Michael Interferometer

Aditya K. Rao[a)]

University of Toronto
MP222, 60 St. George Street, Toronto, Ontario M5S 1A7, Canada.

[a)]adi.rao@mail.utoronto.ca; Student Number: 1008307761

Abstract. ABSTRACT

CONTENTS

INTRODUCTION

$$F = \frac{V_{\max} - V_{\min}}{V_{\max} + V_{\min}} \tag{1}$$

RESULTS & ANALYSIS

DISCUSSION

CONCLUSION

ACKNOWLEDGMENTS

The work conducted by the other lab partners was instrumental in this lab. Thank you to Jack and Billy for their help in setting up the equipment and conducting the experiments. Additionally, thank you to the Teaching Assistant Michael Sloan and Professor Boris Braverman for their guidance and support.

REFERENCES

1. B. Braverman, "Lab 1: Polarization," (2025).
2. G. Van Rossum and F. L. Drake, *Python 3 Reference Manual* (CreateSpace, Scotts Valley, CA, 2009).
3. C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," Nature **585**, 357–362 (2020).
4. The pandas development team, "pandas-dev/pandas: Pandas," (2020).
5. J. Doe, private communication (2024), assistance Recieved.

Appendix

Raw Data

Analysis Code

Toolkit

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 23 12:34:34 2024
```

```
Updated on Mon Oct 14 21:22:09 2024
Updated on Mon Feb 10 09:51:03 2025

Lab Toolkit

@author: Aditya K. Rao
@github: @adirao-projects
"""
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import os
import math
import textwrap

#from uncertainties import ufloat

font = {’family’ : ’DejaVu Sans’,
        ’size’   : 30}

plt.rc(’font’, **font)

def curve_fit_data(xdata, ydata, fit_type, override=False,
                   override_params=(None,), uncertainty=None,
                   res=False, chi=False, uncertainty_x=None,
                   model_function_custom=None, guess=None):

    def chi_sq_red(measured_data:list[float], expected_data:list[float],
               uncertainty:list[float], v: int):
        if type(uncertainty)==float:
            uncertainty = [uncertainty]*len(measured_data)
        chi_sq = 0

        # Converting summation in equation into a for loop
        for i in range(0, len(measured_data)):
            chi_sq += (pow((measured_data[i] \
                - expected_data[i]),2)/(uncertainty[i]**2))

        chi_sq = (1/v)*chi_sq

        return chi_sq


    def residual_calculation(y_data: list, exp_y_data) -> list[float]:
        residuals = []
        for v, u in zip(y_data, exp_y_data):
            residuals.append(u-v)

        return residuals

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

```

```
    def model_function_exp(x, a, b, c):
        return a*np.exp**(b*x)

    def model_function_log(x, a, b):
        return b*np.log(x+a)

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

    def model_function_linear(x, m):
        return m*x

    def model_function_xlnx(x, a, b, c):
        return b*x*(np.log(x)) + c

    def model_function_ln(x, a, b, c):
        return b*(np.log(x)) + c

    def model_function_sqrt(x, a):
        return a*np.sqrt(x)

    model_functions = {
        'linear' : model_function_linear,
        'linear-int' : model_function_linear_int,
        'xlnx' : model_function_xlnx,
        'log' : model_function_log,
        'exp' : model_function_exp,
        'custom' : model_function_custom
        }

    try:
        model_func = model_functions[fit_type]

    except:
        raise ValueError(f'Unsupported fit-type: {fit_type}')


    if not override:
        new_xdata = np.linspace(min(xdata), max(xdata), num=100)


        if type(uncertainty) == int:
            abs_sig =True
        else:
            abs_sig = False

        if guess is not None:
            popt, pcov = curve_fit(model_func, xdata, ydata, sigma=uncertainty,
                                   maxfev=20000, absolute_sigma=abs_sig, p0=guess)
        else:
            popt, pcov = curve_fit(model_func, xdata, ydata, sigma=uncertainty,
                                   maxfev=20000, absolute_sigma=abs_sig)
        param_num = len(popt)

```

```
        exp_ydata = model_func(xdata,*popt)

        deg_free = len(xdata) - param_num

        new_ydata = model_func(new_xdata, *popt)

        residuals = None
        chi_sq = None

        if res:
            residuals = residual_calculation(exp_ydata, ydata)

        if chi:
            chi_sq = chi_sq_red(ydata, exp_ydata, uncertainty, deg_free)

        data_output = {
            'popt' : popt,
            'pcov' : pcov,
            'plotx': new_xdata,
            'ploty': new_ydata,
            'chisq' : chi_sq,
            'residuals' : residuals,
            'pstd' : np.sqrt(np.diag(pcov))
            }

        return data_output

    else:
        return model_func(xdata, *override_params)


def quick_plot_residuals(xdata, ydata, plot_x, plot_y,
                         residuals, meta=None, uncertainty=[], save=False,
                         uncertainty_x=[]):
    """
    Relies on the python uncertainties package to function as normal, however,
    this can be overridden by providing a list for the uncertainties.
    """
    fig = plt.figure(figsize=(14,14))
    gs = gridspec.GridSpec(ncols=11, nrows=11, figure=fig)
    main_fig = fig.add_subplot(gs[:6,:])
    res_fig = fig.add_subplot(gs[8:,:])

    main_fig.grid('on')
    res_fig.grid('on')
    if type(uncertainty) is int:
        uncertainty = [uncertainty]*len(xdata)

    elif len(uncertainty) == 0:
        for y in ydata:
            uncertainty.append(y.std_dev)

    if meta is None:
        meta = {'title' : 'INSERT-TITLE',
```

```
                'xlabel' : 'INSERT-XLABEL',
                'ylabel' : 'INSERT-YLABEL',
                'chisq' : 0,
                'fit-label': "Best Fit",
                'data-label': "Data",
                'save-name' : 'IMAGE',
                'loc' : 'lower right'}

    main_fig.set_title(meta['title'], fontsize = 46)
    if len(uncertainty_x)==0:
        main_fig.errorbar(xdata, ydata, yerr=uncertainty, #xerr=uncertainty_x,
                          markersize='4', fmt='o', color='red',
                          label=meta['data-label'], ecolor='black')
    else:
        main_fig.errorbar(xdata, ydata, yerr=uncertainty, xerr=uncertainty_x,
                          markersize='4', fmt='o', color='red',
                          label=meta['data-label'], ecolor='black')

    main_fig.plot(plot_x, plot_y, linestyle='dashed',
                  label=meta['fit-label'])

    main_fig.set_xlabel(meta['xlabel'])
    main_fig.set_ylabel(meta['ylabel'])
    main_fig.legend(loc=meta['loc'])


    res_fig.errorbar(xdata, residuals, markersize='3', color='red', fmt='o',
                     yerr=uncertainty, ecolor='black', alpha=0.7)
    res_fig.axhline(y=0, linestyle='dashed', color='blue')
    res_fig.set_title('Residuals')
    save_name = meta["save-name"]
    plt.savefig(f'figures/{save_name}.png')

def quick_plot_test(xdata, ydata, plot_x = [], plot_y = [],
                    uncertainty=[]):
    plt.figure(figsize=((14,10)))

    plt.title("Test Plot for data")
    plt.xlabel("X Data")
    plt.ylabel("Y Data")

    if len(uncertainty) != 0:
        plt.errorbar(xdata, ydata, yerr=uncertainty, fmt='o')
    else:
        plt.scatter(xdata, ydata)

    plt.grid("on")
    plt.show()
    plt.savefig('Test.png')
    plt.close()

def block_print(data: list[str], title: str, delimiter='=') -> None:
    """
    Prints a formated block of text with a title and delimiter
```

```

    Parameters
    ----------
    data : list[str]
        Text to be printed (should be input as one block of text).
    title : str
        Title of the data being output.
    delimiter : str, optional
        Delimiter to be used. The default is '='.

    Returns
    -------
    None.

    Examples
    --------
    >>> r_log = 100114.24998718781
    >>> r_dec = 0.007422298127465114
    >>> data = [f'r^2 value (log): {r_log}',
                f'r^2 value (real): {r_dec}']
    >>> block_print(data, 'Regression Coefficient', '=')
    ============================ Regression Coefficient ============================
    r^2 value (log): 100114.24998718781
    r^2 value (real): 0.007422298127465114
    ================================================================================
    """
    term_size = os.get_terminal_size().columns

    breaks = 1
    str_len = len(title)+2
    while  str_len >= term_size:
        breaks += 1
        str_len = math.ceil(str_len/2)


    str_chunk_len = math.ceil(len(title)/breaks)
    str_chunks = textwrap.wrap(title, str_chunk_len)
    output = ''
    for chunk in str_chunks:
        border = delimiter*(math.floor((term_size - str_chunk_len)/2)-1)
        output = f'{border} {chunk} {border}\n'

    output=output[:-1]

    output+= '\n'+ '\n'.join(data) + '\n'
    output+=delimiter*term_size

    print(output)

def numerical_methods(method_type, args=None, custom_method=None):
    def gaussxw(N):

        # Initial approximation to roots of the Legendre polynomial
        a = np.linspace(3,4*N-1,N)/(4*N+2)
```

```
        x = np.cos(np.pi*a+1/(8*N*N*np.tan(a)))

        # Find roots using Newton's method
        epsilon = 1e-15
        delta = 1.0
        while delta>epsilon:
            p0 = np.ones(N,float)
            p1 = np.copy(x)
            for k in range(1,N):
                p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
            dp = (N+1)*(p0-x*p1)/(1-x*x)
            dx = p1/dp
            x -= dx
            delta = max(abs(dx))

        # Calculate the weights
        w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

        return x, w

    def gaussxwab(N,a,b):
        x,w = gaussxw(N)
        return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w

    methods = {
    'gausswx' : gaussxw,
    'gaussxwab' : gaussxwab,
    'custom' : custom_method
    }

    try:
        method = methods[method_type]

    except:
        raise ValueError(f'Unsupported method-type: {method_type}')

    return method(*args)


def interpolation_methods(method_type, args=None, custom_method=None):

    methods = {
    'custom' : custom_method
    }

    try:
        method = methods[method_type]

    except:
        raise ValueError(f'Unsupported method-type: {method_type}')

    return method(*args)
```