Университет ИТМО

Факультет программной инженерии и компьютерной техники

Кафедра вычислительной техники

Курсовая работа № 1 по дисциплине ”Методы цифровой обработки сигналов”

Вариант 12

Выполнил:

Чебыкин И. Б.

Группа: P3401

Проверяющий: Тропченко А. А.

Санкт-Петербург, 2017

Задание

Имеется специализированный процессор для линейной фильтрации сигналов. На вход ему поступают дискретные сигналы в формате целых чисел разрядности n_1. При фильтрации применяются весовые коэффициенты, имеющие разрядность n_2. Все промежуточные и конечные результаты имеют разрядность n_3.

Требуется:

1. Написать программу на любом языке программирования, которая реализует алгоритм линейной фильтрации, используя внутри числа с плавающей точкой;
2. Написать программу, которая реализует тот же алгоритм, но использует целые числа разрядностей n_i; Определить зависимости:
    1. Среднеквадратической погрешности от длины обрабатываемого вектора данных и/или длины ядра преобразования;
    2. Точности от способа формирования малоразрядного результата: с отсечением младших разрядов, с отсечением и увеличением младшего разряда на единицу, с округлением;
    3. Среднеквадратического отклонения от длины обрабатываемого вектора для всех трёх способов;
    4. Среднеквадратического отклонения от длины обрабатываемого вектора при округлении результата с изменением:
    - n_1 на 2 и 4 бита;
    - n_2 на 2 и 4 бита.

Вариант

Вычисление апериодической свёртки при $M = 5, N = [10; 40], n_1 = 8, n_2 = 4, n_3 = 12$.

Математические соотношения

Свёртка — алгоритм, при котором на вход поступают две последовательности — $a(n)$, $b(m)$, где $n \in [0; N), m \in [0; M)$ — и на выходе имеется новая последовательность, представляющая собой произведение элементов a и b такое, что элементы a берутся по возрастающему индексу, а b — по убывающему.

Апериодическая свёртка — частный случай свёртки, при котором входные последовательности не зацикливаются, а дополняются нулями.

Определим $A(n), n \in \mathbb{Z}$ как функцию, которая на $n \in [0; N)$ имеет те же значения, что и $a(n)$, а на остальной области определения — 0. Аналогично определим $B(m)$ как расширение $b(m)$.

Тогда элемент выходной последовательности высчитывается следующим образом:

$$c(k) = \sum_{i=0}^{k} A(i) \cdot B(k-i), k \in [0; N + M - 2]$$

Блок-схема

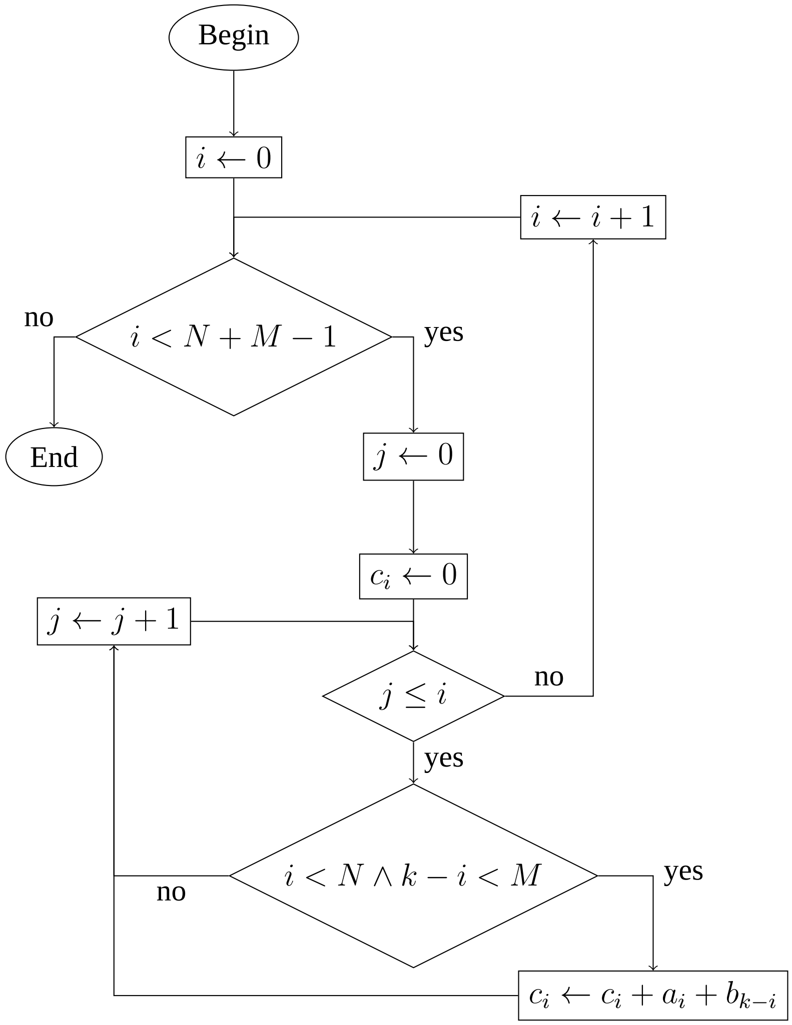

Рис. 1: Блок-схема алгоритма

Выполнение

Числа с плавающей точкой

```
use std::iter::FromIterator;

pub fn fold_f32<B: FromIterator<f32>> (a: &Vec<f32>, b: &Vec<f32>) -> B {
    (0..a.len() + b.len() - 1).map(
        |k| (0 .. k + 1).map(
            |i| *a.get(i).unwrap_or(&0.0) * *b.get(k - i).unwrap_or(&0.0))
        .sum()).collect()
}

#[cfg(test)]
mod tests {
    #[test]
    fn test_fold_32() {
        let a = vec![2.0, 1.0, 3.0, -1.0];
```

```
        let b = vec![-1.0, 1.0, 2.0];
        let c = vec![-2.0, 1.0, 2.0, 6.0, 5.0, -2.0];
        let d : Vec<f32> = super::fold_f32(&a, &b);
        assert_eq!(c, d);
    }
}
```

Числа с ограниченной разрядностью

```
use std::iter::FromIterator;

fn get_sign(a: i32) -> i32 {
    if a > 0 {
        1
    } else if a < 0 {
        -1
    } else {
        0
    }
}

#[derive(Copy, Clone)]
pub enum RoundType {
    Cut,
    Inc,
    Round,
}

fn round(a: &Vec<i32>, size: usize, rtype: RoundType) -> Vec<i32> {
    let s = if a.iter().any(|&x| x < 0) { size - 1 } else { size };

    if let Some(Some(n)) = a.iter().map(
        |&x| (0..31).rev().find(|i| x.abs() & (1 << i) != 0))
        .filter(|x| x.is_some()).max() {
        if n >= s {
            a.iter().map(|&x| {
                let t = x.abs();
                let mo = n + 1 - s;
                get_sign(x) * {
                    match rtype {
                        RoundType::Cut =>
                            t & (!(1i32 << s) << mo),
                        RoundType::Inc =>
                            (1 + (t >> mo)) << mo,
                        RoundType::Round =>
                            if t & (1 << (n - s)) != 0 {
                                (1 + (t >> mo)) << mo
                            } else {
                                t & (!(1i32 << s) << mo)
                            },
                    }
                }
            }).collect()
        } else {
            a.clone()
        }
    } else {
        a.clone()
    }
}

pub fn fold<C: FromIterator<i32>> (a: &Vec<i32>, asize: usize,
                                   b: &Vec<i32>, bsize: usize,
                                   csize: usize, rtype: RoundType) -> C {
    let a2 = round(a, asize, rtype);
    let b2 = round(b, bsize, rtype);
    (0..a.len() + b.len() - 1).map(
```

```
        |k| (0..k + 1).map(
            |i| round(&vec![
                      *a2.get(i).unwrap_or(&0) * *b2.get(k - i).unwrap_or(&0)
            ], csize, rtype)[0])
        .fold(0, |a, c| round(&vec![a + c], csize, rtype)[0])).collect()
}

#[cfg(test)]
mod tests {
    #[test]
    fn test_fold() {
        let a = vec![2, 1, 3, -1];
        let b = vec![-1, 1, 2];
        let c = vec![-2, 1, 2, 6, 5, -2];
        let d : Vec<i32> = super::fold(&a, 8, &b, 8, 8, super::RoundType::Cut);
        assert_eq!(c, d);
    }

    #[test]
    fn test_round() {
        use super::round;
        use super::RoundType;

        assert_eq!(round(&vec![14], 8, RoundType::Cut),   vec![14]);
        assert_eq!(round(&vec![14], 2, RoundType::Cut),   vec![12]);
        assert_eq!(round(&vec![14], 1, RoundType::Cut),   vec![8 ]);
        assert_eq!(round(&vec![14], 8, RoundType::Inc),   vec![14]);
        assert_eq!(round(&vec![14], 2, RoundType::Inc),   vec![16]);
        assert_eq!(round(&vec![14], 1, RoundType::Inc),   vec![16]);
        assert_eq!(round(&vec![14], 8, RoundType::Round), vec![14]);
        assert_eq!(round(&vec![14], 2, RoundType::Round), vec![16]);
        assert_eq!(round(&vec![14], 1, RoundType::Round), vec![16]);
        assert_eq!(round(&vec![31], 8, RoundType::Round), vec![31]);
        assert_eq!(round(&vec![32], 1, RoundType::Round), vec![32]);

        assert_eq!(round(&vec![-1, -1, 4, -1, -1], 2, RoundType::Round),
            vec![0, 0, 4, 0, 0]);
    }
}
```

Графики зависимостей

Для заданного ядра оказалось, что при любой длине вектора заданной точности оказывается достаточно. Действительно, положительная и отрицательная компоненты суммы имеют максимальную разрядность $8 + 3 = 11$ битов. Соответственно, сумма компонент не может превышать 11 битов ни в нижнюю, ни в верхнюю сторону, и один бит отводится на знак.

Погрешность наблюдается лишь при ограничении разрядности входных данных (максимальное СКО — 2) и при ограничении разрядности ядра до двух битов, что приводит к преобразованию ядра в [0, 0, 4, 0, 0] и, соответственно, к существенным отклонениям (СКО порядка сотен).

С другой стороны, для ядра [8, 8, 8, 8, 8] картина совсем иная. Действительно, перемножая числа с разрядностями 8 и 4 бита, имеем число разрядностью 12 битов. Складывая пять таких чисел, получаем число разрядностью в 16 битов.

Для этого ядра также обнаружено, что округление и обрезание нижних битов не приводят к потере точности, в отличие от обрезания и увеличения верхних битов на единицу.

Также по графикам можно установить, что имеется зависимость не от размера входного вектора, но от отношения длины строба ко всей длине вектора. Это отношение в рамках опыта менялось периодически.

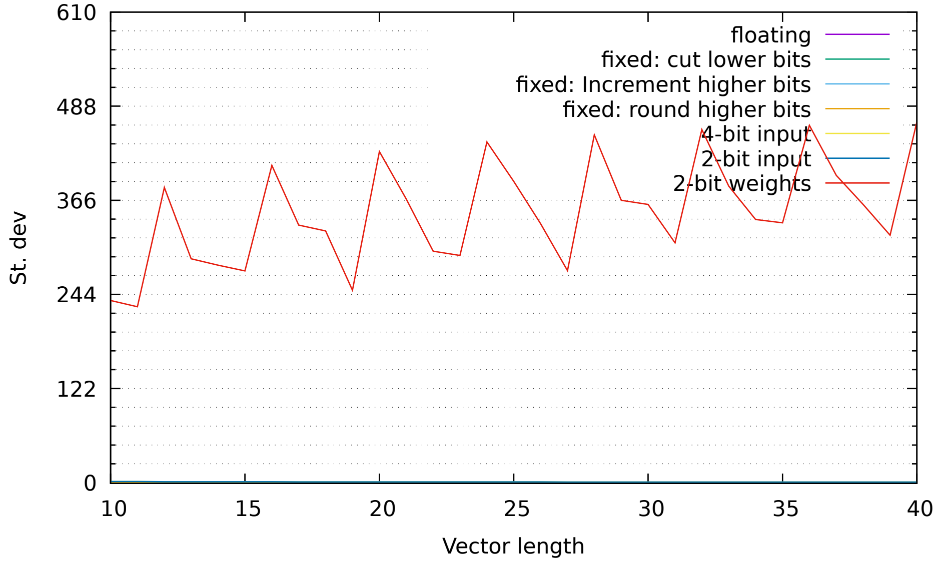

Рис. 2: Зависимость СКО от длины входного вектора для разных видов фильтрации с ядром [-1, -1, 4, -1, -1]

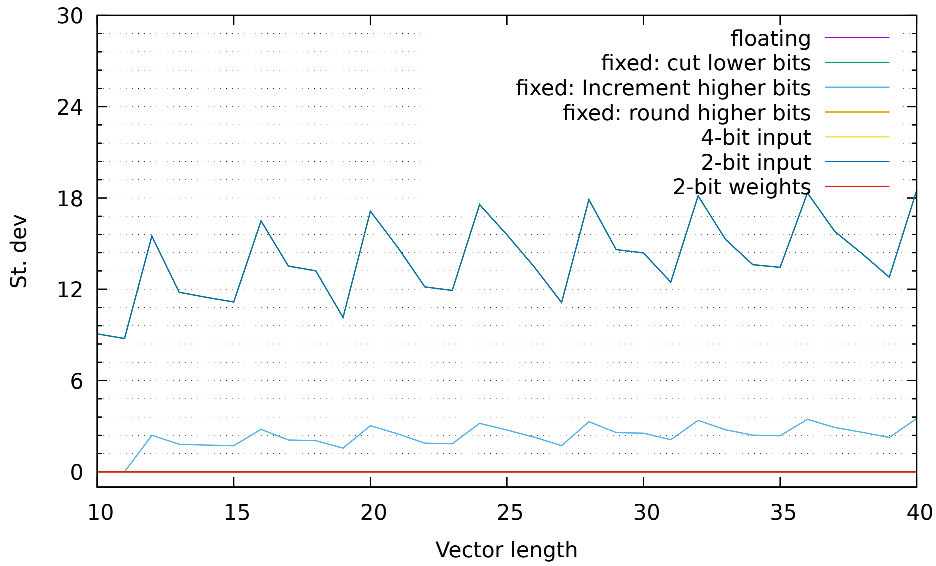

Рис. 3: Зависимость СКО от длины входного вектора для разных видов фильтрации с ядром [8, 8, 8, 8, 8]

Вывод

В результате выполнения работы было выяснено, что длина входного вектора не оказывает существенного влияния на погрешность апериодической свёртки, ограниченной по разрядности. С другой стороны, разрядность ядра крайне важна.