МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №1

по дисциплине: Вычислительная математика
тема: «Погрешности. Приближенные вычисления. Вычислительная устойчивость.»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
пр. Четвертухин Виктор Романович

Белгород 2024 г.

Лабораторная работа №1

«Погрешности. Приближенные вычисления. Вычислительная устойчивость.

Вариант 10

Цель работы: Изучить особенности организации вычислительных процессов, связанные с погрешностями, приближенным характером вычислений на компьютерах современного типа, вычислительной устойчивостью.

1. Запустить и проинтерпретировать результаты работы разных вычислительных схем для простого арифметического выражения на языке Rust.

```rust
fn main() {
    let num1: f32 = 0.23456789;
    let num2: f32 = 1.5678e+20;
    let num3: f32 = 1.2345e+10;
    let result1 = (num1 * num2) / num3;
    let result2 = (num1 / num3) * num2;
    let result3: f64 = num1 as f64 * num2 as f64 / num3 as f64;
    println!("({} * {}) / {} = {}", num1, num2, num3, result1);
    println!("({} / {}) * {} = {}", num1, num3, num2, result2);
    println!(" {} * {} / {} = {}", num1, num2, num3, result3);
}
```

Результаты работы:

```
(0.2345679 * 156780000000000000000) / 12345000000 = 2978983700
(0.2345679 / 12345000000) * 156780000000000000000 = 2978984000
 0.2345679 * 156780000000000000000 / 12345000000 = 2978983717.267449
```

2. Запустить и проинтерпретировать результаты работы разных вычислительных схем для интерационного и неитерационного вычисления на языке Rust.

```rust
// демонстрация накопления погрешности для итерационного процесса
// версия для одинарной точности
fn iter(numbers: &[f32], iterations: i32) {
    println!("Итерационный метод: ");
    for &number in numbers {
        let mut result = number;
        for _ in 0..iterations {
            result = result.sqrt(); // послед. извлечение квадратного корня
        }
        for _ in 0..iterations {
            result = result * result; // послед. возведение числа в квадрат
        }
        let error = (number - result).abs();
        println!(
            "Исх-е значение: {:e}, результат: {:e}, абс-ая погрешность:
```

```
    {:e}, отн-ая погрешность: {:e} (%)",
            number,
            result,
            error,
            error * 100. / number
        );
    }
}

// замена итерации функцией
// версия для одинарной точности с powf
fn non_iter(numbers: &[f32], iterations: i32) {
    println!("Безытерационный метод: ");
    for &number in numbers {
        // извлекаем корень
        let intermediate = number.powf(1.0f32 / (1 << iterations) as f32);
        // восстанавливаем значение
        let result = intermediate.powf((1 << iterations) as f32);
        let error = (number - result).abs();
        println!(
            "Исх-е значение: {:e}, результат: {:e}, абс-ая погрешность:
    {:e}, отн-ая погрешность: {:e} (%)",
            number,
            result,
            error,
            error * 100. / number
        );
    }
}

fn main() {
    let numbers = [
        1.0f32,
        20.,
        300.,
        4000.,
        5e6,
        f32::MIN_POSITIVE,
        f32::MAX * 0.99,
    ]; // вектор с числами одинарной точности

    let iterations = 10; // число итераций

    iter(&numbers, iterations);
    non_iter(&numbers, iterations)
}
```

Результаты работы:

```
Итерационный метод:
Исх-е значение: 1e0, результат: 1e0, абс-ая погрешность:
    0e0, отн-ая погрешность: 0e0 (%)
Исх-е значение: 2e1, результат: 2.000009e1, абс-ая погрешность:
```

```
    8.9645386e-5, отн-ая погрешность: 4.4822693e-4 (%)
Исх-е значение: 3e2, результат: 3.0001422e2, абс-ая погрешность:
    1.4221191e-2, отн-ая погрешность: 4.740397e-3 (%)
Исх-е значение: 4e3, результат: 4.0001064e3, абс-ая погрешность:
    1.0644531e-1, отн-ая погрешность: 2.6611327e-3 (%)
Исх-е значение: 5e6, результат: 4.9994865e6, абс-ая погрешность:
    5.135e2, отн-ая погрешность: 1.027e-2 (%)
Исх-е значение: 1.1754944e-38, результат: 1.17548e-38, абс-ая погрешность:
    1.43e-43, отн-ая погрешность: 1.2159348e-3 (%)
Исх-е значение: 3.3687953e38, результат: 3.3686973e38, абс-ая погрешность:
    9.796404e33, отн-ая погрешность: 2.9079844e-3 (%)
Безытерационный метод:
Исх-е значение: 1e0, результат: 1e0, абс-ая погрешность:
    0e0, отн-ая погрешность: 0e0 (%)
Исх-е значение: 2e1, результат: 2.0000069e1, абс-ая погрешность:
    6.866455e-5, отн-ая погрешность: 3.4332275e-4 (%)
Исх-е значение: 3e2, результат: 3.0000873e2, абс-ая погрешность:
    8.728027e-3, отн-ая погрешность: 2.9093425e-3 (%)
Исх-е значение: 4e3, результат: 4.0001143e3, абс-ая погрешность:
    1.1425781e-1, отн-ая погрешность: 2.8564453e-3 (%)
Исх-е значение: 5e6, результат: 5.000186e6, абс-ая погрешность:
    1.86e2, отн-ая погрешность: 3.72e-3 (%)
Исх-е значение: 1.1754944e-38, результат: 1.175497e-38, абс-ая погрешность:
    2.7e-44, отн-ая погрешность: 2.2649765e-4 (%)
Исх-е значение: 3.3687953e38, результат: 3.3687553e38, абс-ая погрешность:
    3.9956347e33, отн-ая погрешность: 1.1860722e-3 (%)
```

3. С помощью программы на языке Rust вывести на экран двоичное представление машинных чисел одинарной точности стандарта IEEE 754 для записи: числа π, бесконечности, нечисла (NaN), наименьшего положительного числа, наибольшего положительного числа, наименьшего отрицательного числа. Сформулировать обоснование полученных результатов в пунктах 1 и 2, опираясь на двоичное представление машинных чисел.

```rust
fn main() {
    let pi = std::f32::consts::PI;
    let infinity = std::f32::INFINITY;
    let nan = std::f32::NAN;
    let smallest_positive = std::f32::MIN_POSITIVE;
    let largest_positive = std::f32::MAX;
    let smallest_negative = -std::f32::MIN_POSITIVE;
    println!("π: {}", float_to_binary_string(pi));
    println!("Infinity: {}", float_to_binary_string(infinity));
    println!("NaN: {}", float_to_binary_string(nan));
    println!(
        "Smallest Positive Number: {}",
        float_to_binary_string(smallest_positive)
    );
    println!(
        "Largest Positive Number: {}",
```

```
        float_to_binary_string(largest_positive)
    );
    println!(
        "Smallest Negative Number: {}",
        float_to_binary_string(smallest_negative)
    );
}
fn float_to_binary_string(num: f32) -> String {
    let bits = num.to_bits();
    format!("{:032b}", bits)
}
```

Результаты работы:

```
π: 01000000010010010000111111011011
Infinity: 01111111100000000000000000000000
NaN: 01111111110000000000000000000000
Smallest Positive Number: 00000000100000000000000000000000
Largest Positive Number: 01111111011111111111111111111111
Smallest Negative Number: 10000000100000000000000000000000
```

Первое задание:
Точность вычислений зависит от порядка операций. Мантисса числа ограничена, и из-за этого последние цифры вещественных чисел могут округляться некорректно. В первом и втором примере мы получили различные варианты ответов в зависимости от порядка выполнения операций. На порядок и размер чисел стоит обращать внимание, когда порядок может превысить размеры порядка в памяти, тогда мы получим бесконечность или минус бесконечность. В этом случае первый вариант намного логичней, так как порядок числа после умножения числа с отрицательной экспонентой и положительной не будет давать экспоненты больше, чем может содержать память. В третьем примере вещественные числа, хранящиеся в 32-битной схеме расширяются до 64-битной, что увеличивает размер мантиссы и, следовательно, точность вычислений.
Второе задание:
Итерационный метод в целом имеет большую относительную погрешность из-за постепенного накопления ошибок в младших битах при выполнении множественных действий, безытерационный метод лишён таких недостатков, и выполняет действия над числами намного реже, что сокращает кол-во ошибок при сокращении.

4. Подобрать такие входные данные, что в первом случае схема демонстрировала бы заметную потерю в точности, а вторая на тех же входных данных — улучшала бы результат.

```
pub fn make_computation(principal: f32, rate: f32, periods: f32) -> f32 {
    principal * (1.0 + rate).powf(periods)
}
```

```rust
pub fn make_improved_computation(principal: f32, rate: f32, periods: f32) -> f32 {
    principal * (periods * (1.0 + rate).ln()).exp()
}
```

```rust
extern crate algr;

use algr::lab1::task4::{make_computation, make_improved_computation};

fn main() {
    let pri = 1.0f32;
    let rat = -0.9999f32;
    let per = 4.0f32;

    println!("{:.32}", make_computation(pri, rat, per));
    println!("{:.32}", make_improved_computation(pri, rat, per));
    println!("{:.32}", make_computation(pri, rat, per) - make_improved_computation(pri, rat, per));
}
```

Результаты работы:

```
0.00000000000000010006639451092431
0.00000000000000010006634818881001
0.00000000000000000000004632211430
```

Вывод: в ходе лабораторной работы изучили особенности организации вычислительных процессов, связанные с погрешностями, приближенным характером вычислений на компьютерах современного типа, вычислительной устойчивостью. Предложенный улучшенный метод вычисления выражения в большинстве случаев оказывается менее точным, чем его обычная версия, так как содержит множество дополнительных действий, что увеличивает количество ошибок при сокращении.