In [ ]:
```python
import matplotlib.pyplot as plt
import numpy as np

from math import *
from scipy.optimize import curve_fit, minimize
from scipy.stats import poisson
from tqdm import tqdm
from glob import glob

from matplotlib.ticker import FormatStrFormatter
```

Loading Data

This section comprises the loading of data runs. The data is saved in sections of 1.2 hours and so a second function was created to load multiple files together.

Calibration

In [ ]:
```python
channels = [73, 152, 234, 316, 395, 593, 807]
lifetimes_ns = [100, 200, 300, 400, 500, 750, 1000]

lifetimes_us = [el / 1000 for el in lifetimes_ns]

def Linear(x, m, c):
    return m*x + c

calib_pars, calib_cov = curve_fit(Linear, channels, lifetimes_us)

print(calib_pars)
print(np.sqrt(calib_cov[0][0]), np.sqrt(calib_cov[1][1]))

# Error on channel corresponds to peak curve on MCA, error on lifetime corresponds to the uncertainty on the pulse generator / oscilloscope.
plt.errorbar(channels, lifetimes_us, xerr=4, yerr=10/1000, fmt=".", lw=1, capsize=3, color="indianred", label="Calibration data")

plt.plot(np.arange(0, 900), Linear(np.arange(0, 900), calib_pars[0], calib_pars[1]), color="cornflowerblue", label="Least Squares fit")

plt.xlabel("Channel")
plt.ylabel("Lifetime ($\mu$s)")
plt.margins(0)

plt.legend()
```

In [ ]:
```python
def ApplyCalibration(channelNumbers):
    lifetimes = []
    for channel in channelNumbers:
        lifetimes.append(calib_pars[0] * channel + calib_pars[1])

    return lifetimes
```

In [ ]:
```python
def LoadExperimentRun(path, numberOfChannelsToRemove, calibrate=False):

    channelCounts = np.loadtxt(path)[numberOfChannelsToRemove[0]:-numberOfChannelsToRemove[1]]

    channelNumbers = np.arange(numberOfChannelsToRemove[0], len(channelCounts)+numberOfChannelsToRemove[0])

    if calibrate is True:
        channelNumbers = ApplyCalibration(channelNumbers)

    return (channelNumbers, channelCounts)

def LoadMultipleRuns(basePath, filenamePrefix, numberOfChannelsToRemove, calibrate=False):
    #print(basePath + filenamePrefix + "*.txt")
    filePaths = glob(basePath + filenamePrefix + "*.txt")

    filePaths.sort()
    print(f"Loading:\n{filePaths}")

    channelsSample = LoadExperimentRun(filePaths[0], numberOfChannelsToRemove, calibrate)[0]
    countsSample = LoadExperimentRun(filePaths[0], numberOfChannelsToRemove)[1]

    channelCounts = np.zeros(np.shape(countsSample))
    for filePath in filePaths:
        channelCounts = channelCounts + LoadExperimentRun(filePath, numberOfChannelsToRemove, calibrate)[1]

    occurances = 0
    for el in channelCounts:
        occurances += el


    print(f"{occurances} occurances")

    return (channelsSample, channelCounts)
```

In [ ]:
```python
# Example loading
data = LoadMultipleRuns("../data/", "autoDetection_T-800mV_40ns_DMH_", (100, 1), calibrate=False)

plt.scatter(data[0], data[1], marker=".", color="cornflowerblue")
plt.xlabel("Channel")
plt.ylabel("Counts")
```

Data Analysis Pipeline

Re-Binning Data

In [ ]:
```python
def RebinData(data, binFraction, method="mean", verbose=False):
    # Inputs are the data output from LoadExperimentRun, and a binFraction which is a number between 0 and 1 which determines how large the bins are

    bins = list(data[0])
    data = list(data[1])

    binWidth = (bins[-1] - bins[0]) * binFraction

    lowerBound = bins[0]
    numberOfBins = floor((bins[-1] - bins[0]) / binWidth)

    newBins = np.arange(lowerBound, binWidth * (numberOfBins + 1), binWidth)
    newBins = [el + binWidth / 2 for el in newBins]
    #newBins = [floor(el) for el in newBins]

    #topBound = floor(newBins[-1] + binWidth)
    #newBins.append(topBound)

    binnedData = []

    for i, binBound in enumerate(newBins):
        #print(f"{i}/{len(newBins)}")
        if i < len(newBins)-1:
            #binnedData.append(list(data[newBins[i]:newBins[i+1]]))

            binnedData.append(np.array(data)[np.where((bins > newBins[i]) & (bins < newBins[i+1]))])

    # binnedData is a an array which each row containing a bin
    binnedData = np.array(binnedData, dtype="object")

    averagedBinnedData = []
    binnedDataLowerUncertainty = []
    binnedDataUpperUncertainty = []
    for row in tqdm(binnedData, disable= not verbose):

        avg = AverageDataInBin(row, method, verbose=verbose)
        averagedBinnedData.append(avg[0])
        binnedDataLowerUncertainty.append(avg[1])
        binnedDataUpperUncertainty.append(avg[2])

    return (newBins[:-1], averagedBinnedData, binnedDataLowerUncertainty, binnedDataUpperUncertainty)
```

Methods for Averaging Data

In [ ]:

```python
def AverageDataInBin(dataInBin, method="mean", verbose=False):

    if method == "mean":
        averagedRow = sum(dataInBin) / len(dataInBin)
        uncertainty = np.std(dataInBin)

        upperUncertainty = lowerUncertainty = uncertainty / np.sqrt(len(dataInBin))


    elif method == "gaussian":
        averagedRow = np.mean(dataInBin)
        variance = np.sqrt(averagedRow)
        standardError = variance / np.sqrt(len(dataInBin))

        upperUncertainty = lowerUncertainty = standardError

    elif method == "MLE":

        # Finds the maximum likelihood for mu in a poisson distribution
        # To do this we can find the minimum of the negative log likelihood
        model = minimize(MLE_negativeLogLikelihood, np.max(dataInBin) / 2, args=dataInBin, method="Nelder-Mead")

        averagedRow = model.x
        lowerUncertainty, upperUncertainty = poisson.interval(0.68, averagedRow) # 1 sigma uncertainty
        lowerUncertainty, upperUncertainty = (lowerUncertainty / np.sqrt(len(dataInBin)), upperUncertainty / np.sqrt(len(dataInBin)))

        if verbose:
            print(averagedRow, uncertainty)

    return averagedRow, lowerUncertainty, upperUncertainty


# Method of Maximum Likelihood Estimation - Calculate the likelihood of the pmf function of a given mu for each datapoint in the bin
def MLE_negativeLogLikelihood(lamda, data):

    logLikelihoods = []
    for el in data:
        logLikelihoods.append(poisson.pmf(k=el, mu = lamda))

    return -1 * np.sum(logLikelihoods)


def FindFitParameters(data, FittingFunction, sigma, initialPars=None):

    if sigma == None:
        pars, cov = curve_fit(FittingFunction, data[0], data[1], initialPars)

    else:
        pars, cov = curve_fit(FittingFunction, data[0], data[1], initialPars, sigma=sigma, absolute_sigma=True)

    return pars, cov

def ExponentialCurve(x, a, b, c, d):
    y = []

    for point in x:
        y.append(a * exp(- b * point + c) + d)

    return y

#, [10, -0.01, 0.1, 15]
```

In [ ]:
```python
# Example rebinning

data = LoadMultipleRuns("../data/", "autoDetection_T-800mV_40ns_DMH_", (100, 1), calibrate=False)

rebinnedData = RebinData(data, binFraction=0.05, method="gaussian")

fig, axes = plt.subplots(2, 1, figsize=(6, 12))
ax1, ax2 = axes

ax1.scatter(data[0], data[1], marker=".", color="cornflowerblue")
ax1.set_title("Raw Data")

ax2.errorbar(rebinnedData[0], rebinnedData[1], yerr=rebinnedData[2], fmt=".", color="cornflowerblue")
ax2.set_title("Binned Data, size: 5%")

for ax in axes:
    ax.set_ylim(0, 40)
    ax.set_xlabel("Channels")
    ax.set_ylabel("Counts")

plt.tight_layout()
```

Analysis

Finding optimum bin sizes

In [ ]:
```python
fakeDataX = np.linspace(0, 1000, 1000)
fakeDataY = ExponentialCurve(fakeDataX, 1, 0.01, 0, 0)


rebinnedFakeData_20 = RebinData((fakeDataX, fakeDataY), binFraction=0.20, method="gaussian")
rebinnedFakeData_15 = RebinData((fakeDataX, fakeDataY), binFraction=0.15, method="gaussian")
rebinnedFakeData_10 = RebinData((fakeDataX, fakeDataY), binFraction=0.10, method="gaussian")
rebinnedFakeData_5 = RebinData((fakeDataX, fakeDataY), binFraction=0.05, method="gaussian")

fig, axes = plt.subplots(2, 2, figsize=(8,8), sharex=True, sharey=True)
axes = axes.reshape(4)

titles = ["20% bin size", "15% bin size", "10% bin size", "5% bin size"]
fakeData = [rebinnedFakeData_20, rebinnedFakeData_15, rebinnedFakeData_10, rebinnedFakeData_5]

for i, ax in enumerate(axes):
    ax.plot(fakeDataX, fakeDataY, zorder=0, color="cornflowerblue")

    ax.errorbar(fakeData[i][0], fakeData[i][1], yerr=fakeData[i][2], color="indianred", fmt=".", capsize=3, linewidth=1)

    ax.set_title(titles[i])

axes[2].set_xlabel("x (arb.)")
axes[3].set_xlabel("x (arb.)")
axes[0].set_ylabel("y (arb.)")
axes[2].set_ylabel("y (arb.)")

plt.tight_layout()
```

Find Best Bin Size

This section handles the plotting of the uncertainties on the parameters of a least squares fit for **any input function**. This plot can be used to estimate the most optimal bin-size.

In [ ]:

```
def FindOptimumBinFraction(data, binFractions, FitFunction=ExponentialCurve, method="gaussian", initialPars=None):

    parameterUncertainties = []

    print("Assessing bin fractions")
    for binFrac in binFractions:
        #print(f"Testing binFrac: {binFrac}", end="\r")
        rebinnedData = RebinData(data, binFrac, method=method)

        x = rebinnedData[0]
        y = rebinnedData[1]
        y = np.squeeze(y)

        tol = 0.01
        newSigma = []
        for el in rebinnedData[2]:
            if el < tol:
                newSigma.append(tol)
            else:
                newSigma.append(el)

        pars, cov = FindFitParameters((x,y), FitFunction, sigma=newSigma, initialPars=initialPars)

        for i in range(len(cov)):
            parameterUncertainties.append(np.sqrt(cov[i][i]))

    # Normalise and make array of uncertainties

    parameterUncertainties = np.array(parameterUncertainties).reshape(len(binFractions), len(cov))

    #for i, uncertainties in enumerate(parameterUncertainties.T):

        #parameterUncertainties[:,i] = [el / max(uncertainties) for el in uncertainties]

    return (binFractions, parameterUncertainties)
```

In [ ]:
```
data = (fakeDataX, fakeDataY)

binFractions = np.linspace(0.06, 0.01, 1000)

_, parameterUncertainties = FindOptimumBinFraction(data, binFractions, initialPars=[1, 0.01, 0, 0])

#plt.plot(binFractions, parameterUncertainties.T[0])

fig, axes = plt.subplots(4, 1, figsize=(6,8), sharex=True)

fig.text(0.01, 0.5, 'Fit Parameter Normalised Uncertainty', va='center', rotation='vertical')

parameterNames = ["a", "b", "c", "d"]
yLimits = [0.25, 0.25, 0.25, 0.25]

for ax, uncertainties, parameterName, yLim in zip(axes, parameterUncertainties.T, parameterNames, yLimits):

    uncertainties = np.array(uncertainties)

    uncertainties[uncertainties == inf] = np.nan

    uncertainties = uncertainties / np.nanmax(uncertainties)

    ax.plot(binFractions, uncertainties, color="indianred")

    ax.margins(0)
    ax.set_ylabel(parameterName, rotation=0)


    #ax.set_ylim(0, yLim)
    ax.axvline(x=0.045,ymin=0,ymax=1,c="cornflowerblue", linestyle="dashed", clip_on=False, label="Chosen Bin Fraction")

    if ax == axes[-1]:
        ax.set_xlabel("Bin Size (fraction of total number of ch.)")

    if ax == axes[0]:
        ax.set_title("Fit parameter uncertainties for: $f(x) = a e^{(b x + c)} + d$")
        ax.legend()
```

Gaussian Approximation of the Poisson Distribution

In [ ]:
```
data = LoadMultipleRuns("../data/", "autoDetection_T-800mV_40ns_DMH_", (100, 200), calibrate=True)

binFraction = 0.045
binSize = binFraction * data[0][-1]
print(f"Bin Size: {binSize:0.2f} us")

rebinnedData = RebinData(data, binFraction, method="gaussian", verbose=False)

x = rebinnedData[0]
y = rebinnedData[1]
y = np.squeeze(y)

yErr = np.array(list(zip(rebinnedData[2], rebinnedData[3]))).T
yErr = np.squeeze(yErr)
```

In [ ]:
```
gPars, gCov = FindFitParameters((x,y), ExponentialCurve, sigma=rebinnedData[2], initialPars=[15, 0.05, 0.1, 3])

fig, ax = plt.subplots(1, 1, figsize=(6,8))

print(gPars)

fitCurveX = np.linspace(0, max(x), 1000)
ax.plot(fitCurveX, ExponentialCurve(fitCurveX, gPars[0], gPars[1], gPars[2], gPars[3]), color="cornflowerblue", label="Least Squares Fit")

#ax.plot(fitCurveX, ExponentialCurve(fitCurveX, 15, -0.05, 0.1, 3), color="orange", label="Test Function")

ax.errorbar(x, y, yerr=yErr, fmt=".", capsize=3, linewidth=1, color="indianred", label="Gaussian Averaged Data")

ax.set_xlabel("Decay Time ($\mu s$)")
ax.set_ylabel("Counts")
ax.legend()
ax.set_title(f"Gaussian Approximation\nBin Size: {binSize:0.2f} microseconds")
ax.margins(x=0)
ax.hlines([0], 0, max(x) + 0.01*max(x), color="grey", ls="dashed")
#ax.set_yscale("log")
```

Half-life estimation

In [ ]:
```
decayConst = gPars[1]

print(f"Half-life: {log(2) / decayConst:0.5f} us")

decayConstantUncertainty = np.sqrt(gCov[1][1])

print(f"Uncertainty (propagated from fit): {np.sqrt(decayConstantUncertainty**2 * (log(2) / decayConst**2)**2)}")

print()

print(f"Lifetime: {1/ decayConst:0.5f} us")
print(f"Uncertainty (propagated from fit): {np.sqrt(decayConstantUncertainty**2 * (1 / decayConst**2)**2)}")
```

Maximum Likelihood Estimation

In [ ]:
```
data = LoadMultipleRuns("../data/", "autoDetection_T-800mV_40ns_DMH_", (90, 200), calibrate=True)

binFraction = 0.045
binSize = binFraction * data[0][-1]
print(f"Bin Size: {binSize:0.2f} us")

rebinnedData = RebinData(data, binFraction, method="MLE", verbose=False)

x = rebinnedData[0]
y = rebinnedData[1]
y = np.squeeze(y)

yErr = np.array(list(zip(rebinnedData[2], rebinnedData[3]))).T
yErr = np.squeeze(yErr)
```

In [ ]:
```
sig = []
for a, b in zip(np.squeeze(rebinnedData[2]), np.squeeze(rebinnedData[3])):
    sig.append(np.sqrt(a**2 + b**2))


pars, cov = FindFitParameters((x, y), ExponentialCurve, sigma=sig, initialPars=[19, 0.05, 0.1, 1])

print(pars)

fig, ax = plt.subplots(1, 1, figsize=(6,8))

fitCurveX = np.linspace(0, max(x), 1000)

#ax.plot(fitCurveX, ExponentialCurve(fitCurveX, 18, -0.05, 0.1, 1), color="orange", label="Test Function")

ax.plot(fitCurveX, ExponentialCurve(fitCurveX, pars[0], pars[1], pars[2], pars[3]), color="cornflowerblue", label="Least Squares Fit")

ax.errorbar(x, y, yerr=yErr, fmt=".", capsize=3, linewidth=1, color="indianred", label="MLE averaged data")

ax.set_xlabel("Decay Time ($\mu s$)")
ax.set_ylabel("Counts")
ax.legend()
ax.set_title(f"Maximum Likelihood Estimation Method (Poisson)\nBin Size: {binSize:0.2f} microseconds")
ax.margins(x=0)
ax.hlines([0], 0, max(x) + 0.01*max(x), color="grey", ls="dashed")
```

Half-life estimation

In [ ]:
```
decayConst = pars[1]

print(f"Half-life: {log(2) / decayConst:0.5f} us")

decayConstantUncertainty = np.sqrt(cov[1][1])

print(f"Uncertainty (propagated from fit): {np.sqrt(decayConstantUncertainty**2 * (log(2) / decayConst**2)**2)}")

print()

print(f"Lifetime: {1/ decayConst:0.5f} us")
print(f"Uncertainty (propagated from fit): {np.sqrt(decayConstantUncertainty**2 * (1 / decayConst**2)**2)}")
```

Muon flux as a function of Zenith Angle

In [ ]:

```
zenithData = np.loadtxt("../data/zenithAngles.txt", skiprows=1)

angles = zenithData[:,0] # degrees
anglesUncertainty = 2 # degrees

counts = zenithData[:,1]
countsUncertainties = [np.sqrt(mean) for mean in counts]

time = zenithData[:,2] # seconds
timeUncertainty = 1 # seconds, based on my ability to stop the timer and the couter at the same time.

countRates = [c / t for c, t in zip(counts, time)]
countRateUncertainties = [np.sqrt(timeUncertainty**2 * c**2 / t**4 + uc**2 / t**2) for c, uc, t in zip(counts, countsUncertainties, time)]

fig, ax = plt.subplots(1, 1, figsize=(8,8))

ax.errorbar(angles, countRates, xerr=anglesUncertainty, yerr=countRateUncertainties, fmt=".", linewidth=1, capsize=3, color="cornflowerblue")
ax.set_xlabel("Zenith Angle ($^\circ$)")
ax.set_ylabel("Countrate (counts / second)")
```