Error control in scientific modelling (MATH 500, Herbst)

Sheet 5: Floating-point numbers (10 P)

To be handed in via moodle by 27.10.2023

Exercise 1 (2.5 P)

Rewrite the following expressions in order to avoid cancellation for the indicated relations of the arguments or between the arguments. Assume that the input arguments x, y and θ are known exactly. For each improved expression explain shortly (e.g. in 1 bullet point) why it is improved.

1. For $x \approx 0$: $\sqrt{x+1} - 1$
2. For $x \approx y$: $\sin(x) - \sin(y)$
3. For $x \approx y$: $x^2 - y^2$
4. For $x \approx 0$: $\frac{1-\cos(x)}{\sin(x)}$
5. For $a \approx b$ and $|\theta| \ll 1$: $c = \sqrt{a^2 + b^2 - 2ab\cos\theta}$

Hint: Find mathematically equivalent expressions in which cancellation is harmless.

Tip: You might try yourself or use a tool like Herbie (https://herbie.uwplse.org/demo/) to automatically search over expressions. In any case, include a short explanation additionally.

Solution:

1. The initial expression $\sqrt{x+1}-1$ for $x \approx 0$ can lead to significant cancellation errors due to the substraction of two nearly eaqual values. By rationalizing the expression we can get an improved version:

$$\frac{x}{\sqrt{x+1}+1}.$$

2. To avoid cancellation caused by direct subtraction of $\sin(x)$ and $\sin(y)$ where $x \approx y$ we can rewrite the initial expression by applying triginometrical formulas:

$$2\sin\left(\frac{x-y}{2}\right)\cos\left(\frac{x+y}{2}\right).$$

3. To avoid direct substraction we can rewite the original expression by factorizing the difference of the squares which will be numerically more stable in case $x \approx y$:

$$(x+y)(x-y).$$

4. To prevent cancellation error caused by subtraction $1-\cos(x)$ and division by $\sin(x)$ we can employ trigonometric double angle formulas:

$$\tan\left(\frac{x}{2}\right).$$

5. Since the square root of a non-negative value is always non-negative and $\cos^2\theta \approx 0$, the original expression can be rewritten, escaping the difference of squares of almost equal values, as follows:

$$c = |a - b \cdot \cos\theta|.$$

Exercise 2 (2.5 P)

Show that

$$0.1 = \sum_{i=1}^{\infty} 2^{-4i} + 2^{-4i-1}$$

and deduce that $x = 0.1$ has the base 2 representation $0.000\overline{1100}$ (i.e. the last 4 bits periodically repeated). Let $\hat{x} = fl(0.1)$ the IEEE single-precision (`Float32`) version. Show that $\frac{x-\hat{x}}{x} = -\frac{1}{4}u$ where the single-precision unit roundoff is $u = 2^{-24}$.

Solution:

Dealing with geometric series we know that:

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r} \qquad \text{where } |r| < 1.$$

In our case, we have:

$$\sum_{i=1}^{\infty} 2^{-4i} = \sum_{i=1}^{\infty} (\frac{1}{16})^i = \sum_{i=0}^{\infty} (\frac{1}{16})^i - 1 = \frac{1}{1-\frac{1}{16}} - 1 = \frac{1}{15},$$

and

$$\sum_{i=1}^{\infty} 2^{-4i-1} = \frac{1}{2} \sum_{i=1}^{\infty} 2^{-4i} = \frac{1}{2} \cdot \frac{1}{15} = \frac{1}{30}.$$

Therefore,

$$\sum_{i=1}^{\infty} 2^{-4i} + 2^{-4i-1} = \frac{1}{15} + \frac{1}{30} = \frac{3}{30} = 0.1.$$

In order to convert a fraction to binary we need to repeatedly multiply by two, take decimal as the digit and take obtained fraction as the starting point for the next step.

In our case:

$$0.1 \times 2 = 0.2 \rightarrow \mathbf{0}$$

$$0.2 \times 2 = 0.4 \rightarrow \mathbf{0}$$

$$0.4 \times 2 = 0.8 \rightarrow \mathbf{0}$$

$$0.8 \times 2 = 1.6 \rightarrow \mathbf{1}$$

and so on. By repeting the procedure above we get:

$$0.1 \rightarrow 0.0001100110011001100\ldots$$

which shows that 0.1 indeed has the base 2 representation as $0.000\overline{1100}$.

The IEEE single-precision floating-point format (`Float32`) uses 32 bits of computer memory to represent numeric values and the bits are arranged as follows:

$$(-1)^{b_{31}} \times 2^{(b_{30}\dots b_{23})_2 - 127} \times (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}),$$

where b_i with $i = 0, \dots, 31$ denote bits. In our case we have:

$$0.000\overline{1100} = 1.100\overline{1100} \times 2^{-4}.$$

Therefore,

- the exponent is equal to -4 and by adding the bias we get:

$$(-4 + 127)_{10} = (123)_{10} = (1111011)_2 = (b_{29} \dots b_{23})_2$$

- the mantissa can be obtained from $1.100\overline{1100}$ by looking at the fractional part of the binary representation after the binary point and filling out all available bits b_i where $i = 0 \cdots 22$:

$$10011001100110011001100 = b_{22} \cdots b_0$$

- the sign bit b_{31} is 0.

Thus, by converting the IEEE single-precision representation of 0.1 back to binary we get:

$$\frac{x - \hat{x}}{x} = -\frac{0.0\overline{1100} \times 2^{-28}}{0.0\overline{1100} \times 2^{-2}} = -2^{-26} = -\frac{1}{4} 2^{-24} = -\frac{1}{4} u$$

which completes the proof.

□

Exercise 3 (2.5 P)

Providing function implementations, that produce expected answers for all IEEE numbers is sometimes surprisingly tricky. Extra care is in particular needed to ensure that special cases (`±Inf`, `NaN`, `±0` etc.) behave as expected.

Consider the following naive implementation of the `max` function. Does this always produce the expected answer for IEEE arithmetic? If yes, why? If no, suggest an improved version.

```
function max(a, b)
    if a > b
        return a
    else
        return b
    end
end
```

Solution:

Using the naive `max()` function defined above, there are a couple of edge cases that may pose problems. In our solution below, we will discuss these cases and suggest possible corrections.

Let's begin with defining the `naive_max(a,b)` function:

```
function naive_max(a, b)
    if a > b
        return a
    else
        return b
    end
end
```

We also define a function for finding the minimum to compare the results afterwards.

```
function naive_min(a, b)
    if a < b
        return a
    else
        return b
    end
end
```

Let's now go over the problematic cases and try to find a way to overcome the issues.

1 Comparing with Not a Number (NaN)

Let's check what happens to our `naive_max()` function if at least one of the inputs is `NaN`.

```
naive_max(NaN, -1.)
```

```
naive_max(NaN, Inf)
```

```
naive_max(Inf, NaN)
```

```
naive_max(NaN, NaN)
```

The problem here is that `NaN` cannot be a max or a min since it is not comparable to a number and the solution is to return `NaN`.

```
max(NaN, -1.)
```

In this case, we should add an additional check if there is `NaN` given as an input:

```
if isnan(a) || isnan(b)
    return NaN
end
```

2 Comparing `+0` with `-0`

Another problematic case is that the naive functions do not compare correctly 0 with different signs.

```
naive_max(-0.0, 0.0), naive_max(0.0, -0.0)
```

```
naive_min(-0.0, 0.0), naive_min(0.0, -0.0)
```

In order to fix this, we should check if both sides are equal to 0 and then check if they differ by a sign:

```
if a == 0.0 && b == 0.0
    if sign(a) > sign(b)
        return a
    else
        return b
    end
end
```

3 Comparing `+Inf` with `-Inf`

We can see that `naive_max()` function handles the `±Inf` cases correctly:

```
naive_max(-Inf, Inf), naive_max(Inf, -Inf)
```

```
naive_min(-Inf, Inf), naive_min(Inf, -Inf)
```

```
naive_max(-Inf, 0.), naive_max(Inf, 0.)
```

Final corrected function

Now that the problematic cases have been resolved, we can write the `max_corrected()` function, and compare it with the implemented `max()` function from Julia:

```
function max_corrected(a, b)
    """
    The corrected max function
    """
    # NaN case
    if isnan(a) || isnan(b)
        return NaN

    # 0.0 ≠ -0.0 case
    elseif a == 0.0 && b == 0.0
        if sign(a) > sign(b)
            return a
        else
            return b
        end
    else
        return naive_max(a, b)
    end
end
```

Comparing our improved function with `max()` implemented in Julia:

```
max_corrected(NaN, Inf), max(NaN, Inf)
```

```
max_corrected(-0.0, 0.0) == max(-0.0, 0.0)
```

Exercise 4 (2.5 + 0 P)

In this exercise we want to perform an iterative diagonalisation with 16-bit floating point numbers. The underlying idea is to simulate the scenario when we want to perform the diagonalisation of our matrix on a specialised hardware, which can only perform 16-bit operations.

We consider the usual power method implementation

```
begin
    using LinearAlgebra
    using BFloat16s
    using IntervalArithmetic
end
```

```
function power_method(A, u=randn(eltype(A), size(A, 2));
                      tol=1e-6, maxiter=100, verbose=true)
    norm_Δu = NaN
    for i in 1:maxiter
        u_prev = u
        u = A * u
        u /= norm(u)
        norm_Δu = min(norm(u - u_prev), norm(-u - u_prev))
        norm_Δu < tol && break
        verbose && println("$i   $norm_Δu")
    end
    μ = dot(u, A, u)
    norm_Δu ≥ tol && verbose && @warn "Power not converged $norm_Δu"
    (; μ, u)
end
```

And the example matrix

```
A = diagm([-10, 30, 0.2]) + 1e-3 * randn(3, 3)
```

(a) Run the power method using `Float16` and converge the procedure using `tol=1e-6`. Compute the error in the eigenvalue and eigenvector against a diagonalisation with `eigen`, which employs `Float64` precision. Compute the residual norm of the obtained eigenpair in `Float16` and `Float64` precision. What do you observe? Repeat the computation in `BFloat16`.

```
# Example for casting an array to Float16 and BFloat.
let x = [1.1, 2.2, 3.3]
    Float16.(x), BFloat16.(x)
end
```

```
begin
    # Float16
    e_value16, e_vector16 = power_method(Float16.(A))
    # Repeating  calculations for BFloat16
    e_valueB16, e_vectorB16 = power_method(BFloat16.(A))

    e_values, e_vectors = eigen(A)

    e_value64 = e_values[end]
    e_vector64 = e_vectors[:,end]
end
```

```
error_evalue16 = abs(e_value64 - e_value16)
```

```
error_evector16 = norm(e_vector64 - e_vector16)
```

```
residual_norm16 = norm(A * e_vector16 - e_value16 * e_vector16)
```

```
residual_normB16 = norm(A * e_vectorB16 - e_valueB16 * e_vectorB16)
```

```
residual_norm64 = norm(A * e_vector64 - e_value64 * e_vector64)
```

We can see that the residual norm of the obtained eigen pair in both `Float16` and `BFloat16` are bigger than in `Float64` as expected since lower precision can lead to larger numerical errors during the matrix-vector multiplication.

`BFloat16` format uses the same number of bits as `Float16` but allocates bits differently to improve the representation of larger values. As a result, we can see that the residual norms obtained with both formats are similar.

(b) *(optional exercise)* We now move from 16-bit to `Float32` as our working precision. Run the `power_method` on A using `Float32` precision. By employing interval arithmetic in `Float32` precision and by tuning the `tol` parameter appropriately compute the eigenpair in a way that you can computationally prove that your eigenpair is exact to a residual norm below 10^{-5}. What is the highest accuracy (smallest residual) you can provably achieve with `Float32`-only operations?

Hint: To cast a `Float32` array to an appropriate array of `Float32` intervals employ:

```
let
    float32_vector = randn(Float32, 3)
    interval.(float32_vector)
end
```

```
e_value32, e_vector32 = power_method(Float32.(A), tol=1e-8)
```

```
residual_norm32 = norm(A * e_vector32 - e_value32 * e_vector32)
```

```
interval.(norm(A * e_vector32 - e_value32 * e_vector32))
```

```
residual_norm32_interval = norm(interval.(A) * interval.(e_vector32) - interval.
(e_value32) * interval.(e_vector32))
```

In general, `Float32` format has limited precision but as demonstrated in the previous code example we are able to achieve a residual norm on the order of $1e-7$.