南 开 大 学

数 字 信 号 处 理 实 验 报 告

中文题目： **数字信号处理实验报告**

外文题目： Digital Signal Processing Experiment Report

学　　号： 2210556

姓　　名： 廖望

年　　级： 2023 级

专　　业： 计算机科学与技术

系　　别： 计算机科学与技术

学　　院： 计算机学院

指导教师： 李岳

完成日期： 2024 年 12 月

摘 要

本报告介绍了数字信号处理（DSP）基本操作的 C++ 实现与分析，重点在于离散信号序列的处理与操作。实验的主要目标是探索信号序列的定义、扩展、延迟、提前、反转、拉伸、压缩、差分、累加等基本操作，以及加法、乘法和卷积等多序列操作。实验还包括归一化相似性比对和滑动窗口相似性比对等技术。

本实验通过自定义类 `DS` 来实现灵活的信号序列操作，支持有限和无限序列，并且能够实时修改序列，适用于多种 DSP 任务。实验结果表明，所有操作均能正常工作，并支持动态序列扩展、卷积运算和相似性分析。此外，报告也简要分析了滑动窗口的效率优化。

关键词：数字信号处理（DSP）；信号序列操作；卷积与相似性比对

Abstract

This report presents the C++ implementation and analysis of basic Digital Signal Processing (DSP) operations, focusing on the processing and manipulation of discrete signal sequences. The main objective of the experiment is to explore basic operations on signal sequences, including definition, extension, delay, advance, reversal, stretching, compression, differencing, and accumulation, as well as multi-sequence operations such as addition, multiplication, and convolution. The experiment also covers techniques such as normalized similarity comparison and sliding window-based similarity comparison.

In this experiment, a custom class `DS` was used to implement flexible signal sequence operations, supporting both finite and infinite sequences, with the ability to modify the sequence in real time, making it suitable for various DSP tasks. The results show that all operations function correctly, supporting dynamic sequence extension, convolution operations, and similarity analysis. Additionally, the report briefly analyzes the efficiency optimization of the sliding window.

Key Words: Digital Signal Processing (DSP); Signal Sequence Operations; Convolution and Similarity Comparison

目 录

第一章　实验目的

本实验的主要目的如下：

1. 理解并掌握数字信号处理（DSP）中的基本操作；
2. 学习如何在编程环境中实现和处理信号序列；
3. 执行基本和多序列操作，并理解它们的应用；
4. 通过实验探索数字信号处理操作的效率和优化问题。

第二章　实验环境与工具

本实验选用 C++ 编程语言进行实现，使用了标准的输入输出流（I/O）和基础的算法库。通过实现一个名为 DS 的类，定义和操作信号序列。该类支持多种数字信号处理（DSP）操作，如序列延迟、提前、拉伸、压缩、卷积等。

第三章　实验内容

第一节　信号序列定义

在 C++ 中使用 std::vector<double> 类型来存储信号序列，并通过构造函数对信号序列进行初始化。类 DS 支持定义具有起始和结束时间的序列。可以根据需要初始化序列的起始位置、结束位置以及是否为无限序列（左侧或右侧无限）。

```
    // 初始化有限信号序列，起始位置为 l，结束位置为 r，支持左右无限扩展
    DS(int l, int r, bool infinite_left = false, bool infinite_right = false)
        : l(l), r(r), is_infinite_left(infinite_left), is_infinite_right(infinite_right) {
        a.resize(r - l + 1, 0);  // 初始化序列为零
    }
```

该构造函数支持创建一个数字信号序列，并根据 *is_infinite_left* 和 *is_infinite_right* 标志决定是否**支持向左或向右无限扩展**。

第二节　单序列操作

3.2.1　序列读取和修改

通过重载 [] 操作符，我们可以方便地访问和修改序列中的元素（**支持负数下标**）。

```
    double& operator[](int t) {
        if (t < l || t > r) {
            throw std::out_of_range("索引超出范围");
        }
        return a[t - l];
    }
    //例如 a[5] = 1, a[-4] = 2
```

3.2.2 无限长序列的输入

支持无限向左或向右读入（按 Ctrl+D 结束）。

```
// 读入数据，向左或向右扩展
void read_infinite(std::istream& is) {
    double value;
    char direction;

    // 提示用户选择读入的方向
    std::cout << "选择读入的方向 (l: 向左读入, r: 向右读入): ";
    std::cin >> direction;

    if (direction == 'r' || direction == 'R') {
        // 向右扩展序列
        if (is_infinite_right) {
            std::cout << "请输入向右扩展的序列数据 (Ctrl+D 结束): ";
            while (is >> value) {
                r++;
                a.push_back(value);  // 向右插入数据
            }
        } else {
            std::cout << "当前序列不支持向右扩展。" << std::endl;
        }
    }
    else if (direction == 'l' || direction == 'L') {
        // 向左扩展序列
        if (is_infinite_left) {
            std::cout << "请输入向左扩展的序列数据 (Ctrl+D 结束): ";
            while (is >> value) {
                l--;
                a.insert(a.begin(), value);  // 向左插入数
```

```
                        据
29                  }
30              } else {
31                  std::cout << "当前序列不支持向左扩展。" << std
                        ::endl;
32              }
33          } else {
34              std::cout << "无效的选择!" << std::endl;
35          }
36      }
```

3.2.3 前后补零

直接调用 insert() 函数插入 0，更新序列的时间范围：

```
1       // 用零填充序列
2       void pad_zero(int left, int right) {
3           l -= left;
4           r += right;
5           a.insert(a.begin(), left, 0);
6           a.insert(a.end(), right, 0);
7       }
```

3.2.4 序列的延迟与提前

通过修改序列的时间范围（ l 和 r ）执行延迟和提前操作，**无需直接移动序列，大大提升效率**：

```
1       void delay(int n) {
2           if (n < 0) throw std::invalid_argument("延迟不能为负数"
                );
3           l += n;  // 增加左边界
4           r += n;  // 增加右边界
5       }
6       void advance(int n) {
7           if (n < 0) throw std::invalid_argument("提前不能为负数"
                );
```

```
        l -= n;  // 减少左边界
        r -= n;  // 减少右边界
    }
```

3.2.5 序列反转

反转操作将序列中的所有元素顺序反转，并更新时间范围：

```
    void reverse() {
        std::reverse(a.begin(), a.end());
        int _l = l;
        l = -r;  // 更新时间范围
        r = -_l;
        std::swap(is_infinite_left, is_infinite_right);
    }
```

3.2.6 序列拉伸与压缩（上采样与下采样）

根据 factor 参数进行拉伸或压缩，拉伸操作填充 0 来增加序列的长度，压缩操作通过丢弃元素来减少序列的长度。**支持实时输入并采样：**

```
    void stretch(int factor, char direction = 'n') {
        if (factor <= 0) throw std::invalid_argument("拉伸因子
            必须是正数");
        // 具体实现细节
    }

    void compress(int factor, char direction = 'n') {
        if (factor <= 0) throw std::invalid_argument("压缩因子
            必须是正数");
        // 具体实现细节
    }
```

3.2.7 序列差分与累加

差分操作计算相邻元素的差，累加操作计算前缀和。**支持实时输入并计算差分或累加和：**

```
    void difference(char direction = 'n') {
        // 具体实现细节
    }

    void accumulate(char direction = 'n') {
        // 具体实现细节
    }
```

第三节　多序列操作

3.3.1　序列加法与乘法

通过重载加法（operator+）和乘法（operator*）操作符，支持两个序列的逐元素加法和乘法操作。操作后的结果序列大小为两个序列时间范围的并集。**支持实时输入并计算加法或乘法：**

```
    DS operator+(const DS& other) const {
    // 实现加法操作
    }

    DS operator*(const DS& other) const {
        // 实现乘法操作
    }
```

3.3.2　卷积操作

实现线性卷积和圆周卷积。线性卷积将两个序列按给定方式进行加权和，圆周卷积则考虑周期性边界条件。**支持实时输入并计算卷积：**

```
    DS convolution(DS &A, DS &B, char direction = 'n') {
    // 实现线性卷积
    }

    DS convolution_cir(DS &A, DS &B, int k, char direction = 'n
        ') {
        // 实现圆周卷积
    }
```

3.3.3 相似性比对与滑动窗口

相似性比对可以基于**归一化或滑动窗口**来计算两个序列之间的相似度：

```
double similarity_nomalize(DS &A, DS &B, int l) {
// 实现归一化相似性计算
}

double similarity_window(DS &A, DS &B, int l, int k) {
    // 实现滑动窗口相似性计算
}
```

第四章　实验结果

在本实验中，所有操作均按要求实现，并且对于无限长序列输入，系统能够即时处理每一个新的输入并进行相应的操作。以下是部分操作的输出示例：

1. 信号序列定义、读取写入任意位置、无限长序列的输入

```
// 测试信号序列定义、读取写入任意位置、无限长序列的输入
std::vector<double> va = {1,2,3,0,5,6};
DS a(-3, 2, va); //定义a[-3:2]=[1,2,3,0,5,6]
std::cout << a << std::endl;
std::cout<<"写入a[-3] = 6, a[-1] = -2, a[2] = 3后"<<std::
    endl;
a[-3] = 6, a[-1] = -2, a[2] = 3;
std::cout << a << std::endl;
std::cout<<"开始无限长序列的输入："<<std::endl;
a.read_infinite(std::cin);//无限长序列的输入
std::cout<<"更新后的序列为："<<std::endl;
std::cout << a << std::endl;
```

```
→lab   & 'c:\Users\acDsp\.vscode\extensions\ms-v
Microsoft-MIEngine-In-cauiddko.dmu' '--stdout=Mic
-pid=Microsoft-MIEngine-Pid-qkra45wo.5vp' '--dbgE
a[-3:2] = [1, 2, 3, 0, 5, 6]
写入a[-3] = 6, a[-1] = -2, a[2] = 3后
a[-3:2] = [6, 2, -2, 0, 5, 3]
开始无限长序列的输入:
选择读入的方向 (l: 向左读入, r: 向右读入): r
请输入向右扩展的序列数据 (Ctrl+D结束): 1 2 3
^D
更新后的序列为:
a[-3:5] = [6, 2, -2, 0, 5, 3, 1, 2, 3]
```

图 4.1 程序运行截图 1

2. 单序列基本操作

所有操作均支持实时输入。

```
// 测试单序列基本操作
std::vector<double> va = {1,2,3,0,5,6};
DS a(-3, 2, va); //定义a[-3:2]=[1,2,3,0,5,6]
std::cout << a << std::endl;
// a）前、后补零操作
a.pad_zero(2,1);//左侧补2个0，右侧补1个0
std::cout<<"左侧补2个0，右侧补1个0后的序列为："<<std::endl;
std::cout<<a<<std::endl;
// b）序列延迟、提前操作
a.delay(3);//延迟3
std::cout<<"延迟3个时间单位后的序列为："<<std::endl;
std::cout<<a<<std::endl;
a.advance(2);//提前2
std::cout<<"提前2个时间单位后的序列为："<<std::endl;
std::cout<<a<<std::endl;
// c）序列反转操作
a.reverse();
std::cout<<"反转后的序列为："<<std::endl;
std::cout<<a<<std::endl;
// d）序列拉伸、压缩操作（上采样、下采样)
a.stretch(3,'r');//因子=3，向右实时输入并拉伸
a.compress(3,'l');//因子=3，向左实时输入并压缩
// e）序列差分、累加操作
a.difference('r');//向右实时输入并差分
a.accumulate('r');//向右实时输入并累加
```

```
➜lab   & 'c:\Users\acDsp\.vscode\extensions\ms-vscode.cpptools-1.23.2-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '
nt0' '--stdout=Microsoft-MIEngine-Out-zkffelnc.jkn' '--stderr=Microsoft-MIEngine-Error-tps10j54.fcd' '--pid=Microsoft-MIEngine
in\gdb.exe' '--interpreter=mi'
a[-3:2] = [1, 2, 3, 0, 5, 6]
左侧补2个0，右侧补1个0后的序列为：
a[-5:3] = [0, 0, 1, 2, 3, 0, 5, 6, 0]
延迟3个时间单位后的序列为：
a[-2:6] = [0, 0, 1, 2, 3, 0, 5, 6, 0]
提前2个时间单位后的序列为：
a[-4:4] = [0, 0, 1, 2, 3, 0, 5, 6, 0]
反转后的序列为：
a[-4:4] = [0, 6, 5, 0, 3, 2, 1, 0, 0]
当前序列: a[-4:4] = [0, 6, 5, 0, 3, 2, 1, 0, 0]
以3为因子拉伸后的序列a[-12:12] = [0, 0, 0, 6, 0, 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0]
请输入向右拉伸的序列数据 (Ctrl+D结束): 1 2 ^D
以3为因子拉伸后的序列a[-12:15] = [0, 0, 0, 6, 0, 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1]
以3为因子拉伸后的序列a[-12:18] = [0, 0, 0, 6, 0, 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2]
当前序列: a[-12:18] = [0, 0, 0, 6, 0, 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2]
以3为因子压缩后的序列: a[-4:6] = [0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
请输入向左压缩的序列数据 (Ctrl+D结束): 1 2 3 4 5 6 ^D
以3为因子压缩后的序列: a[-4:6] = [0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
以3为因子压缩后的序列: a[-4:6] = [0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
以3为因子压缩后的序列: a[-5:6] = [3, 0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
以3为因子压缩后的序列: a[-5:6] = [3, 0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
以3为因子压缩后的序列: a[-5:6] = [3, 0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
以3为因子压缩后的序列: a[-6:6] = [6, 3, 0, 6, 5, 0, 3, 2, 1, 0, 0, 1, 2]
差分后的序列a[-5:6] = [0, -1, -5, 3, -1, -1, -1, 0, 1, 1]
请输入向右差分的序列数据 (Ctrl+D结束): 1 2 3 ^D
差分后的序列: a[-5:7] = [0, -1, -5, 3, -1, -1, -1, 0, 1, 1, 0]
差分后的序列: a[-5:8] = [0, -1, -5, 3, -1, -1, -1, 0, 1, 1, 0, 1]
差分后的序列: a[-5:9] = [0, -1, -5, 3, -1, -1, -1, 0, 1, 1, 0, 1, 1]
累加后的序列: a[-5:9] = [0, -1, -6, -3, -4, -5, -6, -6, -5, -4]
请输入向右累加的序列数据 (Ctrl+D结束): 1 2 3 ^D
累加后的序列: a[-5:10] = [0, -1, -6, -3, -4, -5, -6, -6, -5, -4, -3]
累加后的序列: a[-5:11] = [0, -1, -6, -3, -4, -5, -6, -6, -5, -4, -3, -1]
累加后的序列: a[-5:12] = [0, -1, -6, -3, -4, -5, -6, -6, -5, -4, -3, -1, 2]
```

图 4.2 程序运行截图 2

3. 多序列操作

所有操作均支持实时输入。为了节省篇幅，仅展示圆周卷积和滑动窗相似性比对的实时输入。

```cpp
//测试多序列操作
std::vector<double> va = {1,2,3,0,5,6};
std::vector<double> vb = {4,3,2,1,5,7};
DS a(0, 5, va); //定义a[0:5]=[1,2,3,0,5,6]
DS b(0, 5, vb); //定义b[0:5]=[4,3,2,1,5,7]
std::cout << "序列a为" <<std::endl;
std::cout << a << std::endl;
std::cout << "序列b为" <<std::endl;
std::cout << b << std::endl;
//加法操作
DS c = a + b;
std::cout << "序列a与序列b的和为" <<std::endl;
std::cout << c << std::endl;
//乘法操作
c = a * b;
std::cout << "序列a与序列b的积为" <<std::endl;
```

```
17      std::cout << c << std::endl;
18      //线性卷积操作
19      c = convolution(a,b);
20      std::cout << "序列a与序列b的线性卷积为" <<std::endl;
21      std::cout << c << std::endl;
22      //8点圆周卷积，向右实时读入
23      c = convolution_cir(a,b,8,'r');
24      std::cout << "序列a为" <<std::endl;
25      std::cout << a << std::endl;
26      std::cout << "序列b为" <<std::endl;
27      std::cout << b << std::endl;
28      //滑动窗相似性比对，l=0,滑动窗长度k=3
29      double ret = similarity_window(a,b,0,3);
30      //归一化相似性比对，l=0
31      ret = similarity_nomalize(a,b,0);
```

```
序列a为
a[0:5] = [1, 2, 3, 0, 5, 6]
序列b为
a[0:5] = [4, 3, 2, 1, 5, 7]
序列a与序列b的和为
a[0:5] = [5, 5, 5, 1, 10, 13]
序列a与序列b的积为
a[0:5] = [4, 6, 6, 0, 25, 42]
序列a与序列b的线性卷积为
a[0:10] = [4, 11, 20, 14, 33, 59, 57, 38, 31, 65, 42]
8-圆周卷积后序列为: a[0:7] = [35, 76, 62, 14, 33, 59, 57, 38]
请输入序列 A 的新数据: 1
请输入序列 B 的新数据: 2
8-圆周卷积后序列为: a[0:7] = [37, 77, 67, 21, 35, 59, 61, 41]
请输入序列 A 的新数据: 3
请输入序列 B 的新数据: 4
8-圆周卷积后序列为: a[0:7] = [46, 83, 70, 36, 56, 65, 73, 53]
请输入序列 A 的新数据: 5
请输入序列 B 的新数据: 6
8-圆周卷积后序列为: a[0:7] = [46, 83, 70, 36, 56, 65, 73, 53]
请输入序列 A 的新数据: 1
请输入序列 B 的新数据: 1
8-圆周卷积后序列为: a[0:7] = [46, 83, 70, 36, 56, 65, 73, 53]
序列a为
a[0:5] = [1, 2, 3, 0, 5, 6]
序列b为
a[0:5] = [4, 3, 2, 1, 5, 7]
l=0,滑动窗相似度为67
请输入序列 A 的新数据: 1
请输入序列 B 的新数据: 2
l=0,滑动窗相似度为69
请输入序列 A 的新数据: 3
请输入序列 B 的新数据: 4
l=0,滑动窗相似度为56
请输入序列 A 的新数据: 4
请输入序列 B 的新数据: 5
l=0,滑动窗相似度为34
请输入序列 A 的新数据: 0
请输入序列 B 的新数据: 1
l=0,滑动窗相似度为32
请输入序列 A 的新数据: 9
请输入序列 B 的新数据: 1
l=0,滑动窗相似度为29
归一化后序列a为
a[0:5] = [-0.353919, -0.160872, 0.0321745, -0.546966, 0.418268, 0.611315]
归一化后序列b为
a[0:5] = [0.0690066, -0.138013, -0.345033, -0.552052, 0.276026, 0.690066]
l=0,归一化相似度为0.825933
→lab
```

图 4.3 程序运行截图 3

4. 滑动窗相似性比对的效率分析和优化

如果每次滑动窗向右移动一个单位都重新计算 *a* 和 *b* 相似性 *ret*，那么时间

复杂度为滑动窗的长度 $O(k)$。

优化算法：

记录 $t-1$ 时刻的相似性 ret'，那么 t 时刻的相似性可以由以下公式算出：

$$ret = ret' - a_{t-k} * b_{t-k} + a_t * b_t$$

时间复杂度为 $O(1)$。

第五章　实验结论

通过本次实验，我掌握了数字信号处理中的基本操作及其在编程环境中的实现方式，理解了如何处理信号序列并进行相关操作。实验还深入探讨了多序列操作、卷积以及相似性比对等高级操作，并对滑动窗口的效率进行了初步分析和优化。