Data Generator and Maximum Likelihood Fitting

Azid Harun

s1579320

November 24, 2018

Abstract

A set of 10,000 random events with a distribution of decay time and angle are generated using box method. A maximum likelihood fit are performed twice where the first one with only decay time information and the other with both decay time and angle information. The best fit values of the physics parameter, F, τ_1 and τ_2 together with proper errors for the first maximum likelihood fit are determined to be 1.0 ± 0.0, 0.0 ± 0.0 and 0.0 ± 0.0, respectively. In the case for the second maximum likelihood, the parameters F, τ_1 and τ_2 together with proper errors are determined to be 0.5459(32), 1.3952(93)μs and 2.7088(215)μs, respectively.

1 Introduction

The objective of this report is to differentiate between a well-behaved fit and a badly behaved fit. The case in this report is about a data set corresponding to a decay of a particular radioactive particle X to a daughter particle D where this decay consists of two contributing components in which each of them have a slight different values of lifetime. These components also have different angular distribution of D with respect to X.

For every event, the observed quantities are identified to be:

- Decay time of X, t
- Decay angle of D with respect to X, θ

The probability density function (PDF) of these two decay components are:

$$P_1(t,\theta;\tau_1) = \frac{1}{N(\tau_1)}(1+\cos^2\theta) \times \exp(\frac{t}{\tau_1}) \tag{1}$$

$$P_2(t,\theta;\tau_2) = \frac{1}{N(\tau_2)}3\sin^2\theta \times \exp(\frac{t}{\tau_2}) \tag{2}$$

where N_i is the normalisation function of the PDF.

2 Methods

2.1 Part 1: Generate and plot simulated data

With the given components of PDF of the decay, a set of 10,000 random events with a distribution of decay times, t and angle θ. This is done by using box method.

For every random sample generation, the maximum value of total PDF of the decay is first evaluated. The interval for the decay times and angles are set up to be in between 0 to 10 μseconds and 0 to 2π, respectively. Random sample of decay time and angle are then generated such that their values are in their corresponding intervals. With these values, the total normalised PDF of the decay is then evaluated and declared as *yval*. A positive random value, *ythrow* is generated such that it is lower than the maximum value of total PDF of the decay.A while loop is then performed such that the decay time and angle will only be taken if *yval* is greater than *ythrow*, otherwise the random sample generation process is then repeated until the condition is fulfilled. The histograms describing the decay times and angle distribution of the data are then plotted using matplotlib package.

The entire process above is then repeated with F = 0.0, 1.0 to study the behaviour of components of the decay PDF.

2.2 Maximum Likelihood fitting

The best value of physics parameters F, τ_1 and τ_2 are determined by maximising the log likelihood function $\mathcal{L}$ where

$$\mathcal{L} = \prod_i P(t_i, \theta_i; F, \tau_1, \tau_2) \tag{3}$$

where P is the total PDF of the decay. This is equivalent to minimise the negative log likelihood, $\mathcal{NLL}$ function where this function can be expressed as shown in Equation 4

$$\mathcal{NLL} = -\log \mathcal{L} = -\sum_i \log P(t_i, \theta_i; F, \tau_1, \tau_2) \tag{4}$$

This process is done by using a Python-friendly minimiser package known as iminuit. iminuit minimise any given function by using Minuit2, an excellent trained code developed and maintained by the scientists at CERN, the worlds leading particle accelerator laboratory. Minuit2 has an outstanding performance in minimising functions compared to other minimiser and it is also one of the minimisers that can compute proper error for any given parameter to be minimised.

In minimising the $\mathcal{NLL}$, the minimiser is fed by the given initial value of the physics parameter F, τ_1 and τ_2. It is then returned the new values of the same parameters in which the $\mathcal{NLL}$ evaluated at these parameters is closer to its minimum point. This process is repeated until the difference between the calculated $\mathcal{NLL}$ at two consecutive steps is smaller than the given threshold value. Having the threshold value of 0.0 implies that the final calculated $\mathcal{NLL}$ is equivalent to the minimum point of the $\mathcal{NLL}$ in Equation 4 and this is possible to be done with iminuit.

The data required for calculating $\mathcal{NLL}$ is provided in a given file named **datafile-Xdecay.txt**. The simplistic error for each of the estimated physics parameter F, τ_1 and τ_2 are calculated such that the idea in calculating this simplistic error is that the error of the estimated parameter is equivalent to 1 unit if $\mathcal{NLL}$ increases by 0.5 from its minimum point.

The methods that is implemented to bring about the idea above are arrays of same size around the estimated value of physics parameter F, τ_1 and τ_2 are created. By using *for* loop, the elements of each arrays that have the same number of index are used to calculate the $\mathcal{NLL}$. A point where the $\mathcal{NLL}$ increased by 0.5 from its minimum point is obtained. The index of this point is then identified and used to find the value of the corresponding parameter considered in this case that has the same number of index . The error of the best estimated parameter is equivalent to the difference between this value and the estimated value of the parameter.

2.2.1 Part 2: ML fit to only the t data

In this part, the maximum likelihood fitting is done by only using the decay time distribution data. This require the decay angle, θ in Equation 1 and 2 to be fixed by iminuit when the minimisation is performed.

2.2.2 Part 3: 2D ML fit to the full t and θ data

Since both decay time and angle distribution are used in this case, there is no need to fix any parameter when the minimisation in process.

2.3 Part 4: The proper errors

The best fit value of the two lifetime parameters, τ_1 and τ_2 are strongly correlated where the value of one of them will decreases while the other increases to achieve the same $\mathcal{NLL}$. Therefore, the simplistic error calculated in the previous parts cannot be considered as true errors because these simplistic errors can only be accepted if the physics parameters are totally uncorrelated.

In determining the true error on a parameter, the parameter is varied such that the minimisation of all the other parameter is done at every points to ensure that the $\mathcal{NLL}$ can be as low as possible. This process is done until the value of $\mathcal{NLL}$ increases by 0.5. This error is also known as the *proper error* and it can be easily calculated by using iminuit.

3 Result

3.1 Part 1

The normalisation functions N_1 and N_2 are determined as shown in Equation 5 and 6 .

$$N_1(\tau_1) \;=\; \int_0^{2\pi}\int_0^{10} (1+\cos^2\theta) \times \exp(\frac{t}{\tau_1})\, dt\, d\theta \;=\; 9.42435 \tag{5}$$

$$N_2(\tau_2) \;=\; \int_0^{2\pi}\int_0^{10} 3\sin^2\theta \times \exp(\frac{t}{\tau_2})\, dt\, d\theta \;=\; 18.7225 \tag{6}$$

The code that generates a set of 10,000 random events with a distribution of decay time and angle is shown in Appendix A.

By setting the value of physics parameters F, τ_1 and τ_2 to be 0.5, 1.0 and 2.0, respectively, the distribution of generated decay time and angle are shown in Figure 1.

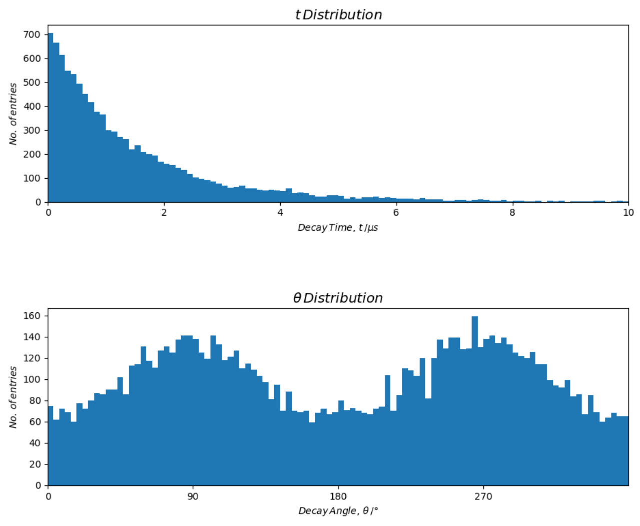
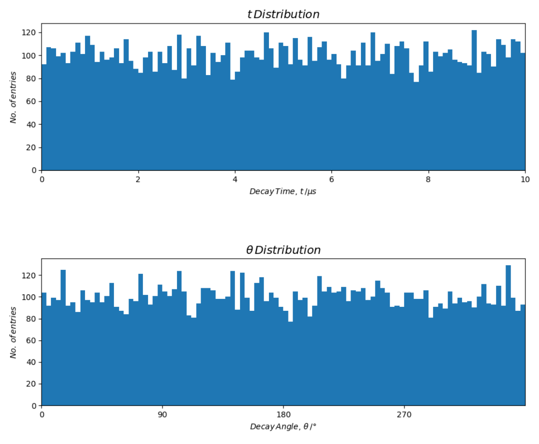


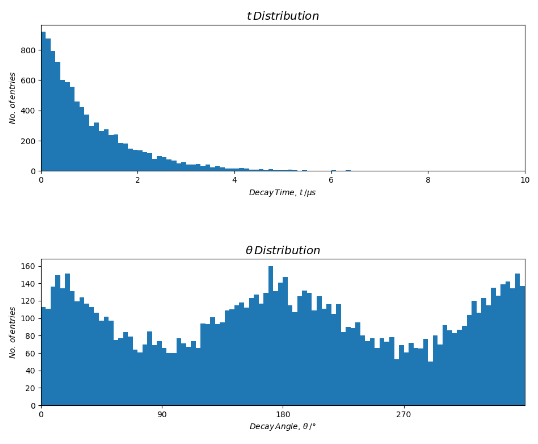
Figure 1: The decay time and angle distributions of the generated data for Fraction = 0.5

In investigating the characteristic of the decay components, the code is reran with the same value of lifetimes but with two different value of Fraction which are 0.0 and 1.0. The histograms of their decay time and angle distributions are shown in Figure 2.

Figure 2 shows that the decay components are not able to be differentiated when the total PDF of the decay are entirely depends on the second PDF (F = 0) and vice versa. This implies that a fit is much easier to distinguish the decay components as the total PDF of the decay depends more on the first PDF.



(a) Fraction = 0.0



(b) Fraction = 1.0

Figure 2: The decay time and angle distribution of the generated data with different fraction

3.2 Part 2

The code that performs the maximum likelihood fitting to the decay time distributions of the given datafile is shown in Appendix B. The best fit value of the physics parameter F, τ_1 and τ_2 together with their simplistic errors for this part are shown in Table 1

Parameter	Estimated Best value	Simplistic Error
F	1.0000	0.0000
τ_1	1.9275	0.2000
τ_2	2.0000	0.2000

Table 1: Minimisation results for Part 2

The plots of $\mathcal{NLL}$ distributions around the best fit value of the physics parameters for this part are shown in Figure 3. These plots exhibit an unstable behaviour due to incomplete data that was given to the minimiser to minimise the $\mathcal{NLL}$.

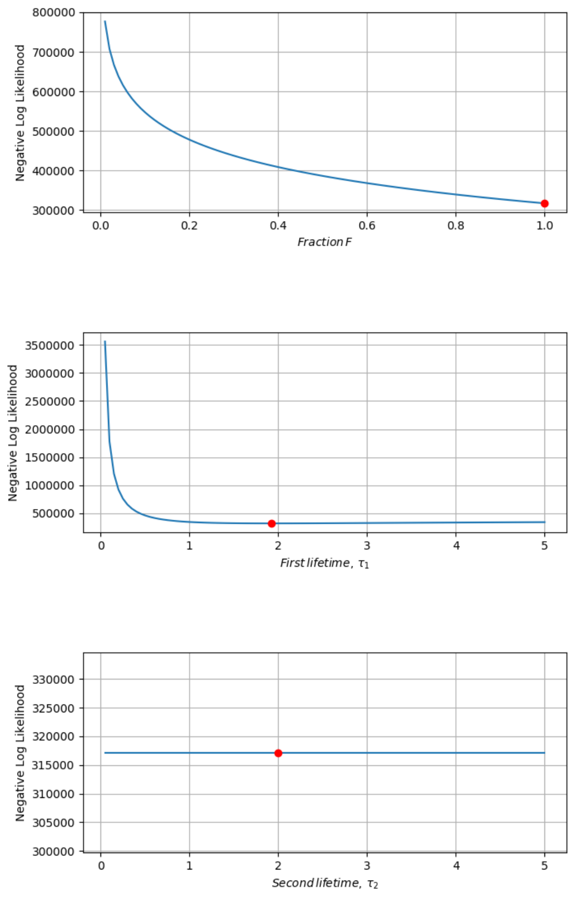


Figure 3: The $\mathcal{NLL}$ distributions around the best fit value of the physics parameters for Part 2.

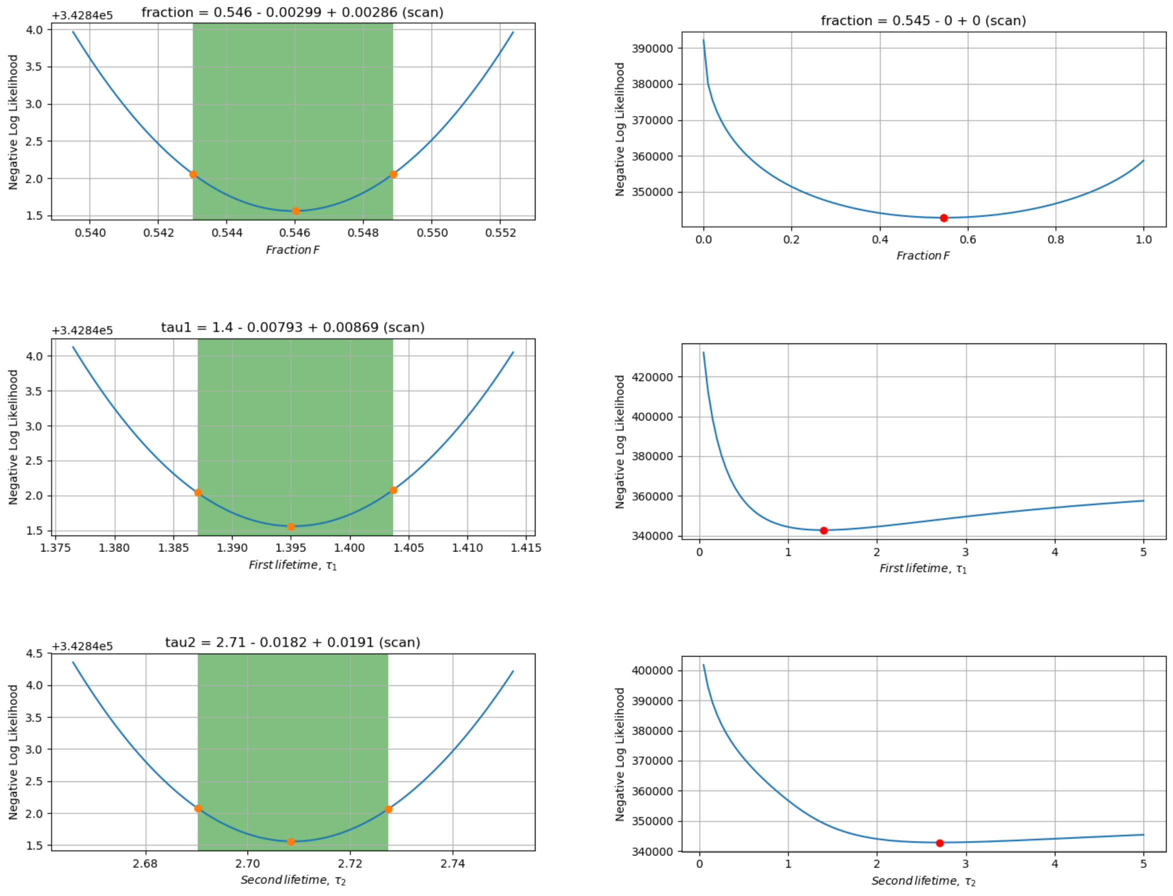
3.3 Part 3

The script that carry out the maximum likelihood fitting to both decay time and angle distributions of the given datafile is shown in Appendix C. The best fit value of the physics parameter F, τ_1 and τ_2 together with their simplistic errors for this part are shown in Table 2.

Parameter	Estimated Best value	Simplistic Error
F	0.5459	0.0437
τ_1	1.3952	0.0884
τ_2	2.7088	0.0167

Table 2: Minimisation results for Part 3

The plots of $\mathcal{NLL}$ distributions around the best fit value of the physics parameters for this part are shown in Figure 4. These plots are much better behaved compared to the previous part because the full decay time ang angle distribution of the given data are fed to the minimiser in minimising $\mathcal{NLL}$.



(a) Around the estimated value

(b) Larger scale of the distribution

Figure 4: The $\mathcal{NLL}$ distributions around the best fit value of the physics parameters for Part 3.

Analysis of Table 1 and 2 tells that the estimation of the best fit values of physics parameter F, τ_1 and τ_2 are better when the maximum likelihood fitting is performed to the full decay time and angle distributions of the given data. Performing a maximum likelihood to only decay time distribution did not give a good estimation of best value of the physics parameter because both of our decay components does not only depend on decay time distribution but both decay time and angle distributions of particle events.

3.4 Part 4

The proper errors for the best fit value of physics parameter F, τ_1 and τ_2 for both Part 2 and 3 are self-calculated and compared with the those from iminuit. These results are shown in Table 3 and 4.

Parameter	Estimated Best value	Proper Error(Minuit)	Proper Error(self)
F	1.0000	0.0000	0.0000
τ_1	1.9275	0.0000	0.0000
τ_2	2.0000	0.0000	0.0000

Table 3: Calculation of proper errors for result in Part 2

Parameter	Estimated Best value	Proper Error(Minuit)	Proper Error(self)
F	0.5459	0.0032	0.0032
τ_1	1.3952	0.0095	0.0093
τ_2	2.7088	0.0217	0.0215

Table 4: Calculation of proper errors for result in Part 3

The difference between the self-calculated proper error and those from iminuit decreases as the increment of the value parameter that was varied in finding the proper error becomes smaller. However, the time taken to calculate these proper errors increases when the increment size is reduced as more re-minimisation of all the other parameters is done in the process.

Both proper and simplistic errors for each of the best fit values of the physics parameters are compared between each other. As a result, it is noticed that most of the proper errors are smaller than the simplistic one in both Part 2 and 3. This is because the method used in calculating the proper error of the parameter is more accurate compared to that of simplistic error. Most of the proper errors of these best fit values of the physics parameters are smaller than those simplistic error of their corresponding physics parameter.

4 Conclusions

It is concluded that a fit is much more easier to differentiate the components of the decay when the total PDF of the decay depends more on the PDF of the first component. Also, the estimation of best fit value of physics parameters F, τ_1 and τ_2 will only give a good and better-behaved result if the maximum likelihood fitting is performed to a full decay time and angle distributions of the data. In this case, the estimated best fit value of these physics parameters F, τ_1 and τ_2 with their proper errors are determined to be 0.5459(32), 1.3952(93)μs and 2.7088(215)μs, respectively.

Appendix A   Part 1 Guide

A.1   Instruction

The command to run this code is

```
python Part1.py
```

A.2   Code (Part1.py)

```
"""
MyPDF, a class for generating random events with decay time and angle distributions according to the PDF
     described in the report.

Authors: Azid Harun

Date : 25/11/2018
"""

import math
import numpy
import scipy.integrate as integrate
import matplotlib.pyplot as plt
import sys

class PDFError(Exception):
    """ An exception class for MyPDF """
    pass

#===============================================
# 2D pdf
class MyPDF:
    """
    Class for generating random events with decay time and angle distributions according to the PDF described
         in the report.

    Properties:
    lifetime1(float)     -  particle first lifetime
    lifetime2(float)     -  particle second lifetime

    t_lolimit(float)     -  lower limit of interval for decay time
    t_hilimit(float)     -  higher limit of interval for decay time

    theta_lolimit(float) -  lower limit of interval for decay angle
    theta_hilimit(float) -  higher limit of interval for decay angle

    shape1(function)     - PDF of first decay component, PDF1
    shape2(function)     - PDF of second decay component, PDF2

    max1(float)          - maximum value of PDF1
    max2(float)          - maximum value of PDF2

    fraction(float)      - fraction of PDF1 being the total PDF of the decay

    Methods:
    * maxVal             - return the maximum value of the total PDF of the decay
    * normalise          - normalise the given pdf
    * evaluate           - evaluate the normalised pdf at give decay time and angle
    * next               - draw N random number from distribution
    * drawSample         - draw a random sample of N events from a pdf using box method
    * plotShape          - plot histograms of the decay time and angle distributions of the generated data
    * writeData          - write out decay times and decay angles generated
    """

     # Constructor
    def __init__(self, t_lolim, t_hilim, theta_lolim, theta_hilim, lifetime1, lifetime2, fraction):
        self.lifetime1 = lifetime1
        self.lifetime2 = lifetime2
        self.t_lolimit = t_lolim
        self.t_hilimit = t_hilim
        self.theta_lolimit = theta_lolim
        self.theta_hilimit = theta_hilim
        self.shape1 = lambda t, theta: (1+math.cos(theta)**2)*(math.exp(-t/lifetime1))
        self.shape2 = lambda t, theta: (3*math.sin(theta)**2)*(math.exp(-t/lifetime2))
        self.max1 = self.shape1(t_lolim, theta_lolim)
        self.max2 = self.shape2(t_lolim, theta_lolim)
        self.fraction = fraction

    # Return the maximum value of the total PDF of the decay
    def maxVal( self ) :
        return self.fraction * self.max1 + (1-self.fraction) * self.max2

    def normalise( self, pdf ) :
        if pdf == 1:
            return integrate.dblquad( self.shape1, self.theta_lolimit, self.theta_hilimit, lambda theta:
                 self.t_lolimit, lambda theta: self.t_hilimit)[0]
        elif pdf == 2:
            return integrate.dblquad( self.shape2, self.theta_lolimit, self.theta_hilimit, lambda theta:
                 self.t_lolimit, lambda theta: self.t_hilimit)[0]
        else:
            raise PDFError('Invalid PDF')

    # Evaluate method (normalised)
    def evaluate( self, t, theta, norm1, norm2, pdf_type):
        pdf1 = self.fraction * (self.shape1(t, theta) / norm1)
        pdf2 = (1-self.fraction) * (self.shape2(t, theta) / norm2)
        if pdf_type == 'all':
            return pdf1 + pdf2
        elif pdf_type == '1':
            return pdf1
        elif pdf_type == '2':
            return pdf2
        else:
            raise PDFError('Invalid PDF type')

    # Draw N random number from distribution
    def next(self, nevents):
        data  = self.drawSample(self, self.t_lolimit, self.t_hilimit, self.theta_lolimit, self.theta_hilimit,
             nevents)
        return data

    @staticmethod
    # To draw a random sample of N events from a pdf using box method
    def drawSample(self, t_lolim, t_hilim, theta_lolim, theta_hilim, nevents):
        times = []
        thetas = []
        for i in range(nevents):
            ythrow = 1.
            yval=0.
            norm1, norm2 = self.normalise(1), self.normalise(2)
            while ythrow > yval:
                tthrow = numpy.random.uniform(t_lolim, t_hilim)
                thetathrow = numpy.random.uniform(theta_lolim, theta_hilim)
                ythrow = self.maxVal() * numpy.random.uniform()
                yval = self.evaluate(tthrow, thetathrow, norm1, norm2, 'all')
            times.append(tthrow)
            thetas.append(thetathrow)
        return (times, thetas)
    @staticmethod
    # function to plot histograms of the decay time and angle distributions
    def plotShape(data, t_lolim, t_hilim, theta_lolim, theta_hilim, nbins ):
        plt.subplot(2, 1, 1)
        plt.hist(data[0], bins=nbins, range=[t_lolim, t_hilim])
        plt.xlim(0, 10)
        plt.ylabel(r'$No.\/of\/entries$')
        plt.xlabel(r'$Decay\/Time,\/t\//\mu s$')
        plt.title(r'$t\/Distribution$', fontsize='x-large')

        plt.subplot(2, 1, 2)
        data_deg = data[1]
        data_deg = [ i/math.pi*180 for i in data_deg ]
        plt.hist(data_deg, bins=nbins, range=[theta_lolim/math.pi*180, theta_hilim/math.pi*180])
        plt.xlim(0, 360)
        plt.xticks(numpy.arange(0, 360, 90))
        plt.ylabel(r'$No.\/of\/entries$')
        plt.xlabel(r'$Decay\/Angle,\/\theta\//\degree$')
        plt.title(r'$\theta\/Distribution$', fontsize='x-large')

        # Display the subplots
        plt.subplots_adjust(hspace=0.6)
        plt.show()

    @staticmethod
    # function to write out decay times and decay angle generated
    def writeData(data, filename):
        with open(filename, 'a') as f:
            for time, angle in zip(data[0], data[1]):
                f.write('{0:0.16f} {1:0.16f}\n'.format(time, angle))

#================================================
# Main code to generate and plot a single experiment

def singleToy( nevents):

    t_lolim, t_hilim         = 0., 10.
    theta_lolim, theta_hilim = 0, 2 * math.pi
    lifetime1, lifetime2     = 1.0, 2.0
    fraction                 = 1.0      # Choose 0.0 / 0.5 / 1.0

    # Create the pdf
    pdf = MyPDF( t_lolim, t_hilim, theta_lolim, theta_hilim, lifetime1, lifetime2, fraction)
    norm1, norm2 = pdf.normalise(1), pdf.normalise(2)

    # Generate a single experiment
    data = pdf.next( nevents)

    # Write decay data in an output textfile
    # pdf.writeData(data, sys.argv[1])

    # Plot function and data
    pdf.plotShape( data, t_lolim, t_hilim, theta_lolim, theta_hilim, 100 )

#================================================
#Main

def main():
    #Perform a single toy
    singleToy(10000)

main()
```

Appendix B Part 2 Guide

B.1 Instruction

The command to run this code is

```
python -W ignore MainPart2.py datafile-Xdecay.txt
```

B.2 Class code (MinuitPart2.py)

```
"""
MinuitPart2, a class for for minimising the negative log likelihood (NLL) to find the best value of physics
     parameter
            F, first and second particle lifetime with only decay times distribution of the given datafile.

Authors: Azid Harun

Date : 25/11/2018

"""

# Import required packages
import numpy as np
import iminuit as im
from iminuit import Minuit

class MinuitError(Exception):
    """ An exception class for Minuit """
    pass


class Minuit(object):
    """
    Class for minimising the NLL with fixed decay angle parameter, theta.

    Properties:
    threshold(float)           -   the minimising threshold value
    fraction_bnd(float, tuple) -   fraction bound
    tau1_bnd(float, tuple)     -   tau1 bound
    tau1_bnd(float, tuple)     -   tau1 bound
    error_size(float)          -   the error of the calculated parameter is 1 unit if the function given
         increases by this value from its minimum point

    Methods:
    * minimise                 -    minimise the function
    * fix0minimise             -    minimise the function with fixed fraction
    * fix1minimise             -    minimise the function with fixed tau1
    * fix2minimise             -    minimise the function with fixed tau2
    * isFinished               -    control the minimiser
    * isExceeded               -    control the proper error finding process
    * errorFinder              -    find the parameter error
    * properErrorFinder        -    calculate the proper error of the parameter
    * simpleErrorFinder        -    calculate the simplistic error of the parameter
    * readData                 -    read the input decay time and angle distributions (t and theta)
    """

#========================================INITIALISER========================================

    def __init__(self, threshold, fraction_range, tau1_range, tau2_range, fn_type):
        self.threshold = threshold
        self.fraction_bnd = fraction_range
        self.tau1_bnd = tau1_range
        self.tau2_bnd = tau2_range
        if fn_type == 'nll':
            self.error_size = 0.5
        elif fn_type == 'chi':
            self.error_size = 1.0
        else:
            raise MinuitError("Invalid function type!")

#========================================MINIMISER========================================

    def minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        theta = 0.0,
                        fix_theta = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        limit_theta = (0.0, 2*np.pi),
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

    def fix0minimise(self, f, x):
        m = im.Minuit( f,
                        fraction=x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        theta = 0.0,
                        fix_theta = True,
                        fix_fraction = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        limit_theta = (0.0, 2*np.pi),
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

    def fix1minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        theta = 0.0,
                        fix_theta = True,
                        fix_tau1 = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        limit_theta = (0.0, 2*np.pi),
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

    def fix2minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        theta = 0.0,
                        fix_theta = True,
                        fix_tau2 = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        limit_theta = (0.0, 2*np.pi),
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m
#==================================MINIMISER PROCESS CONTROL================================

    def isFinished(self,diff):
        if diff == None:
            pass

        elif diff > self.threshold:
            pass

        else:
            finish = 'Minimisation is finished'
            return finish

    def isExceeded(self,diff):
        if diff == None:
            pass

        elif diff < self.threshold:
            pass

        else:
            finish = 'Minimisation is finished'
            return finish

#==================================PARAMETER ERROR CALCULATOR================================

    def properErrorFinder(self, nll, idx, F_tau1_tau2, theyta):
        ini_nll = nll(F_tau1_tau2[0], F_tau1_tau2[1], F_tau1_tau2[2], theyta)
        best = F_tau1_tau2[idx]
        diff = None
        increment = 0
        while not self.isExceeded(diff):
            increment -= 1
            delta = 0.00001 * increment
            F_tau1_tau2[idx] += delta
            # Calculate previous and next NLL value and also their difference.
            if idx == 0:
                m = self.fix0minimise(nll, F_tau1_tau2)
            elif idx == 1:
                m = self.fix1minimise(nll, F_tau1_tau2)
            elif idx == 2:
                m = self.fix2minimise(nll, F_tau1_tau2)
            F_tau1_tau2 = np.array([m.values['fraction'], m.values['tau1'], m.values['tau2']])
            final_nll = m.fval
            diff = np.abs(ini_nll - final_nll)
        return np.abs(m.values[idx] - best)

    @staticmethod
    def simpleErrorFinder(level, f_list, f_min, param, param_list):
        f_errline = min(f_list, key=lambda x:abs(x-(f_min+level)))
        index = f_list.index(f_errline)
        param_error = np.abs(param - param_list[index])
        return param_error

    @staticmethod
    def readData(filename):
        with open(filename, 'r') as f:
            t_list = []
            theta_list = []
            for line in f:
                t, theta = map(float, line.split())
                t_list.append(t)
                theta_list.append(theta)
        return np.array(t_list), np.array(theta_list)
```

B.3 Main code (MainPart2.py)

```
"""
Negative Log Likelihood(NLL) Minimisation, a python script for finding the best estimation
of fraction, first and second lifetime by minimising NLL.

Authors: Azid Harun

Date : 19/10/2018
"""

# Import required packages
import numpy as np
import pylab as pl
import sys
from scipy import *
from MinuitPart2 import Minuit
import scipy.integrate as integrate

# Define Negative Log Likelihood function
def nll(fraction, tau1, tau2, theta):
    shape1 = lambda t, theta: (1+math.cos(theta)**2)*(math.exp(-t/tau1))
    shape2 = lambda t, theta: (3*math.sin(theta)**2)*(math.exp(-t/tau2))
    norm1 = integrate.dblquad( shape1, 0.0, 2*np.pi, lambda theta: 0.0, lambda theta: 10.0)[0]
    norm2 = integrate.dblquad( shape2, 0.0, 2*np.pi, lambda theta: 0.0, lambda theta: 10.0)[0]
    pdf1 = fraction*(1+np.cos(theta)**2)*(np.exp(-t/tau1))/norm1
    pdf2 = (1-fraction)*(3*np.sin(theta)**2)*(np.exp(-t/tau2))/norm2
    pdf = pdf1 + pdf2
    return np.sum(-np.log(pdf))

# Create list to store data
nll_list = []

# Read data from input file
data = Minuit.readData(sys.argv[1])
t = data[0]
```

```
# Define initial straight line parameters, m and c and their range
F_tau1_tau2 = np.array([0.5, 1.0, 2.0])
theyta = 0.0
F_range = (0.0, 1)
tau1_range = (0.0, 5.0)
tau2_range = (0.0, 5.0)
fn_type = 'nll'

# Create a minimiser class
minim = Minuit(0.0, F_range, tau1_range, tau2_range, fn_type)

#====================================MINIMISING PROCESS=====================================

# Loop the process until difference between previous and next NLL value lower than threshold
diff = None
ini_nll = nll(F_tau1_tau2[0], F_tau1_tau2[1], F_tau1_tau2[2], theyta)

while not minim.isFinished(diff):
    # Calculate previous and next NLL value and also their difference.
    m = minim.minimise(nll, F_tau1_tau2)
    F_tau1_tau2 = np.array([m.values['fraction'], m.values['tau1'], m.values['tau2']])
    final_nll = m.fval
    diff = np.abs(ini_nll - final_nll)
    ini_nll = final_nll


#============================CREATING DATA FOR CALC SIMPLISTIC ERROR============================

# Creating data around minimum chi-squared
F_arr = np.arange(0, 2*F_tau1_tau2[0], 2*F_tau1_tau2[0]/200)
tau1_arr = np.arange(0.2, 2*F_tau1_tau2[1]+0.2, 2*F_tau1_tau2[1]/200)
tau2_arr = np.arange(0.2, 2*F_tau1_tau2[2]+0.2, 2*F_tau1_tau2[2]/200)

# F_arr, tau1_arr, tau2_arr = np.delete(F_arr, 0), np.delete(tau1_arr, 0), np.delete(tau2_arr, 0)

for F, tau1, tau2 in zip(F_arr, tau1_arr, tau2_arr):
    nllval = nll(F, tau1, tau2, theyta)
    nll_list.append(nllval)

# Calculate simplistic error for parameters
F_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[0], F_arr)
tau1_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[1], tau1_arr)
tau2_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[2], tau2_arr)

#===============================CREATING DATA FOR CALC PROPER ERROR============================

proper = Minuit(0.5, F_range, tau1_range, tau2_range, fn_type)

F_perror = proper.properErrorFinder(nll, 0, F_tau1_tau2, theyta)
tau1_perror = proper.properErrorFinder(nll, 1, F_tau1_tau2, theyta)
tau2_perror = proper.properErrorFinder(nll, 2, F_tau1_tau2, theyta)

# ================================GENERATE AND DISPLAY RESULTS================================

# Display the result
print('============================================================================')
print('Number of Particle Decay Event          :  {}'.format(len(t)))
print('----------------------------------------------------------------------------')
print('Best Estimated Fraction                 :  {0:0.4f}'.format(m.values['fraction']))
print('Best Estimated Tau 1                    :  {0:0.4f}'.format(m.values['tau1']))
print('Best Estimated Tau 2                    :  {0:0.4f}'.format(m.values['tau2']))
print('------------------------------SIMPLISTIC ERROR--------------------------------')
print('Simplistic error for F                  :  {0:0.4f}'.format(F_error))
print('Simplistic error for tau1               :  {0:0.4f}'.format(tau1_error))
print('Simplistic error for tau2               :  {0:0.4f}'.format(tau2_error))
print('-------------------------------PROPER ERROR-----------------------------------')
print('MINUIT error for F                      :  {0:0.4f}'.format(m.errors['fraction']))
print('MINUIT error for tau1                  :   {0:0.4f}'.format(m.errors['tau1']))
print('MINUIT error for tau2                  :   {0:0.4f}\n'.format(m.errors['tau2']))
print('Calculated error for F                 :   {0:0.4f}'.format(F_perror))
print('Calculated error for tau1              :   {0:0.4f}'.format(tau1_perror))
print('Calculated error for tau2              :   {0:0.4f}'.format(tau2_perror))
print('--------------------------------------------------------------------------------')

#==========================================PLOTTING DATA=====================================

#Plot the result
while True:
    type = (input('Around minimum point? (Y/N)'))

    if type == 'Y':
        pl.subplot(3, 1, 1)
        m.draw_profile('fraction')
        pl.xlabel(r'$Fraction\/F\/$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 2)
        m.draw_profile('tau1')
        pl.xlabel(r'$First\/lifetime,\/\tau_{1}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 3)
        m.draw_profile('tau2')
        pl.xlabel(r'$Second\/lifetime,\/\tau_{2}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplots_adjust(hspace=0.6)
        pl.show()
        break

    elif type == 'N':
        pl.subplot(3, 1, 1)
        m.draw_profile('fraction', bound=(0,1))
        pl.plot(m.values['fraction'], m.fval, 'ro')
        pl.xlabel(r'$Fraction\/F\/$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 2)
        m.draw_profile('tau1', bound=(0,5))
        pl.plot(m.values['tau1'], m.fval, 'ro')
        pl.xlabel(r'$First\/lifetime,\/\tau_{1}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 3)
        m.draw_profile('tau2', bound=(0,5))
        pl.plot(m.values['tau2'], m.fval, 'ro')
        pl.xlabel(r'$Second\/lifetime,\/\tau_{2}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplots_adjust(hspace=0.6)
        pl.show()
        break

    else:
        pass
```

Appendix C Part 3 Guide

C.1 Instruction

The command to run this code is

```
python -W ignore MainPart3.py datafile-Xdecay.txt
```

C.2 Class code (MinuitPart3.py)

```
"""
MinuitPart3, a class for for minimising the negative log likelihood (NLL) to find the best value of physics
     parameter
          F, first and second particle lifetime with both decay times and angle distribution of the given
               datafile.

Authors: Azid Harun

Date : 25/11/2018
"""

# Import required packages
import numpy as np
import iminuit as im
from iminuit import Minuit

class MinuitError(Exception):
    """ An exception class for Minuit """
    pass


class Minuit(object):
    """
    Class for minimising the NLL.

    Properties:
    threshold(float)           -  the minimising threshold value
    fraction_bnd(float, tuple) -  fraction bound
    tau1_bnd(float, tuple)     -  tau1 bound
    tau1_bnd(float, tuple)     -  tau1 bound
    error_size(float)          -  the error of the calculated parameter is 1 unit if the function given
         increases by this value

    Methods:
    * minimise                 -   minimise the function
    * fix0minimise             -   minimise the function with fixed fraction
    * fix1minimise             -   minimise the function with fixed tau1
    * fix2minimise             -   minimise the function with fixed tau2
    * isFinished               -   control the minimiser
    * isExceeded               -   control the proper error finding process
    * errorFinder              -   find the parameter error
    * properErrorFinder        -   calculate the proper error of the parameter
    * simpleErrorFinder        -   calculate the simplistic error of the parameter
    * readData                 -   read the input decay time and angle distributions (t and theta)
    """

#========================================INITIALISER========================================

    def __init__(self, threshold, fraction_range, tau1_range, tau2_range, fn_type):
        self.threshold = threshold
        self.fraction_bnd = fraction_range
        self.tau1_bnd = tau1_range
        self.tau2_bnd = tau2_range
        if fn_type == 'nll':
            self.error_size = 0.5
        elif fn_type == 'chi':
            self.error_size = 1.0
        else:
            raise MinuitError("Invalid function type!")

#========================================MINIMISER========================================

    def minimise(self, f, x):
        m = im.Minuit( f,
                        fraction=x[0],
                        tau1=x[1],
                        tau2=x[2],
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

    def fix0minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        fix_fraction = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

    def fix1minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        fix_tau1 = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m
    def fix2minimise(self, f, x):
        m = im.Minuit( f,
                        fraction = x[0],
                        tau1 = x[1],
                        tau2 = x[2],
                        fix_tau2 = True,
                        limit_fraction = self.fraction_bnd,
                        limit_tau1 = self.tau1_bnd,
                        limit_tau2 = self.tau2_bnd,
                        error_fraction = self.error_size,
                        error_tau1 = self.error_size,
                        error_tau2 = self.error_size,
                        errordef = self.error_size,
                        print_level = 0,
                        pedantic = False
                        )
        m.migrad()
        return m

#==================================MINIMISER PROCESS CONTROL================================

    def isFinished(self,diff):
        if diff == None:
            pass

        elif diff > self.threshold:
            pass

        else:
            finish = 'Minimisation is finished'
            return finish

    def isExceeded(self,diff):
        if diff == None:
            pass

        elif diff < self.threshold:
            pass

        else:
            finish = 'Minimisation is finished'
            return finish

#==================================PARAMETER ERROR CALCULATOR================================

    def properErrorFinder(self, nll, idx, F_tau1_tau2):
        ini_nll = nll(F_tau1_tau2[0], F_tau1_tau2[1], F_tau1_tau2[2])
        best = F_tau1_tau2[idx]
        diff = None
        increment = 0
        while not self.isExceeded(diff):
            increment += 1
            delta = 0.000001 * increment
            F_tau1_tau2[idx] += delta
            # Calculate previous and next NLL value and also their difference.
            if idx == 0:
                m = self.fix0minimise(nll, F_tau1_tau2)
            elif idx == 1:
                m = self.fix1minimise(nll, F_tau1_tau2)
            elif idx == 2:
                m = self.fix2minimise(nll, F_tau1_tau2)
            F_tau1_tau2 = np.array([m.values['fraction'], m.values['tau1'], m.values['tau2']])
            final_nll = m.fval
            diff = np.abs(ini_nll - final_nll)
        return np.abs(m.values[idx] - best)

    @staticmethod
    def simpleErrorFinder(level, f_list, f_min, param, param_list):
        f_errline = min(f_list, key=lambda x:abs(x-(f_min+level)))
        index = f_list.index(f_errline)
        param_error = np.abs(param - param_list[index])
        return param_error

    @staticmethod
    def readData(filename):
        with open(filename, 'r') as f:
            t_list = []
            theta_list = []
            for line in f:
                t, theta = map(float, line.split())
                t_list.append(t)
                theta_list.append(theta)
        return np.array(t_list), np.array(theta_list)
```

C.3 Main code (MainPart3.py)

```
"""
Negative Log Likelihood(NLL) Minimisation, a python script for finding the best estimation
of fraction, first and second lifetime by minimising NLL.

Authors: Azid Harun

Date : 19/10/2018
"""

# Import required packages
import numpy as np
import pylab as pl
import sys
from scipy import *
from MinuitPart3 import Minuit
import scipy.integrate as integrate

# Define Negative Log Likelihood function
def nll(fraction, tau1, tau2):
    shape1 = lambda t, theta: (1+math.cos(theta)**2)*(math.exp(-t/tau1))
    shape2 = lambda t, theta: (3*math.sin(theta)**2)*(math.exp(-t/tau2))
    norm1 = integrate.dblquad( shape1, 0.0, 2*np.pi, lambda theta: 0.0, lambda theta: 10.0)[0]
    norm2 = integrate.dblquad( shape2, 0.0, 2*np.pi, lambda theta: 0.0, lambda theta: 10.0)[0]
    pdf1 = fraction*(1+np.cos(theta)**2)*(np.exp(-t/tau1))/norm1
    pdf2 = (1-fraction)*(3*np.sin(theta)**2)*(np.exp(-t/tau2))/norm2
    pdf = pdf1 + pdf2
    return np.sum(-np.log(pdf))

# Create list to store data
nll_list = []

# Read data from input file
t, theta = Minuit.readData(sys.argv[1])

# Define initial straight line parameters, m and c and their range
F_tau1_tau2 = np.array([0.5, 1.0, 2.0])
F_range = (0.0, 1)
tau1_range = (0.0, 5.0)
tau2_range = (0.0, 5.0)
fn_type = 'nll'

# Create a minimiser class
minim = Minuit(0.0, F_range, tau1_range, tau2_range, fn_type)

#====================================MINIMISING PROCESS=====================================
# Loop the process until difference between previous and next NLL value lower than threshold
diff = None
ini_nll = nll(F_tau1_tau2[0], F_tau1_tau2[1], F_tau1_tau2[2])

while not minim.isFinished(diff):
    # Calculate previous and next NLL value and also their difference.
    m = minim.minimise(nll, F_tau1_tau2)
    F_tau1_tau2 = np.array([m.values['fraction'], m.values['tau1'], m.values['tau2']])
    final_nll = m.fval
    diff = np.abs(ini_nll - final_nll)
    ini_nll = final_nll

#===========================CREATING DATA FOR CALC SIMPLISTIC ERROR============================

# Creating data around minimum chi-squared
F_arr = np.arange(0, 2*F_tau1_tau2[0], 2*F_tau1_tau2[0]/200)
tau1_arr = np.arange(0.2, 2*F_tau1_tau2[1]+0.2, 2*F_tau1_tau2[1]/200)
tau2_arr = np.arange(0.2, 2*F_tau1_tau2[2]+0.2, 2*F_tau1_tau2[2]/200)

for F, tau1, tau2 in zip(F_arr, tau1_arr, tau2_arr):
    nllval = nll(F, tau1, tau2)
    nll_list.append(nllval)

# Calculate simplistic error for parameters
F_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[0], F_arr)
tau1_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[1], tau1_arr)
tau2_error = Minuit.simpleErrorFinder(0.5, nll_list, final_nll, F_tau1_tau2[2], tau2_arr)

#==============================CREATING DATA FOR CALC PROPER ERROR============================

proper = Minuit(0.5, F_range, tau1_range, tau2_range, fn_type)

F_perror = proper.properErrorFinder(nll, 0, F_tau1_tau2)
tau1_perror = proper.properErrorFinder(nll, 1, F_tau1_tau2)
tau2_perror = proper.properErrorFinder(nll, 2, F_tau1_tau2)

# ================================GENERATE AND DISPLAY RESULTS================================

# Display the result
print('================================================================================')
print('Number of Muon Decay Event              :   {}'.format(len(t)))
print('--------------------------------------------------------------------------------')
print('Best Estimated Fraction                 :   {0:0.4f}'.format(m.values['fraction']))
print('Best Estimated Tau 1                    :   {0:0.4f}'.format(m.values['tau1']))
print('Best Estimated Tau 2                    :   {0:0.4f}'.format(m.values['tau2']))
print('------------------------------SIMPLISTIC ERROR----------------------------------')
print('Simplistic error for F                  :   {0:0.4f}'.format(F_error))
print('Simplistic error for tau1               :   {0:0.4f}'.format(tau1_error))
print('Simplistic error for tau2               :   {0:0.4f}'.format(tau2_error))
print('-------------------------------PROPER ERROR-------------------------------------')
print('MINUIT error for F                      :   {0:0.4f}'.format(m.errors['fraction']))
print('MINUIT error for tau1                   :   {0:0.4f}'.format(m.errors['tau1']))
print('MINUIT error for tau2                   :   {0:0.4f}\n'.format(m.errors['tau2']))
print('Calculated error for F                  :   {0:0.4f}'.format(F_perror))
print('Calculated error for tau1               :   {0:0.4f}'.format(tau1_perror))
print('Calculated error for tau2               :   {0:0.4f}'.format(tau2_perror))
print('--------------------------------------------------------------------------------')

#=========================================PLOTTING DATA======================================

#Plot the result
while True:
    type = (input('Around minimum point? (Y/N)'))

    if type == 'Y':
        pl.subplot(3, 1, 1)
        m.draw_profile('fraction')
        pl.xlabel(r'$Fraction\/F\/$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 2)
        m.draw_profile('tau1')
        pl.xlabel(r'$First\/lifetime,\/\tau_{1}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 3)
        m.draw_profile('tau2')
        pl.xlabel(r'$Second\/lifetime,\/\tau_{2}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplots_adjust(hspace=0.6)
        pl.show()
        break

    elif type == 'N':
        pl.subplot(3, 1, 1)
        m.draw_profile('fraction', bound=(0,1))
        pl.plot(m.values['fraction'], m.fval, 'ro')
        pl.xlabel(r'$Fraction\/F\/$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 2)
        m.draw_profile('tau1', bound=(0,5))
        pl.plot(m.values['tau1'], m.fval, 'ro')
        pl.xlabel(r'$First\/lifetime,\/\tau_{1}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplot(3, 1, 3)
        m.draw_profile('tau2', bound=(0,5))
        pl.plot(m.values['tau2'], m.fval, 'ro')
        pl.xlabel(r'$Second\/lifetime,\/\tau_{2}$')
        pl.ylabel('Negative Log Likelihood')

        pl.subplots_adjust(hspace=0.6)
        pl.show()
        break

    else:
        pass
```