A comparison of well and badly behaved minima in parameter estimation

Sean O'Brien

Abstract

This is a meta-analysis of parameter finding techniques. Specifically maximum likelyhood fitting, which finds parameters of a PDF by minimising the NLL function. This technique has very broad application in scientific data analysis and its reliability and pitfalls should be understood.

In particular, the situation of mismatched PDF complexity to data content was examined. The behaviour and stability of fit minima was tested when full and partial data was used to estimate the parameters of a relatively complex PDF. The findings showed that when only the partial data was used the parameter estimation values differed significantly from when the full data was used and that the minima from the partial data were unstable i.e. the results depended on the minimisation initial guess. When the full data was used and the PDF complexity was matched by the data, the fit was stable and gave repeatable results.

Introduction

The data consisted of 100,000 points, each consisting of a t and θ measurement, the significance of which is irrelevant. The 3 parameter PDF (Probability Density Function) $P(t, \theta; F, \tau_1, \tau_2)$ used to model the data was a linear mixing of 2 PDFs $P_1(t, \theta; \tau_1)$ and $P_2(t, \theta; \tau_2)$

$$P_1(t, \theta; \tau_1) = \frac{1}{3\pi\tau_1}(1 + \cos^2\theta)\, e^{-\frac{t}{\tau_1}} \qquad P_2(t, \theta; \tau_2) = \frac{1}{\pi\tau_2}\sin^2\theta\, e^{-\frac{t}{\tau_2}}$$

$$P(t, \theta; F, \tau_1, \tau_2) = FP_1(t, \theta; \tau_1) + (1 - F)P_2(t, \theta; \tau_2)$$

The paramters estimated were F, τ_1, τ_2. This was done by performing a maximum likelyhood fit. Which involves minimising the NLL (Negative Log Likelyhood) related to the parameters in question.

$$\mathrm{NLL}(F, \tau_1, \tau_2) = -\sum_i \log P(t_i, \theta_i; F, \tau_1, \tau_2)$$

- Where each t_i, θ_i were points in the dataset.

The errors on the parameter estimations were also found using the NLL. The points at which the NLL is 0.5 above its minimum value are points that are 1 standard deviation σ from the minima. Justification for this is exculded from the report and exactly how the standard errors were found using this fact is explained further in the simplistic error method and full error method entries in the glossary.

Notes

- Where appropriate, a method or expression will be explained in the glossary and in its first reference will be in This Font
- Function minimisation was performed my Minuit, packaged for python as iminuit `https://iminuit.readthedocs.io/` All other calculations were performed by my own python code, which is included in the appendix of this report.
- The code, data and plots are also available at `https://github.com/mrseanman/NumRecProject`

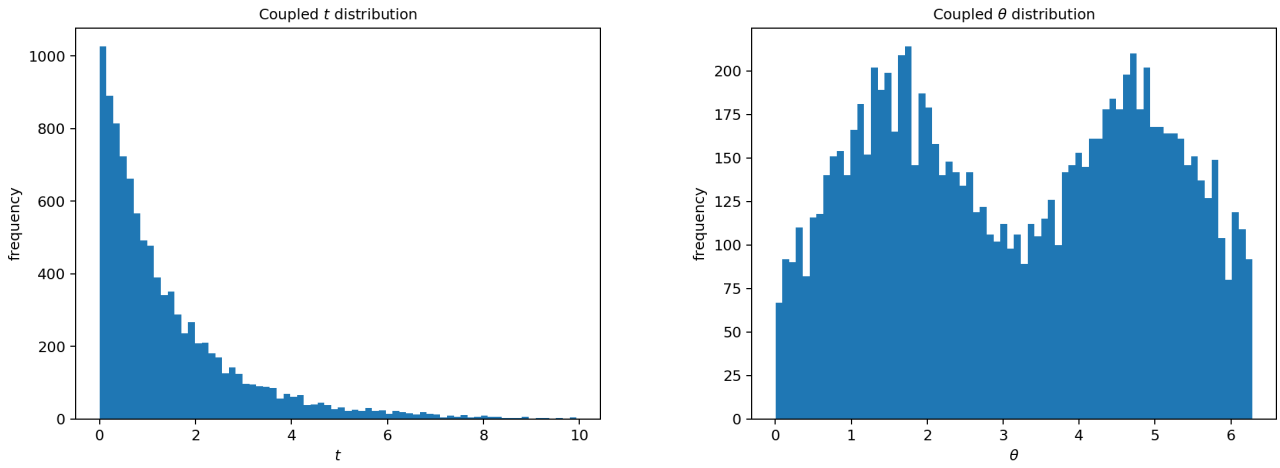

Figure 1: Coupled distributions $F = 0.5, \tau_1 = 1.0, \tau_2 = 2.0$

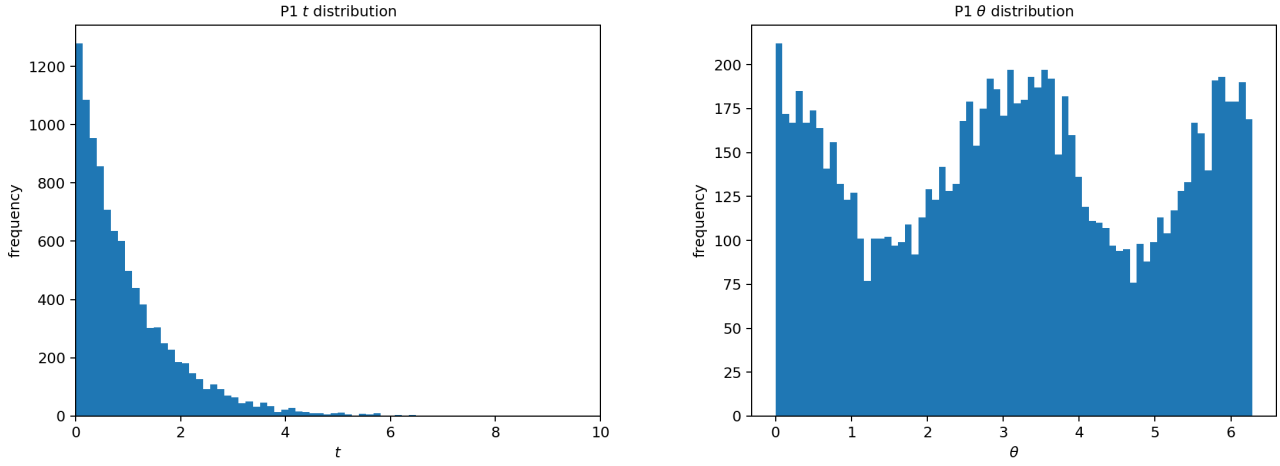

Figure 2: P_1 only distributions $F = 1.0, \tau_1 = 1.0, \tau_2 = 2.0$

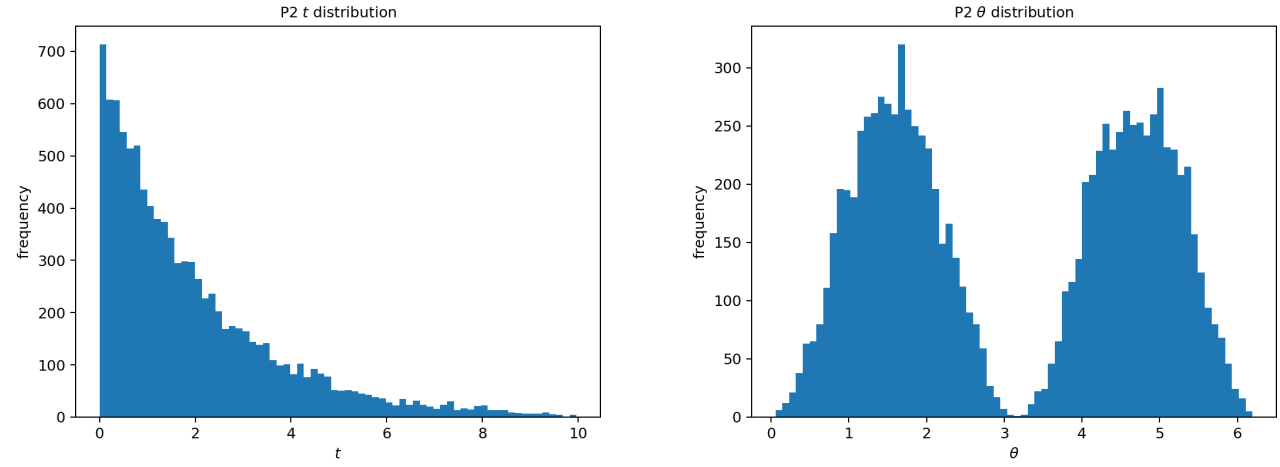

Figure 3: P_2 only distributions $F = 0.0, \tau_1 = 1.0, \tau_2 = 2.0$

1 Investigation of $P(t, \theta)$

10,000 random (t, θ) data points were generated simulating the $P(t, \theta)$ PDF with certain values for F, τ_1, τ_2 The random generator employed the BOX METHOD with the ranges set as $t \in [0, 10]$, $\theta \in [0, 2\pi]$. The finite range of t meant that P_1 and P_2 had to be normalised differently such that. $P_1(t, \theta; \tau_1) = [3\pi\tau_1(1-e^{\frac{-10}{\tau_1}})]^{-1}\,(1+\cos^2\theta)e^{-\frac{t}{\tau_1}}$ and $P_2(t, \theta; \tau_1) = [\pi\tau_2(1-e^{\frac{-10}{\tau_2}})]^{-1}\,\sin^2\theta e^{-\frac{t}{\tau_2}}$ after which $P(t, \theta)$ was defined as normal.
Figures 1-3 show distributions of the simulated data for different values of F.

1.1 Comments on the distributions.

- The t distributions in Figures 1-3 are all of very similar form. What is most significant to this report is that the t distribution of the coupled PDF in Figure 1 where $F = 0.5$ is essentially indistinguishable in form from a simple exponential distribution as in Figures 2 and 3.
- The θ distributions in Figures 1-3 show very different form and importantly, the structure of the θ distribution in Figure 1 shows form that cannot be modeled by P_1 or P_2 alone.

2 Maximum Likelyhood Fit on t Data Only

A maximum likelyhood fit was performed on the (t) data only. To do this the PDF P had to be altered to 'simulate' no θ dependence. This was accomplished by integrating $P(t, \theta)$ over all θ $[0, 2\pi]$ to give us $P_t(t)$. The resulting PDFs were

$$P_{1_t}(t) = \frac{1}{\tau_1}e^{-\frac{t}{\tau_1}} \qquad P_{2_t}(t) = \frac{1}{\tau_2}e^{-\frac{t}{\tau_2}}$$

and similar to before

$$P_t(F, t) = FP_{1_t}(t) + (1-F)P_{2_t}(t)$$

For infinite data points this approach is equivalent to leaving $P(t, \theta)$ as it is defined in the introduction and choosing random θ from a uniform distribution on $[0, 2\pi]$. Although in this case there are many data points, using P_t was chosen as it was more deterministic and afforded more confidence when comparing results of different runs. Having the integrated form also showed analytically the behaviour of the new PDF.

Several of these fits were run with different starting values, which resulted in some different, physically reasonable parameter estimations. The Initial values and resulting parameter estimations along with their errors for some of these runs are shown in Table 1.

The errors shown in Table 1 are the mean of the modulo of the positive and negative errors calculated using the simplistic error method.

Plots of ΔNLL around the parameter minimas in run (2) are given in Figure 4

2.1 Comments on the results

- A large range of starting parameters were tested and all the resulting minima were one of the runs in Table 1. Although by no means was an exhaustive search for all (physically reasonable) local minima carried out.
- The result of run (2) could have been predicted from run (1). Going from run (1) to run (2) we see they relate by swapping τ_1 and τ_2, and mapping $F \to (1-F)$. Inspection of the form of P_t shows that P_t is indeed symmetric under such a change.
- What has occured in run (3) is that a local minimum in which $\tau_1 = \tau_2$ has been found. In this case P_t is independent of F. Thus the error of F for this run in infinite. This is a particularily bad case and in fact inspecing `minuit.is_valid` would have shown that this wasn't even a valid minimum and should be discarded. Because of the nature of this report it is left in.

Table 1: t data fits

	Initial			Fit			Errors		
run	F	$\tau_1(s)$	$\tau_2(s)$	F	$\tau_1(s)$	$\tau_2(s)$	F	$\tau_1(s)$	$\tau_2(s)$
(1)	0.2	3.5	2.3	0.038	0.512	1.92	0.0029	0.036	0.0064
(2)	0.9	1.9	0.5	0.962	1.925	0.513	0.0029	0.0064	0.036
*(3)	0.3	2.0	0.3	0.809	1.87	1.87	-	-	-

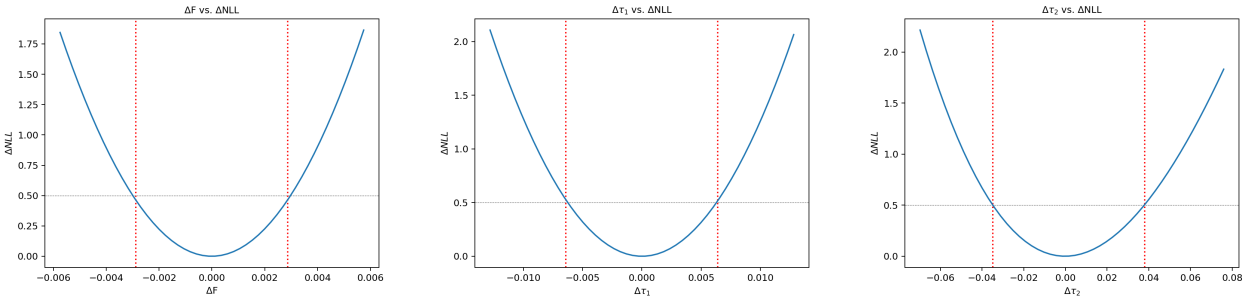

Figure 4: ΔNLL around each parameter minimum

3 Maximum Likelyhood Fit on Full t, θ Data

A maximum likelyhood fit was performed finding the parameters of $P(t, \theta)$ using the full (t, θ) data. A comparison of the fit using full (t, θ) data and (t) data only is shown in Table 2. The (t) only fit is taken from run (2) in Table 1. As in section 2, the errors for the full (t, θ) fit are calculated using the simplistic errors method.

All initial parameters for the (t, θ) fit resulted in the same fit values so a comparison of initial parameters is ommited from Table 2

Table 2: Full (t, θ) and (t) only fits

	Fit			Errors		
Data	F	$\tau_1(s)$	$\tau_2(s)$	F	$\tau_1(s)$	$\tau_2(s)$
(t, θ)	0.540	1.466	2.35	0.0030	0.0087	0.013
(t) run (2)	0.962	1.925	0.513	0.0029	0.0064	0.036

3.1 Comments on the results

- The fit values for both cases in Table 2 are very different. The t data alone is not sufficient to gather an estimate of our fit parameters.
- It is a reasonable assumption that given aribitrarily many points of (t) data, the fit values would not approach the values for the full (t, θ) fit. It is the fact that the θ values break the symmetry between the P_1 and P_2 PDFs that allows for a proper estimation of the parameters.
- The errors on the two fits are quite similar

4 Full Errors on Fits

The simplistic error method used in Section 2 and Section 3 is only accurate if the the parameters are uncorrelated. In this case the parameters are correlated, especially for the case in Section 2 and the t data only fit.

What must be considered when calculating errors is if one parameter is fixed away from the minimum, how would the other parameters change to re-minimise the NLL. For example, let F, τ_1, τ_2 be set at the minima values for the fit in Section 2. Now increase τ_1 by a small ammount. By observing the form of P_t one can see that if τ_1 is now fixed, F and τ_2 should decrease to re-minimise the NLL. So in this case τ_1 is negatively correlated to F and τ_2.

When the parameters are correlated, errors should be calculated using the full errors method. These full error values were calculated for the fits in Section 2 and Section 3. The upper and lower error bounds along with the mean error are shown alongside the fit values in Table 3

Pairwise Error Contours for the fits in Section 2 and Section 3 are given in Figures 5 and 6.

Notes

- For the errors in the Section 2 fit, the `minuit.hesse()` error calculator was not adequate as is warned about in the iminuit documentation. The error values in Table 3 were calculated using my own implementation of the full error method and were checked against the more thorough `minuit.minos()` method.
- The upper and lower bounds for the (t, θ) fit in Section 3 are essentially symmetric about the minimum. The upper and lower bounds for the (t) only fit are less symmetric, which suggests the NLL is less parabolic about the minimum than with the full (t, θ) fit. This is in agreement with the less ellipsoidal shape of the error contours of Section 2.

Table 3: Fits with full errors

	Fit			Error Bounds			Error Mean		
Data	F	$\tau_1(s)$	$\tau_2(s)$	F	$\tau_1(s)$	$\tau_2(s)$	F	$\tau_1(s)$	$\tau_2(s)$
(t, θ)	0.540	1.466	2.35	$^{+0.0032}_{-0.0032}$	$^{+0.0097}_{-0.0096}$	$^{+0.014}_{-0.014}$	0.0032	0.0097	0.0014
(t) run (2)	0.962	1.925	0.513	$^{+0.0059}_{-0.0065}$	$^{+0.0099}_{-0.0096}$	$^{+0.058}_{-0.0056}$	0.0062	0.0097	0.057

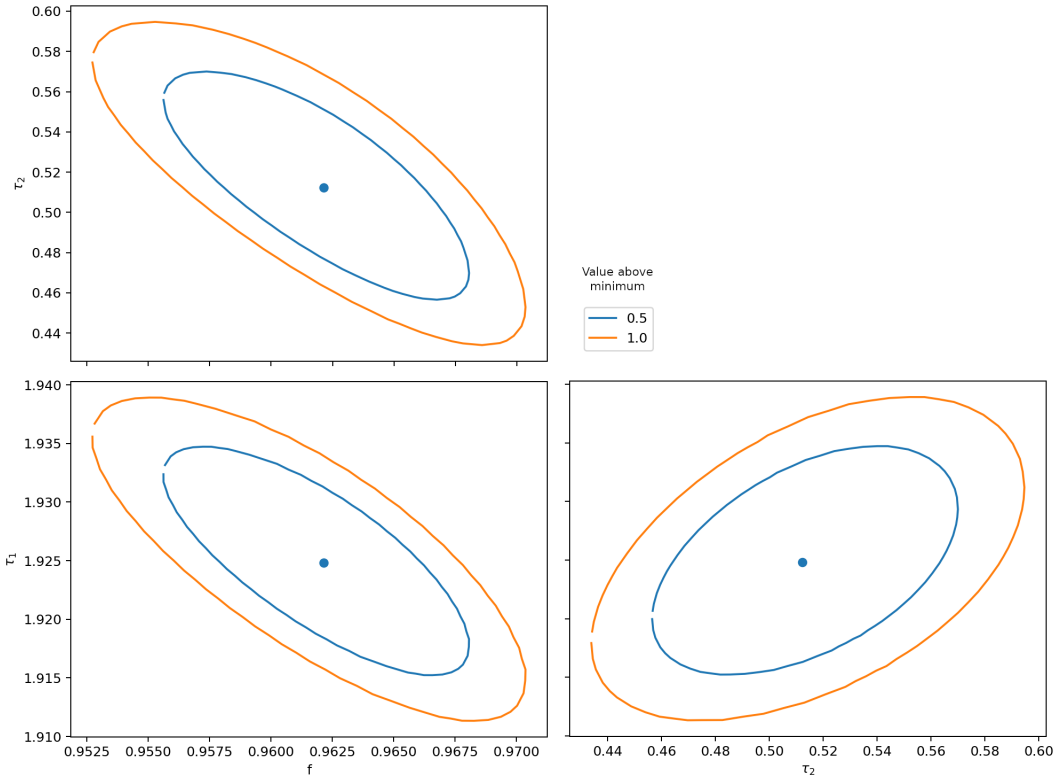

Figure 5: Error contours for fit in Section 2

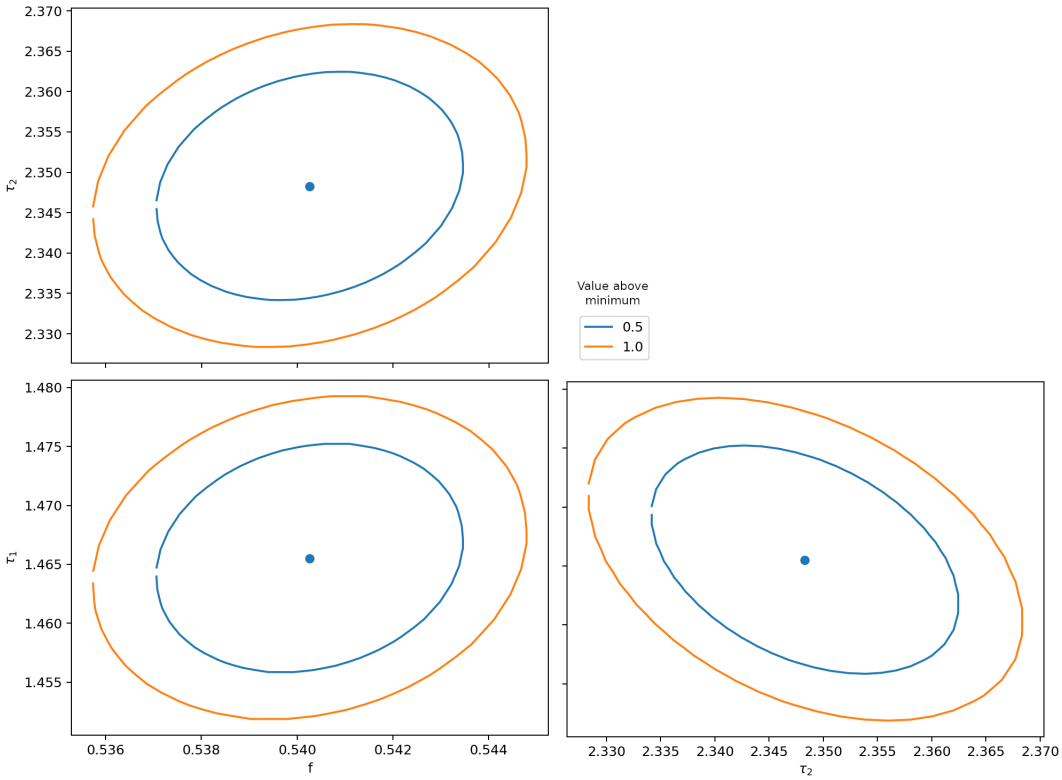

Figure 6: Error contours for fit in Section 3

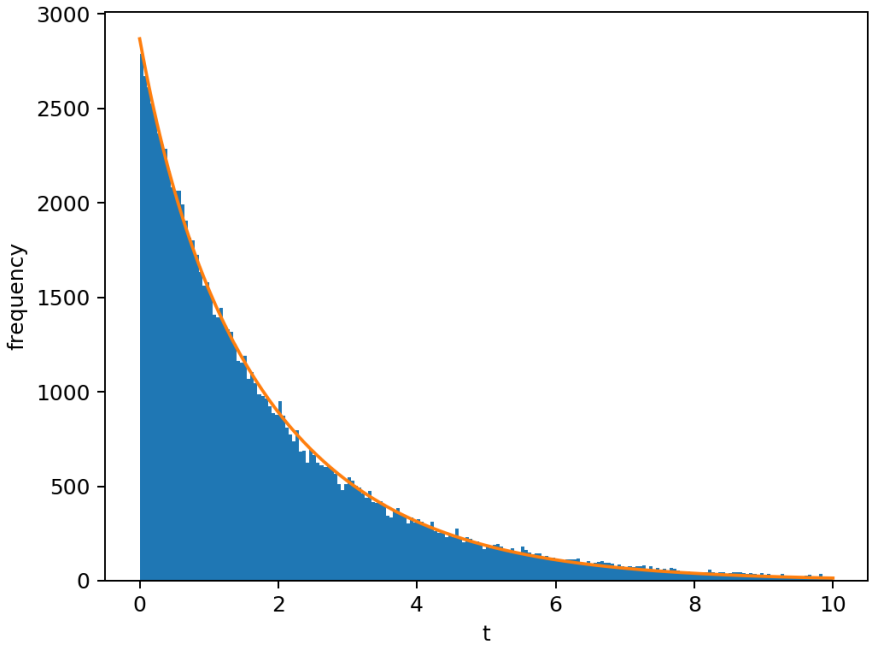

Figure 7: Histogram of t data alongside P_t with values from run (2) in Table 1

Conclusion

The fit in section 2 was unstable and unreliable, but when the final result of the method P_t is examined as is done in Figure 7, it is clear that $P_t(t; 0.962, 1.925, 0.513)$ is a perfectly good model for the t data. The problem was not that the model was incorrect but that it was overcomplex. Any dataset with a simple distribution *can* be accurately modelled by a complex PDF, so long as it has the correct simple part as a component. Issues arise when the parameters of these over complex PDFs are being determined. This is because the NLL is more likely to have multiple local minima and minima along lines (as in run (3) in Table 1) or planes of correlated parameters, rather than at points.

In this simple case, a human observation of the t data would have shown that the data is modelled best by a simple PDF rather than a complex one but in more complex problems such an intervention is not possible. In that case, such non point, poorly behaved minima could be used to suggest that the model is overcomplex or that there is not yet enough data to determine the parameters of the model.

In that sense, this problem was not only an example of bad parameter fitting practices but touched on the subtle concept of the information content of datasets.

Glossary

- **PDF (Pobability Density Function):** A PDF $P(\{s_i^{'}\})$ describes the likelyhood of an event having variables at values near $\{s_i^{'}\}$ i.e. the probability that for each s_i, $s_i^{'} \leq s_i \leq s_i^{'} + ds_i = P(\{s_i^{'}\})ds_1 ds_2 ds_3 ...$ In our model the probability that $t^{'} \leq t \leq t^{'} + dt$ and $\theta^{'} \leq \theta \leq \theta^{'} + d\theta$ is $P(t^{'}, \theta^{'})dtd\theta$
- **Maximum Likelyhood:** The method to find parameters of a model is maximise the probability that our data events happened given our model. The probability of a single event in our data occuring is proportional to the value of our model's PDF evaluated at that data point. Thus the probability of all the data events happening λ is $\prod_n P(d_n)$ where each d_n is a data point. Maximum Likelyhood is the method of maximising λ by changing the parameters of P
- **NLL (Negative Log Likelyhood):** When there are many data points, the value of λ as described for maximum likelyhood becomes very small. To make this method more numerically stable and practical for use with computers, $\log \lambda$ is maximised instead. Programs traditionally compute minima, so conventionally the minimum of the $NLL = -\log \lambda$ is computed. The log turns the product in to a sum and the generalised expression for NLL as given in the introduction is

$$\text{NLL} = -\sum_n \log P(d_n)$$

- **Fit Values:** The values of the parameters at a minima in our maximum likelyhood fit
- **Minima:** A (local unless stated otherwise) minimum point of the NLL on our parameter space.
- **Error Contours:** Points on the 2D parameter surface that lie a constant value above the NLL minimum. Importantly, as the 2 parameters are varied a minimiser is constantly running and changing the remaining parameters. For example take the bottom left graph in Figure 5. The blue line represents (F, τ_1) points where if F and τ_1 are kept fixed and then τ_2 is varied to find a minimum value of NLL. Then that new minimum value is 0.5 above the minimised value of NLL at the blue dot which is our parameter estimation for F and τ_1. These contours can show correlation between parameters and give a sense of what shape the minima is. As a general rule the more eccentric the contour, the more correlated the two parameters are.

Methods

- **Box Method:** A method for generating random values according to a given PDF. Say there is a PDF $P(\{s_i\})$ to be simulated where each s_i has a set, finite range $s_{i_{\text{lower}}} \leq s_i \leq s_{i_{\text{upper}}}$. The method is

1. Find a value M that is a maximum of $P(\{s_i\})$ on the set range.
2. Get random values of s_i from uniform distributions on their respective ranges.

3. Get a random y from a uniform distribution on $[0, M]$
4. If $y \leq P(\{s_i\})$ then return the values $\{s_i\}$
5. If (4) is not satisfied repeat from (2) until (4) is satisfied.

- **Simplistic Error Method:** A method for finding the error on a parameter q_j without re-minimising the remaining parameters $s_{i \neq j}$. The method is

1. Minimise the NLL to find the minimum NLL value l and minima parameters $\{q_i^{\min}\}$. Set $\{q_i^{'}\} \coloneqq \{q_i^{\min}\}$
2. Choose a 'jump value' Δq_j
3. Map $q_j \to q_j + \Delta q_j$
4. In $\{q_i^{'}\}$ set $q_j^{'} \coloneqq q_j$
5. Evaluate $l^{'} = \text{NLL}(\{q_i^{'}\})$
6. If $l^{'} \geq l + 0.5$ then map $\Delta q_j \to -\frac{1}{2}\Delta q_j$
7. Repeat from (3) until the desired accuracy is obtained
8. The final value of $\left| q^{'} - q_j^{\min} \right|$ is the simplistic standard error

- **Full Error Method:** A method for finding the error on a parameter q_j while re-minimising the remaining parameters $s_{i \neq j}$. It is very similar in structure to the simplistic error method.

1. Minimise the NLL to find the minimum NLL value l and minima parameters $\{q_i^{\min}\}$. Set $\{q_i^{'}\} \coloneqq \{q_i^{\min}\}$
2. Choose a 'jump value' Δq_j
3. Map $q_j \to q_j + \Delta q_j$
4. In $\{q_i^{'}\}$ set $q_j^{'} \coloneqq q_j$
5. While fixing $q_j^{'}$ and starting from $\{q^{'}\}$ re-minimise the NLL to find the new minimum NLL value $l^{'}$ and new minima parameters $\{q_i^{\text{re-min}}\}$. Note by fixing $q_j^{'}$ here $q_j^{\text{re-min}} = q_j^{'}$
6. Set $\{q_i^{'}\} \coloneqq \{q_i^{\text{re-min}}\}$
7. If $l^{'} \geq l + 0.5$ then map $\Delta q_j \to -\frac{1}{2}\Delta q_j$
8. Repeat from (3) until the desired accuracy is obtained
9. The final value of $\left| q^{'} - q_j^{\min} \right|$ is the full standard error on q_j

Notes on error methods

- Starting with a positive Δq_j will give a positive error, and visa versa.
- If an accuracy of ϵ is required with a starting jump Δq, then the number n times to re-map Δq_j is given $n = \left\lceil \log_2 \frac{\Delta q_j}{\epsilon} \right\rceil$
- Although structurally very similar, the full error method takes significantly longer to run as it performs many minimisations. The simplistic error method is sufficient for uncorrelated parameters.
- In essence, the full error method describes what is performed by `minuit.minos()`

Appendix

Files

1. Orgnaise.py
2. Plot.py
3. Optimise.py
4. NLL.py
5. Function.py
6. RanGen.py
7. Data.py

(1) Organise.py

```
import numpy as np
from iminuit import Minuit

from Function import Function, FixParams, ComposeFunction
from Optimise import Optimise
from NLL import NLL
from RanGen import RanGen
from Data import Data
from Plot import Plot

'''
Where numbers and instances of all the other classes are moved around
to do useful things.
~Roughly~ a method should correspond to a part in the Report outline

Look here is you want to change the starting values of parameters or
to change other minimiser parameters.
'''
class Organise(object):

    def plotPDF(self):
        F = 0.5
        tau1 = 1.0
        tau2 = 2.0

        numEvents = 10000

        #this is known by me (knowing the PDF)
        #a lazy choice (1 is quite above the max PDF val)
        maxPDFVal = 1.

        tLower = 0.
        tUpper = 10.
        thetaLower = 0.
        thetaUpper = 2.*np.pi
        freeParamRanges = [[tLower, tUpper], [thetaLower, thetaUpper]]

        #random num generator for F = 0.5
        #-----------------------------------------------------------------------
        coupledPDF = Function()
        coupledFixed = FixParams(coupledPDF.fPDF_tRange, 5, [F, tau1, tau2],
            [0,3,4])
```

```
        coupledFixedGen = RanGen(coupledFixed.eval, freeParamRanges, maxPDFVal)

        print("Generating coupled random events...")
        #t in [0]
        #theta in [1]
        t_thetaValsCoupled = coupledFixedGen.manyBox(numEvents)

        #only P1
        #-------------------------------------------------------------------------
        F = 1.
        P1Fixed = FixParams(coupledPDF.fPDF, 5, [F, tau1, tau2], [0,3,4])
        P1FixedGen = RanGen(P1Fixed.eval, freeParamRanges, maxPDFVal)

        print("Generating P1 only random events...")
        t_thetaValsP1 = P1FixedGen.manyBox(numEvents)

        #only P2
        #-------------------------------------------------------------------------
        F = 0.
        P2Fixed = FixParams(coupledPDF.fPDF, 5, [F, tau1, tau2], [0,3,4])
        P2FixedGen = RanGen(P2Fixed.eval, freeParamRanges, maxPDFVal)

        print("Generating P2 only random events...")
        t_thetaValsP2 = P2FixedGen.manyBox(numEvents)

        #plotting
        #-------------------------------------------------------------------------
        plotter = Plot()
        plotter.plotDistributions(t_thetaValsCoupled, "Coupled")
        plotter.plotDistributions(t_thetaValsP1, "P1")
        plotter.plotDistributions(t_thetaValsP2, "P2")

    #finds F, tau1, tau2 values to fit the fThetaIndepPDF to the t data only
    #does some plots
    def fitTVals(self):
        filename = "data/datafile-Xdecay.txt"
        data = Data(filename)
        func = Function()
        dataForNLL = [[item] for item in data.tVals]

        #initial guesses
        fInitial = 0.962
        tau1Initial = 1.925
        tau2Initial = 0.513
        initialGuess = (fInitial, tau1Initial, tau2Initial)

        print("Initial Guesses:")
        print(initialGuess)

        nllCalc = NLL(func.fThetaIndepPDF, dataForNLL, [1], 4)

        self.fit(nllCalc, initialGuess)

        #plotting error contours (takes very long)
        plotter = Plot()
        print("Plots:\n-----------------------")
        plotter.errorCtr(self.minuit, self.fitSoln)

        self.simplisticErrors(nllCalc, self.fitSoln)

        #full errors-------------------------------------------------
```

```
        opt = Optimise()

        fJump = 0.01
        tau1Jump = 0.01
        tau2Jump = 0.01
        posJumps = [fJump, tau1Jump, tau2Jump]

        fAccuracy = 0.00001
        tau1Accuracy = 0.00001
        tau2Accuracy = 0.00001
        accuracys = [fAccuracy, tau1Accuracy, tau2Accuracy]

        fBound = (0.00001,0.999999)
        tau1Bound = (0.00001,20.)
        tau2Bound = (0.00001,20.)
        bounds = (fBound, tau1Bound, tau2Bound)

        print("PosErrs:\n-------------------------")
        posErr = opt.error(nllCalc.evalNLL, self.fitSoln, 0.5,
            posJumps, accuracys, bounds, ("f", "tau1", "tau2"))
        fPosErr, tau1PosErr, tau2PosErr = posErr

        print("NegErrs:\n-------------------------")
        negJumps = [-item for item in posJumps]
        negErr = opt.error(nllCalc.evalNLL, self.fitSoln, 0.5,
            negJumps, accuracys, bounds, ("f", "tau1", "tau2"))
        fNegErr, tau1NegErr, tau2NegErr = [-val for val in negErr]

        fMeanErr = 0.5*(fNegErr + fPosErr)
        tau1MeanErr = 0.5*(tau1NegErr + tau1PosErr)
        tau2MeanErr = 0.5*(tau2NegErr + tau2PosErr)

        print("")
        print("Errors for NLL minimum (full): ")
        print("-----------------------------------------------")
        print("F    :\t+" + str(fPosErr) + "\t-" +
            str(fNegErr) + "\t mean:" + str(fMeanErr))
        print("Tau1:\t+" + str(tau1PosErr) + "\t-" +
            str(tau1NegErr) + "\t mean:" + str(tau1MeanErr))
        print("Tau2:\t+" + str(tau2PosErr) + "\t-" +
            str(tau2NegErr) + "\t mean:" + str(tau2MeanErr))
        print("")

    #finds F, tau1, tau2 values to fit the fPDF to the full (t, theta) data
    def fitFull(self):
        filename = "data/datafile-Xdecay.txt"
        data = Data(filename)
        func = Function()
        fullData = zip(data.tVals, data.thetaVals)
        fullData = [list(item) for item in fullData]

        #initial guesses
        fInitial = 0.6
        tau1Initial = 2.40
        tau2Initial = 1.71
        initialGuess = (fInitial, tau1Initial, tau2Initial)

        nllCalc = NLL(func.fPDF, fullData, [1,2], 5)

        self.fit(nllCalc, initialGuess)
```

```
        self.simplisticErrors(nllCalc, self.fitSoln)

        #plots error contours (takes very long!)
        plotter = Plot()
        print("Plots:\n----------")
        plotter.errorCtr(self.minuit, self.fitSoln)

        #Full errors----------
        opt = Optimise()

        fJump = 0.01
        tau1Jump = 0.01
        tau2Jump = 0.01
        posJumps = [fJump, tau1Jump, tau2Jump]

        fAccuracy = 0.00001
        tau1Accuracy = 0.00001
        tau2Accuracy = 0.00001
        accuracys = [fAccuracy, tau1Accuracy, tau2Accuracy]

        fBound = (0.00001,0.999999)
        tau1Bound = (0.00001,20.)
        tau2Bound = (0.00001,20.)
        bounds = (fBound, tau1Bound, tau2Bound)

        print("PosErrs:\n----------")
        posErr = opt.error(nllCalc.evalNLL, self.fitSoln, 0.5,
            posJumps, accuracys, bounds, ("f", "tau1", "tau2"))
        fPosErr, tau1PosErr, tau2PosErr = posErr

        print("NegErrs:\n----------")
        negJumps = [-item for item in posJumps]
        negErr = opt.error(nllCalc.evalNLL, self.fitSoln, 0.5,
            negJumps, accuracys, bounds, ("f", "tau1", "tau2"))
        fNegErr, tau1NegErr, tau2NegErr = [-val for val in negErr]

        fMeanErr = 0.5*(fNegErr + fPosErr)
        tau1MeanErr = 0.5*(tau1NegErr + tau1PosErr)
        tau2MeanErr = 0.5*(tau2NegErr + tau2PosErr)

        print("")
        print("Errors for NLL minimum (full): ")
        print("-----------------------------------------------")
        print("F   :\t+" + str(fPosErr) + "\t-" + str(fNegErr) +
            "\t mean:" + str(fMeanErr))
        print("Tau1:\t+" + str(tau1PosErr) + "\t-" + str(tau1NegErr) +
            "\t mean:" + str(tau1MeanErr))
        print("Tau2:\t+" + str(tau2PosErr) + "\t-" + str(tau2NegErr) +
            "\t mean:" + str(tau2MeanErr))
        print("")

    #general method for organising a minuit fit of an NLL
    #
    #nllCalc should be an instance of NLL
    def fit(self, nllCalc, initialGuess):

        fBound = (0.00001,0.999999)
        tau1Bound = (0.00001,20.)
        tau2Bound = (0.00001,20.)
        bounds = (fBound, tau1Bound, tau2Bound)
```

```
        fTolerance = 0.001
        tau1Tolerance = 0.001
        tau2Tolerance = 0.001
        tolerances = (fTolerance, tau1Tolerance, tau2Tolerance)

        m = Minuit.from_array_func(nllCalc.evalNLL, initialGuess,
            error=tolerances, limit=bounds, name = ("f", "tau1", "tau2"),
            errordef=0.5, print_level=0)

        m.migrad()

        print("NLL val at minimum:")
        print(m.fval)
        print("")

        soln = [value for (key,value) in m.values.items()]
        NLLMin = nllCalc.evalNLL(soln)
        fMin, tau1Min, tau2Min = soln

        print("Vals for NLL minimum: ")
        print("-----------------------------------------------")
        print("F:\t" + str(fMin))
        print("Tau1:\t" + str(tau1Min))
        print("Tau2:\t" + str(tau2Min))
        print("")

        #for use later on in higher up methods
        self.minuit = m
        self.fitSoln = soln

    #prints simple errors as described by simple error method in report
    def simplisticErrors(self, nllCalc, soln):
        fMin, tau1Min, tau2Min = soln
        NLLMin = nllCalc.evalNLL(soln)

        root = Optimise()
        fJump = 0.001
        tau1Jump = 0.01
        tau2Jump = 0.01
        jumps = [fJump, tau1Jump, tau2Jump]

        fAccuracy = 0.0001
        tau1Accuracy = 0.0001
        tau2Accuracy = 0.0001
        accuracys = [fAccuracy, tau1Accuracy, tau2Accuracy]

        shiftVal = 0.5 + NLLMin

        #upperErrors
        fRoot = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumps,
            accuracys, [1,2])[0]
        tau1Root = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumps,
            accuracys, [0,2])[1]
        tau2Root = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumps,
            accuracys, [0,1])[2]

        fPosErr = fRoot - fMin
        tau1PosErr = tau1Root - tau1Min
        tau2PosErr = tau2Root - tau2Min

        #lowerErrors
```

```
jumpsNeg = [-item for item in jumps]

fRoot = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumpsNeg,
    accuracys, [1,2])[0]
tau1Root = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumpsNeg,
     accuracys, [0,2])[1]
tau2Root = root.equalTo(nllCalc.evalNLL, shiftVal, list(soln), jumpsNeg,
     accuracys, [0,1])[2]

fNegErr = fMin - fRoot
tau1NegErr = tau1Min - tau1Root
tau2NegErr = tau2Min - tau2Root

fMeanErr = 0.5*(fNegErr + fPosErr)
tau1MeanErr = 0.5*(tau1NegErr + tau1PosErr)
tau2MeanErr = 0.5*(tau2NegErr + tau2PosErr)

print("Errors for NLL minimum (simplistic): ")
print("-----------------------------------------------")
print("F    :\t+" + str(fPosErr) + "\t-" +
    str(fNegErr) + "\t mean:" + str(fMeanErr))
print("Tau1:\t+" + str(tau1PosErr) + "\t-" +
    str(tau1NegErr) + "\t mean:" + str(tau1MeanErr))
print("Tau2:\t+" + str(tau2PosErr) + "\t-" +
    str(tau2NegErr) + "\t mean:" + str(tau2MeanErr))
print("")

#plotting error info-----------------------------------------
numXVals = 50
fRange = np.linspace(fMin - 2*fNegErr, fMin + 2*fPosErr, numXVals)
tau1Range = np.linspace(tau1Min - 2*tau1NegErr, tau1Min + 2*tau1PosErr,
    numXVals)
tau2Range = np.linspace(tau2Min - 2*tau2NegErr, tau2Min + 2*tau2PosErr,
    numXVals)

fYvals = [nllCalc.evalNLL([val, tau1Min, tau2Min]) for val in fRange]
tau1Yvals = [nllCalc.evalNLL([fMin, val, tau2Min]) for val in tau1Range
    ]
tau2Yvals = [nllCalc.evalNLL([fMin, tau1Min, val]) for val in tau2Range
    ]

#centering around minimum
fRangeCen = [val-fMin for val in fRange]
tau1RangeCen = [val-tau1Min for val in tau1Range]
tau2RangeCen = [val-tau2Min for val in tau2Range]

fYvalsCen = [val-NLLMin for val in fYvals]
tau1YvalsCen = [val-NLLMin for val in tau1Yvals]
tau2YvalsCen = [val-NLLMin for val in tau2Yvals]

#plots values of function around minimum
plotter = Plot()
plotter.errorInfo(fRangeCen, fYvalsCen, [fNegErr, fPosErr], 'F')
plotter.errorInfo(tau1RangeCen, tau1YvalsCen, [tau1NegErr, tau1PosErr],
    r"$\tau_{1}$")
plotter.errorInfo(tau2RangeCen, tau2YvalsCen, [tau2NegErr, tau2PosErr],
    r"$\tau_{2}$")
```

(2) Plot.py

```
import matplotlib.pylab as pl
import numpy as np

'''
Where all of the plotting and use of matplotlib is done
'''
class Plot(object):

    #plots simulated PDF data
    def plotDistributions(self, t_thetaVals, title):
        tVals = [item[0] for item in t_thetaVals]
        thetaVals = [item[1] for item in t_thetaVals]

        pl.figure(1)
        pl.scatter(thetaVals, tVals, s=1)
        pl.title( title + r" $\theta$ vs $t$ occurence scatter")
        pl.xlabel(r"$\theta$")
        pl.ylabel(r"$t$")

        #plotting individually
        tBins = 70
        pl.figure(2)
        pl.hist(tVals, bins=tBins)
        pl.title( title + r" $t$ distribution")
        pl.xlim((0.0,10.0))
        pl.xlabel(r"$t$")
        pl.ylabel("frequency")

        thetaBins = 70
        pl.figure(3)
        pl.hist(thetaVals, bins=thetaBins)
        pl.title( title + r" $\theta$ distribution")
        pl.xlabel(r"$\theta$")
        pl.ylabel("frequency")

        pl.show()

    #Takes very long (hence printing done after each bit)
    #
    #minuit is instance of Minuit
    #soln is list of values of parameters at NLL minimum
    #
    #make sure saveDestination exists so you dont waste a bunch of time
    def errorCtr(self, minuit, soln):
        saveDestination = "data/results/part5/3/f_tau1.png"

        numPoints = 80
        sigma = 1.0

        #Inner Ring_____________________________________________________________
        minuit.set_errordef(0.5)

        f_tau1Ctr1 = minuit.mncontour('f','tau1', numpoints=numPoints, sigma=
            sigma)[2]
        f_tau1x1 = [val[0] for val in f_tau1Ctr1 ]
        f_tau1y1 = [val[1] for val in f_tau1Ctr1 ]

        print("done 1/6")
```

```
f_tau2Ctr1 = minuit.mncontour('f','tau2', numpoints=numPoints, sigma=
    sigma)[2]
f_tau2x1 = [val[0] for val in f_tau2Ctr1 ]
f_tau2y1 = [val[1] for val in f_tau2Ctr1 ]

print("done 2/6")

tau2_tau1Ctr1 = minuit.mncontour('tau2','tau1', numpoints=numPoints,
    sigma=sigma)[2]
tau2_tau1x1 = [val[0] for val in tau2_tau1Ctr1 ]
tau2_tau1y1 = [val[1] for val in tau2_tau1Ctr1 ]

print("done 3/6")

#Outer Ring_____________________________________________________________
minuit.set_errordef(1.0)

f_tau1Ctr2 = minuit.mncontour('f','tau1', numpoints=numPoints, sigma=
    sigma)[2]
f_tau1x2 = [val[0] for val in f_tau1Ctr2 ]
f_tau1y2 = [val[1] for val in f_tau1Ctr2 ]

print("done 4/6")

f_tau2Ctr2 = minuit.mncontour('f','tau2', numpoints=numPoints, sigma=
    sigma)[2]
f_tau2x2 = [val[0] for val in f_tau2Ctr2 ]
f_tau2y2 = [val[1] for val in f_tau2Ctr2 ]

print("done 5/6")

tau2_tau1Ctr2 = minuit.mncontour('tau2','tau1', numpoints=numPoints,
    sigma=sigma)[2]
tau2_tau1x2 = [val[0] for val in tau2_tau1Ctr2 ]
tau2_tau1y2 = [val[1] for val in tau2_tau1Ctr2 ]

print("done 6/6")

#Watch out for setting where to
pl.figure(1)
pl.title(r"f vs $\tau_{1}$ error contour")
pl.xlabel("f")
pl.ylabel(r"$\tau_{1}$")
pl.plot(f_tau1x1, f_tau1y1, '-', label='0.5')
pl.plot(f_tau1x2, f_tau1y2, '-', label='1.0')
pl.scatter([soln[0]],[soln[1]])
pl.legend(loc='best')
pl.savefig(saveDestination)

pl.figure(2)
pl.title(r"f vs $\tau_{2}$ error contour")
pl.xlabel("f")
pl.ylabel(r"$\tau_{2}$")
pl.plot(f_tau2x1, f_tau2y1, '-', label='0.5')
pl.plot(f_tau2x2, f_tau2y2, '-', label='1.0')
pl.scatter([soln[0]],[soln[2]])
pl.legend(loc='best')
pl.savefig(saveDestination)

pl.figure(3)
pl.title(r"$\tau_{2}$ vs $\tau_{1}$ error contour")
```

```
        pl.xlabel(r"$\tau_{2}$")
        pl.ylabel(r"$\tau_{1}$")
        pl.plot(tau2_tau1x1, tau2_tau1y1, '-', label='0.5')
        pl.plot(tau2_tau1x2, tau2_tau1y2, '-', label='1.0')
        pl.scatter([soln[2]],[soln[1]])
        pl.legend(loc='best')
        pl.savefig(saveDestination)

    #nice plots of value of NLL around minimum
    #Data has to be pre processed to avoid making this method
    #overly complex
    def errorInfo(self, xRange, yVals, errors, label):
        numXVals = 50
        lowerError, upperError = errors

        pl.title(r"$\Delta$" + label + r"_vs._$\Delta$NLL")
        pl.xlabel(r"$\Delta$" + label)
        pl.ylabel("$\Delta_NLL$")
        pl.axvline(x = -lowerError, linestyle=':', color='r')
        pl.axvline(x = upperError, linestyle=':', color='r')
        pl.axhline(y = 0.5, linewidth=0.5, linestyle='--', color='dimgray')
        pl.plot(xRange, yVals)
        pl.show()
```

(3) Optimise.py

```
import copy
import math
import numpy as np
from Function import Function
from iminuit import Minuit

'''
Class for finding minima and roots using simple methods.

Philosphy was to make the methods as general as was reasonable.
So min and root can work with function with any number of numerical
parameters. So long as the function takes the parameters
all bundled in a list.

From a deep fear of mutable lists, there is a lot of copy.deepcopy
This might be lazy but it allows for more certainty and understanding
of the methods behaviour.

In general these methods are kept clean and dumb. Their utility is
more in their understandability rather than their efficiency
or utility, since there are other libraries that do that better anyway.

There is no protection against taking too many iterations.
Try to minimise x^3 and it will stall. God bless ctrl-c

No info about the locality or globality of minima is taken in to account.
'''
class Optimise(object):

    #minimising_____________________________________________________________
    #not actually used in report. Opted for minuit instead
    #But copied from previous work so kept for good measure.
    #
    #returns values of params at function (local) minimum
    #
    #func is function to be minimised only argument must be list params
    #fParamsGuess is list of param starting positions
    #paramsJump is list of initial deltas for parameters
    #paramsAccuracy is list of accuracys on each parameter for minimum
    #paramsToFix is list of indexes of params to... keep fixed
    #
    #Minimises a function in a very simple and dumb way.
    #all other parameters are fixed while one parameter is minimised
    #returns values of parameters at minimum
    def min(self, func, fParamsGuess, paramsJump,
     paramsAccuracy, paramsToFix):
        params = copy.deepcopy(fParamsGuess)
        numParams = len(params)

        #how many times to do the whole minimisation
        totalRepeats = 10

        for i in range(totalRepeats):
            #goes through all the parameters
            #minimises function by minimising each param
            #one at a time while fixing all others
            for paramIndex in range(numParams ):
                if not(paramIndex in paramsToFix):
```

```
                params = self.minSingleParam(func, paramIndex, params,
                    paramsJump[paramIndex], paramsAccuracy[paramIndex])
        return params

    #returns value of parameter at a minimum along the line
    #
    #scans variable for when it passes a minimum.
    #once it passes the jump reverses and is divided by two
    #certain number of passes are done until accuracy is reached
    def minSingleParam(self, func, freeParamIndex,
        fParams, freeParamJump, accuracy):
        params = copy.deepcopy(fParams)

        numIter = int(math.ceil(math.log(freeParamJump/accuracy, 2.)))
        if numIter < 1:
            numIter = 1

        #finds initial direction
        val0 = func(params)
        params[freeParamIndex] += freeParamJump
        val1 = func(params)
        direction = np.sign(val0 - val1)

        val1 = val0
        for i in range(numIter):
            changeDir = False
            while not(changeDir):
                val0 = val1
                params[freeParamIndex] += direction*freeParamJump
                val1 = func(params)
                if val1 > val0:
                    changeDir = True
            #halves the jump at each direction change
            freeParamJump /= 2.
            direction *= -1.
        return params

    #root finding_____________________________________________________________
    #returns values of parameters at root
    #i.e. finds params for when func(params)=0
    #paramsToFix is list of parameter indexes that should be fixed
    #
    #arguments are same as for min
    #
    #Works in a simple and dumb way.
    #all other parameters are kept fixed while one parameter is moved around
    #to find either a root or a minima along that line.
    #Then do the same for all the other parameters
    def root(self, func, fParamsGuess,
        paramsJump, paramsAccuracy, paramsToFix):
        params = copy.deepcopy(fParamsGuess)
        numParams = len(params)
        for paramIndex in range(numParams):
            if not(paramIndex in paramsToFix):
                params = self.rootSingleParam(func, paramIndex, params,
                    paramsJump[paramIndex], paramsAccuracy[paramIndex])
        return params

    #retruns value of paramers at minimum or root along the line.
    #But only value in freeParamIndex is changed
    def rootSingleParam(self, func, freeParamIndex,
```

```
        fParams, freeParamJump, accuracy):
        params = copy.deepcopy(fParams)

        numIter = int(math.ceil(-math.log( abs(accuracy/freeParamJump), 2.)))
        if numIter < 1:
            numIter = 1

        #finds initial direction
        val0 = func(params)
        params[freeParamIndex] += freeParamJump
        val1 = func(params)
        direction = -(np.sign(val1 - val0) * np.sign(val0))
        val1 = val0

        for i in range(numIter):
            val0 = val1
            changeDir = False
            while ((np.sign(val0) == np.sign(val1)) and not(changeDir)):
                val0 = val1
                params[freeParamIndex] += direction*freeParamJump
                val1 = func(params)
                if (abs(val1) - abs(val0) > 0.) and (np.sign(val1) == np.sign(
                    val0)):
                    changeDir = True

            if changeDir:
                print(freeParamIndex)
                print("Changed Direction while finding root!")

            freeParamJump /= 2.
            direction *= -1
        return params

    #returns params where func = x
    #i.e. finds root of func - x
    def equalTo(self, func, x, fParamsGuess,
        paramsJump, paramsAccuracy, paramsToFix):
        paramsGuess = copy.deepcopy(fParamsGuess)
        self.func = func
        self.valueToFind = x
        root = self.root(self.funcMinus, paramsGuess, paramsJump, paramsAccuracy
            , paramsToFix)
        return root

    def funcMinus(self, params):
        return self.func(params) - self.valueToFind

    #error finding ____________________________________________________________
    #returns full error on all the parameter minimums as described in the
    #full error method in the report.
    #
    #func is function in question
    #minParams is list of values of params at func minimum.
    #errorDef for NLL is 0.5
    #jumps is list of starting deltas for scan
    #accuracys is list of accuracys for errors on params
    #limits is tuple of duple (lowerLim, upperLim) for minimiser scan.
    #   i.e. ((param1Lower, param1Upper), (param2Lower, param2Upper)...)
    #names is tuple of str to call each parameter
    #
    #Positive jumps will give positive error and visa versa
```

```
#Finds where func = valAtMin + errorDef by scanning each
#param all while constantly minimising func and updating
#the parameters that are not being scanned accordingly.
def error(self, func, minParams, errorDef,
    jumps, accuracys, limits, names):
    valAtMin = func(minParams)
    valAtError = valAtMin + errorDef

    errors = []
    for index in range(len(minParams)):
        paramError = self.errorSingleParam(valAtError, func, index,
            minParams, jumps[index], accuracys[index], limits, names)

        errors.append(paramError)
        print("Done: " + str(index) + "/" + str())

    return errors

#returns param_at_error - param_at_minimum
#
#valAtError is same as errorDef for error
#funcInit is function in question
#fParams is list of values of parameters at minimum of funcInit
#rest of parameters are the same as for error
#
#very similar method to rootSingleParam but after jumping the free param
#the remaining params are changed while fixing the free param so as to
#minimise func as best as possible.
def errorSingleParam(self, valAtError, funcInit,
 freeParamIndex, fParams, freeParamJump, accuracy, limits, names):
    #all this because we are finding root of func-valAtError
    self.valueToFind = valAtError
    self.func = funcInit
    func = self.funcMinus

    params = copy.deepcopy(fParams)

    #creates a well formatted tuple of which vars to fix
    fix = tuple([index==freeParamIndex for index in range(len(params))])

    fitArgs = dict(fix=fix,
                   limit=limits, name = tuple(names),
                   errordef=1, print_level=0)
    m = Minuit.from_array_func(func, tuple(params), **fitArgs)

    numIter = int(math.ceil(-math.log( abs(accuracy/freeParamJump), 2.)))
    if numIter < 1:
        numIter = 1

    #finds initial direction
    val0 = func(params)
    params[freeParamIndex] += freeParamJump
    val1 = func(params)
    direction = -(np.sign(val1 - val0) * np.sign(val0))

    for i in range(numIter):
        val0 = val1
        changeDir = False
        while ((np.sign(val0) == np.sign(val1)) and not(changeDir)):
            val0 = val1
            params[freeParamIndex] += direction*freeParamJump
```

```
            #forming new initial conditions
            for i in range(len(params)):
                m.values[i] = params[i]

            m.migrad()

            #extracting new params
            for i in range(len(params)):
                params[i] = m.values[i]

            #m.fval is value of func at last computed minimum
            val1 = m.fval

            if (abs(val1) - abs(val0) > 0.) and (np.sign(val1) == np.sign(
                val0)):
                changeDir = True

        if changeDir:
            print(freeParamIndex)
            print("Changed Direction while finding root!")

        freeParamJump /= 2.
        direction *= -1

    return params[freeParamIndex] - fParams[freeParamIndex]
```

(4) NLL.py

```
from Data import Data
from Function import Function, FixParams
import math
import copy

'''
An instance of this class is used to return the value of
the NLL (negative log likelyhood) of a particular PDF
attatched to a dataset.

An instance of NLL can only be used to evaluate the NLL
for one PDF with one set of free params and one dataset.

The PDF values attributed to the data are fixed using FixParams
class and evalNLL then takes the remaining free parameters as arguments.

The idea behind this class was to make evalNLL as fast as
possible. Therefore an instance of NLL is a bit memory
heavy (at least compared to other ways of doing the same thing)
'''
class NLL(Function):

    def __init__(self, pdf, data, dataParamsIndex, numPDFParams):
        self.pdf = pdf
        self.numPDFParams = numPDFParams
        self.data = data
        self.numData = len(self.data)
        self.dataParamsIndex = dataParamsIndex
        self.formFixedPDFs()

    #calls all the elements in self.dataFixedPDFs with the
    #remaining free params in params.
    def evalNLL(self, params):
        runningNLL = 0.
        for i in range(self.numData):
            L = -math.log(self.dataFixedPDFs[i].eval(params))
            runningNLL +=  L
        return runningNLL

    #takes data attatched to PDFs and uses it to form a big list of
    #fixed parameter PDFs (one for every data point)
    #These now only need to be called with the remaining free params.
    def formFixedPDFs(self):
        fixedParamPDFs = []
        for i in range(self.numData):
            singleDataFixedPDF = FixParams(self.pdf, self.numPDFParams, self.
                data[i], self.dataParamsIndex)
            fixedParamPDFs.append(copy.deepcopy(singleDataFixedPDF))
        self.dataFixedPDFs = fixedParamPDFs
```

(5) Function.py

```
#File for classes related to Function.
#Contains Function itself and classes for 'modulating'
#instances and methods in Function

import numpy as np
import copy

'''
The methods of this class all return evaluations of PDFs relating to the
report.

As a rule, the functions take all their parameters bundled in a list.
This makes for a more uniform interface between these functions and
other methods that interact with them.

Most importantly it allows for outside methods to deal with functions of
that require different numbers of parameters cleanly.

The methods in this class are dumb, more useful things are done with
them using the other classes in this file.
'''
class Function(object):

    #P1_____________________________________________________
    #this assumes t in [0,infinity]
    def p1PDF(self, params):
        t, theta, tau1 = params
        norm1 = 3*np.pi * tau1
        val = (1. + (np.cos(theta))**2.)*np.exp(-t/tau1)
        return val/norm1

    #same as above but a different normalisation constant because
    #t in [0,10]
    #theta in [0, 2pi]
    def p1PDF_tRange(self, params):
        t, theta, tau1 = params

        norm1 = -3*np.pi * ( tau1*np.exp(-10./tau1) - tau1)
        val = (1. + (np.cos(theta))**2.)*np.exp(-t/tau1)
        return val/norm1

    #P2_____________________________________________________
    #this assumes t in [0,infinity]
    def p2PDF(self, params):
        t, theta, tau2 = params

        norm2 = 3*np.pi * tau2
        val = 3*((np.sin(theta))**2.)*np.exp(-t/tau2)
        return val/norm2

    #same as above but a different normalisation constant because
    #t in [0,10]
    #theta in [0, 2pi]
    def p2PDF_tRange(self, params):
        t, theta, tau2 = params

        norm2 = -3*np.pi * ( tau2*np.exp(-10./tau2) - tau2)
        val = 3*((np.sin(theta))**2.)*np.exp(-t/tau2)
        return val/norm2
```

```
    #PFull_____________________________________________
    #Full PDF P
    #f * pdf1  + (1-f)*pdf2
    def fPDF(self, params):
        f, t, theta, tau1, tau2 = params

        val = f*self.p1PDF([t, theta, tau1]) + (1-f)*self.p2PDF([t, theta, tau2
            ])
        return val

    #equivalent of fPDF for when
    #t in [0,10]
    def fPDF_tRange(self,params):
        f, t, theta, tau1, tau2 = params
        val = f*self.p1PDF_tRange([t, theta, tau1]) + (1-f)*self.p2PDF_tRange([t
            , theta, tau2])
        return val

    #P_Theta_Independent_____________________________________
    #P1 and P2 equivalent for no theta dependence
    # = P1 and P2 integrated over all theta [0,2pi]
    #only one because they both end up the same
    def thetaIndepPDF(self, params):
        t, tau = params
        norm = tau
        val = np.exp(-t/tau)
        return val/norm

    #equivalent of fPDF for no theta dependence
    def fThetaIndepPDF(self, params):
        f, t, tau1, tau2 = params
        val = f*self.thetaIndepPDF([t,tau1]) + (1-f)*self.thetaIndepPDF([t,tau2
            ])
        return val



'''
An instance of this class is used when we want to reduce the number
of parameters for a method in Function.

A use case of this is when certain parameters are fixed by data
and we want our function to only need the remaining free parameters.

e.g. function.fPDF requires 5 values in params.
t and theta are fixed by a data point.
FixParams.eval now only requires 3 values in freeParams.

This might seem like a strange workaround, but it allows for
the minimisers etc. to be as dumb and as general as possible.

In principal this can be used with methods not in Function
'''
class FixParams(object):
    def __init__(self, func, numOfParams, fixParamsVals, indexOfFixParams):
        self.func = func
        self.numOfParams = numOfParams
        self.fixParamsVals = fixParamsVals
        self.indexOfFixParams = indexOfFixParams
```

```
        self.indexOfFreeParams = [index for index in range(self.numOfParams)
            if not(index in indexOfFixParams)]

    def eval(self, freeParams):
        params = range(self.numOfParams)
        for index, val in enumerate(self.fixParamsVals):
            params[self.indexOfFixParams[index]] = val
        for index, val in enumerate(freeParams):
            params[self.indexOfFreeParams[index]] = val
        return self.func(params)


'''
Modulates instances of Function

An instance of this class is used to compose two functions and create
a new method evalCompose that requires only one set of parameters
and to python just looks like a normal method.

This might seem like a strange workaround but when we send a function
to a minimiser, there is no easy way to give the minimiser/root finder
a composition of functions.

A use case is when we want to find where a function f(params) = val.
This is equivalent to finding the root of f(params) - val.
So we compose (lambda x: x-val) with f(params) and then find the
root of FixParams.eval()

This is to make everything else that interacts with functions
as dumb and as general as possible.
'''
class ComposeFunction(object):
    def __init__(self, funcToCompose, funcInitial):
        self.funcInitial = funcInitial
        self.funcToCompose = funcToCompose

    def evalCompose(self, params):
        return self.funcToCompose(self.funcInitial(params))
```

(6) RanGen.py

```
import numpy as np
import sys

'''
For generating pseudo random numbers according to a certain PDF
An instance of RanGen can only be used with one particular PDF
'''
class RanGen(object):

    #paramRanges to be given as...
    #[[p1Lower, p1upper], [p2lower, p2upper], ....]
    def __init__(self, pdf, paramRanges, maxPDFVal):
        self.pdf = pdf
        self.numParams = len(paramRanges)
        self.paramRanges = paramRanges
        self.maxPDFVal = maxPDFVal

    def evaluate(self, params):
        return self.pdf(params)

    #returns random value in range according to PDF
    #uses box method as described in report
    def nextBox(self):
        foundVal = False
        while not(foundVal):
            evalParams = []
            for i in range(self.numParams):
                evalParams.append(np.random.uniform(self.paramRanges[i][0], self.
                    paramRanges[i][1]))
            y = np.random.uniform(0., self.maxPDFVal)

            if self.pdf(evalParams) >= y:
                foundVal = True
        return evalParams

    def manyBox(self, numEvents):
        vals = []
        for i in range(numEvents):
            #shows a little progress indicator
            sys.stdout.write("\r" + str(100*i/numEvents)[0:3] + "%")
            vals.append(self.nextBox())
            sys.stdout.flush()
        print("")

        return vals
```

(7) Data.py

```
'''
Class to wrap up data from a file.
Specific to the data found in..
data/datafile-Xdecay.txt
which has format..

tVal0    thetaVal0
tVal1    thetaVal1
.
.

'''

class Data(object):

    #dataset points to the data file
    def __init__(self, dataFileName):
        self.importData(dataFileName)

    def importData(self, dataFileName):
        self.tVals = []
        self.thetaVals= []

        dataText = open(dataFileName, "r")
        lines = dataText.readlines()

        #number of datapoints
        self.numData = len(lines)

        for line in lines:
            vals = line.split()
            self.tVals.append(float(vals[0]))
            self.thetaVals.append(float(vals[1]))

        dataText.close()
```