Error control in scientific modelling (MATH 500, Herbst)

# Sheet 5: Floating-point numbers

## Exercise 1

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 $\theta$ 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 $|\theta| \approx 0$ : $c = \sqrt{a^2 + b^2 - 2ab\cos\theta}$

*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.**

*(taken from Higham 1.3)*

The following expressions are cancellation-free or feature subtraction operations only for expressions involving the input arguments.

1. Use $\frac{x}{\sqrt{x+1}+1}$
2. Use $2\sin\left(\frac{x-y}{2}\right)\cos\left(\frac{x+y}{2}\right)$
3. Use $(x-y)(x+y)$
4. Use $\frac{\sin(x)}{1+\cos(x)}$ or $\tan(\frac{x}{2})$
5. Use

$$
\begin{aligned}
c &= \sqrt{a^2 + b^2 - 2ab\cos(\theta)} \\
&= \sqrt{a^2 + b^2 - 2ab(\cos(\theta) + 1 - 1)} \\
&= \sqrt{a^2 + b^2 - 2ab - 2ab(\cos(\theta) - 1)} \\
&= \sqrt{(a-b)^2 + 4ab\sin^2(\theta/2)}
\end{aligned}
$$

where we have used the half-angle identity $1 - \cos(\theta) = 2\sin^2(\theta/2)$.

# Exercise 2

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.**

*(taken from Higham 2.5)* We can use the geometric series

$$\sum_{i=1}^{\infty} r^i = \frac{1}{1-r} \quad \text{for } |r| < 1.$$

We have

$$\sum_{i=1}^{\infty} 2^{-4i} + 2^{-4i-1} = \frac{3}{2}\sum_{i=1}^{\infty}(2^{-4})^i = \frac{3}{2}\frac{2^{-4}}{1-2^{-4}} = \frac{3}{2}\frac{1}{15} = \frac{1}{10} = 0.1.$$

Since $x = 0.1$ in base $2$ thus reads as

$$x = 0.1100\,1100\,1100\,1100\,1100\,1100\,|\,1100\,\ldots \times 2^{-3},$$

**rounding** to 24 bits yields

$$\hat{x} = 0.1100\,1100\,1100\,1100\,1100\,1101 \times 2^{-3}.$$

Thus

$$\begin{aligned}
\hat{x} - x &= (0.001 - 0.000\overline{1100})_2 \times 2^{-21} \times 2^{-3} \\
&= \left(\frac{1}{8} - \frac{1}{10}\right) \times 2^{-24} \\
&= \frac{1}{40} \times 2^{-24},
\end{aligned}$$

and conclude $(x - \hat{x})/x = -\frac{1}{4} \times 2^{-24} = -\frac{1}{4}u$

```
0.30000000000000004
```

```
0.1 + 0.2
```

# Exercise 3

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.**

*(taken from Higham 2.22)*

The problem with the simple implementation is the treatment of `NaN` as in particular `x > y` is `false` as soon as $x$ or $y$ are `NaN`. This breaks the symmetry in the arguments:

```
max_naive (generic function with 1 method)
```

```
    # split: solution
    function max_naive(a, b)
        if a > b
            return a
        else
            return b
        end
    end
```

```
1.0
```

```
    # split: solution
    max_naive(NaN, 1.0)
```

```
NaN
```

```
    # split: solution
    max_naive(1.0, NaN)
```

An additional subtlety is that

```
    # split: solution
    md"An additional subtlety is that"
```

```
false
```

```
    # split: solution
    0.0 > -0.0
```

or more precisely that `-0.0` and `0.0` are equal whereas we would like `max(-0.0, 0.0)` to yield the positive sign. An alternative is to catch the `NaN` early and introduce a signbit check of the difference:

```
max_better (generic function with 1 method)
```

```
    # split: solution
    function max_better(a, b)
        if isnan(a)
            return a
        elseif isnan(b)
            return b
        elseif signbit(a - b)
            return b
        else
            return a
        end
    end
```

max(x::**T**, y::**T**) *where T<:Union{Float32, Float64}* in Base.Math at math.jl:834

```
    @which max(1.0, 2.0)
```

Testing some special cases:

```
    # split: solution
    md"Testing some special cases:"
```

```
0.0
```

```
    # split: solution
    max_better(0.0, -0.0)
```

```
-0.0
```

```
    # split: solution
    max_better(-0.0, -0.0)
```

```
NaN
```

```
    # split: solution
    max_better(1.0, NaN)
```

```
NaN
```

```
    # split: solution
    max_better(NaN, 1.0)
```

# Exercise 4

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
```

```

```

```
power_method (generic function with 2 methods)
```

```
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 = 3×3 Matrix{Float64}:
 -9.99879     -0.000176402  -0.00038136
  0.00144313  29.9992       -0.000930122
  0.00137257  -0.000872902   0.20037
```

```
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 $||\boldsymbol{Ax} - \lambda \boldsymbol{x}||$ of the obtained `Float16`-eigenpair in `Float16` and `Float64` precision. What do you observe? Repeat the computation in `BFloat16`.

```
([1.1, 2.2, 3.3], [BFloat16(1.10156), BFloat16(2.20312), BFloat16(3.29688)])
```

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

```
# Your code and answer here
```

```
Eigen{Float64, Float64, Matrix{Float64}, Vector{Float64}}
values:
3-element Vector{Float64}:
 -9.998786247196868
  0.20036946277793974
 29.99921203809707
vectors:
3×3 Matrix{Float64}:
  1.0         -3.73918e-5   4.40999e-6
 -3.60831e-5   3.12152e-5  -1.0
 -0.00013458   1.0          2.92934e-5
```

```
    # split: solution
    begin
        λ, X = eigen(A)
        μ_ref = λ[end]
        u_ref = X[:, end]
        Eigen(λ, X)
    end
```

```
Float16(0.000977)
```

```
    eps(Float16)
```

```
1.0e-20
```

```
    1e-20
```

```
1.0000000000000001e-20
```

```
    nextfloat(1e-20)
```

```
9.999999999999998e-21
```

```
    prevfloat(1e-20)
```

```
2.220446049250313e-16
```

```
    eps(Float64)
```

```
# split: solution
let
    (;μ, u) = power_method(Float16.(A); tol=1e-6)
    α = sign(dot(u, u_ref))

    @show abs(μ - μ_ref)       # Exact only to 1e-3
    @show norm(u - α * u_ref)  # Exact to 1e-7
    @show norm(Float16.(A) * u - μ * u)  # Residual computed in working precision
    @show norm(A * u - μ * u)  # 2 orders larger than the one above!
end;
```

```
1   1.595
2   0.05695
3   0.01877
4   0.006264
5   0.002087
6   0.0006905
7   0.0002441
8   7.725e-5
9   2.575e-5
10   8.6e-6
11   2.86e-6
abs(μ - μ_ref) = 0.0007879619029296236
norm(u - α * u_ref) = 2.407438042468105e-7
norm(Float16.(A) * u - μ * u) = Float16(9.6e-6)
norm(A * u - μ * u) = 0.007880820917779327
```

```
# split: solution
let
    (;μ, u) = power_method(BFloat16.(A); tol=1e-6)
    α = sign(dot(u, u_ref))

    @show abs(μ - μ_ref)       # Exact only to 1e-3
    @show norm(u - α * u_ref)  # Exact to 1e-7
    @show norm(BFloat16.(A) * u - μ * u)  # Residual computed in working precision
    @show norm(A * u - μ * u)  # 2 orders larger than the one above!
end;
```

```
1   BFloat16(1.0078125)
2   BFloat16(0.49609375)
3   BFloat16(0.171875)
4   BFloat16(0.057617188)
5   BFloat16(0.01928711)
6   BFloat16(0.006439209)
7   BFloat16(0.0021514893)
8   BFloat16(0.0007171631)
9   BFloat16(0.00023937225)
10   BFloat16(8.010864f-5)
11   BFloat16(2.670288f-5)
12   BFloat16(8.881092f-6)
13   BFloat16(2.9802322f-6)
abs(μ - μ_ref) = 0.0007879619029296236
norm(u - α * u_ref) = 2.413179873249909e-7
norm(BFloat16.(A) * u - μ * u) = BFloat16(9.536743f-6)
norm(A * u - μ * u) = 0.0007880211585519866
```

**solution (a)** The following points should be noticed by the students:

- The accuracy of the estimated eigenvalue in 16-bit arithmetic is very low. The eigenvector is of much higher accuracy, closer to what was requested via `tol`.
- This is a puzzling artifact of reduced precision, since the eigenvalue should converge quadratically, hence be more accurate.
- As a result the residual norm (if computed in `Float64` arithmetic) is also much larger than `tol`. In contrast, the residual norm computed in $16$-bit arithmetic is two orders of magnitude smaller, closer to `tol` and gives away the impression that everything went fine. We cannot
- Overall we get a much lower quality eigenpair than what one would expect from the selected `tol` parameter and from the $16$-bit computed residual norm.
- We see that for ill-conditioned settings or reduced precision care should be taken and even a residual smaller than the desired accuracy — if computed in a too low working precision — might not be sufficient to ensure convergence.
- `BFloat16` versus `Float16` makes no real difference.

(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:

```
[[0.0752853f0, 0.0752854f0], [-1.05412f0, -1.05411f0], [0.106079f0, 0.10608f0]]
```

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

**solution (b)** Employing e.g. `tol=1e-6` is sufficient to get the upper bound of the interval resulting in the residual computation below $10^{-5}$, thus we have achieved the goal of the problem. Proving a residual norm much below $10^{-6}$ cannot be achieved.

Syntax error

# Syntax errors

1. Expected `end`
   from This cell: *line 3*

2. extra tokens after end of expression
   from This cell: *line 3*

3. invalid identifier
   from This cell: *line 10*

```
    # split: solution
    let
        (;μ, u) = power_method(Float32.(A)); tol=1e-10)
        @show norm(Float32.(A) * u - μ * u)  # Computation in Float32

        # Computation using intervals:
        @show norm(
            interval.(A) * interval.(u) - interval(μ) * interval.(u)
        )
    end;
```