



《微波遥感》课程集中实习

实习四 相位解缠编程

实习报告

目录

一、实习目的	1
二、实习原理	1
2.1 相位解缠	1
2.1.1 相位缠绕与相位解缠的概念	1
2.1.2 相位解缠方法概述	1
2.2 支切线法相位解缠	2
2.2.1 支切线法概述	2
2.2.2 前提条件	2
2.2.3 残差点检测	2
2.2.4 支切线连接	3
2.2.5 相位解缠	4
三、实习数据	5
四、实习内容	5
4.1 总体流程	5
4.2 导入真实相位	5
4.3 相位缠绕	6
4.4 残差点检测	7
4.5 支切线生成	8
4.6 相位解缠	12
五、实习结果与分析	18
5.1 主要函数说明	18
5.2 运行结果	19
5.3 正确解缠结果	19
5.4 不同支切线结果分析	20
六、实习心得	22

一、实习目的

理解并掌握相位解缠的基本原理和解缠过程，编程实现简单的枝切法相位解缠方法，对结果进行分析。

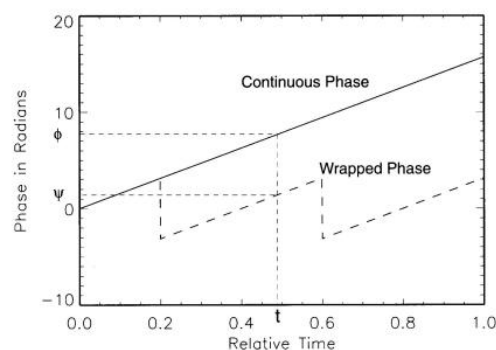
枝切法相位解缠:理解枝切法相位解缠基本原理（参考课程 PPT）。理解相位再缠绕过程，残差点检测的方法，枝切线的构成，解缠路径的生成以及解缠结果的输出。比较不同枝切线相位解缠的结果，并对结果进行分析。

二、实习原理

2.1 相位解缠

2.1.1 相位缠绕与相位解缠的概念

相位缠绕 (phase wrapping) 就是将真实相位的 2π 整周部分消除，剩下非整周的主值部分 $[-\pi, \pi)$ 。其逆向过程即为相位解缠 (phase unwrapping)，核心是恢复 2π 整周数。



由于干涉图像包含的相位信息是缠绕相位，因此相位解缠是 InSAR 数据处理流程中的一个关键环节，它通常也是 InSAR 产品的一大误差源。

2.1.2 相位解缠方法概述

自 20 世纪 70 年代末至今，人们已经发展了几十种相位解缠算法，各类相位解缠算法都有利有弊。因此如何选择合适的相位解缠算法，能达到既快速又精确的解缠效果成为了重点及难点。

相位解缠算法主要可以分为三大类：

(1) 第一类是以枝切法为代表的基于路径跟踪的相位解缠算法。

它主要是通过沿着预先确定的一致性路径进行相邻像元的相位差值积分来实现相位解缠。积分时路径要避开一些低质量、不一致的区域，这是路径跟踪算法的核心思想。这些方法都是一种局域算子，即误差被限制在局部区域内不会传播。

(2) 第二类是以最小二乘法为代表的基于最小范数思想的相位解缠算法。

它是通过在整体上使缠绕相位的梯度与真实相位的梯度差的平方最小来实现相位解缠。它与路径跟踪法不同的地方是，最小二乘法是一种全局性的优化算子。

(3) 第三类是以最优估计为基础的最小网络费用流算法。

网络流算法为了避免影像过大产生的影响，采取了分块策略、瓦片算法，将一幅大尺寸的干涉影像图分成若干尺寸稍小点的干涉图，或“瓦片”组合成一幅马赛克图。这样就可以来解缠这些分解开的瓦片干涉图来减少内存的需求。网络流算法通过寻求最少费用流来达到解缠的目的。

2.2 支切线法相位解缠

2.2.1 支切线法概述

枝切线算法是由 Goldstein 提出的经典路径跟踪算法，该算法运算较快，对内存需求小，基本上没有利用的辅助信息，在残差点过于密集的情况下，枝切线难以正确设置，采取最邻近原则也不能保证枝切线放置的位置是最合理的，导致造成解缠误差。算法本质上依赖接于点状不连续区域的检测，所以不适合与大面积的不连续区域的解缠。

其基本原理为：残差点导致解缠与路径有关，所以相位解缠路径只要避开这些残差点，就可以实现和路径无关，而且能保证解缠结果准确。Goldstein 枝切法就是利用连接残差点的支切线（branch），让解缠路径躲开这些区域，解缠就可以顺利进行。算法核心为如何连接残差点，通过支切线（branch）连接距离最近的正负残差点，并使它们达到平衡（balance），即正负残差点的代数和为 0。

2.2.2 前提条件

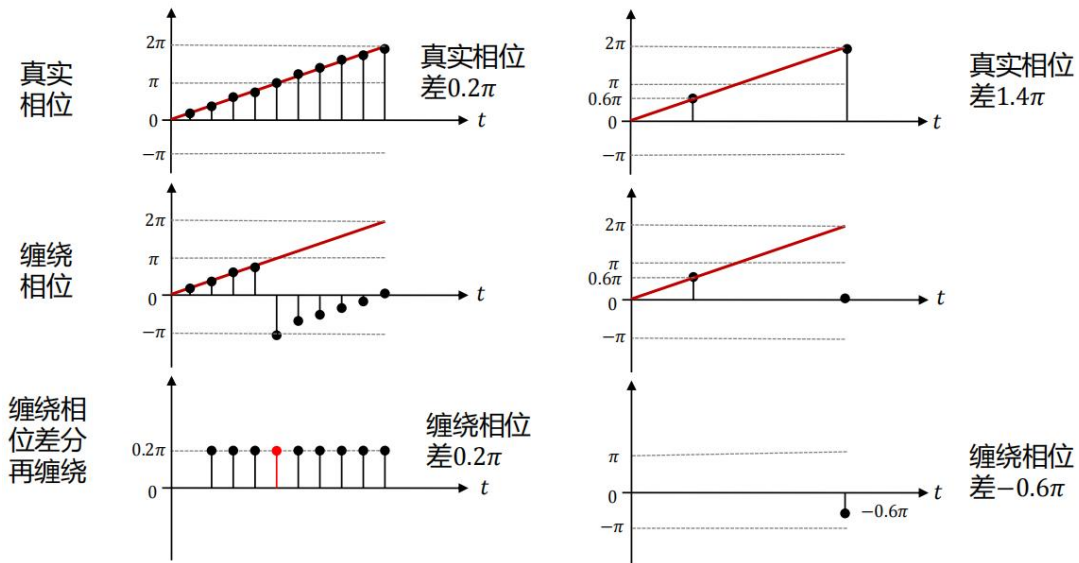
枝切线算法的前提条件是真实相位应满足采样定理。

采样定理，又称香农采样定律、奈奎斯特采样定律，是信息论，特别是通讯与信号处理学科中的一个重要基本结论。其内容为为了不失真地恢复模拟信号，采样频率应该大于等于模拟信号频谱中最高频率的 2 倍。

在相位解缠中则为：对连续相位进行离散采样，离散相位两点之间的相位差不应大于 1/2 周期，即 π 。否则，会出现相位混叠现象，无法正确恢复真实相位。

真实相位满足 $-\pi \leq \Delta\{\varphi(n)\} < \pi$

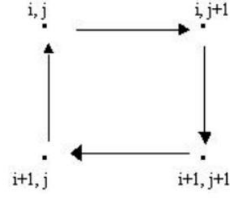
真实相位不满足 $-\pi \leq \Delta\{\varphi(n)\} < \pi$



2.2.3 残差点检测

在每一点(i,j)上计算相邻点相位梯度，判断路径一致性，不一致路径的点为残差点。相位图中，最小环路路径是一个 2x2 像素块。所以残差点其实不是指一个像素，而是指一个 2x2 像素块，一般标记左上角像素为残差点。

2x2 像素块环路积分的结果只有 3 种情况（残差点极性，charge）：0， ± 1 （ 2π 倍数）。当环路积分结果等于 0，这是一个正常的位置；等于+1，定义为正残差点；等于-1，定义为负残差点。

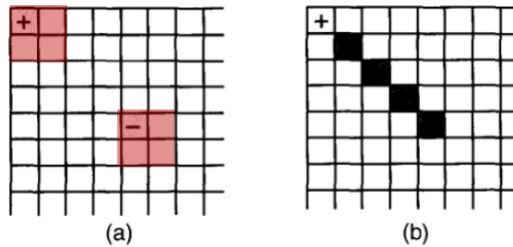
$$\begin{aligned}\Delta\varphi_1 &= \varphi_{i,j} - \varphi_{i+1,j} \\ \Delta\varphi_2 &= \varphi_{i+1,j} - \varphi_{i+1,j+1} \\ \Delta\varphi_3 &= \varphi_{i+1,j+1} - \varphi_{i,j+1} \\ \Delta\varphi_4 &= \varphi_{i,j+1} - \varphi_{i,j}\end{aligned}$$


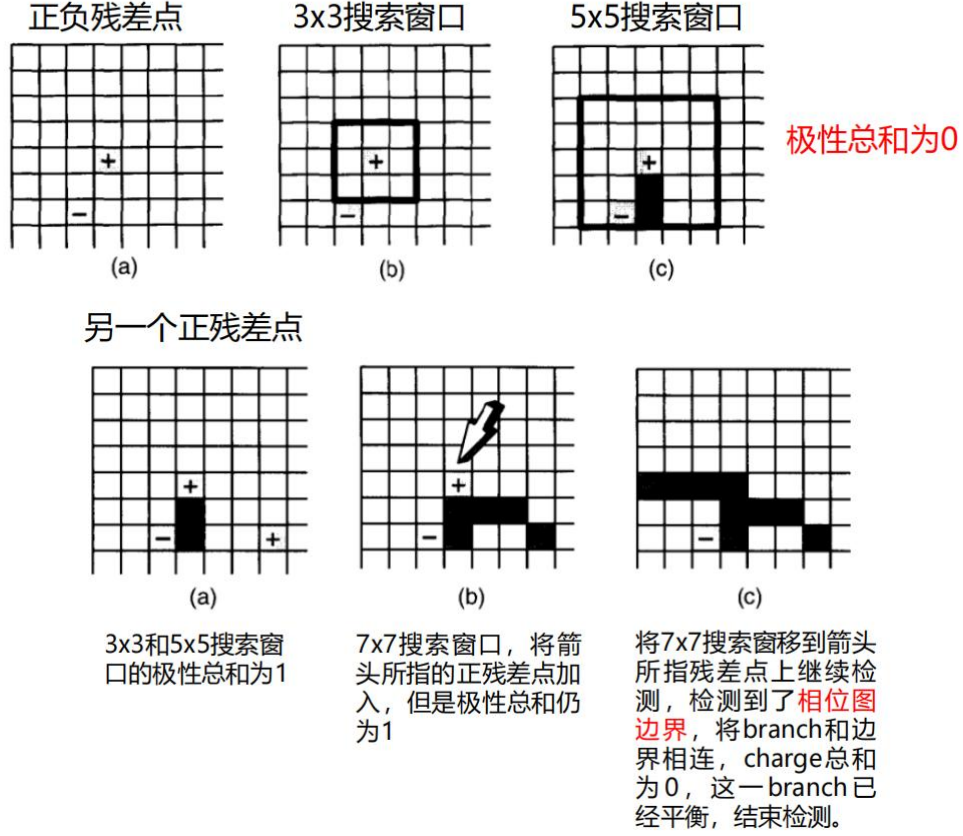
$$W(\Delta\varphi) = \begin{cases} \Delta\varphi, & |\Delta\varphi| \leq 0.5 \\ \Delta\varphi + 1, & |\Delta\varphi + 1| \leq 0.5 \\ \Delta\varphi - 1, & |\Delta\varphi - 1| \leq 0.5 \end{cases}$$

$$S = \sum_{i=1}^4 W(\Delta\varphi_i) \begin{cases} = 0 & \text{一致性路径} \\ \neq 0 & \text{不一致性路径} \end{cases}$$

2.2.4 支切线连接

- (1) 首先检测出所有正负残差点。
- (2) 将一个 3x3 搜索窗口放在某个残差点位置，寻找窗口内别的残差点。
 - (a) 如果找到了，将它们用 **branch** 连接起来。
 - ①如果另一个残差点的极性与当前残差点相反，就认为这个 **branch** 是平衡的；
 - ②如果另一个残差点的极性和当前残差点相同，那搜索窗口就放在新检测到的这个残差点位置，继续寻找残点，直到找到极性相反的残点并将它们用 **branch** 连接起来且他们极性总和为 0。
 - (b) 如果在搜索窗口中无残差点被检测到，那么搜索窗口增大，变成 5x5 窗口，继续从开始的那个残差点来检测别的残差点。
- (3) 残差点连接：
 - (a) 在搜索窗中检测到了残差点，要将它们用 **branch** 连接起来。如果检测到的这个残差点之前还未被 **branch** 连接过，它的极性需要加到 **charge** 里面来，如果连接过了就不需要加；
 - (b) 如果搜索窗遇到了相位图边界，直接把 **branch** 和边界连接，同时认定这一 **branch** 的 **charge** 总和为 0；
 - (c) 残差点实际是一个 2x2 像素的小区域，一般标记这个小区域左上角的那个像素为残差点，但在残差点的连接中，不是必须连接被标记的两个像素，而是选择两个小区域最短的路径将他们连接起来。





2.2.5 相位解缠

使用 Itoh 一维相位解缠方法进行解缠。一维相位解缠可以总结为：解缠相位可以通过累加缠绕相位差值的缠绕相位而求得。

遍历全部相位值，比较前后两个位置的相位值，求后一点与前一个点的相位差，如果相位差大于 π ，则后一个点的相位减 2π ；如果相位差小于 $-\pi$ ，则后一个点的相位加 2π ；如果相位差大于 $-\pi$ 小于 π ，则继续比较下一位置的相位。相位差全部完成后，从第一个缠绕相位开始，累加相位差，即为解缠相位。

可以表示为下面的式子：

$$\varphi_{uw}(m) = \varphi_w(0) + \sum_{n=0}^{m-1} W\{\Delta\{\varphi_w(n)\}\}, m = 1, \dots, N.$$

$$\varphi_w(n) = \varphi(n) \pm 2\pi k_1(n)$$

$$\Delta\{\varphi_w(n)\} = \Delta\{\varphi(n)\} \pm 2\pi\Delta\{k_1(n)\}$$

$$W\{\Delta\{\varphi_w(n)\}\} = \Delta\{\varphi(n)\} \pm 2\pi[\Delta\{k_1(n)\} + k_2(n)]$$

三、实习数据

本次实习提供了一个 8×8 的模拟数据代替 SAR 数据，作为真实相位，如下所示：

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

四、实习内容

4.1 总体流程

根据支切线法相位解缠的原理，程序需要实现以下 5 方面内容：

- (1) 输入真实相位：实习提供 8×8 的模拟数据代替 SAR 数据，将其导入作为真实相位；
- (2) 相位缠绕：将真实相位缠绕，得到缠绕相位；
- (3) 残差点检测：利用缠绕相位，检测其中的残差点；
- (4) 支切线生成：根据残差点检测结果，按照支切线相关算法，生成支切线；
- (5) 相位解缠：根据支切线，对缠绕相位进行解缠，恢复真实相位。



4.2 导入真实相位

初始化一个二维数组 `TrueMatrix`，用于存储某个区域的真实相位信息。

具体步骤如下：

(1) 动态内存分配：首先，代码通过动态内存分配创建了一个 `double` 类型的二维数组 `TrueMatrix`，其尺寸为 8×8 。这是通过两次循环完成的，外层循环为每一行分配内存，内层循环则为每一行中的每个元素分配内存。

(2) 初始化为零：在分配完内存后，代码通过两个嵌套的 `for` 循环将 `TrueMatrix` 中的所有元素初始化为 0.0。这里需要注意的是，内层循环的条件应该是 `j < col` 而不是 `j < row`，可能是原代码的一个小错误。

(3) 赋真实相位值：接下来，定义了一个二维数组 `data`，它直接存储了预设的真实相

位数据，尺寸同样为 8x8。然后，代码再次通过两个嵌套的 for 循环，将 data 中的值逐个复制到之前初始化的 TrueMatrix 中。

```
//初始化某区域的真实相位
int row = 8, col = 8;
double** TrueMatrix=new double*[row];
for (int i = 0; i < row; i++) {
    TrueMatrix[i] = new double[col];
}
for (int i = 0; i < row; i++)
    for (int j = 0; j < col; j++)
        TrueMatrix[i][j] = 0.0;

double data[8][8] = {0.0,0.0,0.3,0.0,0.0,0.3,0.0,0.0},{0.0,0.3,0.6,0.3,0.3,0.6,0.3,0.0},
{0.0,0.0,0.9,0.6,0.6,0.9,0.0,0.0},{0.0,0.0,1.2,0.9,0.9,1.2,0.0,0.0}, {0.0,0.0,1.2,0.9,0.9,1.2,0.0,0.0},
{0.0,0.0,0.9,0.6,0.6,0.9,0.0,0.0},{0.0,0.3,0.6,0.3,0.3,0.6,0.3,0.0},{0.0,0.0,0.3,0.0,0.0,0.3,0.0,0.0} }
;

for (int i = 0; i < row; i++)
{
    for (int j = 0; j < col; j++)
    {
        TrueMatrix[i][j] = data[i][j];
    }
}
```

4.3 相位缠绕

本部分主要进行真实相位的缠绕，生成缠绕相位，作为相位解缠编程的输入数据。
SrcMatrix 为真实相位，WrapMatrix 存储缠绕相位。

步骤如下：

- (1) 定义缠绕相位矩阵与分配内存
- (2) 遍历真实相位矩阵
- (3) 使用缠绕函数缠绕每个值

```
/*将一个数值缠绕到[-0.5,0.5)*/
double WrapPhase(double Phase) {
    Phase = Phase - floor(Phase + 0.5);
    return Phase;
}

/*将真实相位矩阵转为缠绕相位矩阵*/
void WrapPhase(int row, int col, double** SrcMatrix, double** WrapMatrix) {
    for (int i = 0; i < row; i++) {
        for (int j = 0; j < col; j++) {
            WrapMatrix[i][j] = WrapPhase(SrcMatrix[i][j]);
        }
    }
}
```


主函数部分：

```
//初始化某区域的缠绕相位
double** WrapMatrix = new double* [row];
for (int i = 0; i < row; i++) {
    WrapMatrix[i] = new double[col];
}
for (int i = 0; i < row; i++)
    for (int j = 0; j < row; j++)
        WrapMatrix[i][j] = 0.0;

WrapPhase(row, col, TrueMatrix, WrapMatrix);
```

4.4 残差点检测

本部分主要对缠绕相位进行残差点的计算，寻找正/负残差点，生成一个 0/1/-1 标记的矩阵。WrapMatrix 为缠绕相位，ChargeMatrix 记录残差点的极性。

步骤如下：

- (1) 定义残差点标记矩阵与分配内存
- (2) 矩阵初始化，全部设为 0
- (3) 遍历缠绕相位矩阵，求 $\Delta 1$ 、 $\Delta 2$ 、 $\Delta 3$ 、 $\Delta 4$
- (4) 求和 $\Delta 1 + \Delta 2 + \Delta 3 + \Delta 4$ ，根据其正、负、零符号判断正/负残差点和正常点。

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.0	→ -0.4	0.3	0.3	-0.4	0.3	0.0
0.0	↑ +1	0.0	↓ -0.1	-0.4	-0.4	-0.1	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0
0.0	-1	0.3	-0.4	0.3	0.3	-0.4	0.3
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

```
void ResidueDetection(int row, int col, double** WrapMatrix, double** ChargeMatrix) {
    //ChargeMatrix[row][col] = { 0 };
    for (int i = 0; i < row - 1; i++) {
        for (int j = 0; j < col - 1; j++) {
            double delta1 = WrapPhase(WrapMatrix[i][j + 1] - WrapMatrix[i][j]);
            double delta2 = WrapPhase(WrapMatrix[i + 1][j + 1] - WrapMatrix[i][j + 1]);
            double delta3 = WrapPhase(WrapMatrix[i + 1][j] - WrapMatrix[i + 1][j + 1]);
            double delta4 = WrapPhase(WrapMatrix[i][j] - WrapMatrix[i + 1][j]);
            double sum = delta1 + delta2 + delta3 + delta4;
            if (sum > 0.0001) ChargeMatrix[i][j] = 1;
            if (sum < -0.0001) ChargeMatrix[i][j] = -1;
        }
    }
}
```

主函数部分：

```
//初始化残差点数组
double** ChargeMatrix = new double* [row];
for (int i = 0; i < row; i++) {
    ChargeMatrix[i] = new double[col];
}
for (int i = 0; i < row; i++)
    for (int j = 0; j < col; j++)
        ChargeMatrix[i][j] = 0.0;

ResidueDetection(row, col, WrapMatrix, ChargeMatrix);
```

4.5 支切线生成

本部分主要根据上一步得到的残差点，按照 Goldstein 支切线算法生成支切线，得到一个 0/1 值标记的支切线标记 line 矩阵。

(1) 初始化：分配内存给两个临时矩阵 (Residue_mark 和 window_mark) 来跟踪已处理的残留点和搜索窗口。同时初始化向量来存储残留点的位置。

(2) 残留点检测：遍历 ChargeMatrix 来查找残留点 (charge 不为 0 的位置)，计数它们，并记录其位置。

(3) 创建枝切线：

- 对于找到的每个残差点：
- 检查它是否已经被标记。如果没有，则继续。
- 确定该位置的电荷。
- 进入一个循环以使用分支切割中和 charge：
 - 在扩展的窗口 (3x3, 5x5, 7x7) 内搜索可能的残差点：
 - 如果残留点在边界上，直接将其与边缘连接。
 - 否则，在窗口内寻找其他残留点。如果找到，使用 Bresenham 算法将它们连接起来，更新 charge，并标记新位置以进行进一步处理。
 - 如果当前窗口大小内没有建立连接，则移动到下一个更大的窗口，直到 charge 得到 balance。

(4) 结果：line 矩阵将根据创建的分支切割进行更新，标记出连接残差点并中和 charge 分布的路径。

```
//使用 Bresenham 算法画线
void drawLine(double** line, int x1, int y1, int x2, int y2) {
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = (x1 < x2) ? 1 : -1;
    int sy = (y1 < y2) ? 1 : -1;
    int err = dx - dy;

    while (true) {
        line[y1][x1] = 1; // 设置点为 1
    }
}
```

```

        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x1 += sx;
        }
        if (e2 < dx) {
            err += dx;
            y1 += sy;
        }
    }
}

void Link(double** line, int row1, int col1, int row2, int col2)//连接残差点{
    //使用 Bresenham 算法画线
    drawLine(line, col1, row1, col2, row2);
}

void BranchCut(int row, int col, double** line, double** ChargeMatrix){
    double** Residue_mark = new double* [row];
    for (int i = 0; i < row; i++) Residue_mark[i] = new double[col];
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            Residue_mark[i][j] = 0;

    double** window_mark = new double* [row];
    for (int i = 0; i < row; i++) window_mark[i] = new double[col];
    for (int i = 0; i < row; i++)
        for (int j = 0; j < col; j++)
            window_mark[i][j] = 0;

    //记录残差点位置和数量
    int Residue_num = 0;
    vector<int>Residue_row;
    vector<int>Residue_col;
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            line[i][j] = 0;
            if (ChargeMatrix[i][j] != 0)
            {
                Residue_num++;
                Residue_row.push_back(i);
            }
        }
    }
}

```

```

        Residue_col.push_back(j);
    }
}

for (int i = 0; i < Residue_num; i++)
{
    int nowrow = Residue_row[i];
    int nowcol = Residue_col[i];
    if (Residue_mark[nowrow][nowcol] == 0) Residue_mark[nowrow][nowcol] = 1; //检查
    残差点是否被标记
    int charge = ChargeMatrix[nowrow][nowcol];
    int break_mark = 0;

    while (charge != 0)
    {
        //3*3 窗口、5*5 窗口、7*7 窗口依次搜索
        for (int halfwindow = 1; halfwindow <= 3; halfwindow++)
        {
            // 残差点位于边界
            if (nowrow == 0 || nowcol == 0 || nowrow == row - 1 || nowcol == col - 1)
            {
                line[nowrow][nowcol] = 1;
                Residue_mark[nowrow][nowcol] = 1;
                charge = 0;
                break;
            }
            // 残差点不位于边界
            for (int x = nowrow - halfwindow; x <= nowrow + halfwindow; x++)
            {
                for (int y = nowcol - halfwindow; y <= nowcol + halfwindow; y++)
                {
                    break_mark = 0;
                    // 当搜索窗口遇到了相位图边界，直接把 branch 和边界连接
                    if (abs(x - nowrow) == halfwindow || abs(y - nowcol) ==
halfwindow)
                    {
                        if (x <= 0 || x >= row - 1 || y <= 0 || y >= col - 1)
                        {
                            // 当窗口遇到上边界
                            if (x <= 0) {
                                x = 0;
                                y = nowcol;
                                Link(line, nowrow, nowcol, x, y);

```

```

        break_mark = 1;
    }
    // 当窗口遇到下边界
    if (x >= row - 1) {
        x = row - 1;
        y = nowcol;
        Link(line, nowrow, nowcol, x, y);
        break_mark = 1;
    }
    // 当窗口遇到左边界
    if (y <= 0) {
        x = nowrow;
        y = 0;
        Link(line, nowrow, nowcol, x, y);
        break_mark = 1;
    }
    // 当窗口遇到右边界
    if (y >= col - 1) {
        x = nowrow;
        y = col - 1;
        Link(line, nowrow, nowcol, x, y);
        break_mark = 1;
    }
    charge = 0;
}
// 找到其他残差点
if (ChargeMatrix[x][y] != 0 && window_mark[x][y] != 1) {
    Link(line, nowrow, nowcol, x, y); // 进行连接
    if (Residue_mark[x][y] != 1)
    {
        charge = charge + ChargeMatrix[x][y];
        Residue_mark[x][y] = 1;
    }
    if (charge == 0) {
        break_mark = 1;
    }
    else {
        nowrow = x;
        nowcol = y;
        window_mark[x][y] = 1;
        break_mark = 1;
    }
}
}
}

```

```

        if (break_mark == 1) break;
    }
    if (break_mark == 1) break;
}
if (break_mark == 1) break;
}
}
}
}
}

```

主函数部分：

```

//初始化枝切线数组
double** line = new double* [row];
for (int i = 0; i < row; i++) {
    line[i] = new double[col];
}
for (int i = 0; i < row; i++)
    for (int j = 0; j < col; j++)
        line[i][j] = 0.0;

BranchCut(row, col, line, ChargeMatrix);

```

4.6 相位解缠

本部分主要根据上一步得到的支切线结果，首先绕开支切线，对支切点之外的点解缠，然后对支切线上的点解缠，得到解缠相位矩阵，通过 `UnWrap` 和 `UnwrapPhase` 两个函数实现，`WrapMatrix` 为缠绕相位，`branchcut` 枝切线数组，`UnWrappedMatrix` 存储解缠相位。

支切线外的点解缠步骤如下：

- (1) 定义解缠相位矩阵，及各种标记矩阵，并初始化为全 0；
- (2) 将第一个非支切点作为已知点；
- (3) 循环找到一个符合条件的种子点，以开始解缠；
- (4) 从一个种子点开始，不停循环寻找解缠路径，直到死胡同；
- (5) 循环结束，解缠所有支切线外的点。

支切线上的点解缠步骤如下：

- (1) 统计支切点的个数和每个支切点的位置；
- (3) 循环处理每个支切点；
- (4) 查看支切点的 4 邻域，若能解缠则解缠；
- (5) 记录已解缠支切点数量，当其小于支切点总数时，继续循环；
- (6) 循环结束，解缠所有支切线上的点。

```

double UnWrap(double wrap, double unwrap) {
    //单点相位解缠
    double difference = wrap - WrapPhase(unwrap); //缠绕相位差分
    double wrapdif = WrapPhase(difference); // 差分结果再缠绕
}

```

```

        return unwrap + wrapdif; // 加上已知值
    }

    void UnwrapPhase(int row, int col, double** WrapMatrix, double** branchcut, double**
UnWrappedMatrix){
        double** UnWrapFlag = new double* [row];
        for (int i = 0; i < row; i++) UnWrapFlag[i] = new double[col]; //标记是否已解缠，邻接表
        double** medium_mark = new double* [row];
        for (int i = 0; i < row; i++) medium_mark[i] = new double[col];
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                UnWrapFlag[i][j] = 0;
                medium_mark[i][j] = 0;
                UnWrappedMatrix [i][j] = 0;
            }
        }

        // 第一个非枝切线点，将其解缠相位视为基准
        int break_mark = 0;
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (branchcut[i][j] == 0)
                {
                    UnWrappedMatrix[i][j] = WrapMatrix[i][j];
                    UnWrapFlag[i][j] = 1;
                    break_mark = 1;
                    break;
                }
            }
            if (break_mark == 1) break;
        }

        while (1) {
            int mediumrow = 100;
            int mediumcol = 100;
            // 寻找解缠中心点
            break_mark = 0;
            for (int i = 0; i < row; i++) {
                for (int j = 0; j < col; j++) {
                    // 寻找未作为解缠中心的点作为解缠中心
                    if (UnWrapFlag[i][j] == 1 && medium_mark[i][j] == 0) {
                        medium_mark[i][j] = 1;
                        mediumrow = i;
                        mediumcol = j;
                    }
                }
            }
        }
    }

```



```

        break_mark = 1;
        break;
    }
}
if (break_mark == 1) break;
}
if (mediumrow == 100 && mediumcol == 100) break;
int mark = 0;
while (mark == 0) {
    mark = 1;
    //左邻域
    if (mediumcol - 1 >= 0 && mark == 1) {
        if (UnWrapFlag[mediumrow][mediumcol - 1] == 0 &&
branchcut[mediumrow][mediumcol - 1] == 0) {
            mark = 0;
            UnWrappedMatrix[mediumrow][mediumcol - 1] =
UnWrap(WrapMatrix[mediumrow][mediumcol - 1], UnWrappedMatrix[mediumrow][mediumcol]);
            UnWrapFlag[mediumrow][mediumcol - 1] = 1;
            mediumcol = mediumcol - 1; //中心点变为左邻域点
        }
    }
    //右邻域
    if (mediumcol + 1 < col && mark == 1) {
        if (UnWrapFlag[mediumrow][mediumcol + 1] == 0 &&
branchcut[mediumrow][mediumcol + 1] == 0) {
            mark = 0;
            UnWrappedMatrix[mediumrow][mediumcol + 1] =
UnWrap(WrapMatrix[mediumrow][mediumcol + 1], UnWrappedMatrix[mediumrow][mediumcol]);
            UnWrapFlag[mediumrow][mediumcol + 1] = 1;
            mediumcol = mediumcol + 1; //中心点变为右邻域点
        }
    }
    // 上邻域
    if (mediumrow - 1 >= 0 && mark == 1) {
        if (UnWrapFlag[mediumrow - 1][mediumcol] == 0 &&
branchcut[mediumrow - 1][mediumcol] == 0) {
            mark = 0;
            UnWrappedMatrix[mediumrow - 1][mediumcol] =
UnWrap(WrapMatrix[mediumrow - 1][mediumcol], UnWrappedMatrix[mediumrow][mediumcol]);
            UnWrapFlag[mediumrow - 1][mediumcol] = 1;
            mediumrow = mediumrow - 1; //中心点变为上邻域点
        }
    }
}
}

```

```

        //上邻域
        if (mediumrow + 1 < row && mark == 1) {
            if (UnWrapFlag[mediumrow + 1][mediumcol] == 0 &&
branchcut[mediumrow + 1][mediumcol] == 0) {
                mark = 0;
                UnWrappedMatrix[mediumrow + 1][mediumcol] =
UnWrap(WrapMatrix[mediumrow + 1][mediumcol], UnWrappedMatrix[mediumrow][mediumcol]);
                UnWrapFlag[mediumrow + 1][mediumcol] = 1;
                mediumrow = mediumrow + 1; //中心点变为上邻域点
            }
        }
    }
}

//解缠枝切线上的点
int line_num = 0;
vector<int> line_row; // 储存行号
vector<int> line_col; // 储存列号
for (int i = 0; i < row; i++) {
    for (int j = 0; j < col; j++) {
        if (branchcut[i][j] != 0) {
            line_num++;
            line_row.push_back(i);
            line_col.push_back(j);
        }
    }
}
int unwrappedline_num = 0;
while (unwrappedline_num < line_num) {
    for (int i = 0; i < line_num; i++) {
        int mediumrow = line_row[i];
        int mediumcol = line_col[i];
        int mark = 0;

        // 左邻域
        if (mediumcol - 1 >= 0 && mark == 0) {
            if (UnWrapFlag[mediumrow][mediumcol - 1] == 1 &&
UnWrapFlag[mediumrow][mediumcol] == 0) {
                mark = 1;
                UnWrappedMatrix[mediumrow][mediumcol] =
UnWrap(WrapMatrix[mediumrow][mediumcol], UnWrappedMatrix[mediumrow][mediumcol - 1]);
                UnWrapFlag[mediumrow][mediumcol] = 1;
            }
        }
    }
}

```

```

        // 右邻域
        if (mediumcol + 1 <= col && mark == 0) {
            if (UnWrapFlag[mediumrow][mediumcol + 1] == 1 &&
UnWrapFlag[mediumrow][mediumcol] == 0) {
                mark = 1;
                UnWrappedMatrix[mediumrow][mediumcol] =
UnWrap(WrapMatrix[mediumrow][mediumcol], UnWrappedMatrix[mediumrow][mediumcol + 1]);
                UnWrapFlag[mediumrow][mediumcol] = 1;
            }
        }

        //上邻域
        if (mediumrow - 1 >= 0 && mark == 0) {
            if (UnWrapFlag[mediumrow-1][mediumcol] == 1 &&
UnWrapFlag[mediumrow][mediumcol] == 0) {
                mark = 1;
                UnWrappedMatrix[mediumrow][mediumcol] =
UnWrap(WrapMatrix[mediumrow][mediumcol], UnWrappedMatrix[mediumrow-1][mediumcol]);
                UnWrapFlag[mediumrow][mediumcol] = 1;
            }
        }

        //下邻域
        if (mediumrow + 1 <= row && mark == 0) {
            if (UnWrapFlag[mediumrow + 1][mediumcol] == 1 &&
UnWrapFlag[mediumrow][mediumcol] == 0) {
                mark = 1;
                UnWrappedMatrix[mediumrow][mediumcol] =
UnWrap(WrapMatrix[mediumrow][mediumcol], UnWrappedMatrix[mediumrow + 1][mediumcol]);
                UnWrapFlag[mediumrow][mediumcol] = 1;
            }
        }
        if (mark == 1) {
            unwrappedline_num++;
        }
    }
    delete[] UnWrapFlag;
    delete[] medium_mark;
}

```

主函数部分：

```
//初始化解缠矩阵
```

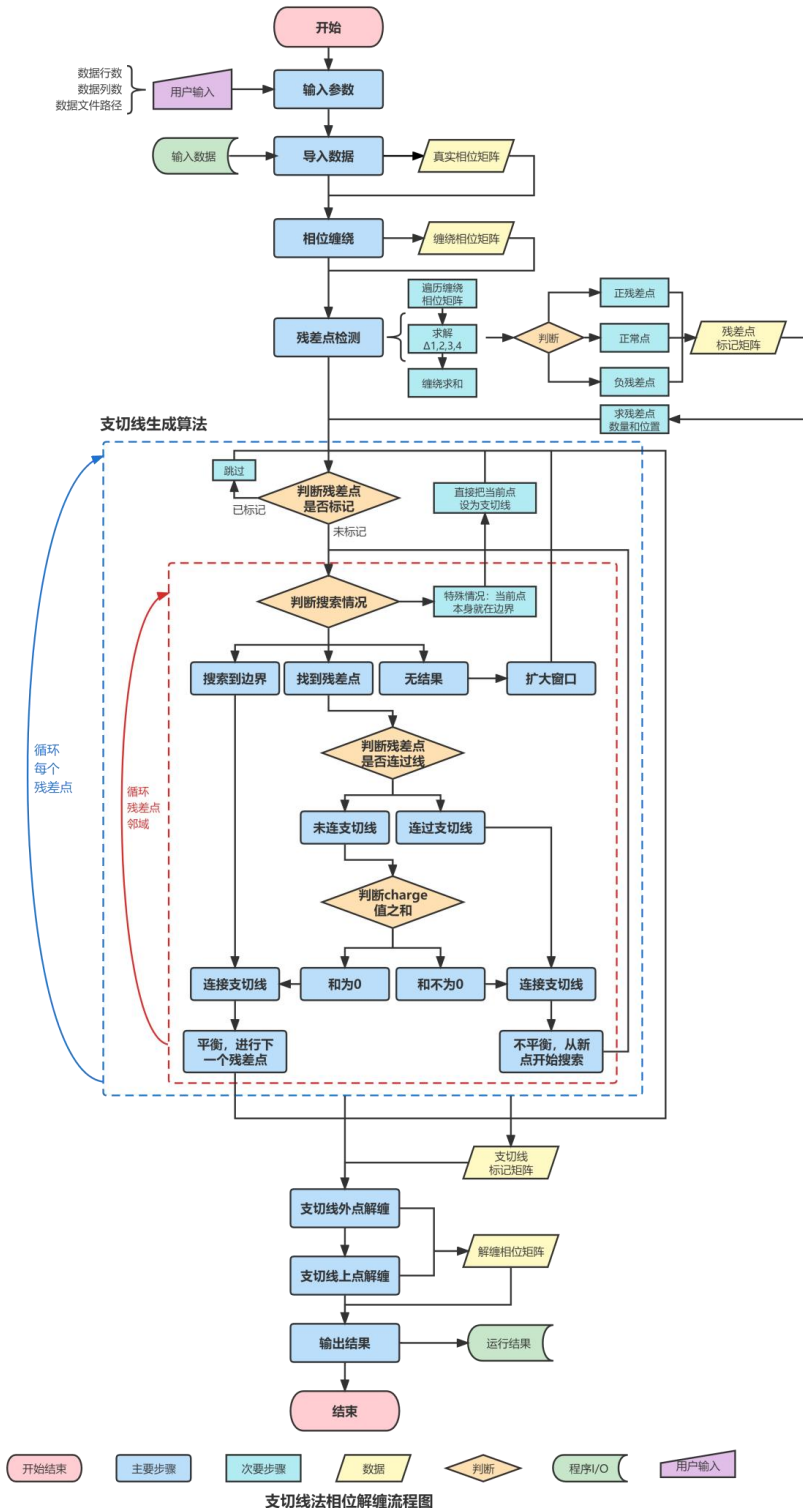
```
double** UnWrappedMatrix = new double* [row];
for (int i = 0; i < row; i++) {
    UnWrappedMatrix[i] = new double[col];
}
for (int i = 0; i < row; i++)
    for (int j = 0; j < col; j++)
        UnWrappedMatrix[i][j] = 0.0;

UnwrapPhase(row, col, WrapMatrix, line, UnWrappedMatrix);
```

五、实习结果与分析

5.1 主要函数说明

- (1) WrapPhase: 对真实相位进行缠绕;
- (2) ResidueDetection: 残差点检测;
- (3) BranchCut: 支切线生成, 包含 Link 和 drawLine 函数;
- (4) UnwrapPhase: 相位解缠, 包含 UnWrap 函数。



5.2 运行结果

程序运行结果如下两图所示：

初始真实相位：								新枝切线连接数组：							
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
缠绕相位：								旧枝切线解缠结果：							
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.0	-0.7	-1.0	-1.0	-0.7	0.0	0.0
0.0	0.3	-0.4	0.3	0.3	-0.4	0.3	0.0	0.0	0.0	0.3	-0.4	-0.7	-0.7	-0.4	0.3
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0
0.0	0.3	-0.4	0.3	0.3	-0.4	0.3	0.0	0.0	-1.0	-0.7	-0.4	-0.7	-0.7	-0.4	-0.7
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	-1.0	-1.0	-0.7	-1.0	-1.0	-0.7	-1.0
残差点数组：								新枝切线解缠结果：							
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0
0.0	1.0	0.0	0.0	0.0	0.0	-1.0	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9	0.6	0.6	0.9	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0
0.0	-1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.9	0.6	0.6	0.9	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0

5.3 正确解缠结果

当支切线正确方式连接时，可得到正确的解缠结果。各步骤正确结果记录如下各表所示：

(1) 真实相位：

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

(2) 缠绕相位：

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.3	-0.4	0.3	0.3	-0.4	0.3	0.0
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0
0.0	0.3	-0.4	0.3	0.3	-0.4	0.3	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

(3) 残差点检测:

0	0	0	0	0	0	0	0
0	1	0	0	0	-1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	-1	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(4) 支切线连接:

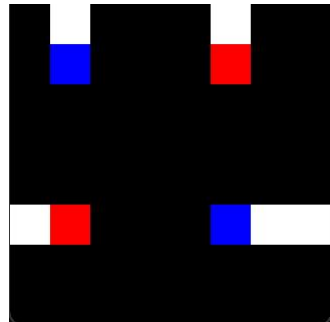
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

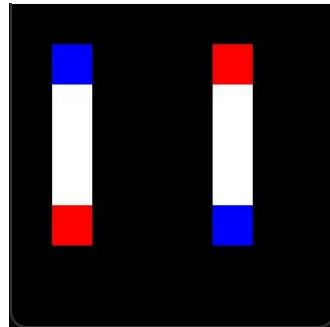
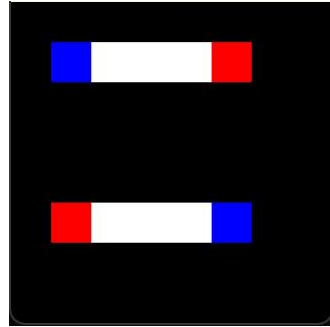
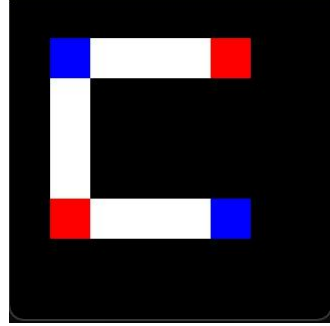
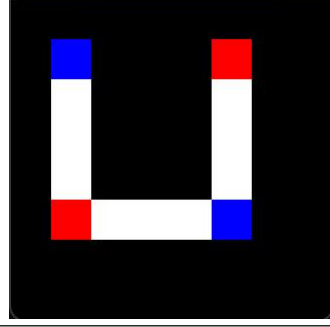
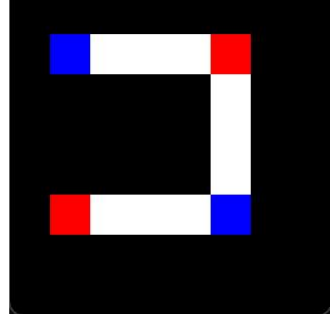
(5) 解缠相位:

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

5.4 不同支切线结果分析

除了算法规定的默认情况外,手动使支切线的连接方式改变,对比几种不同支切线的结果,总结如下表所示(枝切线连接方式图像中红色为负残差点,蓝色为正残差点,白色为枝切线):

枝切线连接方式	相位解缠结果	正确性																																																																
	<table><tr><td>0.0</td><td>0.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>-0.4</td><td>-0.7</td><td>-0.7</td><td>-0.4</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>0.6</td><td>0.6</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>-1.0</td><td>-0.7</td><td>-0.4</td><td>-0.7</td><td>-0.7</td><td>-0.4</td><td>-0.7</td><td>-1.0</td></tr><tr><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td></tr></table>	0.0	0.0	-0.7	-1.0	-1.0	-0.7	0.0	0.0	0.0	0.3	-0.4	-0.7	-0.7	-0.4	0.3	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	-0.1	0.6	0.6	-0.1	0.0	0.0	-1.0	-0.7	-0.4	-0.7	-0.7	-0.4	-0.7	-1.0	-1.0	-1.0	-0.7	-1.0	-1.0	-0.7	-1.0	-1.0	不正确, 红色区域比真实值少 1
0.0	0.0	-0.7	-1.0	-1.0	-0.7	0.0	0.0																																																											
0.0	0.3	-0.4	-0.7	-0.7	-0.4	0.3	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	-0.1	0.6	0.6	-0.1	0.0	0.0																																																											
-1.0	-0.7	-0.4	-0.7	-0.7	-0.4	-0.7	-1.0																																																											
-1.0	-1.0	-0.7	-1.0	-1.0	-0.7	-1.0	-1.0																																																											

	<table><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.9</td><td>0.6</td><td>0.6</td><td>0.9</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>1.2</td><td>0.9</td><td>0.9</td><td>1.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>1.2</td><td>0.9</td><td>0.9</td><td>1.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.9</td><td>0.6</td><td>0.6</td><td>0.9</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr></table>	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	全部正确
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0																																																											
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0																																																											
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0																																																											
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
	<table><tr><td>0.0</td><td>0.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>-0.4</td><td>-0.7</td><td>-0.7</td><td>-0.4</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>-1.0</td><td>-0.7</td><td>-0.4</td><td>-0.7</td><td>-0.7</td><td>-0.4</td><td>-0.7</td><td>-1.0</td></tr><tr><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td><td>-0.7</td><td>-1.0</td><td>-1.0</td></tr></table>	0.0	0.0	-0.7	-1.0	-1.0	-0.7	0.0	0.0	0.0	0.3	-0.4	-0.7	-0.7	-0.4	0.3	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	-1.0	-0.7	-0.4	-0.7	-0.7	-0.4	-0.7	-1.0	-1.0	-1.0	-0.7	-1.0	-1.0	-0.7	-1.0	-1.0	不正确，红色区域比真实值少 1
0.0	0.0	-0.7	-1.0	-1.0	-0.7	0.0	0.0																																																											
0.0	0.3	-0.4	-0.7	-0.7	-0.4	0.3	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
-1.0	-0.7	-0.4	-0.7	-0.7	-0.4	-0.7	-1.0																																																											
-1.0	-1.0	-0.7	-1.0	-1.0	-0.7	-1.0	-1.0																																																											
	<table><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr></table>	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	不正确，红色区域比真实值少 1
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
	<table><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.9</td><td>0.6</td><td>0.6</td><td>0.9</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>1.2</td><td>0.9</td><td>0.9</td><td>1.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>1.2</td><td>0.9</td><td>0.9</td><td>1.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr></table>	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	不正确，红色区域比真实值少 1
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0																																																											
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0																																																											
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
	<table><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.2</td><td>-0.1</td><td>-0.1</td><td>0.2</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>-0.1</td><td>-0.4</td><td>-0.4</td><td>-0.1</td><td>0.0</td><td>0.0</td></tr><tr><td>0.0</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.3</td><td>0.6</td><td>0.3</td><td>0.0</td></tr><tr><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td><td>0.3</td><td>0.0</td><td>0.0</td></tr></table>	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0	0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0	0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0	0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0	不正确，红色区域比真实值少 1
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	0.2	-0.1	-0.1	0.2	0.0	0.0																																																											
0.0	0.0	-0.1	-0.4	-0.4	-0.1	0.0	0.0																																																											
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0																																																											
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0																																																											

由上面结果可以发现，标准的 Goldstein 算法的所有枝切线最终都与边界相连，但是最终造成了解缠结果的错误，采取认为方式读取多种枝切线矩阵，发现仅有当枝切线矩阵仅包含两条竖着的枝切线时，解缠结果才正确。

之所以会出现上述结果，原因是原数据的下图区域之间的差值大于 0.5，不满足 Nyquist

采样定律所造成，由于出现了频谱混叠。

0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	1.2	0.9	0.9	1.2	0.0	0.0
0.0	0.0	0.9	0.6	0.6	0.9	0.0	0.0
0.0	0.3	0.6	0.3	0.3	0.6	0.3	0.0
0.0	0.0	0.3	0.0	0.0	0.3	0.0	0.0

一旦解缠路径经过此处就无法恢复出真实相位，且由于解缠具有传递性，最后造成整幅影响受影响。当竖直方向存在两条支切线时，恰好可以避免解缠路径经过，因此解缠可以恢复出全图真实相位。

六、实习心得

本次实习中，我也或多或少的遇到了一些问题，总结如下：

（1）一是枝切线的生成。在编写支切线算法时，别看理论课程中支切线的生成原理可以用几句话概括，但在编程时仔细思考就能发现其背后需要考虑的情况太多了。此时需要冷静下来，按照计算机的逻辑拆分问题，逐个实现。同学的枝切线算法运用到了斜率、反正切的运算，十分复杂，我通过查阅资料了解到了 **Bresenham** 算法来生成枝切线，仅用十行就能实现相同功能，大大提高了效率。

（2）二是关于不同枝切线解缠结果的思考。在初次完整实现了程序后，安装算法规定的情况运行起来却发现总有几个结果不对，与真值相差了一整周。起初我认为是自己代码的问题，但仔细检查后也没有收获。这时我观察原数据才突然明白，发生问题的地方，数据之间的差值太大了，不满足采样定理才造成了错误。

（3）三是实际 **SAR** 影像解缠流程的思考。在实际应用中，由于噪声、欠采样等因素的影响，相位解缠成为一个非常具有挑战性的任务。本次实习数据是 8×8 的模拟相位，但是在实际应用中 **SAR** 影像更大，解缠操作更繁琐，会涉及到相位干涉图生成、相位滤波等等操作才能得到更加准确的相位信息，涉及的特殊情况只会更复杂。

通过这次实习，我不仅学会了如何使用枝切法进行 **SAR** 影像相位解缠，还掌握了如何针对具体问题进行算法优化。这对我今后的科研工作和工程实践具有很大的帮助。总之，实习过程中的挑战和困难使我不断成长，也让我更加深入地理解了 **InSAR** 技术的原理和应用。我相信，在未来的学习和工作中，我会继续努力，为 **InSAR** 技术的发展和应用做出自己的贡献

本次实习让我能够将《微波遥感》课程中的理论知识在编程中实际运用，提高了我解决问题的能力，感谢给我提供帮助的老师和同学们。