ramsauerCode

April 10, 2023

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

# Change default colours to personal colour scheme
import matplotlib as mpl
mpl.rcParams['axes.prop_cycle'] = mpl.cycler(color=["indianred",␣
 ↪"cornflowerblue", "mediumseagreen", "plum", "sandybrown"])

rootPath = r"/home/daraghhollman/Main/UCD_PASS_Labs/RamsauerTownsend/Data"

def LoadFile(path, skiprows=2):
    data = np.array(np.loadtxt(path, skiprows=skiprows))
    return data
```

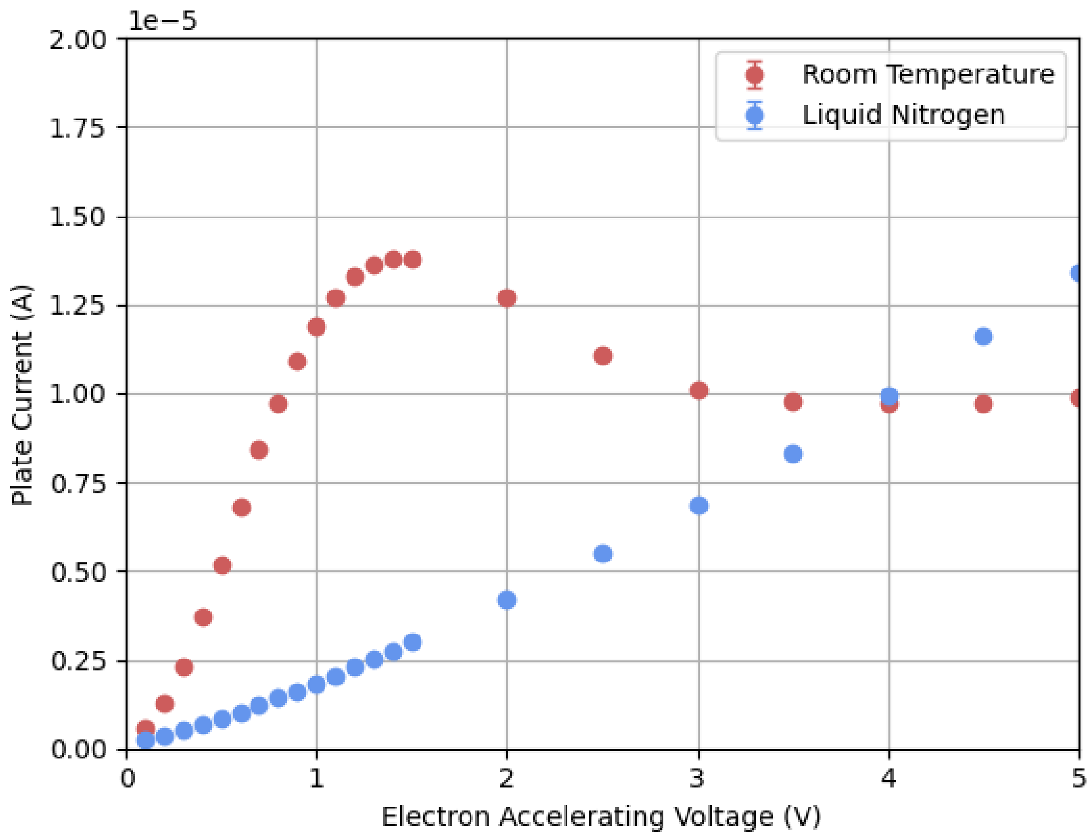
[ ]:
```python
def Current(voltage, resistance):
    return voltage / resistance
```

[ ]:
```python
def ExtractData(data, uncertainnty=False):
    inputVoltage = data[:,0]
    plateCurrent = [Current(el/1000, 10000) for el in data[:,1]] # note␣
 ↪conversion from milivolts to volts
    shieldCurrent = [Current(el/1000, 100) for el in data[:,2]]

    if uncertainnty:
        plateCurrentUncertainty = [Current(0.05/1000, 10000) for el in data[:
 ↪,3]]
        return [inputVoltage, plateCurrent, shieldCurrent,␣
 ↪plateCurrentUncertainty]
    else:
        return [inputVoltage, plateCurrent, shieldCurrent]
```

[ ]:
```python
warmData = ExtractData(LoadFile(rootPath + r"/warmDataLess.txt"),␣
 ↪uncertainnty=True)

coldData = ExtractData(LoadFile(rootPath + r"/coldData.txt"),␣
 ↪uncertainnty=False)
```

```
plt.errorbar(warmData[0], [el*10 for el in warmData[1]], yerr=warmData[3],␣
 ↪fmt="o", capsize=3, linewidth=1, label="Room Temperature")
plt.errorbar(coldData[0], coldData[1], yerr=Current(0.05/1000, 10000), fmt="o",␣
 ↪capsize=3, linewidth=1, label="Liquid Nitrogen")

plt.legend()
plt.grid()
plt.xlim(0, 5)
plt.ylim(0, 2e-5)
plt.xlabel("Electron Accelerating Voltage (V)")
plt.ylabel("Plate Current (A)")
```

[ ]: Text(0, 0.5, 'Plate Current (A)')



[ ]:
```
def ProbabilityOfScattering(warmIp, warmIs, coldIp, coldIs):
    return 1 - warmIp * coldIs / (warmIs * coldIp)

def dPdWIp(warmIp, warmIs, coldIp, coldIs):
    return - coldIs / (warmIs * coldIp)
```

```python
def dPdWIs(warmIp, warmIs, coldIp, coldIs):
    return warmIp * coldIs / (warmIs**2 * coldIp)

def dPdCIp(warmIp, warmIs, coldIp, coldIs):
    return warmIp * coldIs / (warmIs * coldIp**2)

def dPdCIs(warmIp, warmIs, coldIp, coldIs):
    return - warmIp / (warmIs * coldIp)

def ProbabilityOfScatteringUncertainty(warmIp, warmIs, coldIp, coldIs, uWarmIp,␣
↪uWarmIs, uColdIp, uColdIs):

    warmIpContribution = dPdWIp(warmIp, warmIs, coldIp, coldIs)**2 * uWarmIp**2
    warmIsContribution = dPdWIs(warmIp, warmIs, coldIp, coldIs)**2 * uWarmIs**2
    coldIpContribution = dPdCIp(warmIp, warmIs, coldIp, coldIs)**2 * uColdIp**2
    coldIsContribution = dPdCIs(warmIp, warmIs, coldIp, coldIs)**2 * uColdIs**2

    return np.sqrt(warmIpContribution + warmIsContribution + coldIpContribution␣
↪+ coldIsContribution)

def dIdV(R):
    return 1/R

def CurrentUncertainty(uV, R):
    return np.sqrt(dIdV(R)**2 * uV**2)
```

[ ]:

```python
plateVoltageLists = [warmData[1], coldData[1]]
shieldVoltageLists = [warmData[2], coldData[2]]

plateCurrentLists = []
for list in plateVoltageLists:
    newCurrentList = []
    for voltage in list:
        newCurrentList.append(Current(voltage, 10000))
    plateCurrentLists.append(newCurrentList)

shieldCurrentLists = []
for list in shieldVoltageLists:
    newCurrentList = []
    for voltage in list:
        newCurrentList.append(Current(voltage, 100))
    shieldCurrentLists.append(newCurrentList)

probabilityOfScattering = [ProbabilityOfScattering(a, b, c, d) \
    for a, b, c, d in zip(plateCurrentLists[0], shieldCurrentLists[0],␣
↪plateCurrentLists[1], shieldCurrentLists[1])]
```

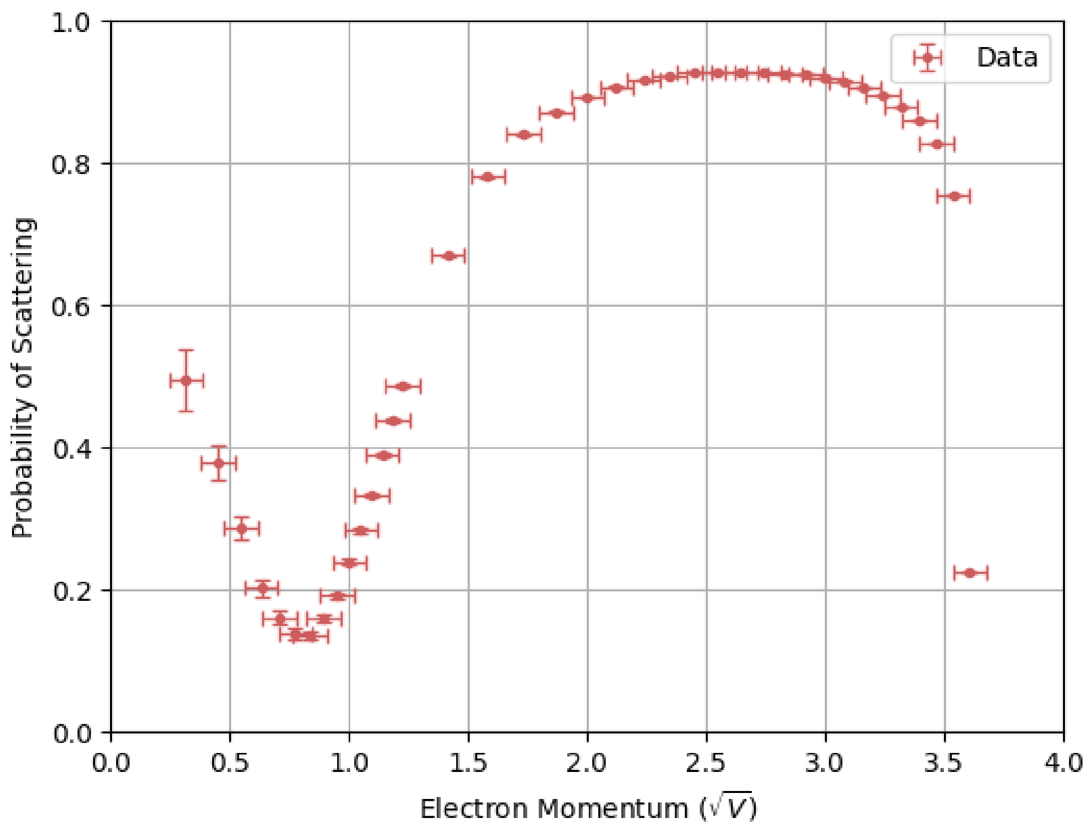
```
warmShieldVoltageError = []
for i in range(len(warmData[2])):
    if warmData[2][i] < 200/1000:
        warmShieldVoltageError.append(0.05/1000)
    elif warmData[2][i] > 200/1000:
        warmShieldVoltageError.append(0.5/1000)

coldShieldVoltageError = []
for i in range(len(coldData[2])):
    if coldData[2][i] < 200:
        coldShieldVoltageError.append(0.05/1000)
    elif coldData[2][i] > 200:
        coldShieldVoltageError.append(0.5/1000)

probabilityOfScatteringUncertainty = []
for warmIp, warmIs, coldIp, coldIs, uWarmIp, uWarmIs, uColdIp, uColdIs in␣
 ↪zip(plateCurrentLists[0], \
    shieldCurrentLists[0], plateCurrentLists[1], shieldCurrentLists[1], \
        [CurrentUncertainty(0.05/1000, 10000)]*len(plateCurrentLists[0]), \
        [CurrentUncertainty(uV, 100) for uV in warmShieldVoltageError], \
            [CurrentUncertainty(0.05/1000, 10000)]*len(plateCurrentLists[0]), \
            [CurrentUncertainty(uV, 100) for uV in coldShieldVoltageError]):

    probabilityOfScatteringUncertainty.append(\
        ProbabilityOfScatteringUncertainty(warmIp, warmIs, coldIp, coldIs,␣
 ↪uWarmIp, uWarmIs, uColdIp, uColdIs))

print(probabilityOfScatteringUncertainty)
probabilityOfScatteringUncertainty = [el/10000 for el in␣
 ↪probabilityOfScatteringUncertainty]
```

```
[431.9289149848619, 252.5157400027043, 169.58129208652815, 122.83559315840829,
94.68138126811247, 76.01240056729162, 62.505310648180185, 52.46985201923111,
44.74463634535257, 38.095797486444965, 33.17852604257133, 29.025188399706074,
25.528922932959098, 22.754087941288464, 20.468115656736533, 13.547138748092992,
10.09115224738238, 8.000144895217508, 6.577475278420045, 5.57813835677166,
4.807737777535432, 4.212666880320136, 3.7342773839934065, 3.3660041689175566,
3.086644132642144, 2.849764657974558, 2.641193438687383, 2.4296813283876197,
2.2496255794021005, 2.0842950651428516, 1.936576219360492, 1.827656714480186,
1.7453829559190897, 1.6776639942688392, 1.5829601627239867, 1.4542950453929377,
1.2765219351608958, 1.1894341574327283, 1.451619884551446, 1.6440654576782265,
1.2261232482605633, 0.9450554210838951]
```

[ ]:
```
electronMomentum = [np.sqrt(el) for el in warmData[0]]
```

```
plt.errorbar(electronMomentum, probabilityOfScattering, xerr=np.sqrt(0.005),␣
↪yerr=probabilityOfScatteringUncertainty, fmt=".", capsize=3, linewidth=1,␣
↪label="Data")

plt.legend()
plt.grid()
plt.xlim(0, 4)
plt.ylim(0, 1)
plt.xlabel("Electron Momentum ($\sqrt{V}$)")
plt.ylabel("Probability of Scattering")
```

[ ]: Text(0, 0.5, 'Probability of Scattering')

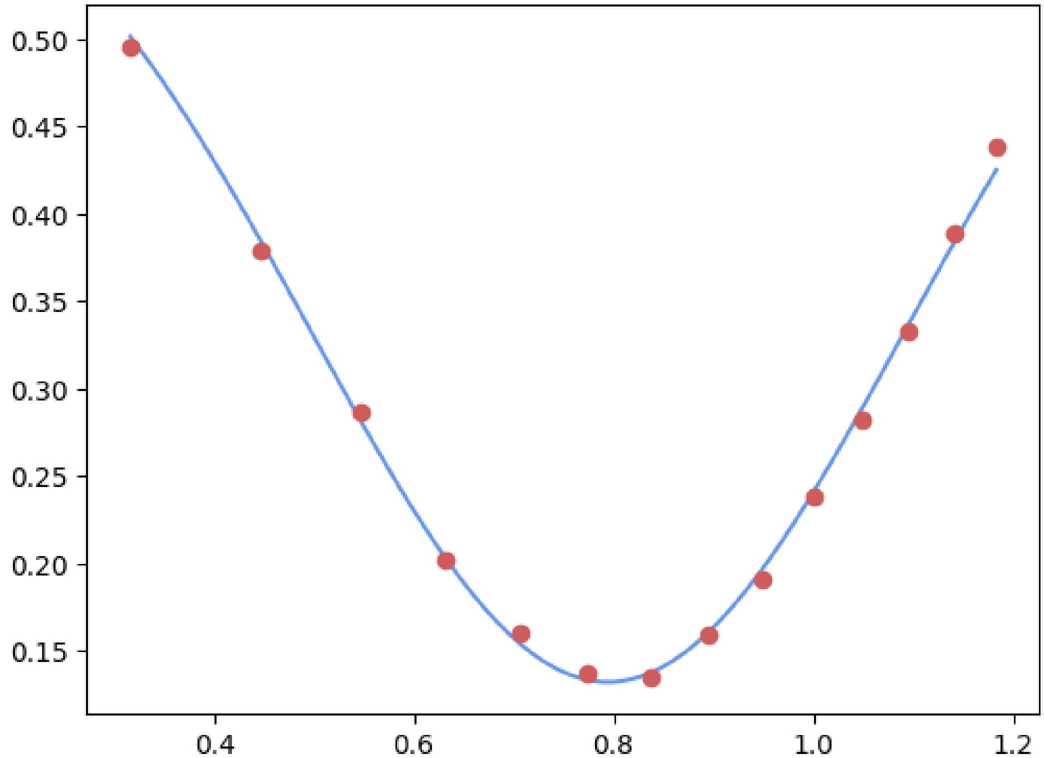


[ ]:
```
def GaussianFunction(x, mu, std, scale, height):
    ans = []
    for el in x:
        ans.append(- scale * np.exp(-(el-mu)**2/(2*std**2)) + height)
    return ans

def FWHM(standardDeviation):
```

```python
    return 2 * np.sqrt(2 * np.log(2)) * standardDeviation

def CalculateMinimumProbability(electronMomentum, probabilityOfScattering,␣
 ↪showPlot=False):
    dipLocation = electronMomentum[0:14]
    dipProbability = probabilityOfScattering[0:14]

    pars, cov = curve_fit(GaussianFunction, dipLocation, dipProbability, [0.8,␣
 ↪0.5, 1, 1.15])

    xRange = np.linspace(np.min(dipLocation), np.max(dipLocation), 100)

    if showPlot:
        plt.errorbar(dipLocation, dipProbability, fmt="o")
        plt.plot(xRange, GaussianFunction(xRange, pars[0], pars[1], pars[2],␣
 ↪pars[3]))
    print(cov)
    return [pars[0], np.sqrt(cov[0][0])]

CalculateMinimumProbability(electronMomentum, probabilityOfScattering,␣
 ↪showPlot=True)
```

```
[[ 5.50751763e-06 -1.07751624e-05 -2.08896894e-05 -2.14370733e-05]
 [-1.07751624e-05  2.08683233e-04  4.02427148e-04  4.29699475e-04]
 [-2.08896894e-05  4.02427148e-04  8.39228547e-04  8.74228691e-04]
 [-2.14370733e-05  4.29699475e-04  8.74228691e-04  9.21005543e-04]]
```

```
[ ]: [0.7943370769321534, 0.0023468100957812162]
```

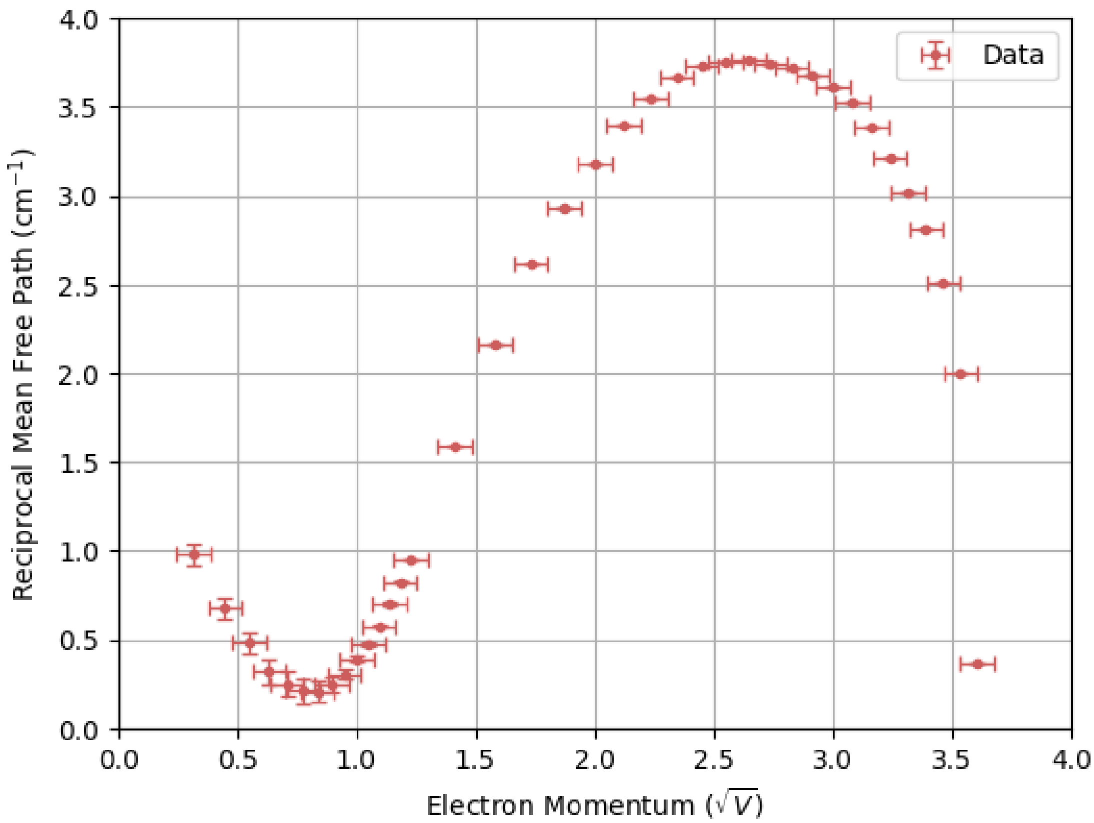


[ ]:
```
def DensityCrossSection(probabilityOfScattering):
    return - np.log(1 - probabilityOfScattering) / 0.7
```

[ ]:
```
densityCrossSectionUncertainty = [np.sqrt((((1-1/probabilityOfScattering)/0.
 ↪7)**2 * (uncertainnty)**2) \
    for probabilityOfScattering, uncertainnty in zip(probabilityOfScattering,␣
 ↪probabilityOfScatteringUncertainty)]
```

[ ]:
```
reciprocalMeanFreePath = [DensityCrossSection(el) for el in␣
 ↪probabilityOfScattering]

plt.errorbar(electronMomentum, reciprocalMeanFreePath, xerr=np.sqrt(0.005),␣
 ↪yerr=densityCrossSectionUncertainty, fmt=".", capsize=3, linewidth=1,␣
 ↪label="Data")

plt.legend()
plt.grid()
plt.xlim(0, 4)
plt.ylim(0, 4)
plt.xlabel("Electron Momentum ($\sqrt{V}$)")
plt.ylabel("Reciprocal Mean Free Path (cm$^{-1}$)")
```

```
[ ]: Text(0, 0.5, 'Reciprocal Mean Free Path (cm$^{-1}$)')
```

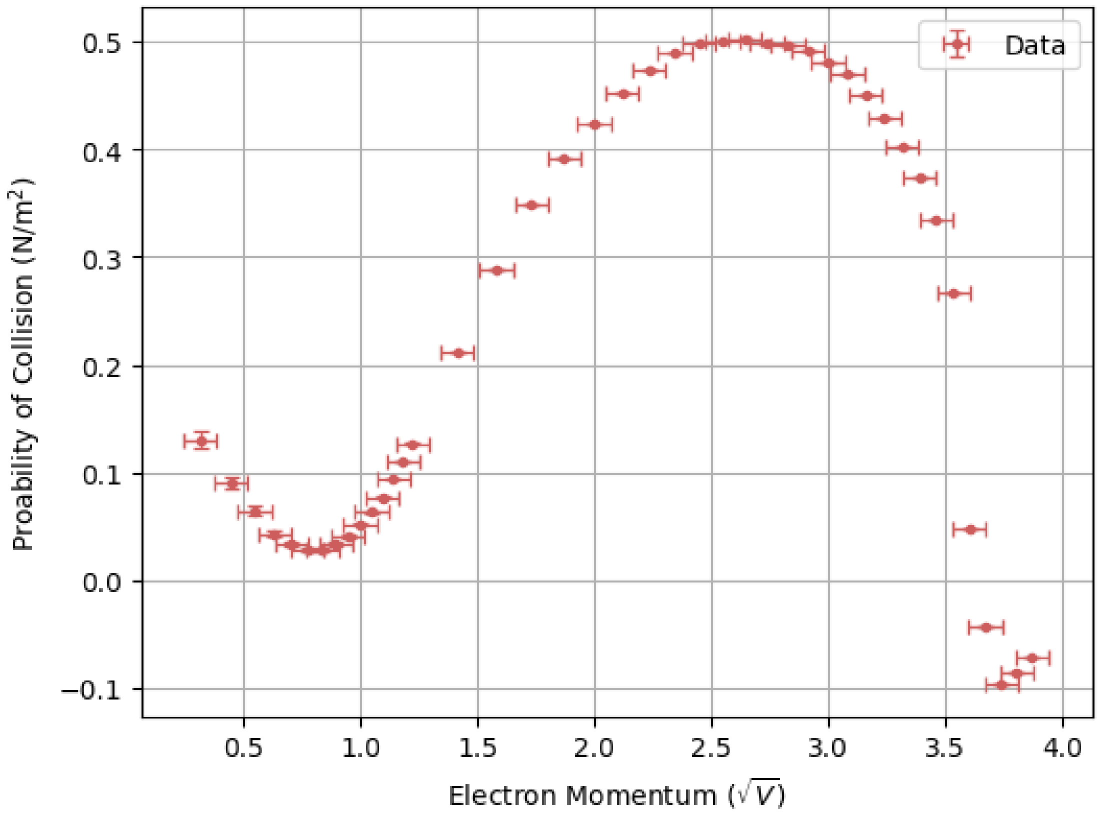



```
[ ]: def ProbabilityOfCollision(pressure, reciprocalMeanFreePath):
         return pressure * reciprocalMeanFreePath
```

```
[ ]: torr = 133.322 # pascals

     probabilityOfCollision = [ProbabilityOfCollision(1e-3 * torr, el) for el in␣
      ↪reciprocalMeanFreePath]
     probabilityOfCollisionUncertainty = [np.sqrt((ProbabilityOfCollision(1e-3 *␣
      ↪torr, reciprocalMeanFreePath))**2 * \
         densityCrossSectionUncertainty**2) for reciprocalMeanFreePath,␣
      ↪densityCrossSectionUncertainty in \
             zip(reciprocalMeanFreePath, densityCrossSectionUncertainty)]

     plt.errorbar(electronMomentum, probabilityOfCollision, xerr=np.sqrt(0.005),␣
      ↪yerr=probabilityOfCollisionUncertainty, fmt=".", capsize=3, linewidth=1,␣
      ↪label="Data")

     plt.legend()
```

```
plt.grid()
plt.xlabel("Electron Momentum ($\sqrt{V}$)")
plt.ylabel("Proability of Collision (N$/$m$^2$)")
```

[ ]: Text(0, 0.5, 'Proability of Collision (N$/$m^2)')



[ ]:
```
extensionData = LoadFile(rootPath + r"/extensionsData.txt")
reversePolarityAccelerationVoltage = extensionData[:,0]
reversePolarityShieldVoltage = [el/1000 for el in extensionData[:,1]]

reversePolarityShieldCurrent = [Current(voltage, 100) for voltage in␣
↪reversePolarityShieldVoltage]
```

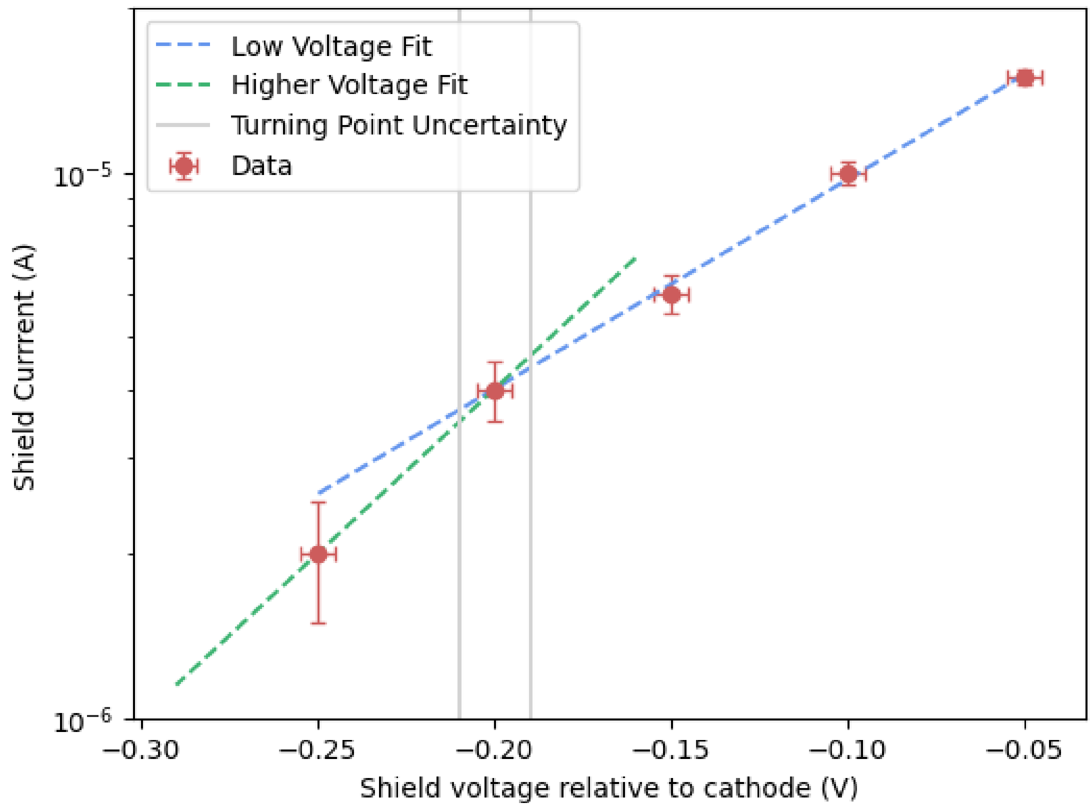
[ ]:
```
def ExpLine(x, A, B):
    return A * np.exp(B*x)
```

[ ]:
```
plt.errorbar(reversePolarityAccelerationVoltage[0:-2],␣
↪reversePolarityShieldCurrent[0:-2], xerr=0.005, yerr=Current(0.05/1000,␣
↪100), fmt="o", capsize=3, linewidth=1, label="Data")

xRangeRight = np.arange(-0.05, -0.26, -0.01)
```

```
xRangeLeft = np.arange(-0.16, -0.30, -0.01)
parsRight, covRight = curve_fit(ExpLine, reversePolarityAccelerationVoltage[0:
 ↪3], reversePolarityShieldCurrent[0:3])
parsLeft, covLeft = curve_fit(ExpLine, reversePolarityAccelerationVoltage[3:
 ↪-1], reversePolarityShieldCurrent[3:-1])

parsLowerFit, covLowerFit = curve_fit(ExpLine, [-.2, -0.25],␣
 ↪[reversePolarityShieldCurrent[3]-Current(0.05/1000, 100),␣
 ↪reversePolarityShieldCurrent[4]-Current(0.05/1000, 100)])
parsUpperFit, covUpperFit = curve_fit(ExpLine, [-.2, -0.25],␣
 ↪[reversePolarityShieldCurrent[3]+Current(0.05/1000, 100),␣
 ↪reversePolarityShieldCurrent[4]+Current(0.05/1000, 100)])

plt.plot(xRangeRight, ExpLine(xRangeRight, parsRight[0], parsRight[1]),␣
 ↪label="Low Voltage Fit", linestyle="--")
plt.plot(xRangeLeft, ExpLine(xRangeLeft, parsLeft[0], parsLeft[1]),␣
 ↪label="Higher Voltage Fit", linestyle="--")

#plt.plot(xRangeLeft[4:10], ExpLine(xRangeLeft[4:10], parsLowerFit[0],␣
 ↪parsLowerFit[1]), linestyle="dotted", color="black", label="Slope␣
 ↪Uncertainty")
#plt.plot(xRangeLeft[4:10], ExpLine(xRangeLeft[4:10], parsUpperFit[0],␣
 ↪parsUpperFit[1]), linestyle="dotted", color="black")

plt.vlines(x=-0.1997+0.01, ymin=0, ymax=1e-4, color="lightgrey", label="Turning␣
 ↪Point Uncertainty")
plt.vlines(x=-0.1997-0.01, ymin=0, ymax=1e-4, color="lightgrey")

plt.yscale("log")
plt.ylim(1e-6, 2e-5)
plt.ylabel("Shield Currrent (A)")
plt.xlabel("Shield voltage relative to cathode (V)")
plt.legend()

print(f"Right line parameters: {parsRight}")
print(f"Left line parameters: {parsLeft}")
print(f"Slope uncertainty: {np.max([abs(parsLeft[1] - parsUpperFit[1]),␣
 ↪abs(parsLeft[1] - parsLowerFit[1])])}")
```

```
/home/daraghhollman/.local/lib/python3.10/site-
packages/scipy/optimize/_minpack_py.py:906: OptimizeWarning: Covariance of the
parameters could not be estimated
  warnings.warn('Covariance of the parameters could not be estimated',
```

```
Right line parameters: [2.34936550e-05 8.84565998e+00]
Left line parameters: [6.40000000e-05 1.38629436e+01]
Slope uncertainty: 3.0830135965451717
```



Low Voltage Fit
Higher Voltage Fit
Turning Point Uncertainty
Data
10^{-5}
10^{-6}
Shield Currrent (A)
−0.30
−0.25
−0.20
−0.15
−0.10
−0.05
Shield voltage relative to cathode (V)