ELEC362
Project - 1D Graphs Digitiser

Name: Junhao Zhang
Student ID: 201377244

Content

1 How the Programme Works

The basic idea of the algorithm of the 1D graphs digitiser is to open an image file, then let the user click on certain points to calculate the calibration factor based on the input values that the user has been input. Once the programme complete the calibration, it allows the user to click on the graph and transfer the pixel coordinates to the real-world coordinates based on the calibration. The following list illustrates how the programme works in detail:

- The first step is to open an image file in order to calibrate prior to measuring the graph. The programme loads the image file and stores it for any further operation. The programme will automatically look for .jpg and .png images at first.
- The next step is to let the user calibrate on the image. The progarmme calculates the calibration factor based on the four clicked pixels on the graph and the real-world values that the user has input.
- When completing the calibration, the programme uses the calibration factor to calculate the real-world coordinate of any pixel that the user clicks based on the calibration factor. A QVector is used to stores all the data points that the user clicks and they can also be exported as text file if it is necessary.
- Additionally, there is an automatic measuring mode where the programme will automatically collect many valid points on the graph in the range set by the user.

The most important part of the programme is the calculation of the calibration factor. In order to do that, the following formula is used:

$$calibration\ factor = \frac{real\ length}{pixel\ length}$$

The formula above calculates the calibration which has the unit: real length/pixel. The pixel length is calculated by subtracting the y pixel coordinates of two clicked points on the graph. The real length is based on the graph given by the user, and any valid number can be accepted by the programme.

Another necessary parameter for the calibration Is the offset. It is of large possibility that the origin of the pixel coordinate is quite different from the real-world coordinate, therefore, it is necessary to calculate the offset between the two coordinates:

$$offset = real\ length\ from\ the\ origin \\ -\ calibration\ facotr \times pixel\ length\ from\ the\ origin$$

Once these two parameters are calculated based on the graph, the programme will be able to convert any pixel coordinates to real-world coordinates based on the graph using the formula:

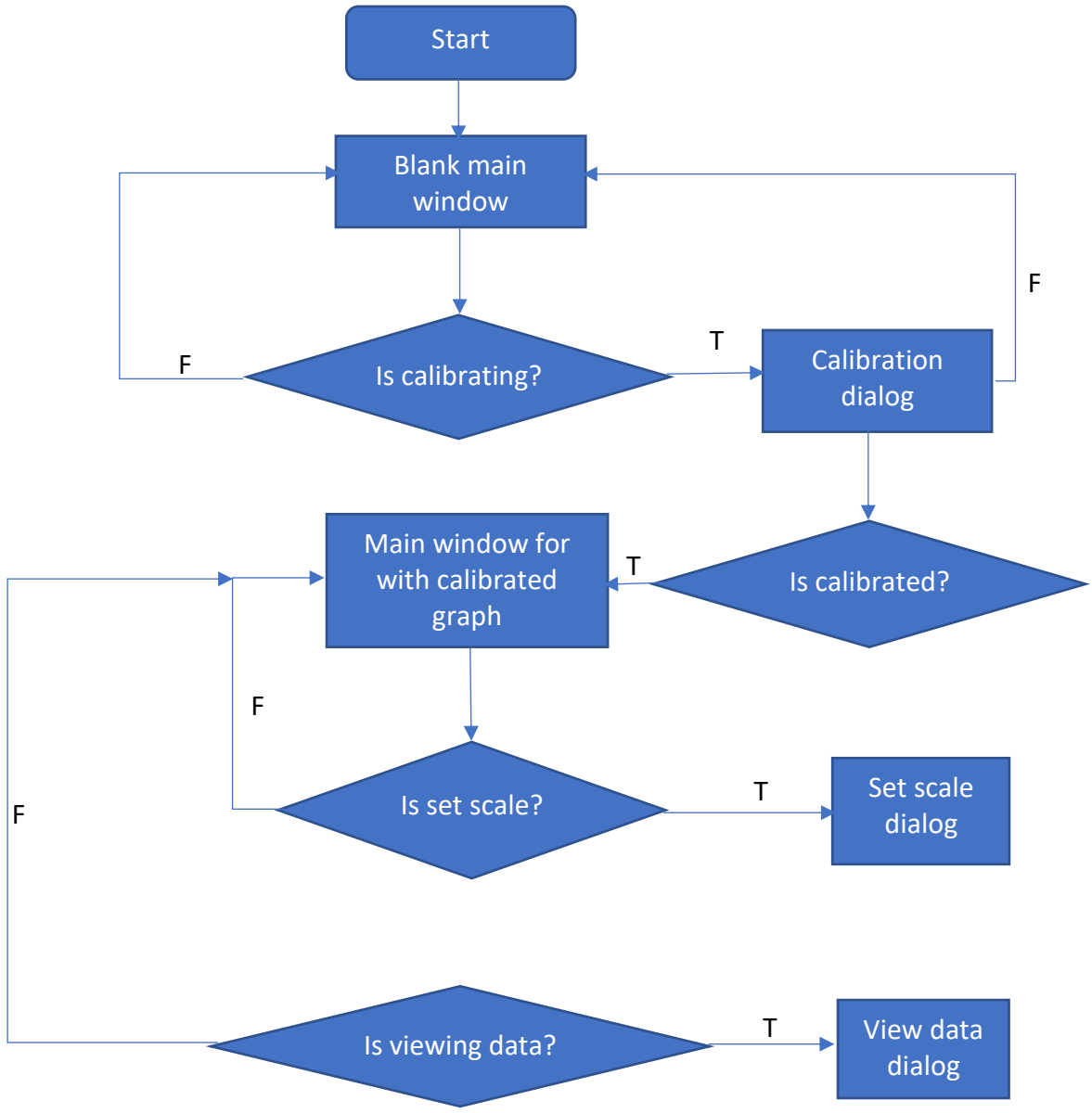
$$linear\ coordinate = offset + pixel\ coordinate \times calibration\ factor$$

And for log scale coordinate:

$$log\ coordinate = 10^{offset + pixel\ coordinate \times calibration\ factor}$$

1.1 Window Invoking Algorithm

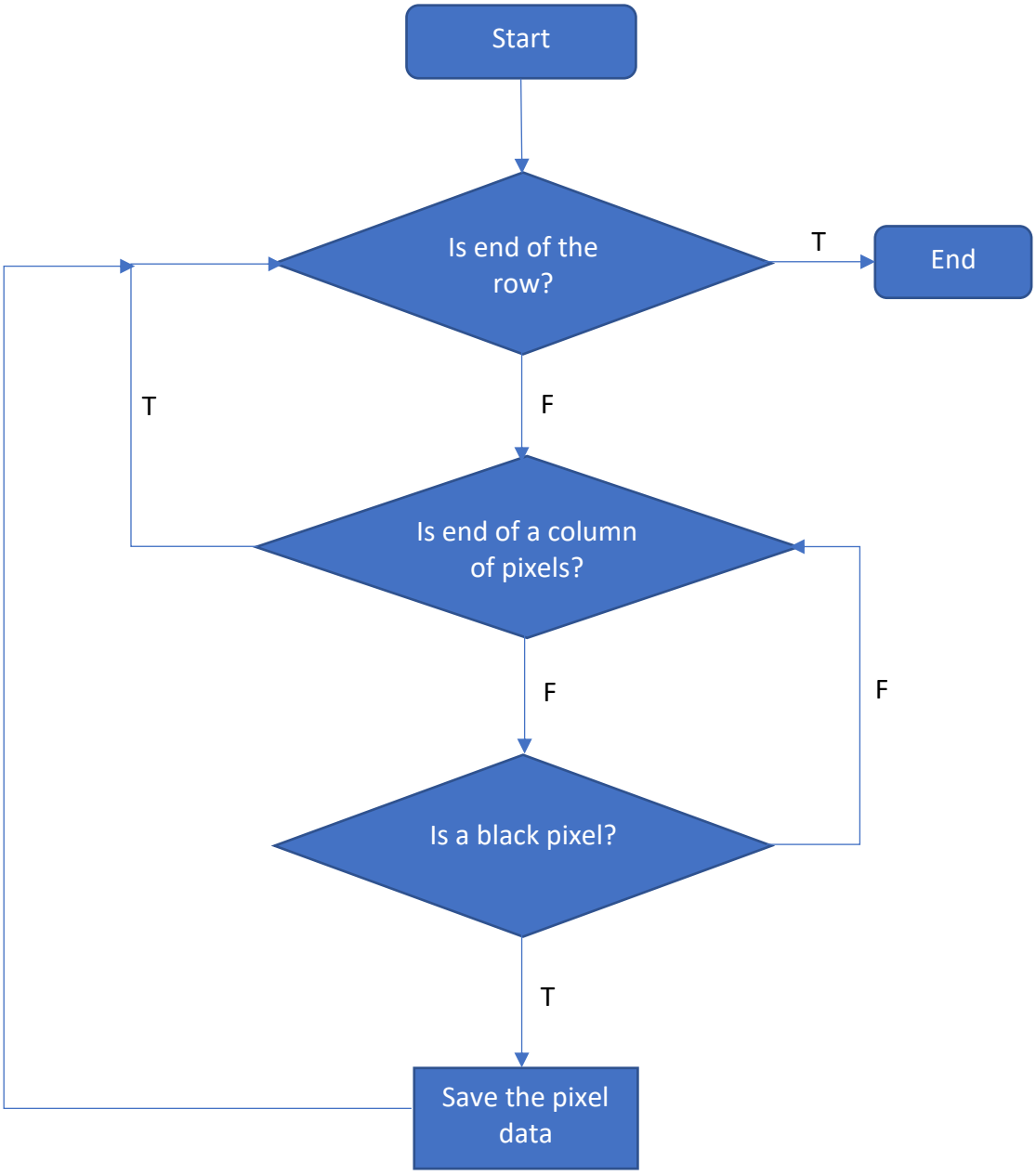
The following flow chart shows the overall algorithm of the programme and shows what window is called at different conditions.



As shown in the flow chart, there are 4 different windows in total for different purposes when called. The main window is always at presence and the other 3 dialog windows are only called when it is operated by the user. It is worth noting that the main window has two different state: blank state and image loaded state. In blank state, the user must firstly add a calibration to the programme and the main window will load the calibrated graph and become the window with a loaded image for measuring.

1.2 Automatic Mode Algorithm

The programme has an automatic measuring mode that tracks the points on the graph automatically. The main idea to implement this is based on the colour different between the line and the background of the graph. As specified in the project description, the plotted line is solid and has a black colour against a white background. Therefore, it is possible to let the programme look for black pixels from white pixels in a specified range set by the user.



As illustrated in the flow chart, the automatic mode is constituted of two loops: a main loop for scanning the row coordinate and a nested loop for scanning a single column. The loop start to traverse the first column of pixel and try to find a black pixel then recorded its corresponding parameters, then the outer loop move to the next column, allowing the inner

loop to traverse again the next column to look for the next black pixel. As a result, all the black pixels of the graph within the selected range will be recorded, thereby the line Is automatically measured by the programme.

2 User Instructions

2.1 Calibration

The Graph Digitiser software has a main window and 3 dialog windows. Once the software is opened, the user should see a black main window shown in Figure 1.

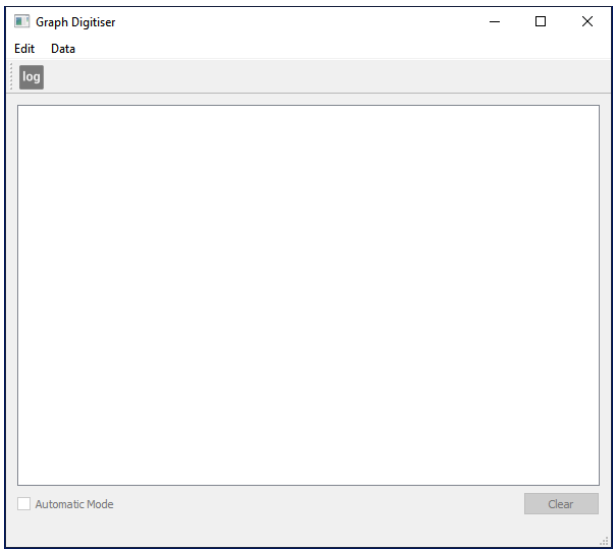


Figure 1. Blank Main Window

In this window, the user is only allowed to import a graph for calibration first, before which any other operation Is not allowed. The user should go to **Edit->Add Calibration** to add a graph for calibration. Once clicked, the calibration dialog window will show up as shown in Figure 2.

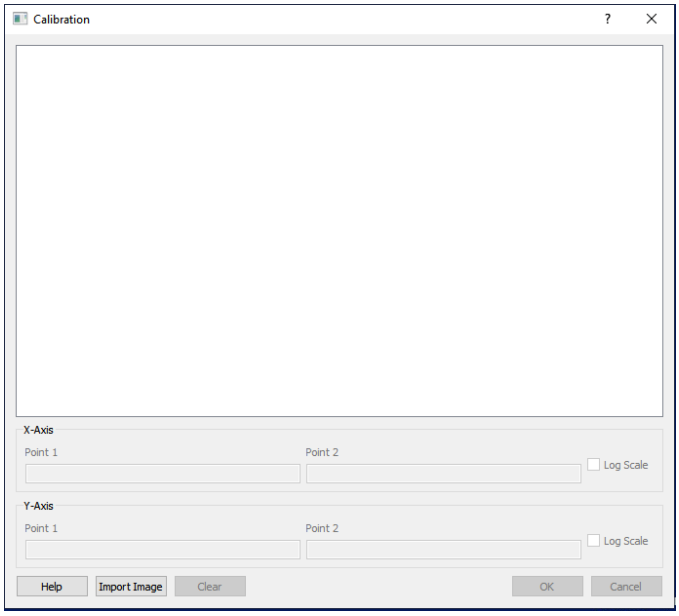


Figure 2. Calibration Dialog

By clicking **Import Image** button, the user can import a graph with a suffix of .jpg or .png for calibration.

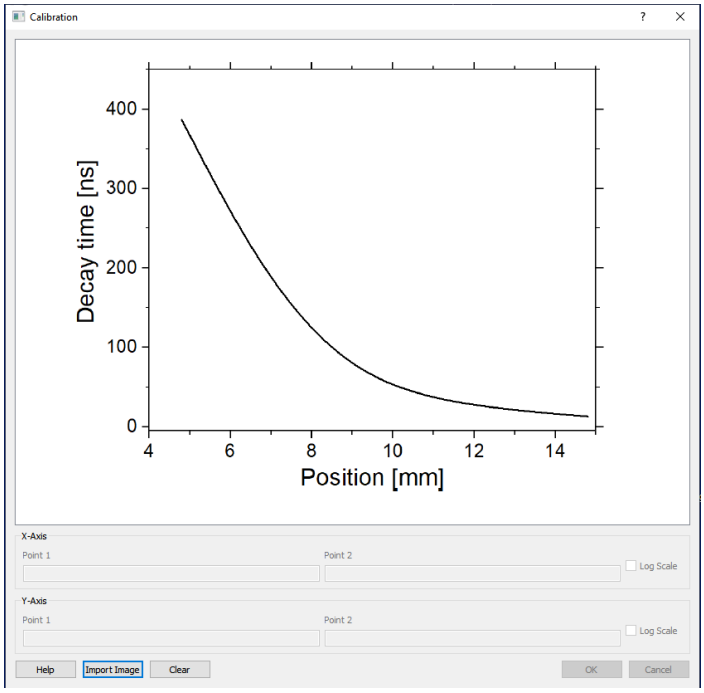


Figure 3. Graph Imported

When an image is successfully imported to the software, the calibration dialog window will show the imported Image and allow the user to click on the graph for calibration. There are totally four points need to be calibrated on the graph: two for the x-axis and two for the y-axis. The order of which the points are clicked is important so it is of great important for the user to follow:

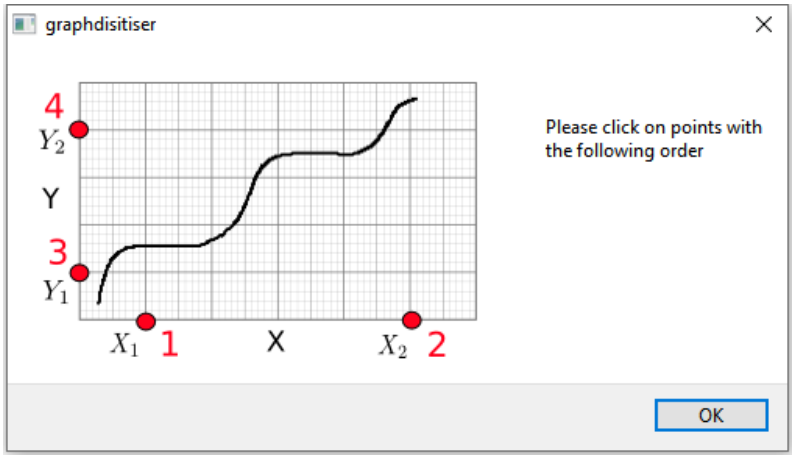


Figure 4. Help Message

The user can also click the **Help** button to review this introduction of how to calibrate the axes. After 4 points are clicked, the software will allow the user to enter the corresponding coordinates of each points, so that it can calibrate the graph from pixel to its real-world coordinates.

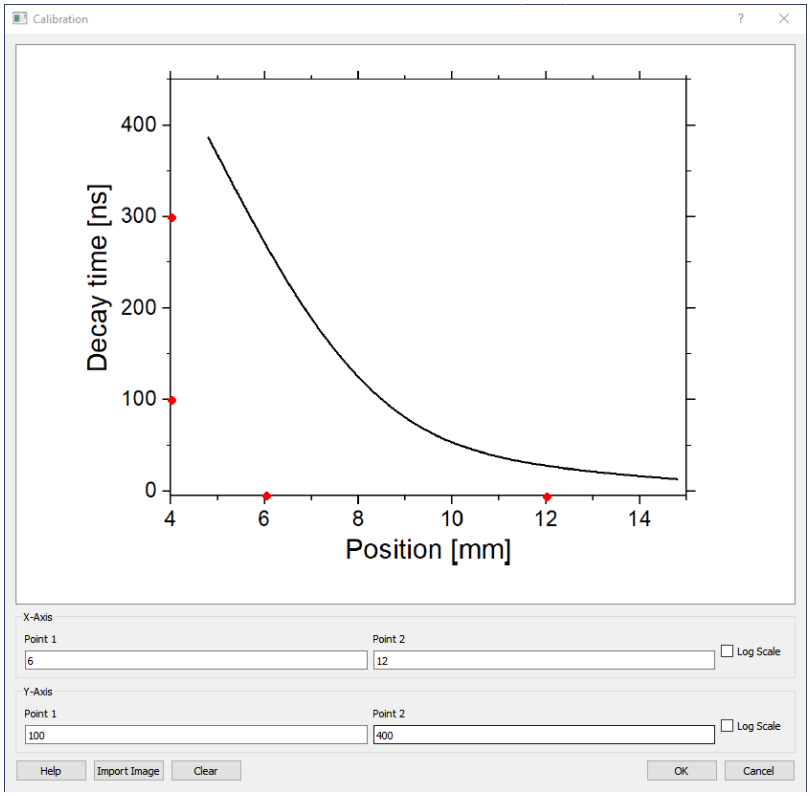


Figure 5. Calibrating

As shown in Figure 5, the software takes the input from the user to calibrate the x-y axes. At this stage, the user can either click **OK** to complete the calibration or click **Clear** to clear all current input, including all the clicked points on the graph. Additionally, there are two checkboxes that allow the user to determine whether the axis is log scale or linear scale.

It is worth noting that the user cannot enter two identical value for the two points of the same axis, as the software will not accept this kind of input:

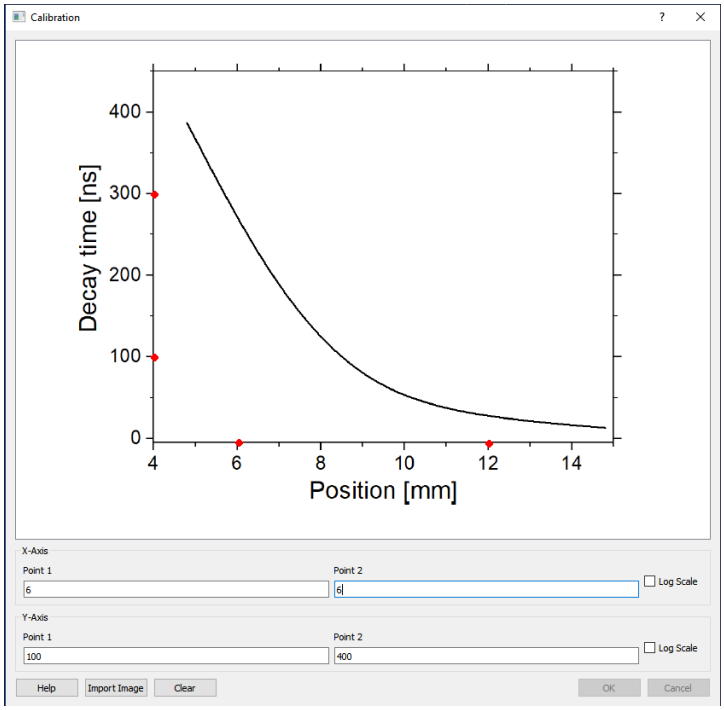


Figure 6. Wrong Values

As shown in Figure 6, the **OK** button will be disabled if the software detects any invalid input.

2.2 Manual Digitising

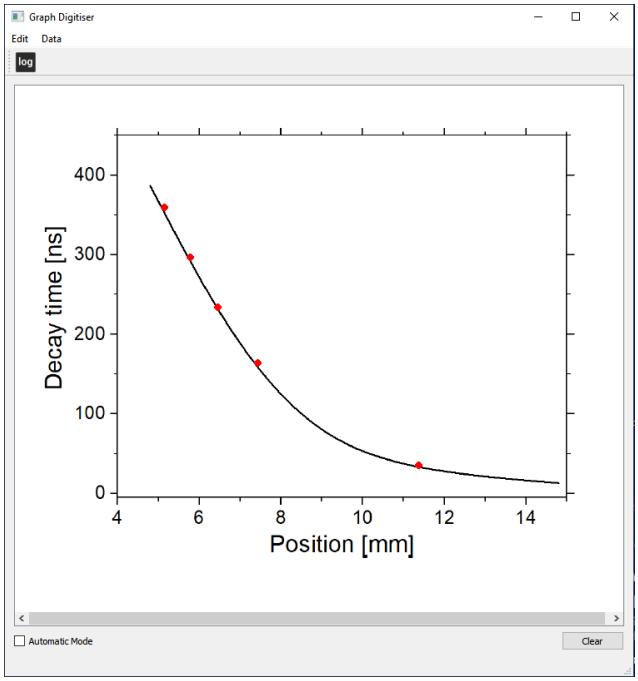


Figure 7. Main Window with Image

Once calibrated, the software will come back to the main window with the previously calibrated graph. The user is now able to click on any points of the graph and the corresponding coordinates will be recorded. By going to **Data->View Data**, the user can view

the coordinates of current clicked points on the graph. The user can also export the data set by going to **Data->Export Data** and the software will allow the user to save the data set to the computer as text file including .csv or .txt file.

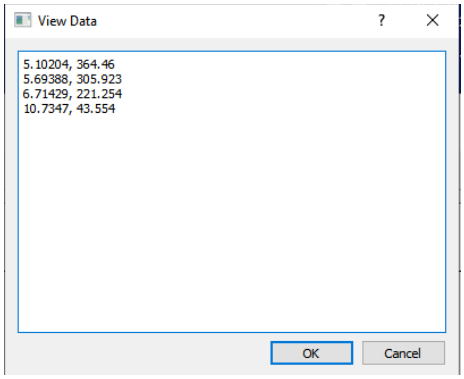


Figure 8. Coordinates of Clicked Points

The user is able to change the calibrated axes between linear scale and log scale whenever it is necessary. There are 3 ways to do this:

1. Select the **Log Scale** checkbox at the calibration dialog to calibrate the axis as log scale.
2. **Right-click** on the graph at main window and select **Set Scale**.
3. Go to **Edit->Set Scale**.

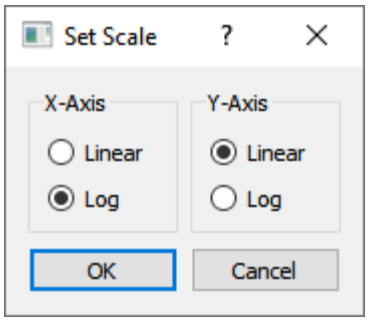


Figure 9. Set Scale Dialog

2.3 Automatic Digitising

The software also has an automatic mode which allows the user to set a range and it will measure the intermediate points automatically without manually operation. To enable this function, the user should select the **Automatic Mode** checkbox on the main window:

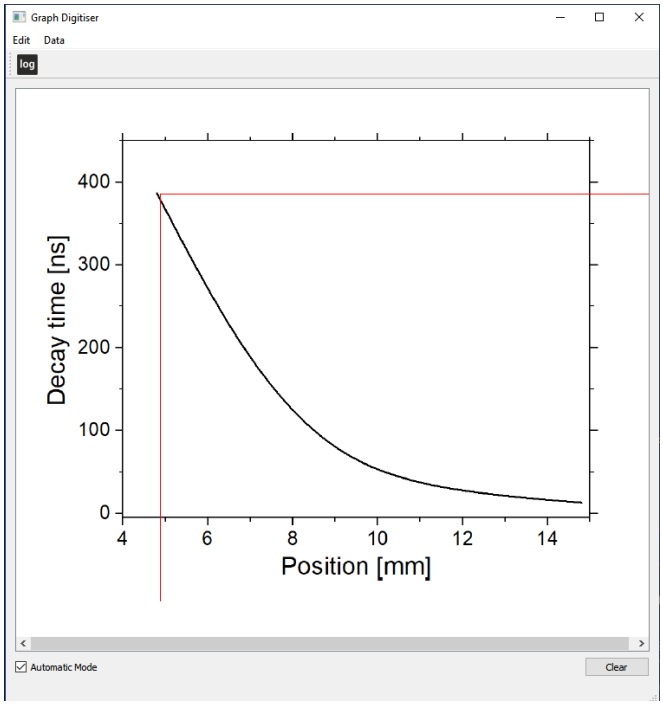


Figure 10. Auto Mode 1

In auto mode, the user should firstly select one upper-left point as the start of the range.

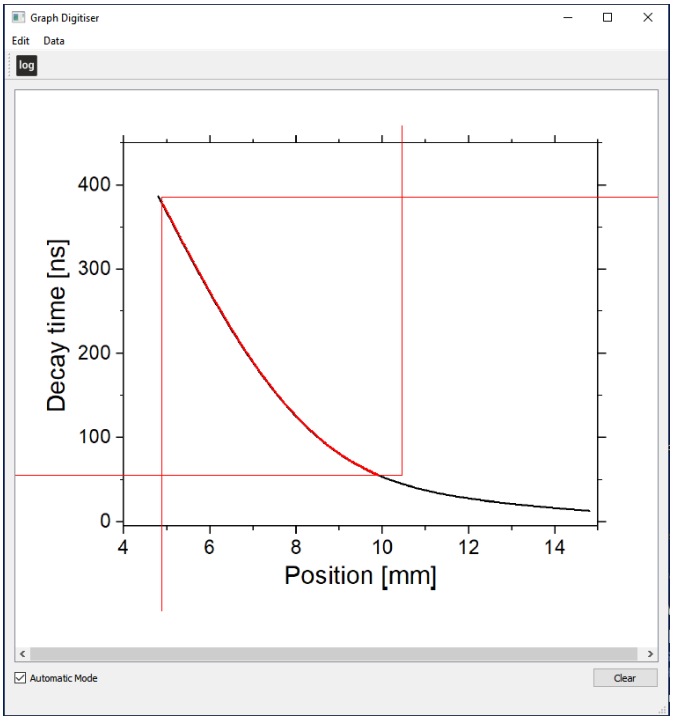


Figure 11. Auto Mode

Then the user should select one lower-right point to complete the range. As shown in Figure 10, the line inside the rectangle is then automatically plotted by the programme and the relating data set is generated.

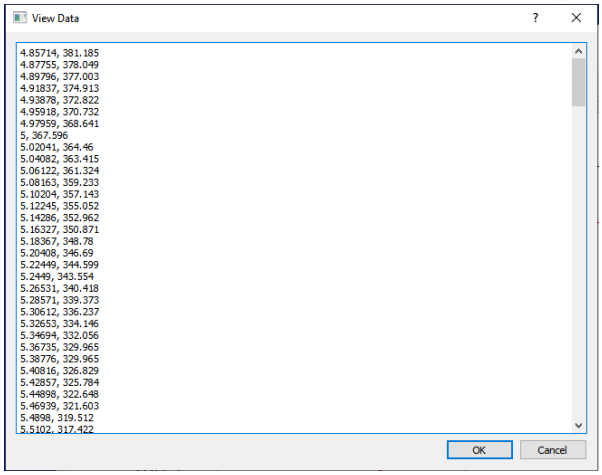


Figure 12. Auto Mode Data

3 Testing and Verification Attempts

Figure 12 is used as the test graph to test the functionality of the software, for both the manual digitising and the automatic digitising.

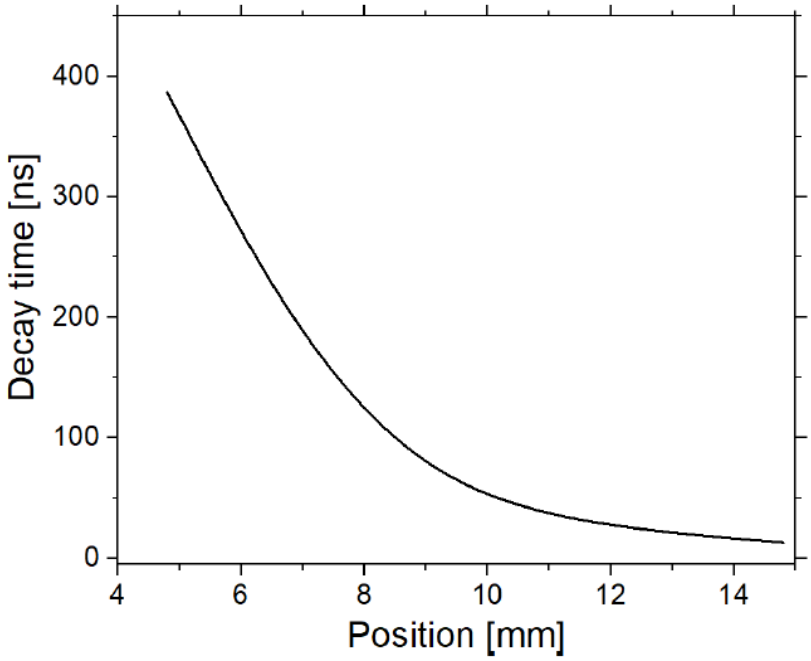


Figure 13. Test Graph

For the manual digitising, a number of points on the line are clicked and the data set is stored to plot another graph in order to compare the original graph and the graph with the digitising data.

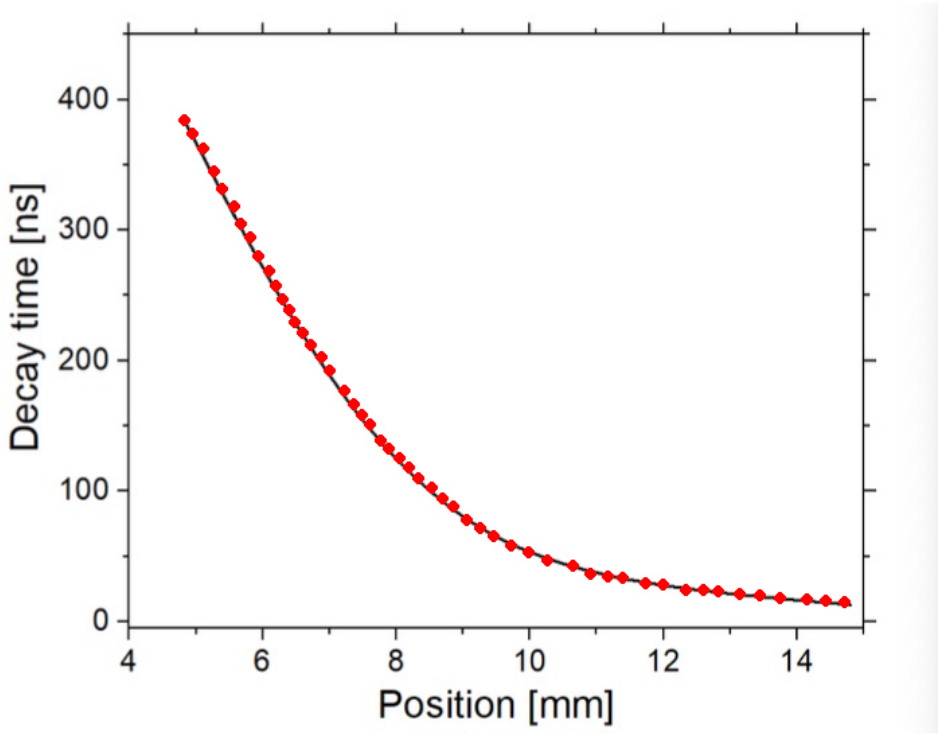


Figure 14. Clicked Points

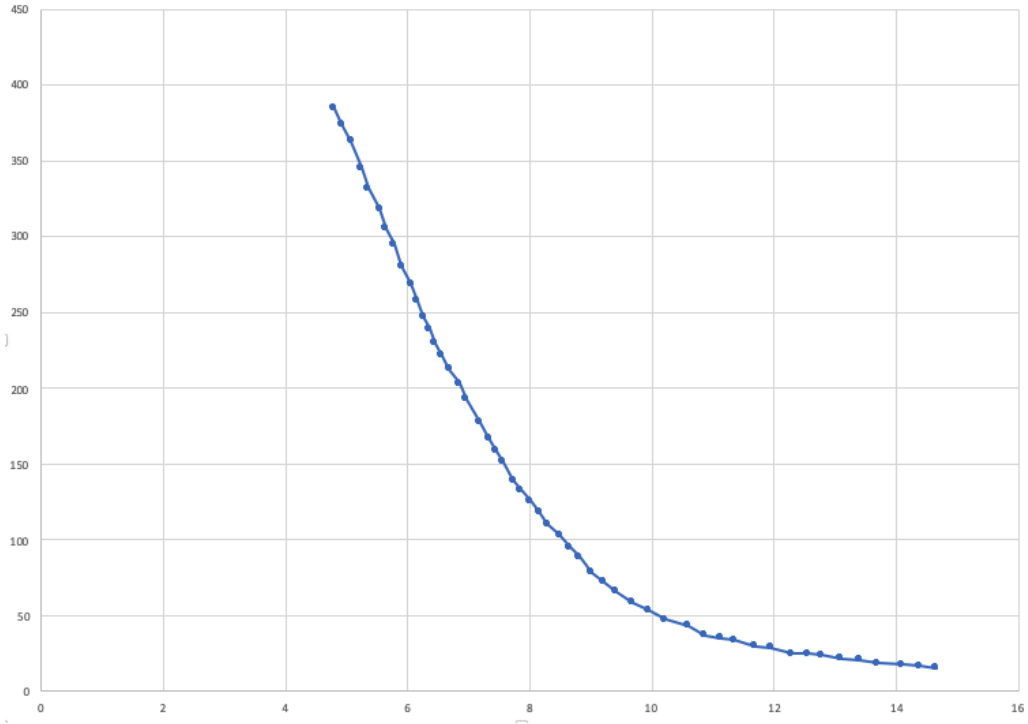


Figure 15. Generated Graph

And the automatic digitising function is also tested:

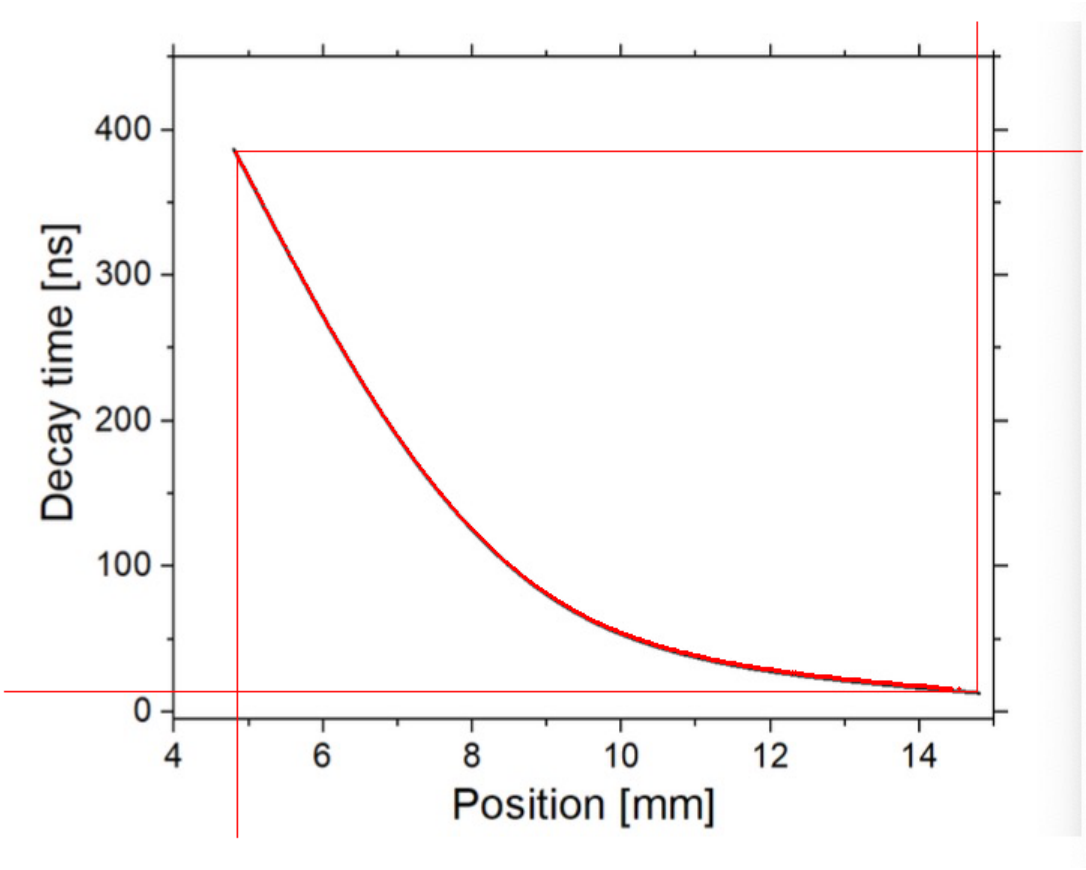


Figure 16. Automatically Digitised Points

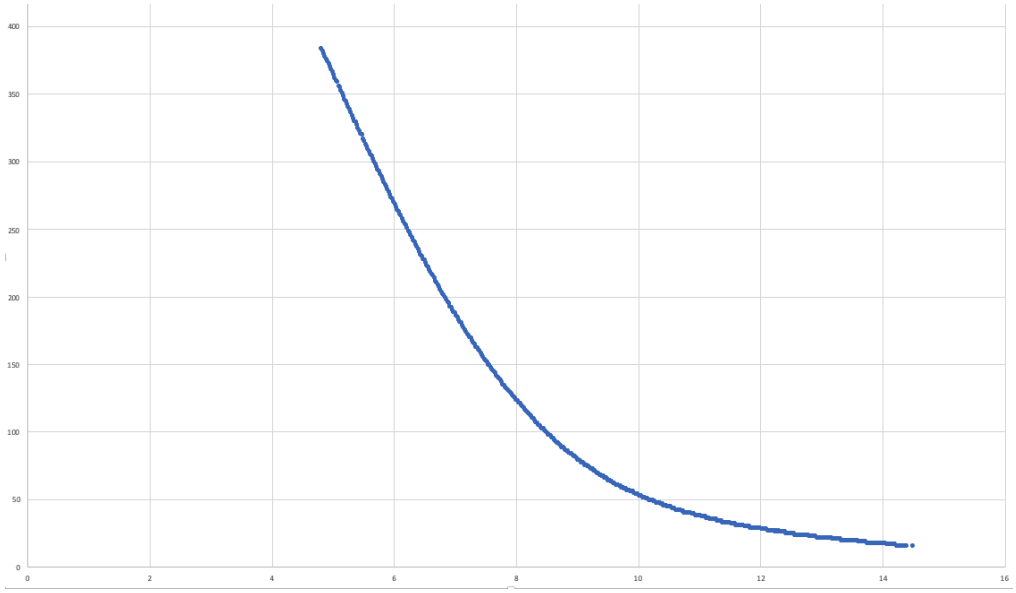


Figure 17. Generated Graph

As shown in the figure above, the digitised data from the original graph are both accurate to restore the line of the original graph.

Next it is necessary to test the log scale function to ensure that the software can convert the axes scale correctly.

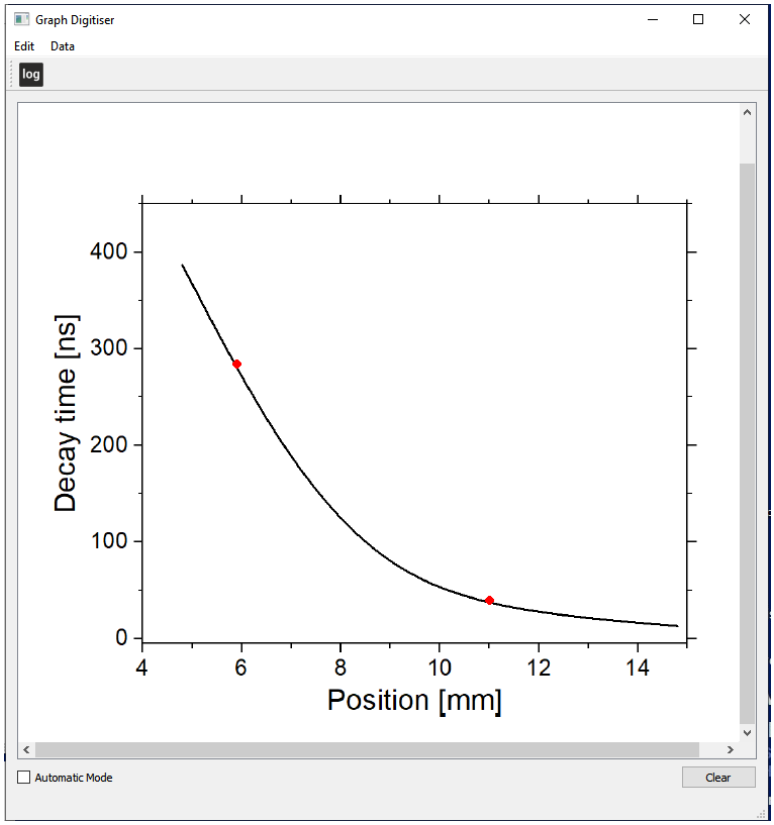


Figure 18. Log Scale

In Figure 18, the x-axis is changed to log scale and Figure 19 shows the coordinates of the two points:

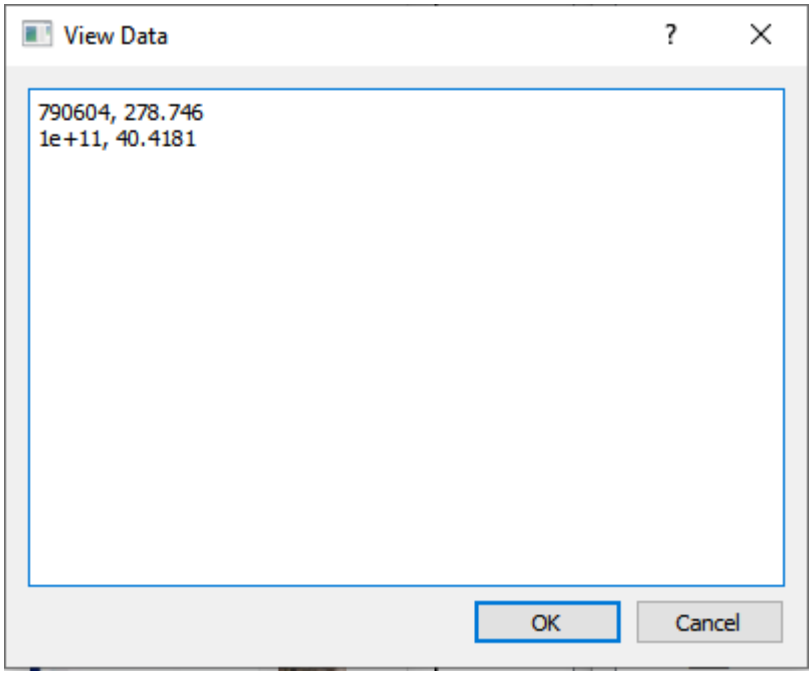


Figure 19. Data Set

As shown in Figure 19, the x-axis coordinates is successfully converted to log scale for this calibrated graph.

4 Appendix: source code

4.1 "calibrationdialog.h"

```
#ifndef CALIBRATIONDIALOG_H
#define CALIBRATIONDIALOG_H

#include <QDialog>
#include <QGraphicsScene>
#include <QtMath>
#include <QPixmap>
#include <QFileDialog>
#include <QVector>
#include <QPointF>
#include <QMouseEvent>
#include <QDoubleValidator>
#include <QGraphicsEllipseItem>
#include <QMessageBox>

namespace Ui {
class CalibrationDialog;
}

class CalibrationDialog : public QDialog
{
    Q_OBJECT

public:
    explicit CalibrationDialog(QWidget *parent = nullptr);
    ~CalibrationDialog();

    void mousePressEvent(QMouseEvent *event);

    void calibrationCheck();

    double getXCalibrationFactor() {return x_calibration_factor;}
    double getYCalibrationFactor() {return y_calibration_factor;}
    double getXOffset() {return x_offset;}
    double getYOffset() {return y_offset;}
    bool getIsXLog() {return isXLog;}
    bool getIsYLog() {return isYLog;}

    void setXCalibrationFactor(bool log) {isXLog = log;}
    void setYCalibrationFactor(bool log) {isYLog = log;}

    bool getIsCalibrated() {return isCalibrated;}

    QString getFileName() {return filename;}


private slots:

    void on_CalibrationDialog_accepted();
```

```
    void on_lineEdit_x_point_1_textEdited(const QString &text_x_1);

    void on_lineEdit_x_point_2_textEdited(const QString &text_x_2);

    void on_lineEdit_y_point_1_textEdited(const QString &text_y_1);

    void on_lineEdit_y_point_2_textEdited(const QString &text_y_2);

    void on_checkBox_x_stateChanged(int arg1);

    void on_checkBox_y_stateChanged(int arg1);

    void on_importImageButton_clicked();

    void on_helpButton_clicked();

    void on_clearButton_clicked();

private:
    Ui::CalibrationDialog *ui;

    QGraphicsScene *p_scene = new QGraphicsScene(this);

    QPixmap *p_map = new QPixmap();

    QMessageBox tip;

    QVector<QPointF> calibrationVector;

    QString filename;

    QString *p_x_point_1 = nullptr;
    QString *p_x_point_2 = nullptr;
    QString *p_y_point_1 = nullptr;
    QString *p_y_point_2 = nullptr;

    bool isXLog = false;
    bool isYLog = false;
    bool isCalibrated = false;
    bool isCalibrating = false;

    bool hasTextX1 = false;
    bool hasTextX2 = false;
    bool hasTextY1 = false;
    bool hasTextY2 = false;

    double x_point_1_num;
    double x_point_2_num;
    double y_point_1_num;
    double y_point_2_num;

    double x_offset;
    double y_offset;
    double x_calibration_factor;
    double y_calibration_factor;

    void _disableWidgets();
};

#endif // CALIBRATIONDIALOG_H
```

4.2 "callibrationdialog.cpp"

```
#include "calibrationdialog.h"
#include "ui_calibrationdialog.h"

CalibrationDialog::CalibrationDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::CalibrationDialog)
{
    ui->setupUi(this);
    ui->graphicsView->setScene(p_scene);
    ui->graphicsView->setCursor(Qt::CrossCursor);

    ui->lineEdit_x_point_1->setValidator(new
QDoubleValidator(0,1000,3,this));
    ui->lineEdit_x_point_2->setValidator(new
QDoubleValidator(0,1000,3,this));
    ui->lineEdit_y_point_1->setValidator(new
QDoubleValidator(0,1000,3,this));
    ui->lineEdit_y_point_2->setValidator(new
QDoubleValidator(0,1000,3,this));

    ui->clearButton->setEnabled(false);
    ui->buttonBox->setEnabled(false);
    ui->checkBox_x->setEnabled(false);
    ui->checkBox_y->setEnabled(false);
    ui->groupBox_x->setEnabled(false);
    ui->groupBox_y->setEnabled(false);
    ui->label_point_1->setEnabled(false);
    ui->label_point_2->setEnabled(false);
    ui->label_point_3->setEnabled(false);
    ui->label_point_4->setEnabled(false);
    ui->lineEdit_x_point_1->setEnabled(false);
    ui->lineEdit_x_point_2->setEnabled(false);
    ui->lineEdit_y_point_1->setEnabled(false);
    ui->lineEdit_y_point_2->setEnabled(false);

    tip.setText("Please click on points with the following order");
    tip.setStandardButtons(QMessageBox::Ok);
    tip.setIconPixmap(QPixmap(":/img/img/tip.png"));
    tip.setDefaultButton(QMessageBox::Ok);
}

CalibrationDialog::~CalibrationDialog()
{
    delete ui;

    delete p_scene;
    delete p_map;

    delete p_x_point_1;
    delete p_x_point_2;
    delete p_y_point_1;
    delete p_y_point_2;
}

void CalibrationDialog::mousePressEvent(QMouseEvent *event)
{
    QPoint remapped = ui->graphicsView->mapFromParent(event->pos());
```

```
    if (ui->graphicsView->rect().contains(remapped) && event->button() ==
Qt::LeftButton)
    {
         QPointF mousePoint = ui->graphicsView->mapToScene(remapped);

         if(isCalibrating)
         {
             double radius = 4;

             p_scene->addEllipse(mousePoint.x()-radius,
                                 mousePoint.y()-radius,
                                 radius*2.0, radius*2.0,
             QPen(Qt::red), QBrush(Qt::red, Qt::SolidPattern));

             calibrationVector.push_back(mousePoint);

             calibrationCheck();
         }
    }
}

void CalibrationDialog::on_CalibrationDialog_accepted()
{
    double x_length, y_length;

    p_x_point_1 = new QString(ui->lineEdit_x_point_1->text());
    p_x_point_2 = new QString(ui->lineEdit_x_point_2->text());
    p_y_point_1 = new QString(ui->lineEdit_y_point_1->text());
    p_y_point_2 = new QString(ui->lineEdit_y_point_2->text());

    x_point_1_num = p_x_point_1->toDouble();
    x_point_2_num = p_x_point_2->toDouble();
    y_point_1_num = p_y_point_1->toDouble();
    y_point_2_num = p_y_point_2->toDouble();

    x_length = x_point_2_num - x_point_1_num;
    y_length = y_point_2_num - y_point_1_num;

    // calculate the x-y calibration factors unit: real_length/pixel
    x_calibration_factor = x_length / (calibrationVector[1].x() -
calibrationVector[0].x());
    y_calibration_factor = y_length / (calibrationVector[3].y() -
calibrationVector[2].y());

    // calculate the offset of the origin
    x_offset = x_point_1_num - x_calibration_factor *
calibrationVector[0].x();
    y_offset = y_point_1_num - y_calibration_factor *
calibrationVector[2].y();

    isCalibrated = true;
}

void CalibrationDialog::calibrationCheck()
{
    if(calibrationVector.size() == 4)
    {
        isCalibrating = false;

        ui->groupBox_x->setEnabled(true);
        ui->groupBox_y->setEnabled(true);
```

```
        ui->label_point_1->setEnabled(true);
        ui->label_point_2->setEnabled(true);
        ui->label_point_3->setEnabled(true);
        ui->label_point_4->setEnabled(true);
        ui->lineEdit_x_point_1->setEnabled(true);
        ui->lineEdit_x_point_2->setEnabled(true);
        ui->lineEdit_y_point_1->setEnabled(true);
        ui->lineEdit_y_point_2->setEnabled(true);
    }
}



void CalibrationDialog::on_lineEdit_x_point_1_textEdited(const QString
&text_x_1)
{
    if(!text_x_1.isEmpty())
        hasTextX1 = true;
    else
        hasTextX1 = false;

    if(!text_x_1.isEmpty() && hasTextX2)
        ui->checkBox_x->setEnabled(true);
    else
        ui->checkBox_x->setEnabled(false);

    if(!text_x_1.isEmpty() && hasTextX2 && hasTextY1 && hasTextY2
        && text_x_1 != ui->lineEdit_x_point_2->text()
        && ui->lineEdit_y_point_1->text() != ui->lineEdit_y_point_2-
>text())
        ui->buttonBox->setEnabled(true);
    else
        ui->buttonBox->setEnabled(false);
}

void CalibrationDialog::on_lineEdit_x_point_2_textEdited(const QString
&text_x_2)
{
    if(!text_x_2.isEmpty())
        hasTextX2 = true;
    else
        hasTextX2 = false;

    if(!text_x_2.isEmpty() && hasTextX1)
        ui->checkBox_x->setEnabled(true);
    else
        ui->checkBox_x->setEnabled(false);

    if(!text_x_2.isEmpty() && hasTextX2 && hasTextY1 && hasTextY2
        && text_x_2 != ui->lineEdit_x_point_1->text()
        && ui->lineEdit_y_point_1->text() != ui->lineEdit_y_point_2-
>text())
        ui->buttonBox->setEnabled(true);
    else
        ui->buttonBox->setEnabled(false);
}



void CalibrationDialog::on_lineEdit_y_point_1_textEdited(const QString
&text_y_1)
{
```

```
    if(!text_y_1.isEmpty())
        hasTextY1 = true;
    else
        hasTextY1 = false;

    if(!text_y_1.isEmpty() && hasTextY2)
        ui->checkBox_y->setEnabled(true);
    else
        ui->checkBox_y->setEnabled(false);

    if(!text_y_1.isEmpty() && hasTextX2 && hasTextY1 && hasTextY2
        && text_y_1 != ui->lineEdit_y_point_2->text()
        && ui->lineEdit_x_point_1->text() != ui->lineEdit_x_point_2-
>text())
        ui->buttonBox->setEnabled(true);
    else
        ui->buttonBox->setEnabled(false);
}

void CalibrationDialog::on_lineEdit_y_point_2_textEdited(const QString
&text_y_2)
{
    if(!text_y_2.isEmpty())
        hasTextY2 = true;
    else
        hasTextY2 = false;

    if(!text_y_2.isEmpty() && hasTextY1)
        ui->checkBox_y->setEnabled(true);
    else
        ui->checkBox_y->setEnabled(false);

    if(!text_y_2.isEmpty() && hasTextX2 && hasTextY1 && hasTextY2
        && text_y_2 != ui->lineEdit_y_point_1->text()
        && ui->lineEdit_x_point_1->text() != ui->lineEdit_x_point_2-
>text())
        ui->buttonBox->setEnabled(true);
    else
        ui->buttonBox->setEnabled(false);
}

void CalibrationDialog::on_checkBox_x_stateChanged(int arg1)
{
    if(arg1 == 0)
        isXLog = false;
    else
        isXLog = true;
}

void CalibrationDialog::on_checkBox_y_stateChanged(int arg1)
{
    if(arg1 == 0)
        isYLog = false;
    else
        isYLog = true;
}

void CalibrationDialog::on_importImageButton_clicked()
{
    filename = QFileDialog::getOpenFileName(this,tr("Import
Graph"),"*.png",tr("Images (*.png *.jpg)"));
```

```
    if(!filename.isNull())
    {
        isCalibrating = true;
        calibrationVector.clear();

        p_map = new QPixmap(filename);

        p_scene->clear();
        p_scene->addPixmap(*p_map);

        ui->clearButton->setEnabled(true);

        _disableWidgets();

        tip.exec();
    }
}

void CalibrationDialog::on_helpButton_clicked()
{
    tip.exec();
}

void CalibrationDialog::on_clearButton_clicked()
{
    isCalibrating = true;
    hasTextX1 = false;
    hasTextX2 = false;
    hasTextY1 = false;
    hasTextY2 = false;

    calibrationVector.clear();

    p_scene->clear();
    p_scene->addPixmap(*p_map);

    ui->lineEdit_x_point_1->clear();
    ui->lineEdit_x_point_2->clear();
    ui->lineEdit_y_point_1->clear();
    ui->lineEdit_y_point_2->clear();

    ui->checkBox_x->setCheckState(Qt::Unchecked);
    ui->checkBox_y->setCheckState(Qt::Unchecked);

    _disableWidgets();
}

void CalibrationDialog::_disableWidgets()
{
    ui->buttonBox->setEnabled(false);
    ui->checkBox_x->setEnabled(false);
    ui->checkBox_y->setEnabled(false);
    ui->groupBox_x->setEnabled(false);
    ui->groupBox_y->setEnabled(false);
    ui->label_point_1->setEnabled(false);
    ui->label_point_2->setEnabled(false);
    ui->label_point_3->setEnabled(false);
    ui->label_point_4->setEnabled(false);
    ui->lineEdit_x_point_1->setEnabled(false);
    ui->lineEdit_x_point_2->setEnabled(false);
```

```
    ui->lineEdit_y_point_1->setEnabled(false);
    ui->lineEdit_y_point_2->setEnabled(false);
}
```

4.3 "setscaledialog.h"

```
#ifndef SETSCALEDIALOG_H
#define SETSCALEDIALOG_H

#include <QDialog>

namespace Ui {
class SetScaleDialog;
}

class SetScaleDialog : public QDialog
{
    Q_OBJECT

public:
    explicit SetScaleDialog(bool isXLog, bool isYLog, QWidget *parent =
nullptr);
    ~SetScaleDialog();

    bool getIsXLog() {return isXLog;}
    bool getIsYLog() {return isYLog;}

private slots:

    void on_SetScaleDialog_accepted();

private:
    Ui::SetScaleDialog *ui;

    bool isXLog = false;
    bool isYLog = false;
};

#endif // SETSCALEDIALOG_H
```

4.4 "setscaledialog.cpp"

```
#include "viewdatadialog.h"
#include "ui_viewdatadialog.h"

ViewDataDialog::ViewDataDialog(const QVector<QPointF> &points, QWidget
*parent) :
    QDialog(parent),
    ui(new Ui::ViewDataDialog),
    points(points)
{
    ui->setupUi(this);
    ui->textEdit->setReadOnly(true);

    setDataText(points);
}

ViewDataDialog::~ViewDataDialog()
```

```
{
    delete ui;
}

void ViewDataDialog::setDataText(const QVector<QPointF> &points)
{
    QString text;

    for(int i = 0; i < points.size(); i++)
    {
       text.append(QString::number(points[i].x()));
       text.append(", ");
       text.append(QString::number(points[i].y()));
       text.append('\n');
    }

    ui->textEdit->setText(text);
}
```

4.5 "viewdatadialog.h"

```
#ifndef VIEWDATADIALOG_H
#define VIEWDATADIALOG_H

#include <QDialog>
#include <QVector>
#include <QPointF>
#include <QString>
#include <QDebug>

namespace Ui {
class ViewDataDialog;
}

class ViewDataDialog : public QDialog
{
    Q_OBJECT

public:
    explicit ViewDataDialog(const QVector<QPointF> &points, QWidget *parent
= nullptr);
    ~ViewDataDialog();

    void setDataText(const QVector<QPointF> &points);

private:
    Ui::ViewDataDialog *ui;

    QVector<QPointF> points;
};

#endif // VIEWDATADIALOG_H
```

4.6 "viewdatadialog.cpp"

```
#include "viewdatadialog.h"
#include "ui_viewdatadialog.h"
```

```
ViewDataDialog::ViewDataDialog(const QVector<QPointF> &points, QWidget
*parent) :
    QDialog(parent),
    ui(new Ui::ViewDataDialog),
    points(points)
{
    ui->setupUi(this);
    ui->textEdit->setReadOnly(true);

    setDataText(points);
}

ViewDataDialog::~ViewDataDialog()
{
    delete ui;
}

void ViewDataDialog::setDataText(const QVector<QPointF> &points)
{
    QString text;

    for(int i = 0; i < points.size(); i++)
    {
       text.append(QString::number(points[i].x()));
       text.append(", ");
       text.append(QString::number(points[i].y()));
       text.append('\n');
    }

    ui->textEdit->setText(text);
}
```

4.7 "graphdigitiser.h"

```
#ifndef GRAPHDIGITISER_H
#define GRAPHDIGITISER_H

#include <QMainWindow>
#include <QGraphicsScene>
#include <QFileDialog>
#include <QContextMenuEvent>
#include <QMenu>
#include <QMessageBox>
#include <QDebug>
#include <QGraphicsEllipseItem>
#include "setscaledialog.h"
#include "calibrationdialog.h"
#include "viewdatadialog.h"

QT_BEGIN_NAMESPACE
namespace Ui { class GraphDigitiser; }
QT_END_NAMESPACE

class GraphDigitiser : public QMainWindow
{
    Q_OBJECT

public:
    GraphDigitiser(QWidget *parent = nullptr);
    ~GraphDigitiser();
```

```
    void mousePressEvent(QMouseEvent *event);
    void contextMenuEvent(QContextMenuEvent *event);
    QPointF pixelToReal(const QPointF &point);

    void constructMenus();
    void autoMeasuring();

private slots:

    void on_actionAdd_Calibration_triggered();

    void on_actionSet_Axes_Scales_triggered();

    void on_actionView_Data_triggered();

    void on_actionExport_Data_triggered();

    void on_checkBox_stateChanged(int arg1);

    void on_clearButton_clicked();

private:
    Ui::GraphDigitiser *ui;

    QGraphicsScene *p_scene = new QGraphicsScene(this);

    QMenu *p_menu = new QMenu(this);

    QPixmap *p_map = new QPixmap();

    QVector<QPointF> dataVector;
    QVector<QPointF> endPointsVector;

    double x_calibration_factor;
    double y_calibration_factor;
    double x_offset;
    double y_offset;

    bool isXLog = false;
    bool isYLog = false;

    bool isMeasuring = false;
    bool isAutoMode = false;
};
#endif // GRAPHDIGITISER_H
```

4.8 "graphdigitiser.cpp"

```
#include "graphdigitiser.h"
#include "ui_graphdigitiser.h"

GraphDigitiser::GraphDigitiser(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::GraphDigitiser)
{
    ui->setupUi(this);
    ui->graphicsView->setScene(p_scene);
    ui->graphicsView->setCursor(Qt::CrossCursor);
```

```
    ui->clearButton->setEnabled(false);
    ui->actionSet_Axes_Scales->setEnabled(false);
    ui->actionView_Data->setEnabled(false);
    ui->actionExport_Data->setEnabled(false);
    ui->checkBox->setEnabled(false);

    constructMenus();
}

GraphDigitiser::~GraphDigitiser()
{
    delete ui;

    delete p_scene;
    delete p_menu;
    delete p_map;
}

void GraphDigitiser::mousePressEvent(QMouseEvent *event)
{
    if(isMeasuring)
    {
        QPoint remapped = ui->graphicsView->mapFromParent(event->pos());

        if (ui->graphicsView->rect().contains(remapped) && event->button()
== Qt::LeftButton)
        {
            QPointF mousePoint = ui->graphicsView->mapToScene(remapped);

            if(isAutoMode)
            {
                mousePoint.setY(mousePoint.y()-58);
                endPointsVector.push_back(mousePoint);


                if(endPointsVector.size() == 1)
                {
                    p_scene->addLine(mousePoint.x(),mousePoint.y(),
                                     p_map->width(),mousePoint.y(),
                                     QPen(Qt::red));

                    p_scene->addLine(mousePoint.x(),mousePoint.y(),
                                     mousePoint.x(),p_map->height(),
                                     QPen(Qt::red));
                }
                else
                {
                    p_scene->addLine(mousePoint.x(),mousePoint.y(),
                                     0,mousePoint.y(),
                                     QPen(Qt::red));

                    p_scene->addLine(mousePoint.x(),mousePoint.y(),
                                     mousePoint.x(),0,
                                     QPen(Qt::red));
                }

                if(endPointsVector.size() == 2)
                {
                    if(endPointsVector[0].x() < endPointsVector[1].x() &&
endPointsVector[0].y() < endPointsVector[1].y())
                    {
```

```
                        autoMeasuring();
                        QMessageBox::about(this,tr("Auto Mode"),tr("Data is
generated successfully"));
                    }
                    else
                    {
                        QMessageBox::about(this,tr("Auto Mode"),tr("The
selected range is invalid"));

                        ui->clearButton->click();
                    }
                }
            }
            else
            {
                mousePoint.setY(mousePoint.y()-58);
                dataVector.push_back(pixelToReal(mousePoint));

                double radius = 4;

                p_scene->addEllipse(mousePoint.x()-radius,
                                    mousePoint.y()-radius,
                                    radius*2.0, radius*2.0,
                QPen(Qt::red), QBrush(Qt::red, Qt::SolidPattern));
            }
        }
    }
}

void GraphDigitiser::autoMeasuring()
{
    isAutoMode = false;

    QImage image(p_map->toImage());

    double x_start = endPointsVector[0].x();
    double x_end = endPointsVector[1].x();
    double y_start = endPointsVector[0].y();
    double y_end = endPointsVector[1].y();

    for(int i = 0; i < x_end-x_start; i++)
    {
        for(int j = 0; j < y_end-y_start; j++)
        {
            double radius = 1;

            QColor color(image.pixelColor(x_start+i, y_start+j));

            if(color.rgb() <= 0xFF646464)
            {
                QPointF point(x_start+i, y_start+j);

                p_scene->addEllipse(point.x()-radius,
                                    point.y()-radius,
                                    radius*2.0, radius*2.0,
                QPen(Qt::red), QBrush(Qt::red, Qt::SolidPattern));

                dataVector.push_back(pixelToReal(point));

                break;
            }
```

```
        }
    }
}

void GraphDigitiser::contextMenuEvent(QContextMenuEvent *event)
{
    p_menu->popup(event->globalPos());
}

QPointF GraphDigitiser::pixelToReal(const QPointF &point)
{
    double x, y;

    if(isXLog)
        x = qPow(10, x_offset + point.x() * x_calibration_factor);
    else
        x = x_offset + point.x() * x_calibration_factor;

    if(isYLog)
        y = qPow(10, y_offset + point.y() * y_calibration_factor);
    else
        y = y_offset + point.y() * y_calibration_factor;

    QPointF real_point(x,y);

    return real_point;
}

void GraphDigitiser::constructMenus()
{
    p_menu->addAction(ui->actionSet_Axes_Scales);
}

void GraphDigitiser::on_actionAdd_Calibration_triggered()
{
    CalibrationDialog *p_calibration_dialog = new CalibrationDialog(this);
    p_calibration_dialog->exec();

    if(p_calibration_dialog->getIsCalibrated())
    {
        ui->clearButton->setEnabled(true);
        ui->actionSet_Axes_Scales->setEnabled(true);
        ui->actionView_Data->setEnabled(true);
        ui->actionExport_Data->setEnabled(true);
        ui->checkBox->setEnabled(true);
        ui->checkBox->setCheckState(Qt::Unchecked);

        x_calibration_factor = p_calibration_dialog-
>getXCalibrationFactor();
        y_calibration_factor = p_calibration_dialog-
>getYCalibrationFactor();

        x_offset = p_calibration_dialog->getXOffset();
        y_offset = p_calibration_dialog->getYOffset();

        isXLog = p_calibration_dialog->getIsXLog();
        isYLog = p_calibration_dialog->getIsYLog();

        p_map = new QPixmap(p_calibration_dialog->getFileName());

        p_scene->clear();
```

```
        p_scene->addPixmap(*p_map);

        dataVector.clear();
        endPointsVector.clear();

        isMeasuring = true;
    }
}

void GraphDigitiser::on_actionSet_Axes_Scales_triggered()
{
    SetScaleDialog *p_set_scale_dialog = new SetScaleDialog(isXLog,
isYLog);
    p_set_scale_dialog->exec();

    isXLog = p_set_scale_dialog->getIsXLog();
    isYLog = p_set_scale_dialog->getIsYLog();
}

void GraphDigitiser::on_actionView_Data_triggered()
{
    ViewDataDialog *p_view_data_dialog = new ViewDataDialog(dataVector);
    p_view_data_dialog->exec();
}

void GraphDigitiser::on_actionExport_Data_triggered()
{
    QString text;

    for(int i = 0; i < dataVector.size(); i++)
    {
       text.append(QString::number(dataVector[i].x()));
       text.append(", ");
       text.append(QString::number(dataVector[i].y()));
       text.append('\n');
    }


    QString filename = QFileDialog::getSaveFileName(this,
        tr("Save Data"),
        "untitled.csv",
        tr("Comma Seperated Values (*.csv);;Text File (*.txt);;All Files
(*)"));

    if(filename.isEmpty())
        return;
    else
    {
        QFile file(filename);

        if(!file.open(QIODevice::WriteOnly))
        {
            QMessageBox::information(this,tr("Unable to open file"),
                                     file.errorString());
            return;
        }

        QDataStream out(&file);
        out << text;
    }
}
```

```
void GraphDigitiser::on_checkBox_stateChanged(int arg1)
{
    if(arg1 == 0)
        isAutoMode = false;
    else
    {
        isAutoMode = true;

        p_scene->clear();
        p_scene->addPixmap(*p_map);

        endPointsVector.clear();
        dataVector.clear();

        QMessageBox::about(this, tr("Auto Mode"),
                           tr("Set a range for auto measuring\nFirstly
click on the upper-left corner of the graph\nSecondly click on the lower-
right corner of the graph"));
    }
}

void GraphDigitiser::on_clearButton_clicked()
{
    dataVector.clear();
    endPointsVector.clear();

    p_scene->clear();
    p_scene->addPixmap(*p_map);

    ui->checkBox->setCheckState(Qt::Unchecked);
}
```

4.9 "main.cpp"

```
#include "graphdigitiser.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    GraphDigitiser w;
    w.show();

    return a.exec();
}
```