted values for the each particle's mass in $\frac{MeV}{c^2}$, the magnitude of the magnetic field - 1.5 T, and the speed of light - 0.3 $\frac{mm}{ps}$.

Before running the program, make sure that the images that are being used for the analysis are in the "img" directory, all of the constants are set inside of the "constants.py" file, and that the images are correctly being referenced in the "analysis.py" file. If all of that is in order, navigate to your working directory in the shell and type `python3 analysis.py` to run the analysis. The program takes about 5 seconds to complete and will list all of the information in the shell.

## 4 Program Results

For the kaon, it was found that the negative pion had a radius of curvature of 0.5977 mm and made an angle of 28.12 degrees with the kaon's trajectory. It contributed 237.22 $\frac{MeV}{c}$ toward the kaon's momentum and 302.94 MeV toward the kaon's energy. The positive pion had a radius of curvature of 0.6959 mm and made an angle of 56.10 degrees with the kaon's trajectory. It contributed 174.65 $\frac{MeV}{c}$ to the kaon's momentum and 342.77 MeV to the kaon's energy. The kaon had a total momentum of 411.87 $\frac{MeV}{c}$ and total energy of 645.71 MeV. This led to it's calculated mass being 497.29 $\frac{MeV}{c^2}$.

For the lambda particle, the negative pion had a radius of curvature of 0.0192 mm and made an angle of 44.22 degrees with the lambda particle's trajectory.It contributed 6.20 $\frac{MeV}{c}$ to the lambda particle's momentum and 139.67 MeV to the lambda particle's energy. The proton had a radius of curvature of 0.3347 mm and made an angle of 4.41 degrees with the lambda particle's trajectory. It contributed 150.19 $\frac{MeV}{c}$ to the lambda particle's momentum and 950.31 MeV to the lambda particle's energy. The lambda particle had a total momentum of 156.38 $\frac{MeV}{c}$ and total energy of 1089.98 MeV. This led to it's calculated mass being 1078.71 $\frac{MeV}{c^2}$.

## 5 Analysis

Both calculated masses were very close to their respective accepted values. The accepted value of the kaon's mass is 497.611 $\frac{MeV}{c^2}$. Our calculated value was 497.29 $\frac{MeV}{c^2}$, which has an absolute error of 0.32 and a relative error of 0.0006. The accepted value for the lambda particle is 1115.6 $\frac{MeV}{c^2}$ compared to our calculated value of 1078.71 $\frac{MeV}{c^2}$. This results an in absolute error of 36.89 and a relative error of 0.03. The program seems to be very precise at calculating the mass values. This is where the scale factor comes in to play. While creating the program, it was noticed that the results were consistently off by the same scale value, 0.15. Once applying this scale factor, everything lined up and the program could complete is task. It still remains a mystery what this scale factor is and where it comes from, but I believe it to be a per-computer scale factor. I might have miscalculated the conversion factor between pixels and length, or even the magnification factor. It might be due to the multiplication of two scale factors that causes some offset in the actual values. All that is known about it is that its some scaling quantity that offsets the errors, be it scaling errors or round off errors, or errors that I caused while drawing the paths.

## 6 Conclusion

The kaon's mass was calculated to be 497.29 $\frac{MeV}{c^2}$ with a relative error of 0.0006. The lambda particle's mass was calculated to be 1078.71 $\frac{MeV}{c^2}$ with a relative error of 0.03. Thanks to the scaling factor, these errors were very low. What the program returned on the information about the child particles seemed to be quite accurate as well. The angles of each of the child particles had the right magnitudes relative to each other and the source image, and their radius curvatures lined up as well. There are a lot of improvement to be made to the program, namely in how it goes about calculating the path information. Optimizing the loops and cutting down the time it takes to find the sagitta are on the list of things to do. In terms of the constants, with more time it would've been great to find out what exactly the scale factor is and where it was coming from by testing the program on multiple computers, monitor sizes, and with different image sizes. There might even be a calibration function that could be made to find the optimal scale factor to reduce the error on both the masses of both particles. Lastly, another way to expand the program is to include the lifetimes of the parent particles. Since the distance and momentum are already obtained, it should be relatively straightforward to find their lifetime, since we know when they were produced and when they decayed.

## Links and References

[1] *Code repository*, `https://github.com/hippmatthew/bubblechamber`.

[2] *Lab reference manual*, source of the source image and particle image.

[3] *Sagitta image*, `https://en.wikipedia.org/wiki/Sagitta_(geometry)#/media/File:Sagitta.svg`.

[4] *Pixlr e*, `https://pixlr.com/e/#home`.