地球物理软件技术

《课程设计报告》

地球物理数据曲线编辑软件设计

制作: 赖利朋

班级: 21 级地球物理学 1 班

学号: 202105050122

目录：

一、课程设计要求

1.1 功能需求

地球物理学数据可视化软件需要实现的功能如下：

（1）数据：对多种地球物理数据格式读取保存导出，以曲线的形式表示。

（2）编辑：实现撤销重做与清空，方便对原始数据编辑。

（3）查看：可对生成曲线进行放大/缩小/移动拖拽/还原等操作。

（4）帮助：提供软件有关信息以及操作的入门文档。

1.2 设计原则

为了方便地球物理工作者处理有关数据，使系统功能齐全，操作简便，最大限度地降低用户的上手难度，从而满足用户的需求，项目的开发工作秉承以下原则：

（1） **软件依赖：**软件可独立运行于 Windows 系统，不含第三方(软件、平台)支持，软件应含安装程序，安装完成后可直接运行。

（2） **用户体验：**软件界面美观简洁，功能全面，操作简便，容错纠错能力强，且运行稳定。

（3） **面向对象原则：**面向对象编程灵活、易扩展，模块化的思想使得代码更易于理解、修改和维护，且面向对象可以更好地映射真实世界的概念和关系，使代码更加容易被理解。

（4） **程序可读性：**为了方便代码编写与理解，编程时需加上完善注释，易与后续的功能完善与问题查找。

二、思路及可行性分析

（1）点结构体 PointXY

对于地球物理数据，多为 XY 坐标类型，则可用 Point 类型存储，为提升效率，本程序采用自定义的 PointXY 结构体，并添加了标记该点是否被选中的标志，相比于 C#内置的 Point 类型，性能效率有所提升。

（2）一条曲线类 List<PointXY>

一条曲线由多个点连线所组成，本程序采用的是 C#内置的数据结构列表 List，并提供了下标访问，返回数据点个数等，相比于自定义实现数组存储，C#内置的 List 具有更高的性能，且无需担心数组越界访问。

（3）一组曲线类(包含多条曲线) List<Curve>

一组曲线类包含多条曲线，它存在的目的是，方便管理打包一段时期内的曲线列表。考虑如下应用场景：有一组曲线，任意修改其中的一条曲线，过段时间后撤销该操作，即恢复原来的曲线。此时只需要定义 Stack<Curves>，并只要将更改过的一组曲线压栈，需要撤销时，只需弹出栈顶元素，即可完成撤销一步；

而若不使用该一组曲线类，也需要定义栈 Stack<Curve>，把发生修改的曲线压入栈中，撤销时也能根据 Stack 弹出栈顶元素，但是问题出在绘图时，无法分辨出该用哪一条数据，若是对每条曲线打标记，由于可以变化很多次，也会造成栈的容量过大，或是需要撤销的次数过多，才能恢复到目标情况。

对于将一组曲线当成栈的属性即 Stack<Curves>，其优点为操作便捷且容易理解，代码简单，但是其缺点也是很明显的，即对于没有进行修改的曲线，它也压入了栈中，因为它们是一组曲线是一个整体，这会造成资源的冗余，内存开销较大及性能相对会差。

三、需求分析

（1）数据读取

地球物理数据多种多样，有诸如 XY 坐标类型的点形式的 TXT 文本文件以及具有表头信息的 EXCEL 文件，特殊的还有近似于倒三角的高密度电法数，对此需要对不同的数据类型，设计不同的数据读入接口，并提醒用户选择以对应的数据读取，否则会抛出提示。

（2）数据转换

由于读入的是地球物理数据即逻辑坐标，而显示屏上是像素位置即设备坐标，故需要将地球物理数据与像素数据进行映射，映射的公式如下：

$$D_x = \text{winRect.}\,Left + (L_x - minx) \times \text{winRect.Width}/(maxx - minx)$$

$$L_x = (D_x - \text{winRect.Left}) * (maxx - minx)/(\text{winRect.Width}) + minx$$

其中，D_x为设备(Device)的横坐标，L_x为逻辑的横坐标，$winRect.Left$为窗口的左边界，同理$winRect.Width$为窗口的宽度，$\min x$,$\max x$为该组数据中的极小极大值；对于纵坐标 y，只需要将$winRect.Left$更换为$winRect.Bottom$，将$winRect.Width$更换为$winRect.Height$，即横坐标为从左到右绘图，对于纵坐标，从下到上绘图。

（3）绘图显示

对于一系列的数据点，使用浅蓝绿色方框标注，并使用 C#中的 DrawLine 方法，将数据点连线，形成一条曲线，若用户需要拖拽数据点以更改数据，则需要将选择点标红，以表示选择了该点，在拖拽过程中，鼠标一直按住左键，同时调用重绘事件，直至松开鼠标。

四、软件设计

（1） 对不同数据读取

1. 对二进制数据读取：调用 C#中的内置类 BinaryReader，对二进制数据进行读取。先读取数据点的个数，而后根据数据点的个数，使用 for 循环对同一行数据数据分别读取 x 和 y，并加入到全局已经定义的曲线中。
2. 对 MT 正演 TXT 数据读取：调用 C#中的内置类 StreamReader，对文本数据进行读取。实现定义分隔符数据 spilteChars，String 类中的 Spilit 方法可根据预先定义的分隔符对文本进行分割，得到分割好的字符串 string 后，其本质是一个字符数组，故根据索引即可访问频率和相位的数据，而后使用 Parse 方

法，将字符数据转换为浮点型数据。

3. 对 AMT 的 EDI 数据读取：对于 EDI 数据，需要先寻找其标志位“>FREQ”和“ZXXR”，以标志位作为开始行，进行读取，其中“>FREQ”为 x 坐标，“ZXXR”为 y 坐标，由于二者是分隔开读取的，故在读取“>FREQ”时可将 y 赋初值，待读取到“ZXXR”的值时，再取出对应点的纵坐标进行赋值。
4. 对高密度电测深 Excel 数据读取：其表头为隔离系数、点距和电阻率，其中隔离系数表示从高到低以层排列，点距表示距离$winRect.Left$的距离，电阻率表示相对于水平线起伏的高度。对此，读出最大的隔离系数，确定层数，根据层数以及窗口的高度$winRect.Height$分隔每个隔离系数对应的一层，并确定水平线，后调用 Draw 方法对对应的一层绘图，并约定不超出边界。

（2） 导出数据

导出数据即是按照数据的原始格式，对其进行导出操作，若只修改了某个数据点，则应只修改其对应的数据，而其他的格式则应保留，并输出。

故当读取数据时，则应保留数据的原始格式，比如 Excel 数据的表头，TXT 格式的文本标识符，若文本标识符过多时，则应采用 String 数组表示，并在标志位上输出已经修改好的 Points 数据。

（3） 全局的数据极小极大值

在**需求分析**中讨论了将逻辑坐标与设备坐标的互相转换，其中需要曲线的$\min x, \max x, \min y, \max y$值，由于每条曲线的极小极大值可能不一样，当绘制多条曲线时，会导致比例问题，故比较全部的曲线，并取全局的极值；特殊的是，对于高密度电测深数据，由于其数据格式特殊，故采用了额外的变量来表示高密度电测深数据的极值。

上述写法定义导致代码臃肿，因为对于每一条曲线，实际上都存储了全局的极小极大值，这显然是多余的。

（4） 移动/点击/松开鼠标控制

首先需要使用一个枚举类型描述鼠标状态的情况，即：正常、点击鼠标、松开鼠标，并用标志变量记录鼠标是否点击到数据点(若鼠标点击到显示数据点的浅蓝绿色方框内，则视为点击到数据)，并使用鼠标事件参数 MouseEventArgs 中的 Location 方法确定此时点击的位置，此时切换鼠标状态为“点击鼠标”。

在鼠标移动函数中，要求鼠标状态必须为“点击鼠标”状态，并要求鼠标移动范围不能超出窗口范围，此时根据鼠标移动的距离，映射到被鼠标选中的数据点的逻辑坐标中坐标变化的值，在绘图中再映射为设备坐标的值，至此，就实现了数据点跟着鼠标移动的动态过程。

当移动至目标位置时，松开鼠标，将鼠标状态改为“松开鼠标”，此时移动鼠标，选择数据点，数据点也不会和鼠标移动，至此就完成了鼠标控制的整套逻辑。

（5） 放大/缩小/平移实现

与(4)鼠标控制同理，需要枚举定义鼠标状态，即：正常 Normal、放大 ZoomIn、缩小 ZoomOut 和平移 ZoomMove，效果为选中以上按钮，鼠标变化为十字 Cross，并可通过拉取矩形框，在鼠标移动函数 MouseMove 中，来选中放大/缩小区域。

对于放大功能，新选中的矩形框成为新的设备极小极大值，并更新，调用重

新绘图。

对于缩小功能，记录矩形边界，并设置偏移量 Offset=200，即要求，向左上右下共 4 个边界扩展，为了防止越界，规定极小值取 Max(0, x1 - Offset), 极大值取 Min(winRect.Width, x2 + Offset)。

对于平移功能，将鼠标设置为箭头模式，为和鼠标选中数据点相区分，定义标识变量 bMouseDrag，判断是否为拖拽效果，其实现为：记录鼠标与点击时位置的偏移量，并将所有数据点都加上该偏移量，并调用重新绘图，至此可实现动态鼠标拖拽效果。

（6） 撤销实现

首先需要定义一组曲线类 Curves，以及方便进行撤销操作的栈结构 Stack<Curves>，由于要确定当前绘制的是哪一组曲线，故定义 CurrentCurves 并初始化为 null，以及确定是否曲线被修改，添加标识符 Modified=false。

读入数据时，把数据压入栈中，并把将当前数据赋值给 CurrentCurves，点击并移动数据点，将标识符 Modified=true，直至松开鼠标，则将当前数据（相对于原始数据已经变化）压入栈中。

点击编辑栏的“撤销”按钮，Pop 栈顶元素，并将其转移到另外一个栈，表示撤销的历史记录，随后调用 PictureBox 的重绘函数。

在绘图时，使用栈的 Peek()方法，提取栈顶元素，但是不进行删除，将栈顶赋值给 CurrentCurves，对 CurrentCurves 一组曲线列表，依次提取每一条曲线中的每一个点进行绘图。

（7） 属性框和列表框显示

为了明确当前的数据，需要添加列表框 ListBox，并增加选中效果，在读取完数据后，获取数据的文件名，并添加到列表框，并使用 Item.Add 方法对列表框进行更新 UpdateListBox。

为了能实时变更数据的线宽 LineWidth、线颜色 LineColor 等其他属性，增加 PropertyGrid，并添加其选择对象变换 SelectedObjectsChanged 和属性值变换的方法 PropertyValueChanged，为了能使用 PropertyGrid 变更属性，需要在曲线类 Curve 中，使用 get 和 set 方法定义曲线属性，即 public Color LineColor { get; set; } = Color.Red;

（8） 深拷贝实现

由于在 C#中，对象的赋值为浅拷贝，在如下场景中会出错：把当前一组曲线 Curves 压入栈中即 undoCurves.Push(CurrentCurves)，若后续对 CurrentCurves 的值变更，那么栈中原先存储的 CurrentCuvres 的值也会变更，而不是存储的原来的值，这就达不到撤销的效果。

因此，需要对于 Curves、Curve 和 PointXY 这些自定义类编写其拷贝构造函数，常用的方法有：反射序列化和 Mapper。

五、编码实现

当鼠标点击在 pictureBox1 范围内时，根据鼠标指针形状执行相应操作。如果是箭头形状，表示选点鼠标点击，触发选点逻辑并重新绘图。如果是十字标形

状，表示矩形拉框鼠标点击，改变相应状态

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
    Point p = e.Location;
    p1 = p; // p1记录被点击时的位置

    if (p1.X >=0 && p1.X <= pictureBox1.Width && p1.Y >=0  && p1.Y <= pictureBox1.Height)
    {
        if (Cursor == Cursors.Arrow) // 箭头
        {
            bMouseDown = true; // 选点鼠标点击，状态改变为true
            if (DoSelect(p))
            {
                pictureBox1.Invalidate(); //若判断成功则重新绘图
            }
        }

        if (Cursor == Cursors.Cross) // 十字标
        {
            bMouseDrag = true; // 矩形拉框鼠标点击，状态改变为true
        }
    }
}
```

图 1 点击鼠标

鼠标放大/缩小状态下拖动，触发图像重新绘制。在 pictureBox1 内，鼠标箭头拖动选中点，触发点移动并标记曲线修改。曲线移动状态下，鼠标十字标触发整体曲线移动，随鼠标移动所有点，同时重新绘制图像。

```
private void pictureBox1_MouseMove(object sender, MouseEventArgs e)
{
    Point p = e.Location;
    // 如果鼠标正在拖动且是放大状态
    if ((MouseState == MouseStateEnum.ZoomIn || MouseState == MouseStateEnum.ZoomOut) && bMouseDrag && Cursor == Cursors.Cross)
    {
            pictureBox1.Invalidate();
    }

    if (p.X >= 0 && p.X <= pictureBox1.Width && p.Y >= 0 && p.Y <= pictureBox1.Height)
    {
        if (CalAllcurves_SelectedCount() > 0 && Cursor == Cursors.Arrow && bMouseDown)
        {
            ModifyAllcurves_SeletedPoint(p1, p); // 移动选择的点
            p1 = p;
            Modified = true; // 在鼠标拖动中，修改点后，标记曲线已经被修改
            pictureBox1.Invalidate();
        }
        if (MouseState == MouseStateEnum.ZoomMove && Cursor == Cursors.Cross && bMouseDrag)
        {
            ModifyAllcurvesByMoving(p1, p); // 随着鼠标的移动，所有点都动
            p1 = p;
            pictureBox1.Invalidate();
        }
    }
}
```

图 2 移动鼠标

根据鼠标状态和位置，执行不同的操作。如果是选点鼠标点击结束，记录修改并触发图像重新绘制。如果是放大或缩小状态，更新曲线的逻辑坐标矩形并重新绘制图像。最后，标记鼠标拖动状态结束

```
private void pictureBox1_MouseUp(object sender, MouseEventArgs e)
{
    Point p = e.Location;
    p2 = p; // p2记录鼠标松手时的位置

    if (p2.X >= 0 && p2.X <= pictureBox1.Width && p2.Y >= 0 && p2.Y <= pictureBox1.Height && Cursor == Cursors.Arrow)
    {
        bMouseDown = false; // 选点鼠标点击，状态改变为true
        if (Modified) // 如果修改过了，就把当前所有的数据压入栈中
        {
            undoCurves.Push(currentCurves);
        }
        pictureBox1.Invalidate();
    }

    if (MouseState == MouseStateEnum.ZoomIn && Cursor == Cursors.Cross)
    {
        // 获得拉框的四个角点坐标轴 屏幕坐标 设备坐标
        float x1 = Math.Min(p1.X, p2.X);
        float x2 = Math.Max(p1.X, p2.X);
        float y1 = Math.Min(p1.Y, p2.Y);
        float y2 = Math.Max(p1.Y, p2.Y);

        for (int i = 0; i < AllCurves.Pcurves.Count; i ++)
        {
            Curve a = AllCurves.Pcurves[i];
            PointXY LeftPoint = new PointXY(x1, y1);
            PointXY RightPoint = new PointXY(x2, y2);
            LeftPoint = a.DPtoLP(LeftPoint); // 设备坐标转为逻辑坐标
            RightPoint = a.DPtoLP(RightPoint);
            a.lpRect.x1 = Math.Min(LeftPoint.x, RightPoint.x);
            a.lpRect.y1 = Math.Min(LeftPoint.y, RightPoint.y);
            a.lpRect.x2 = Math.Max(LeftPoint.x, RightPoint.x);
            a.lpRect.y2 = Math.Max(LeftPoint.y, RightPoint.y);
        }

        pictureBox1.Invalidate();
        bMouseDrag = false; // 对松手的该点标记状态
        p1 = p2 = new Point(-1, -1);
    }
    else if (MouseState == MouseStateEnum.ZoomOut && Cursor == Cursors.Cross)
    {
        // 获得拉框的四个角点坐标轴 屏幕坐标 设备坐标
        float x1 = Math.Min(p1.X, p2.X);
        float x2 = Math.Max(p1.X, p2.X);
        float y1 = Math.Min(p1.Y, p2.Y);
        float y2 = Math.Max(p1.Y, p2.Y);

        // 扩展边界 -> 左上角(x1, y1)减 -> 右下角(x2, y2)加
        const int Offset = 200; // 偏移量
        x1 = Math.Min(0, x1 - Offset);
        x2 = Math.Max(this.pictureBox1.Width, x2 + Offset);
        y1 = Math.Min(0, y1 - Offset);
        y2 = Math.Max(this.pictureBox1.Height, y2 + Offset);

        for (int i = 0; i < AllCurves.Pcurves.Count; i++)
        {
            // Curve a = curves[i];
            Curve a = AllCurves.Pcurves[i];
            PointXY LeftPoint = new PointXY(x1, y1);
            PointXY RightPoint = new PointXY(x2, y2);
            LeftPoint = a.DPtoLP(LeftPoint); // 设备坐标转为逻辑坐标
            RightPoint = a.DPtoLP(RightPoint);
            a.lpRect.x1 = Math.Min(LeftPoint.x, RightPoint.x);
            a.lpRect.y1 = Math.Min(LeftPoint.y, RightPoint.y);
            a.lpRect.x2 = Math.Max(LeftPoint.x, RightPoint.x);
            a.lpRect.y2 = Math.Max(LeftPoint.y, RightPoint.y);
        }

        pictureBox1.Invalidate();
        bMouseDrag = false; // 对松手的该点标记状态
        p1 = p2 = new Point(-1, -1);
    }
    bMouseDrag = false; // 对松手的该点标记状态
}
```

图 3 松开鼠标

LPtoDP 函数将逻辑坐标映射到设备坐标，考虑了坐标范围和窗口大小。DPtoLP 函数执行相反的操作，将设备坐标映射回逻辑坐标。

```
public PointXY LPtoDP(PointXY LP)        // 逻辑坐标 -> 设备坐标
{
    float minx = Math.Min(lpRect.x1, lpRect.x2);
    float miny = Math.Min(lpRect.y1, lpRect.y2);
    float maxx = Math.Max(lpRect.x1, lpRect.x2);
    float maxy = Math.Max(lpRect.y1, lpRect.y2);

    float DPx = winRect.Left + (LP.x - minx) * winRect.Width / (maxx - minx);
    float DPy = winRect.Bottom - (LP.y - miny) * winRect.Height / (maxy - miny);

    return new PointXY(DPx, DPy, LP.Selected); // 构造函数需要状态一致
}

6 个引用
public PointXY DPtoLP(PointXY DP)        // 设备坐标 -> 逻辑坐标
{
    float minx = Math.Min(lpRect.x1, lpRect.x2);
    float miny = Math.Min(lpRect.y1, lpRect.y2);
    float maxx = Math.Max(lpRect.x1, lpRect.x2);
    float maxy = Math.Max(lpRect.y1, lpRect.y2);

    float LPx = (DP.x - winRect.Left) * (maxx - minx) / (winRect.Width) + minx;
    float LPy = -(DP.y - winRect.Bottom) * (maxy - miny) / (winRect.Height) + miny;

    return new PointXY(LPx, LPy, DP.Selected); // 构造函数需要状态一致
}
```

图 5 逻辑设备坐标相互转换

通过迭代逻辑坐标系中的点，使用 LPtoDP 函数将其转换为设备坐标。根据曲线是否被选择，使用红色或灰色画笔绘制曲线，同时在选中的点周围绘制相应的矩形框，突出显示选中的点。适用于图形用户界面中的曲线可视化。

```
public void Draw(Graphics g, bool SelectedLine)
{
    float dotSize = 12;
    for (int i = 0; i < Points.Count - 1; i++)
    {
        PointXY p1 = Points[i];  //获取逻辑点坐标
        PointXY p2 = Points[i + 1];  //获取逻辑点坐标
        p1 = LPtoDP(p1);
        p2 = LPtoDP(p2);  //转换成屏幕设备坐标

        Color grayColor = Color.Gray;
        Pen grayPen = new Pen(grayColor);

        Pen linePen = new Pen(LineColor);
        linePen.Width = LineWidth;
        if (SelectedLine) // 如果是当前选择的曲线，则标记为红色
        {
            Pen redPen = new Pen(LineColor, LineWidth);
            g.DrawLine(redPen, p1.x, p1.y, p2.x, p2.y);

            if (p1.Selected) // 给当前选中的点绘制红色矩形框，表示当前选中该点
            {
                g.DrawRectangle(Pens.Red,
                p1.x - dotSize / 2,
                p1.y - dotSize / 2,
                dotSize, dotSize);
            }
            else // 如果没有选中该点，则绘制为亮蓝色
            {
                g.DrawRectangle(Pens.DarkCyan,
                p1.x - dotSize / 2,
                p1.y - dotSize / 2,
                dotSize, dotSize);
            }
        }
        else // 如果不是当前选择的曲线，则标记为灰色
        {
            g.DrawLine(grayPen, p1.x, p1.y, p2.x, p2.y);
        }
    }
}
```

图 6 一条曲线绘图

从 Excel 文件中读取数据，每组数据包括一个隔离因子、横坐标和纵坐标。根据隔离因子的变化，将数据分组为不同的曲线。读取完成后，将曲线对象添加到 CurrentCurves.Pcurves 列表中，并计算全局 x 坐标范围。最后，更新 ListBox 和刷新绘图区域。

```
private void 高密度电阻率ToolStripMenuItem_Click(object sender, EventArgs e)
{
    ClearDataWhenReadNewData(); // 读数据之前先清楚别的数据
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Excel Files| *.xls; *.xlsx; *.xlsm; *.csv"; // 仅显示 Excel 文件
    openFileDialog.ValidateNames = true;
    openFileDialog.Multiselect = false; // 单文件
    openFileDialog.Title = "选择 Excel 文件";

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        FileStream fs = new FileStream(openFileDialog.FileName, FileMode.Open, FileAccess.Read);
        IExcelDataReader reader = ExcelReaderFactory.CreateReader(fs);

        DataSet result = reader.AsDataSet();                // 读取数据
        DataTable table = result.Tables[0];                // 获取第一个工作表

        string IsolationFactor = "1";
        Curve a = new Curve();                  // 点距为x, 视电阻率为y
        for (int i = 1; i < table.Rows.Count; i++) // 从索引 1 开始，跳过第一行
        {
            DataRow row = table.Rows[i];
            float x = 0, y = 0;
            for (int j = 0; j < table.Columns.Count; j++)
            {
                string cellValue = row[j].ToString();

                if (j == 0)                             // 如果是第一条 或 与前一条不同 就说明此时是新的曲线
                {
                    if (cellValue.Equals(IsolationFactor) == false)
                    {
                        AllCurves.Pcurves.Add(a); // 先把旧的加进去
                        IsolationFactor = cellValue;
                        a = new Curve(); // 后创建新的
                    }
                }
                else if (j == 1)
                {
                    x = float.Parse(cellValue);
                    if (x < global_minx) global_minx = x;
                    if (x > global_maxx) global_maxx = x;
                }
                else if (j == 2)
                {
                    y = float.Parse(cellValue);
                    a.Add(x, y);
                }
            }
        }
        AllCurves.Pcurves.Add(a); // 防止遗漏最后一条曲线
        Console.WriteLine("The Count of curves:", AllCurves.Pcurves.Count);

        float GridHeight = pictureBox1.Height / (AllCurves.Pcurves.Count); // 格子高度
        for (int i = 0; i < AllCurves.Pcurves.Count; i ++)                // 添加上下边界
        {
            float GridUp = i * GridHeight / 2;
            float GridDown = (i + 1) * GridHeight / 2;
            grids.Add((GridUp, GridDown));
        }
        Console.WriteLine("The Count of grid: ", grids.Count);
        UpdateListBox();
        reader.Close();
        fs.Close();
        HighDensityElectrical = true;
        pictureBox1.Invalidate(); // 调用在picutureBox1中绘图
    }
}
```

图 7 高密度电阻率处理流程

当选中曲线且属性发生变化时，根据变化的属性名称更新曲线的对应属性值（如颜色、线宽），然后刷新绘图区域以反映变化。

```
private void propertyGrid1_PropertyValueChanged(object s, PropertyValueChangedEventArgs e)
{
    // 检查所选曲线是否存在并且属性网格中的属性值已更改
    if (curSelected >= 0 && curSelected < AllCurves.Pcurves.Count && e.ChangedItem != null)
    {
        // 获取所选曲线
        Curve selectedCurve = AllCurves.Pcurves[curSelected];

        // 检查更改的属性名称，并根据不同的属性名称更新曲线属性
        switch (e.ChangedItem.Label)
        {
            case "LineColor":
                selectedCurve.LineColor = (Color)e.ChangedItem.Value;
                break;
            case "LineWidth":
                selectedCurve.LineWidth = (float)e.ChangedItem.Value;
                break;
                // 添加其他属性的处理逻辑...
        }

        // 重新绘制曲线
        pictureBox1.Invalidate();
    }
}
```

图 8 属性框关联

当用户选择“撤销”时，从撤销栈中取出上一个状态的曲线数据，放入重做栈，并刷新绘图区域。

```
private void 撤销ToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (undoCurves.Count > 0)
    {
        Curves a = undoCurves.Pop(); // 取出了当前的栈顶，那么画图时，实际上是栈顶的前一个元素
        redoCurves.Push(a);
    }
    pictureBox1.Invalidate(); // 撤销后 重绘
}
```

图 9 撤销重做

实现了曲线对象的深拷贝操作。通过创建新的曲线对象，复制曲线上的每个点，并复制其他相关属性，以生成原曲线对象的独立副本。最后，返回这个深拷贝后的曲线对象。

```
public Curve DeepCopy()
{
    Curve curve = new Curve();

    // curve.Test_Add_Data();

    PointXY p = new PointXY();
    for (int i = 0; i < Points.Count; i ++)
    {
        curve.Points.Add(p);
    }

    for (int i = 0; i < this.Points.Count; i ++)
    {
        curve.Points[i] = this.Points[i].DeepCopy();
    }
    curve.minx = this.minx;
    curve.maxx = this.maxx;
    curve.miny = this.miny;
    curve.maxy = this.maxy;
    curve.winRect = this.winRect;
    curve.dataRect = this.dataRect;
    curve.lpRect = this.lpRect;
    curve.LineWidth = this.LineWidth;
    curve.LineColor = this.LineColor;

    return curve;
}
```

图 10 深拷贝实现

六、系统调试

（1） 逻辑设备坐标转换

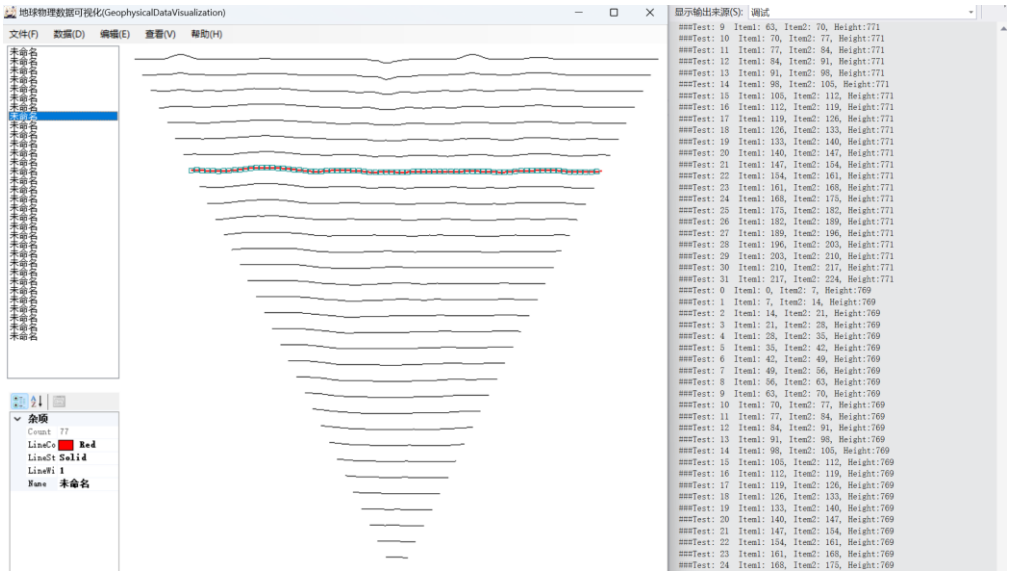
以下是调试的代码：

```
PointXY p1 = Points[i];  //获取逻辑点坐标
Console.WriteLine("p1的逻辑点坐标 x:{0}, y:{1}", p1.x, p1.y);
PointXY p2 = Points[i + 1];  //获取逻辑点坐标
p1 = LPtoDP(p1);
Console.WriteLine("p1经过了逻辑点到设备点的转换");
Console.WriteLine("p1的设备点的坐标 x:{0}, y:{1}", p1.x, p1.y);
```

下面是结果：

```
p1的逻辑点坐标 x:0.5, y:10
p1经过了逻辑点到设备点的转换
p1的设备点的坐标 x:0, y:467
p1的逻辑点坐标 x:1, y:18
p1经过了逻辑点到设备点的转换
p1的设备点的坐标 x:6.428571, y:425.4889
```

（2） 高密度电阻率读取



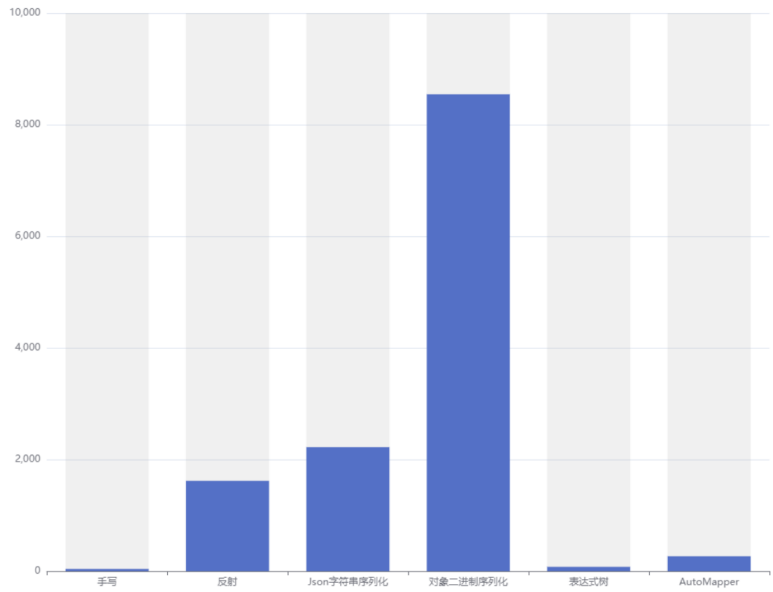
七、结论

（1） 对单例模式分析

受到**《设计模式》**的启发，可以使用**单例模式**，即对于整个曲线类，只提供全局接口 public static void GetGlobalMaxMin(float global_minx, float global_miny, float global_maxx, float global_maxy)，并将其极小极大值属性设置为 private static float minx, maxx，此时可以保证曲线类只有这一个接口，并且可以被外界直接访问，有助于减小内存开销和提高运行效率。

（2） 对深拷贝算法分析

根据 C#几种深拷贝方法探究及性能比较[2]，可以得到如下性能比较图表



对于一般简单的对象深拷贝，推荐直接手写，复杂对象深拷贝，推荐使用表达式树。当然，如果创建对象中还涉及到构造函数初始化，那又是不同的情况。

（3） 个人心得

这是我第一次独立写的一个小项目，之前是跟着写过几个(神经网络车辆识别，Hack 语言转汇编指令，Android 的 app 和 tcmalloc 高并发内存池)，虽说跟着网络教程写了这些项目，但是仍感觉有些朦胧，经常是看了教程的代码怎么写，然后模仿，总觉得缺少了点为什么要这么写，能不能有其他写法的思考。

当自己真正的去实现一些细节，才会理解，逻辑的连贯、编程的严密性和调试技巧。比如对于撤销功能的调试，我查了一晚上才发现是浅拷贝的问题，之前也学过深浅拷贝，但是常是学了，没有怎么用；以及常忽视了备份的问题，在原代码上增加功能发现有错误，但是想回退之前的版本，这才知道 Git 重要性。

同时，也感到自己编写的代码仍然有比较大缺陷，因为缺少《软件工程》《设计模式》等知识，总觉得编的代码不够逻辑严密及缺少美观，只能说初步实现了要求吧，这作为一个开始，还是带来了不错的体验的。

参考资料

[1] https://learn.microsoft.com/zh-cn/dotnet/csharp/
[2] https://www.quarkbook.com/?p=1210
[3] https://zhuanlan.zhihu.com/p/355303892