Floating Point Challenges and Overcoming them in Rust Lang

Understanding and Mitigating Inaccuracies in Real-World Calculations

About me

LinkedIn: prabhat25
Github: artech-git

Hobbies and Interest

- Love Linux 🐧 🐧
- Systems programming
- Music / DJ 🎹
- Cycling 🚴

Experience

- Multiple programming jobs during college
- Contributed to different open source project's in rust
- Interned as Rust developer for AfterShoot, one of the most thriving startup's in domain of photography

Content

Introduction

Importance of floating point calculations

The Patriot Missile Failure, which occurred during the Gulf War in 1991, was attributed to various factors, including a floating-point-related calculation error.

Time Accumulation Error: The Patriot missile system tracked incoming Scud missiles using time measurements. A floating-point representation was used to represent time in tenths of a second. However, this representation led to a gradual accumulation of rounding errors over time due to limited precision.

Rounding Error Propagation: The Patriot's tracking radar system relied on a time calculation to predict the position of the incoming Scud missile. The rounding errors in this time calculation were used to determine the missile's position, and these errors propagated over time, causing the predicted position to deviate from the actual position.

Introduction

Preliminaries before proceeding

- **Real-World Representation:** Floating-point numbers provide a way to approximate and represent real numbers in computers.
- **Scientific Notation:** They use scientific notation, expressing a number as a sign, significand (mantissa), and exponent.
- **Precision and Range:** Floating-point offers a trade-off between precision and range, suitable for a wide spectrum of numerical values.
- **Limited Precision:** Due to finite bits, they exhibit rounding errors and minor inaccuracies in calculations.
- **Versatile Usage:** Vital in scientific, engineering, and financial computations, enabling accurate modeling of real-world phenomena.
- **Mitigation Strategies:** Developers apply strategies to manage rounding errors and ensure reliable results.

Introduction

The IEEE 754 standard is a widely used specification for representing and performing arithmetic operations on floating-point numbers in computers. It defines how real numbers, including both integers and fractions, can be represented using a finite number of bits in binary format. The standard was first published by the Institute of Electrical and Electronics Engineers (IEEE) in 1985 and has since been revised and updated.

The IEEE 754 standard defines several formats for representing floating-point numbers, including single precision (32-bit), double precision (64-bit), and extended precision (80-bit or higher).

Precision

Single Precision (32-bit): In the IEEE 754 single precision format, a floating-point number is represented using 32 bits, which are divided into three components:

Sign bit (1 bit): Represents the sign of the number. 0 for positive, 1 for negative.
Exponent bits (8 bits): Represent the exponent of the number in a biased form. This allows the representation of both very small and very large numbers.
Significand bits (23 bits): Also known as the fraction or mantissa, these bits represent the fractional part of the number.

The value of a single precision floating-point number is calculated as follows:

(-1)^S * 2^(E - Bias) * 1.F

Where:

- **S is the sign bit.**
- **E is the exponent value decoded from the exponent bits.**
- **Bias is a constant (127 for single precision) used to encode the exponent in a biased form.**
- **F is the fractional value represented by the significand bits as a binary fraction.**

Precision

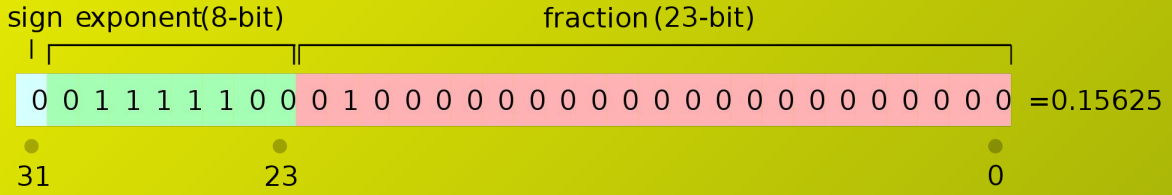
Double Precision (64-bit): In the IEEE 754 double precision format, a floating-point number is represented using 64 bits, with the following components:

Sign bit (1 bit): As in single precision.
Exponent bits (11 bits): Represent the exponent of the number with a larger range.
Significand bits (52 bits): Represent the fractional part.

The value of a double precision floating-point number is calculated in a similar way to single precision, but with a larger exponent range and more significand bits.

How Floating Point numerical works

Single Precision (32 bit)



$$2.345 = \underbrace{12345}_{\text{significand}} \times \underbrace{10^{\overbrace{-3}^{\text{exponent}}}}_{\text{base}}$$

1. Introduction

Limited Precision and Rounding Errors

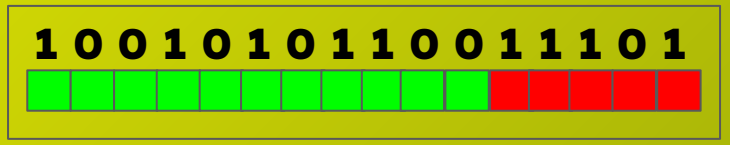
- Floating-point numbers, a cornerstone of scientific and engineering computations, are represented using a finite number of bits.
- Standardized by IEEE 754, floating-point representation includes both the significand and exponent to capture a wide range of values.
- However, due to the finite number of bits available for representation, real numbers often cannot be represented exactly.
- This inherent limitation leads to rounding errors, where the closest representable value is used, resulting in slight inaccuracies.

Addressing Limited Precision

- Developers need to be aware of limited precision when designing algorithms and systems that involve floating-point computations.
- Employ techniques to mitigate rounding errors and manage precision loss.
- Numerical libraries, proper algorithm design, and careful consideration of tolerance thresholds are essential to achieve accurate results.

2. Limited Precision

- **Limited Precision and Rounding Errors**
  - **Floating-point numbers have finite precision due to the fixed number of bits used for representation in the significand or mantissa .**
  - **Rounding errors occur when real numbers cannot be represented exactly.**



2. Limited Precision

Problem

```rust
fn main() {
    let a: f64 = 0.1;
    let b: f64 = 0.2;
    let sum: f64 = a + b;
    println!("Sum: {}", sum); // Output might not be exactly 0.3
}
```

2. Limited Precision

Solution

- **Use Fixed-Point Arithmetic:**
  - **Represent numbers as scaled integers to avoid precision issues.**
  - **Perform arithmetic operations on scaled integers to maintain precision.**
- **BigDecimal Library:**
  - **Utilize the `num-bigint` or `rust-dec` crate for arbitrary precision decimal arithmetic.**
- **Use F64/F32 Wisely:**
  - **Choose the appropriate floating-point type (f64 or f32) based on required precision.**
  - **Be aware of limitations and precision trade-offs for each type.**

3: Rounding Errors

- **Rounding Errors and Precision Loss occurs since not all real numbers can be exactly represented in a limited precision floating-point format.**

- **Rounding errors can accumulate as a result of approximations made during arithmetic operations.**

- **Errors can lead to small discrepancies between the expected mathematical result and the actual result obtained when using floating-point arithmetic.**

3: Rounding Errors

Problem

```rust
fn main() {
    let x: f64 = 1.0;
    let y: f64 = 1e-15;
    let sum: f64 = x + y;
    println!("Sum: {}", sum); // Output might not reflect y's contribution
    // Sum: 1.000000000000001
}
```

3: Rounding Errors

Solution

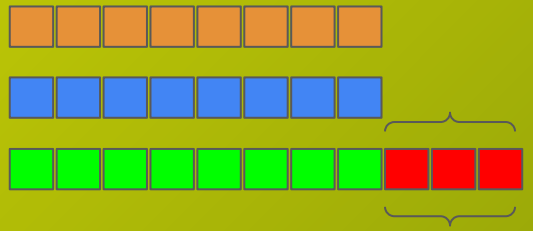
- **Use Decimal Types:**
  - Utilize libraries like `rust_decimal` to work with decimal-based arithmetic, which can minimize rounding errors.
- **Round Only When Necessary:**
  - Avoid unnecessary rounding during intermediate calculations.
  - Perform rounding only when presenting results to users or external systems.
- **Avoid Cumulative Rounding:**
  - Minimize the number of rounding operations in a sequence to prevent cumulative errors.
- **Avoid Divisions:**
  - Divisions can amplify rounding errors; try to use multiplication or other operations when possible to achieve the end result.
- **Interval Arithmetic:**
  - Use interval arithmetic libraries to represent ranges of possible values instead of relying solely on point estimates.

4: Loss of Significance

Loss of significance is a consequence of the limited precision of floating-point representations, where the available bits are allocated to the most significant digits

Loss of Significance in Subtraction

- **Subtraction of nearly equal numbers can lead to loss of significant digits.**
- **Catastrophic cancellation can occur.**



4: Loss of Significance

Problem

```rust
fn main() {
    let a: f64 = 1.000001;
    let b: f64 = 1.0000001;
    let difference: f64 = a - b;
    println!("Difference: {}", difference); // Loss of significant digits
    //Difference: 0.000000899999998593466
}
```

4: Loss of Significance

Solution

- **Reorder Operations:**
  - **Rearrange mathematical operations to minimize the loss of significant digits during calculations.**
- **Use Alternative Formulas:**
  - **Utilize algebraic manipulations to rewrite formulas and reduce subtractive cancellations.**

- **Use Taylor Series Expansion:**
  - **Approximate functions using Taylor series expansion to maintain precision in calculations.**
- **Arbitrary-Precision Arithmetic:**
  - **Use crates like `num-bigint` or `rust-gmp` for arbitrary-precision arithmetic when utmost precision is required.**

5: Associativity and Order of Operations

```rust
fn main() {
    let a: f64 = 1.0e20;
    let b: f64 = -1.0e20;
    let c: f64 = 1.0;

    let result1 = (a + b) + c;
    let result2 = a + (b + c);

    println!("Result 1: {}", result1);
    // Result 1: 1
    println!("Result 2: {}", result2);
    // Result 2: 0
}
```

Associativity and Order of Operations

- **Floating-point operations are not always associative.**
- **Changing order can lead to different results.**

5: Associativity and Order of Operations

- **Parentheses and Explicit Grouping:**
  - **Use parentheses to explicitly group operations and control the order of evaluation to ensure correct results.**
- **Reorder Operations:**
  - **Rearrange operations to minimize intermediate rounding and error accumulation. Consider commutative operations that are less sensitive to order.**
- **Use Fused Multiply-Add (FMA) Operations:**
  - **FMA operations can improve accuracy by performing multiplication and addition in a single step, reducing rounding errors.**

6: Comparisons and Equality Testing

Comparisons and Approximate Equality

- **Direct equality comparisons for floating-point numbers can be problematic.**
- **Use epsilon-based comparisons for approximate equality.**

```rust
const EPSILON: f64 = 1e-9;

fn approx_eq(a: f64, b: f64) -> bool {
    (a - b).abs() < EPSILON
}

fn main() {
    let a: f64 = 0.1 + 0.2;
    let b: f64 = 0.3;

    if approx_eq(a, b) {
        println!("Approximately equal");
    }
}
```

6: Comparisons and Equality Testing

- **Using Epsilon Comparison:**
  - **Instead of checking for exact equality (`a == b`), use an epsilon value to allow for a small difference between two numbers.**
  - **Example: `if (a - b).abs() < epsilon`**
- **Relative Error Comparison:**
  - **Compare the relative error (difference divided by one of the numbers) against a threshold.**
  - **Example: `if (a - b).abs() / a < threshold`**
- **ULP (Units in the Last Place) Comparison:**
  - **Compare the difference between two numbers in terms of their ULPs to account for precision.**
  - **Example: Use the `f64::ulps_eq` function from the `float_cmp` crate.**
- **Comparing with Specific Tolerance:**
  - **Define a tolerance value based on your application's requirements and compare within that range.**
  - **Example: `if (a - b).abs() < tolerance`**

7. NaN and Infinity Handling

NaN and Infinity Handling

- **Floating-point types include special values: NaN (Not-a-Number) and infinity.**
- **Arise from operations like division by zero or square root of negative number.**

```
fn main() {
    let a: f64 = 0.0;
    let b: f64 = 0.0;

    let result = a / b;

    if result.is_nan() {
        println!("Result is NaN");
    } else if result.is_infinite() {
        println!("Result is Infinite");
    }
}
```

8. Compiler Optimizations

```rust
#[inline(never)]
fn perform_calculation(a: f64, b: f64) -> f64 {
    a + b * 2.0
}

fn main() {
    let x: f64 = 0.1;
    let y: f64 = 0.2;

    let result = perform_calculation(x, y);

    println!("Result: {}", result);
    // Result: 0.5
}
```

Compiler Optimizations and Precision

- **Compilers optimize floating-point calculations for performance.**
- **May reorder or combine operations, affecting precision.**
- **Use stable compiler flags to control optimizations.**

8. Compiler Optimizations

Use compiler attributes:

Rust provides attributes that allow you to control the behavior of floating-point operations. One such attribute is `#[target_feature]`, which allows you to enable specific CPU features and optimizations, including some that might be related to "fast-math" behavior. For example:

```
#![feature(target_feature)]

#[target_feature(enable = "fast-math")]
fn main() {
    // Your code here
}
```

Use compiler flags:

You can use Cargo's feature flags to enable specific optimization options provided by the LLVM compiler backend. For example, you can set the RUSTFLAGS environment variable to include optimization flags:

```
RUSTFLAGS="-C target-cpu=native -C llvm-args=-enable-fast-math" cargo build
```

9: Accumulative Errors

Accumulative Errors in Iterations

- **Iterative algorithms can accumulate rounding errors.**
- **Strategies to minimize error accumulation:**
  - **Kahan summation.**
  - **Compensated summation.**

```rust
fn kahan_sum(arr: &[f64]) -> f64 {
    let mut sum: f64 = 0.0;
    let mut c: f64 = 0.0;

    for &x in arr {
        let y = x - c;
        let t = sum + y;
        c = (t - sum) - y;
        sum = t;
    }

    sum
}

fn main() {
    let data = [0.1, 0.2, 0.3, 0.4];
    let result = kahan_sum(&data);

    println!("Kahan Sum: {}", result);
    // Kahan Sum: 1
}
```

10. Strategies for Overcoming Limitations (Part 1)

Mitigation Strategies

- **Choose appropriate data types (f32, f64) for precision requirements.**
- **Leverage numerical computation libraries for enhanced accuracy.**
  a. **rug**
  b. **num-bigint-dig**
  c. **Inari**
  d. **FastFp**
  e. **float_cmp**
  f. **rust_decimal**
- **Avoid direct equality comparisons, use epsilon-based comparisons.**
- **Perform error propagation analysis to estimate potential inaccuracies.**

11: Strategies for Overcoming Limitations (Part 2)

More Mitigation Strategies

- **Consider using arbitrary precision libraries for critical computations.**
- **Minimize accumulated errors in iterative algorithms using techniques like Kahan summation.**
- **Utilize stable compiler flags to control floating-point behavior.**
- **Thoroughly test numerical code with diverse inputs to identify and address inaccuracies.**