# Plot digitizer

March 22, 2022

Author: Ruiyang Liu

## 0.1 Introduction

PlotDigitizer is an application that is designed to calibrate a 2D plot with new axes for academic usages.

Journal plots are the fruit of those academic researchers. Whilst the digital data of the plots is lost due to the loss of disc memory and the lack of cloud services, PlotDigitizer is intended to recover the data.

## 0.2 Problem

The journal plots are usually image files in the format of png, jpeg, gif, etc, saved from pdf or downloaded from online resources. The coordinate of a point in the graph will never be known unless the raw data that is used to plot the graph is obtained or the function of the curve is provided by the author. However, the former case is almost impossible because journal articles rarely includes the raw data or the source of data file is rarely attached. The latter case is only found when the curve is simple enough. Therefore, the plot digitizer has the advantage of getting the approximate the coordinate of any points labelled by the user and output the coordinates. The coordinates can be further used to rebuild the graph and obtain the formula of the trendlines by other software such as Microsoft Excel.

## 0.3 Program specification

Plot digitizer can be opened in the form of .app (application of MacOS), .py and ran by a single command line.

The code will allow:

- Browsing and opening a local image file (png, gif, jpg, etc) in the user's computer.
- Display the image as a plot using matplotlib with backend 'TkAgg' which is compatible with the Tkinter GUI.
- Calibrate the axis of the plot by transforming the built-in coordinate system of matplotlib into the coordinate system expected in the plot.
- Labelling points in the plot and obtain their coordinates in the new coordinate system.
- Output of the coordinates into a csv file.

### 0.4 Approach

The calibration requires two points for a 2D coordinate system.

1. Calculate the ratio between the scale of matplotlib axes and the scale of the new axes:

$$\text{scale}_x = \frac{(x_2 - x_1)_{\text{new}}}{(x_2 - x_1)_{\text{mat}}}$$

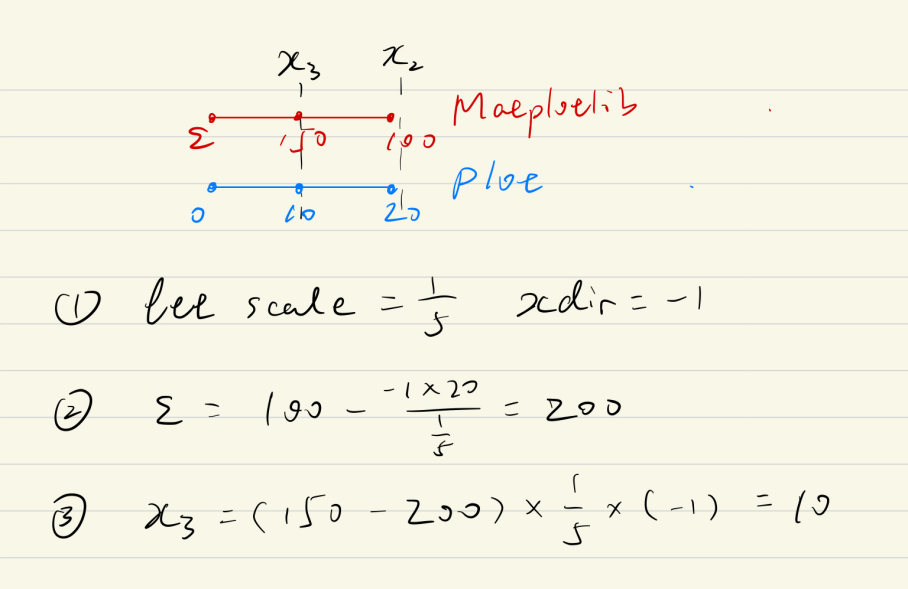
2. Calculate the origin of the new coordinate system in the matplotlib coordinate system. xdir takes the value of 1 and -1 that indicates the direction of new axes relative to the matplotlib axes:

$$\text{origin}_x = (x_2)_{\text{mat}} - \frac{\text{xdir} \cdot (x_2)_{\text{new}}}{\text{scale}_x}$$

3. Calculate the coordinate of a novel point $x_3$ in the new coordinate system:

$$(x_3)_{\text{new}} = ((x_3)_{\text{mat}} - \text{origin}_x) \cdot \text{scale}_x \cdot \text{xdir}$$

An example of calculation:




The calculated value (10) of a novel point is the same as the value shown on the plot axis.

### 0.5 Imports (main.py)

The main.py is an outer framework used to start the main body (plot_digitizer.py) of this program (plot_digitizer.py) in `matplotlib`, so the class Click() from plot_digitizer.py is imported. `mpl_interactions` contains the functions that allows the user to zoom and pan the plot in matplotlib.

The plot is imported by `matplotlib.image` and labelling is done by `matplotlib.pyplot`. `os, stat and subprocess` are the modules for system processes such as opening a file and searching for working directory. `PIL` is used to parse te PIL.UnidentifiedImageError when a file is opened with improper format. Although the matplotlib has a GUI, `tkinter` is used to browse files by filedialog and pop up message by simpledialog and messagebox.

[ ]:
```
import os, stat, subprocess
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import PIL

from plot_digitizer import Click

from tkinter import *
from tkinter import filedialog
from tkinter import simpledialog
from tkinter import messagebox

from mpl_interactions import panhandler, zoom_factory
```

### 0.6 Load image file and open README.txt

When matplotlib is used with tkinter, the default backend (e.g., `MacOSX` in MacOS) needs to be changed to `TkAgg`, otherwise it will crash.

`Tk()` is the ground floor where other widgets are built upon, the size and title of the GUI is set. The OPEN button and README button is linked to the function open_img and read_me.

In the `open_img()`, a pop up window is displayed to browse files in the local directory. There will be three possible outcomes: 1) A image file is opened, no error is raised; 2) Other file type is opened which can not be loaded by matplotlib, raise a `PIL.UnidentifiedImageError`; 3) The pop up window is closed without opening a file, raise an `AttributeError`. Each outcome is provided with a pathway to continue the program without a crash.

In the `read_me()`, use `os.walk(topdown=True)` to search for the file name 'README.txt' either from the closest working directory to the main.py or the PlotDigitizer.app. It is useful because there are other README.txt in the imported modules such as matplotlib.

If an image is opened, display it by `plt.show()`.

[ ]:
```
matplotlib.use('TKAgg') # backend
matplotlib.rcParams['toolbar'] = 'None' # remove matplotlib toolbar

ROOT = Tk()
ROOT.geometry('300x150+300+150')
ROOT.title('plot digitizer')
ROOT.resizable(width=False, height=False) # fix tkinter window

def read_me():
```

```
    name = 'README.txt'

    path = os.path.dirname(os.path.abspath(__file__)) # return the absolute␣
↪directory of the running script

    for root, dirs, files in os.walk(path, topdown=True): # open the first␣
↪README topdown
        if name in files: #and 'build' not in root.split(os.sep): # 'build'␣
↪folder contains README.txt of other modules,
            #print (root.split(os.sep))
            file = os.path.join(root, name)
            os.chmod(file, stat.S_IREAD) # stat.S_IREAD read only for owner
            subprocess.run(['open', file], check=True)
            return ''

    simpledialog.messagebox.showwarning(title='Warning', message='No README.txt␣
↪found')

def open_img():

    while True:
        try:
            filename = filedialog.askopenfilename(title='Open')
            ROOT.withdraw() # hide GUI
            # img = mpimg.imread('/Users/Rui/Desktop/cvece1.gif') # the scale of␣
↪x and y axis is based on the size of image
            img = mpimg.imread(filename)

            break

        except PIL.UnidentifiedImageError:
            simpledialog.messagebox.showwarning(title='Warning', message='Cannot␣
↪identify image file')

            pass

        except AttributeError: # in the case of pressing cancel
            ROOT.deiconify() # reveal GUI
            return '' # jump out of def

    fig, ax1 = plt.subplots(figsize=(6, 4))
    ax1.imshow(img)
    ax1.axis('off')

    Click(ax1, ROOT)
```

```
    plt.show()
```

### 0.7 Import and setting variables in class Click (plot_digitizer.py)

Matplotlib has its own button and cursor wideget differed from Tkinter. Abbreviations such as `ms`, `self.annot` for the fucntions are used to make the code look neat and brief. The variables that are initialized will be used either in calculation or as the conditions for the `if` statements. For example, no label points at the beginning so labelpoint list and output list is empty. `ax.figure.canvas.mpl_connect()` is used to bind different matplotlib event to functions, such as a mousing click and mouse movement.

[ ]:
```
import sys
import csv
import matplotlib.pyplot as plt
from matplotlib.widgets import Cursor
from matplotlib.widgets import Button

from tkinter import simpledialog
from tkinter import filedialog

ms = simpledialog.messagebox.showwarning
qs = simpledialog.askstring
sf = filedialog.asksaveasfile

class Click():

    def __init__(self, ax, ROOT):

            #
            #for the entire code, please see main.py and plot_digitizer.py
            #
            self.press_event = self.ax.figure.canvas.
 ↪mpl_connect('button_press_event', self.onpress)
            self.ax.figure.canvas.mpl_connect('button_release_event', self.
 ↪onrelease)
            self.ax.figure.canvas.mpl_connect('motion_notify_event', self.onmove)
            self.ax.figure.canvas.mpl_connect('close_event', self.handle_close)

            self.cursor = Cursor(ax, horizOn=True, vertOn=True, useblit=True,
                                 color='r', linewidth=1)
            # Creating an annotating box
            self.annot = ax.annotate("", xy=(0, 0), xytext=(-40, 40),␣
 ↪textcoords="offset points",
                                     bbox=dict(boxstyle='round4', fc='linen',␣
 ↪ec='k', lw=1),
                                     arrowprops=dict(arrowstyle='-|>')
```

```
            self.calpoint = []
            self.labelpoint = []
            self.output = []
            self.x_before = []
            self.y_before = []
            self.x_after = []
            self.y_after = []
            self.counter = 0
            self.x_cache = 9999
            self.y_cache = 9999

            self.xdir = 1  # direction of calibrated x axis, default 1 indicates␣
↪from left (negative) to right (positive)
            self.ydir = 1  # direction of calibrated y axis, default 1 indicates␣
↪from bottom (negative) to top (positive)
            #
            #
            #
```

### 0.8 Add buttons

Four buttons are added to the matplotlib GUI and they are connected to different function based on their actual usage.

`_b_connect()` is used to disconnect the button functions when an input dialog is popped up by `simpledialog.askstring`, otherwise multiple pop up window will clash. `con` is initially set to True. It is only necessary when the code runs in the command line interface or as an application. The IDE such as pycharm will freeze the matplotlib until the input is complete.

[1]:
```
    def _add_buttons(self):

        self.button_cal = Button(plt.axes([0.18, 0.05, 0.12, 0.075]), 'Cal/
↪Recal')
        self.button_showscatter = Button(plt.axes([0.375, 0.05, 0.1, 0.075]),␣
↪'Show')
        self.button_delscatter = Button(plt.axes([0.55, 0.05, 0.1, 0.075]),␣
↪'Delete')
        self.button_save = Button(plt.axes([0.725, 0.05, 0.1, 0.075]), 'Save')

    def _b_connect(self, con):

        if con:
            self.bc = self.button_cal.on_clicked(self.activate_cal)
            self.bss = self.button_showscatter.on_clicked(self.show_scatter)
            self.bds = self.button_delscatter.on_clicked(self.del_scatter)
            self.bs = self.button_save.on_clicked(self.save)
```

```
        else:
            self.button_cal.disconnect(self.bc)
            self.button_showscatter.disconnect(self.bss)
            self.button_delscatter.disconnect(self.bds)
            self.button_save.disconnect(self.bs)
```

### 0.9 Define a click function

The code below defines what to do if the user is clicking on the plot. After each click, `counter` is increased by 1. After the calibration button is pressed, the first two click is to draw the calibrate point by `cal()`. The rest of the click is to draw the label points. In the following code, it also prevents user from drawing the points if the user is holding and dragging the left mouse button. When the user is moving the mouse over a label point, the point will be automatically annotated by the `hover()`.

[ ]:
```
    def onclick(self, event):

        if event.inaxes == self.ax:
            if event.button == self.button:
                if self.cal_pressed and self.counter < 2 and self.activated:
                    self.cal(event, self.ax)
                elif self.cal_pressed and self.counter == 2 and self.activated:
                    self.scaling(self.x_before, self.y_before, self.x_after,␣
↪self.y_after)
                    ms(title='Info', message='Now you can start labeling...')
                elif self.cal_pressed and self.counter > 2 and self.activated:
                    self.label(event, self.ax)

        plt.pause(0.01) # avoid clicking too fast that causes crash

    def onpress(self, event):

        self.press = True

    def onmove(self, event):

        if self.press:
            self.move = True

        if event.inaxes == self.ax:  # hovering
            for p in self.labelpoint:
                boo = p.contains(event)[0]
                if boo:
                    self.hover(p)

    def onrelease(self, event):
```

```
        if self.press and not self.move:
            self.onclick(event)
        self.press = False; self.move = False
```

### 0.10 Activate calibration

The code defines what to do if the calibration button is pressed. It requires the user to input the direction of the new axes relative to the matplotlib axes. It requires re-input if the input value is not 1 (aligned) or 2 (reversed). It is noticed that the origin of the matplotlib coordinate system for imported images is set at the left-top corner by default, it means that the value on y axis increases from top to bottom, which must be considered into calculation. When the pop up window is cancelled without an input, which is the same as entering a `None`, the program will do nothing. It is important to reset some variables if cancel is pressed, the `counter` will not change so `activae_cal()` will run again (see `onclick()` in the previous block). It works the same way if the calibration button is pressed again, variables should be reset.

[ ]:
```
    def activate_cal(self, event):

        self.ax.figure.canvas.mpl_disconnect(self.press_event)  # disable the␣
↪press event when input
        self._b_connect(False)  # disable the buttons when input

        if self.cal_pressed:
            ms(title='Info', message='Now recalibration starts...')

        self.activated = False
        self.counter = 0
        self.x_before = []
        self.y_before = []
        self.x_after = []
        self.y_after = []
        self.x_cache = 9999
        self.y_cache = 9999

        self.annot.set_visible(False)

        try:  # try/except used here because calpoint has been removed after␣
↪scaling, remove twice will cause ValueError
            for p in self.calpoint:
                p.remove()

        except ValueError:
            pass

        for p in self.labelpoint:
            p.remove()
```

```
        self.calpoint = []
        self.labelpoint = []
        self.cal_pressed = True

        x_dir = qs(title='Enter the direction of calibrated X axis',
                   prompt='From left (negative) to right (postive) press 1,␣
↪otherwise press 2')

        while True:
            if x_dir == str(1):
                self.xdir = 1
                break
            elif x_dir == str(2):
                self.xdir = -1
                break
            elif x_dir == None:  # it is tested by print(x_dir) that None will␣
↪be returned when the cancel button is pressed
                self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
                                                                    self.
↪onpress)  # re-enable the press event
                self._b_connect(True)
                ms(title='Warning', message='Calibration paused !')
                return ''  # Then jump out of activate_cal and do nothing
            else:
                x_dir = qs(title='Enter the direction of calibrated X axis',
                           prompt='Invalid input ! From left (negative) to right␣
↪(postive) press 1, otherwise press 2')

        y_dir = qs(title='Enter the direction of calibrated Y axis',
                   prompt='From bottom (negative) to top (postive) press 1,␣
↪otherwise press 2')

        while True:
            if y_dir == str(1):
                self.ydir = -1  # the origin of matplotlib axis for import image␣
↪is at the top left corner
                break
            elif y_dir == str(2):
                self.ydir = 1
                break
            elif y_dir == None:
                self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
```

```
                                                                    self.
↪onpress)  # re-enable the press event
                self._b_connect(True)
                ms(title='Warning', message='Calibration paused !')
                return ''
            else:
                y_dir = qs(title='Enter the direction of calibrated Y axis',
                           prompt='Invalid input ! From bottom (negative) to top␣
↪(postive) press 1, otherwise press 2')

        ms(title='Info',
           message='Continue plotting two points and enter their calibrated␣
↪coordinates to finish calibration...')
        self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
                                                          self.onpress)  #␣
↪re-enable the press event
        self._b_connect(True)

        self.activated = True
```

### 0.11 Start calibration

After `self.activated = True` that means calibration is activated, now this block of code analyzes the first two clicks to draw two calibration points. When the user clicks on the plot, the program pops out a window to ask for the coordinates of the place he clicked in the new coordinate system. It parses the error when the input is not a digit or the cancel is pressed. It also compares the coordinates of calibration point 1 and 2. They should not be the same, otherwise the ratio between the scale of matplotlib axes and the scale of new axes will be wrong.

[ ]:
```
    def cal(self, event, ax):

        self.press = False
        self.ax.figure.canvas.mpl_disconnect(self.press_event)  # disable the␣
↪press event when input
        self._b_connect(False)  # disable the buttons when input

        x, y = event.xdata, event.ydata
        sc = ax.scatter(x, y, marker='x')
        ax.figure.canvas.draw()
        print(x, y)
        self.annot.xy = (x, y)
        text = 'Calpoint {:d}'.format(self.counter + 1)
        self.annot.set_text(text)
        self.annot.set_visible(True)

        while True:
```

```
            try:
                x_in = qs(title=None,
                          prompt='enter the calibrated X value for calibration␣
↪point %d: ' % (self.counter + 1))
                if float(x_in) == self.x_cache:
                    ms(title='Warning', message='Must enter a different X value !
↪ ! !')
                else:
                    self.x_cache = float(x_in)
                    self.x_after.append(float(x_in))
                    break
            except ValueError:
                ms(title='Warning', message='Invalid input ! ! !')
                pass
            except TypeError:  # if cancel is pressed, a TypeError is raised.␣
↪Then jump out of the cal and do nothing
                sc.remove() # remove the last calpoint drawn in the graph
                if self.x_before != []:
                    self.annot.xy = (self.x_before[-1], self.y_before[-1]) #␣
↪re-annotate calpoint 1 to give a clear view
                    text = 'Calpoint {:d}'.format(self.counter) # counter = 1
                    self.annot.set_text(text)
                else:
                    self.annot.set_visible(False)
                self.press = True
                self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
                                                                    self.
↪onpress)  # re-enable the press event
                self._b_connect(True)
                return ''

        while True:
            try:
                y_in = qs(title=None,
                          prompt='enter the calibrated Y value for calibration␣
↪point %d: ' % (self.counter + 1))
                if float(y_in) == self.y_cache:
                    ms(title='Warning', message='Must enter a different Y value !
↪ ! !')
                else:
                    self.y_cache = float(y_in)
                    self.y_after.append(float(y_in))
                    break
            except ValueError:
                ms(title='Warning', message='Invalid input ! ! !')
```

```
                pass
            except TypeError:
                self.x_cache = 9999
                del self.x_after[-1]  # remove the last x_after value from the␣
↪list
                sc.remove()

                if self.x_before != []:
                    self.annot.xy = (self.x_before[-1], self.y_before[-1]) #␣
↪re-annotate calpoint 1 to give a clear view
                    text = 'Calpoint {:d}'.format(self.counter) # counter = 1
                    self.annot.set_text(text)
                else:
                    self.annot.set_visible(False)

                self.press = True
                self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
                                                                    self.
↪onpress)  # re-enable the press event
                self._b_connect(True)
                return ''

        self.calpoint.append(sc)
        self.x_before.append(x)
        self.y_before.append(y)

        self.counter += 1

        self.press = True
        self.press_event = self.ax.figure.canvas.
↪mpl_connect('button_press_event',
                                                          self.onpress)  #␣
↪re-enable the press event
        self._b_connect(True)  # enable the buttons
```

## 0.12 Scaling, labelling and hovering

The scaling and labelling follows the underlying mathematical logic in the Approach block. The two calibration points is removed given that in most cases the calibration points are not useful (because it is easier to have more accurate calibration if the two points are on x and y axis respectively). In `hover()`, `get_offsets()` will return the coordinates of a matplotlib event.artist, combining with the `for` loop in `onmove()` to annotate the scatter points when the user is placing the mouse over the points. The label points are stored in a `labelpoint` list as matplotlib events, which can be either deleted or set invisible by `event.remove()` and `event.set_visible(False)`.

[ ]:
```
    def scaling(self, x_before, y_before, x_after, y_after):

        print(self.x_before, self.x_after)

        upper_x = x_after[1] - x_after[0]
        lower_x = x_before[1] - x_before[0]
        self.scale_x = abs(upper_x / lower_x)  # ratio of displacement between␣
↪matplotlib axis and image axis
        print('scale x:', self.scale_x)

        upper_y = y_after[1] - y_after[0]
        lower_y = y_before[1] - y_before[0]
        self.scale_y = abs(upper_y / lower_y)
        print('scale y:', self.scale_y)

        self.origin_x = x_before[1] - self.xdir * x_after[1] / self.scale_x
        self.origin_y = y_before[1] - self.ydir * y_after[1] / self.scale_y
        print('new origin:', self.origin_x, self.origin_y)

        self.counter += 1

        self.annot.set_visible(False)

        for p in self.calpoint:
            p.remove()

    def label(self, event, ax):

        x, y = event.xdata, event.ydata
        self.labelpoint.append(ax.scatter(x, y, marker='x'))
        for p in self.labelpoint:
            p.set_visible(True)
        ax.figure.canvas.draw()
        new_x = (x - self.origin_x) * self.scale_x * self.xdir
        new_y = (y - self.origin_y) * self.scale_y * self.ydir
        print(new_x, new_y)
        self.output.append([new_x, new_y])
        self.annot.xy = (x, y)
        text = "({:.2g}, {:.2g})".format(new_x, new_y)
        self.annot.set_text(text)
        self.annot.set_visible(True)

    def hover(self, p):

        [[x, y]] = p.get_offsets()

        new_x = (x - self.origin_x) * self.scale_x * self.xdir
```

```
        new_y = (y - self.origin_y) * self.scale_y * self.ydir

        try:
            self.annot.xy = (x, y)
            text = '({:.2g}, {:.2g})'.format(new_x, new_y)
            self.annot.set_text(text)
            self.annot.set_visible(True)
            self.ax.figure.canvas.draw()

        except ValueError:
            pass
```

### 0.13 Button: Show, delete and save

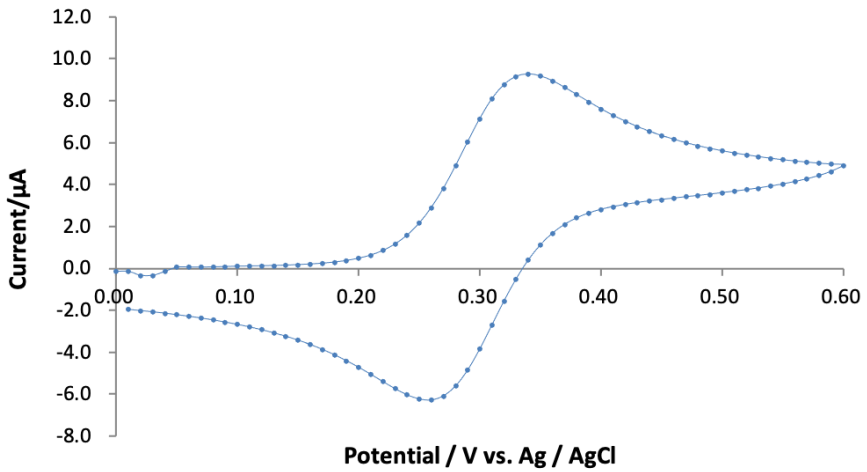
`show_scatter()` allows the annotation and scatter to be hidden or shown. `del_scatter()` deletes all label points in the plot. `save()` uses `filedialog.asksaveasfile()` to choose a local directory and save all label points by `csv.writer.writerow()` in a csv file. All the three functions iterate every events (scatters) in the `labelpoint` list and the `calpoint` list.

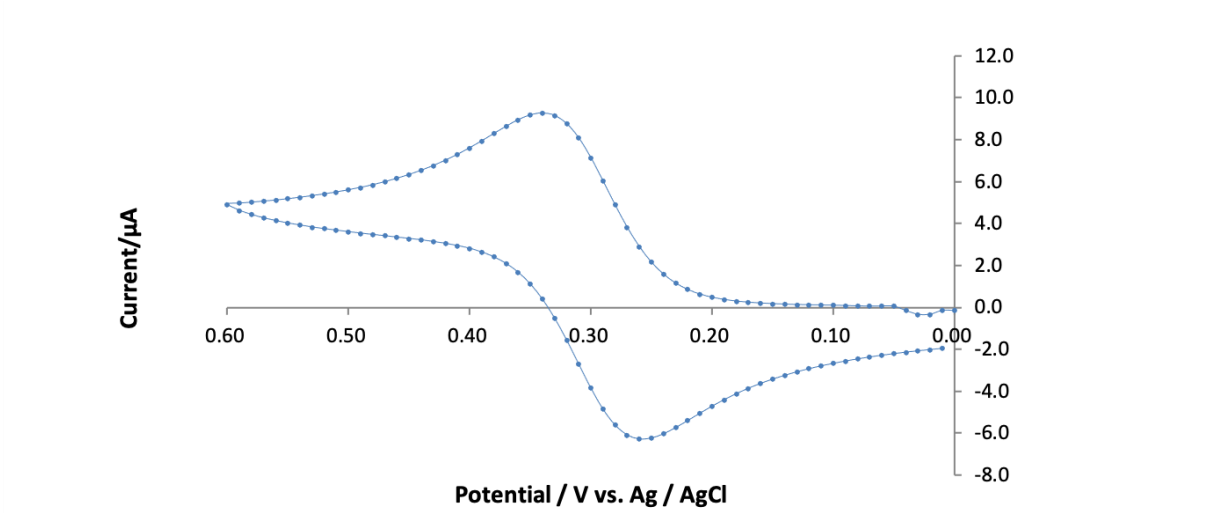
[ ]:
```
    def del_scatter(self, event):

        self.annot.set_visible(False)
        try:
            for p in self.labelpoint:
                p.remove()
        except ValueError:
            pass

        self.labelpoint = []
        self.output = []

    def show_scatter(self, event):

        if self.showscatter_pressed:
            self.annot.set_visible(True)
            for p in self.calpoint:
                p.set_visible(True)
            for p in self.labelpoint:
                p.set_visible(True)
            self.showscatter_pressed = False
        else:
            self.annot.set_visible(False)
            for p in self.calpoint:
                p.set_visible(False)
            for p in self.labelpoint:
                p.set_visible(False)
            self.showscatter_pressed = True
```

```
    def save(self, event):

        file = sf(initialfile='Untitled.csv',
                defaultextension='.csv',
                title='Save as',
                filetypes=[("All Files", "*.*"), ("Comma-Separated Values File",␣
↪"*.csv")])

        f = csv.writer(file)

        try:
            for p in self.output:
                f.writerow(p)
            file.close()
            ms(title='Info', message='Data saved !')

        except AttributeError:# when cancel is pressed
            pass
```

## 0.14 Tests and Sample outputs

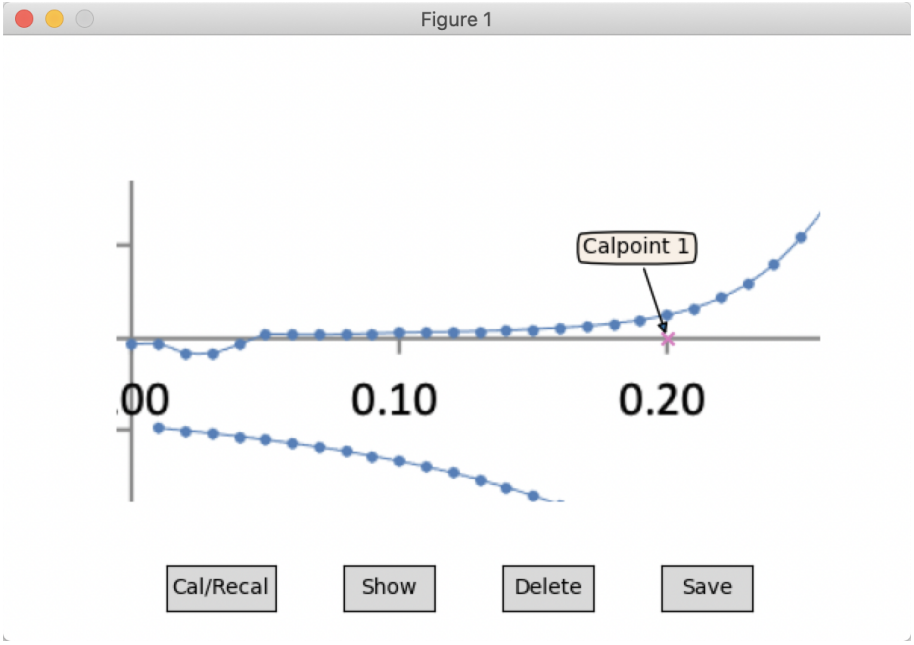
Two image files with normal axes and inverted x axis respectively are tested. They are produced using Microsoft Excel and the raw data are real experimental data from the CHEM0025/26 electrochemistry experiment. Each image is calibrated and labelled with three random points, the output csv file is produced.

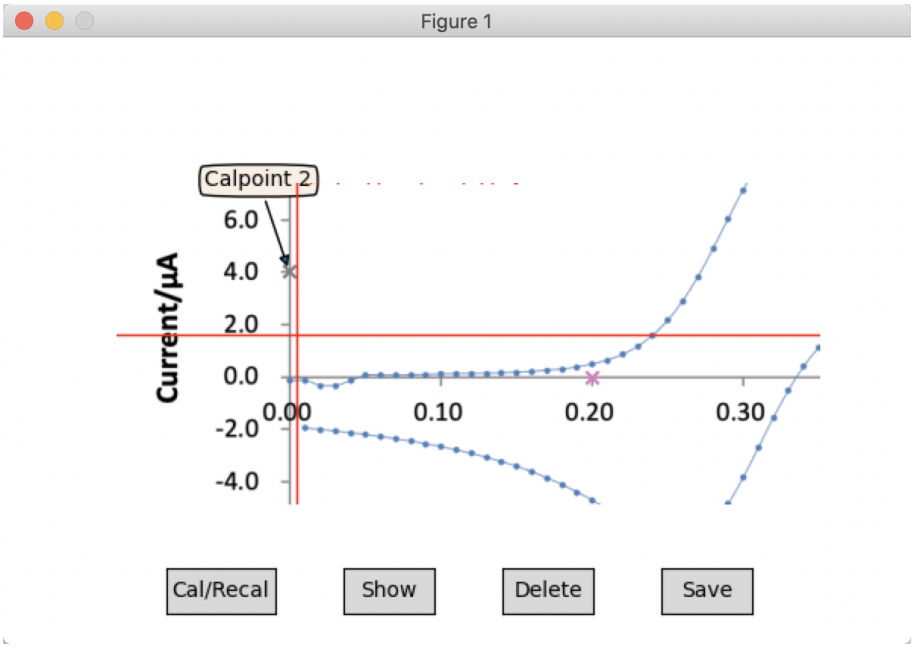


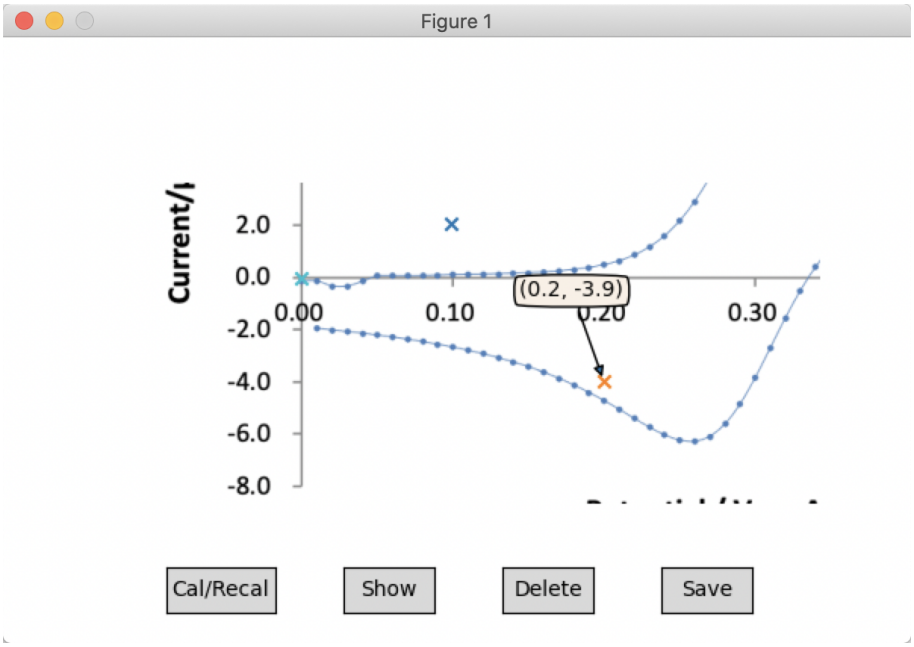



### 0.14.1 IMG 1

It is calibrated at (0.2, 0) and (0, 4) and intended to label the points at (0,0), (0.1, 2) and (0.2, -4).






| Point | x | y |
|---|---|---|
| (0, 0) | 0.000415876 | -0.001691562 |
| (0.1, 2) | 0.099119024 | 2.049390059 |
| (0.2, -4) | 0.199773484 | -3.896520849 |

### 0.14.2 IMG 2

It is calibrated at (0.3, 0) and (0, 6) and intended to label the points at (0,0), (0.2, 8) and (0.1, -2).

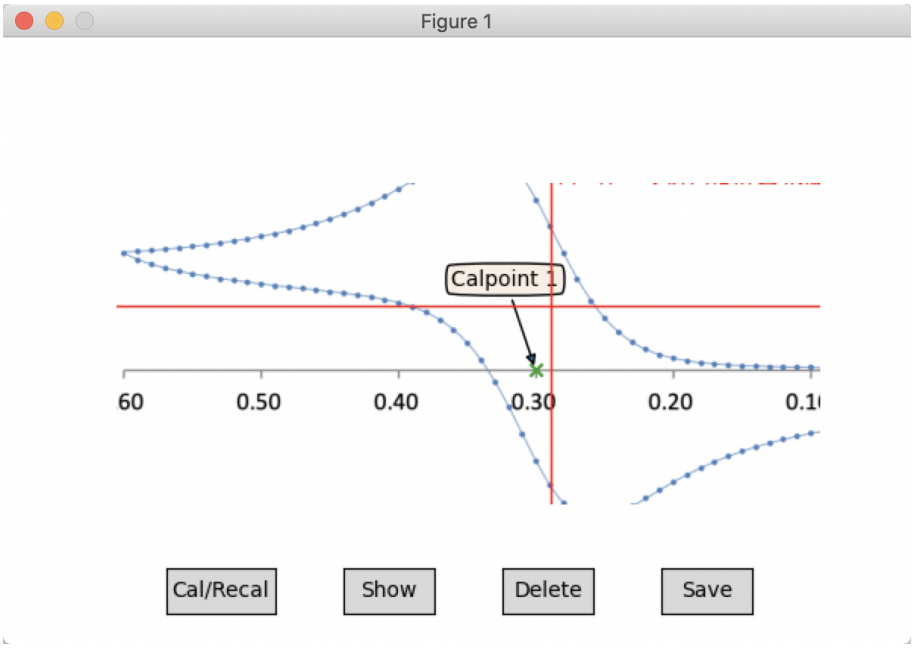


Figure 1
Calpoint 1
60
0.50
0.40
0.30
0.20
Cal/Recal
Show
Delete
Save

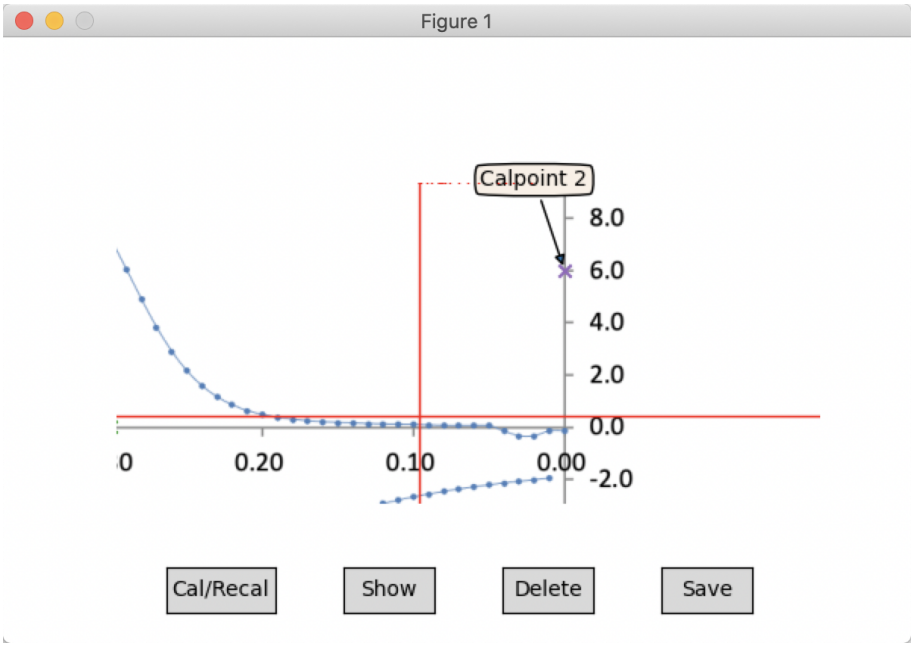
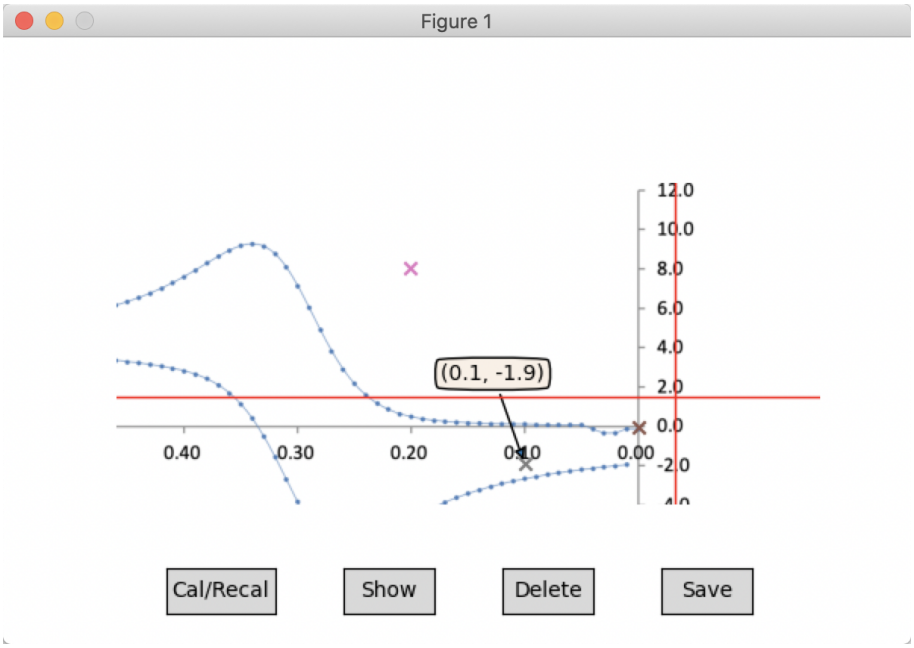


Figure 1
Calpoint 2
8.0
6.0
4.0
2.0
0.0
-2.0
0
0.20
0.10
0.00
Cal/Recal
Show
Delete
Save




| Point | x | y |
|---|---|---|
| (0, 0) | -0.000227714 | -0.018087366 |
| (0.2, 8) | 0.200517667 | 8.00656048 |
| (0.1, -2) | 0.099788754 | -1.929644544 |

By comparing the expected output and the actual output, it proves that the program can digitalize the plot well. The sign of x and y is correct and the value of error is far less than 1% in most cases. The error of output strongly depends on the accuracy of labelling on the screen, the user can get better output if he zooms in the plot and label carefully.

## 0.15 Installation

- To start the PlotDigitizer as .app in MacOS:

  Free online download:

  https://liveuclac-my.sharepoint.com/:u:/g/personal/zccarli_ucl_ac_uk/Ecc850WYIbBJpY5_jqSdEN8BmWvpa8092f37UaTmriBHsw?e=zMYshF

  Unzip the file and double click the PlotDigitizer.app to start

- To start the PlotDigitizer as .py in IDE:

  The libraries that you need:

  os, sys, stat, subprocesses, matplotlib, PIL, tkinter, mpl_interactions, csv, numpy

  Place main.py and plot_digitizer.py in the same folder (working directory) and run main.py in IDE

- To run the main.py in command line interface:

Open the command line interface as administrator, enter <cd /the/working/directory>, in the next line enter <python main.py>

### 0.16 Usage

PlotDigitizer is now started and two options are displayed:

- <OPEN>

  Browse image files in your computer, formats such as png, jpg, gif are accepted

- <README>

  Open README.txt to help

After an image file is opened, it will be displayed by the matplotlib module using TkAgg backend which is compatible with the Tkinter GUI. Then you will see 4 buttons and the built-in scrolling, zooming and panning functions provided by mpl-interactions:

- <Cal/Recal>

  Start calibrating the axis, press again at any moment (unless a pop up input dialog is displayed) to recalibrate

- <Show>

  Show/hide the scatter points and annotations

- <Del>

  Only available after calibration is complete. Delete all label points

- <Save>

  Print out all label points (exclude calibration points), save as a csv file in your computer

- <Scrolling, zooming and panning>

  Scroll to zoom the plot and hold right mouse button to pan

Please follow the instructions of the pop up messages.

After calibration is complete, click any place in the plot to start labelling.

### 0.17 Improvements

Theoretically, the user gets 100% the same coordinate system expected in the plot if the calibration is completed without errors. The improvements should be focused on how to rebuild the plot efficiently. It is obvious that labelling all the data points by hands is time-consuming for a complex graph, the above plots are good example. In most journal plots, the background is white and the data points / curves are black, which provides an excellent platform for extraction by color. It solves the problems when only curves (no data points) are shown in the plot. The time cost is less scaled than that in the manual labelling (this programme) but it requires more advanced computational knowledge instead. Nevertheless, this program acts as a good approach for those who wants to analyze simple journal plots.