WROCŁAW UNIVERSITY OF SCIENCE AND TECHNOLOGY

Scientific Calculations - Task List 1

Marcel Jerzyk

October 6, 2019

1 TASK - EPS, ETA & MAX

The first task on the list is about designating *eps*, *eta* and *max* on all types of floating point numbers (*Float16, Float32, Float64*) iteratively and comparing to these from C language (from float.h).

1.1 GETTING TO THE SOLUTION

Started of with looking up for the numbers as provided in Julia language and printing them out:

eps(Float16), nextFloat(Float16(0.0)), floatmax(Float16)

Then, prepared values with according names to begin calculations to find those numbers.

Code 1: EPS

```
while (1 + (f16eps / 2)) > 1
    global f16eps /= 2
end
```

- Take half if: "/next half/ + 1" is grater than: "1"
- If while ends, it means that we reached macheps for this type, in "human" words:
- 1 + /something so tiny it looks like 0 in this type/ IS NOT greater than 1, so loop ends.

Code 2: ETA

```
while (f16eta / 2) > 0
    global f16eta /= 2
end
```

- Take half if: "/next half/" is greater than: "0"
- If while ends, it means that we reached eta for this type; in "human" words:
- /something so tiny it looks like 0 in this type/ IS NOT greater than 0, so loop ends.

Code 3: MAX

```
while (f16max*2) != typemax(Float16)
    global f16max *= 2
end
f16max *= (2-f16eps)
```

- Double the number if: "/next duplication/" will not be "inf"
- If while ends, it means that we reached half max for this type.
- To reach the max we need to subtract as little as its possible from "2" before next doubling to get as close as possible
- As little as its possible is previously calculated EPS

1.2 Results

```
Build-in Values:
Float16: Macheps [0.000977]                 NextFloat: [6.0e-8]          RealMax: [6.55e4]
Float32: Macheps [1.1920929e-7]             NextFloat: [1.0e-45]         RealMax: [3.4028235e38]
Float64: Macheps [2.220446049250313e-16]    NextFloat: [5.0e-324]        RealMax: [1.7976931348623157e308]

Results:
Float16: Macheps [0.000977]                 NextFloat: [6.0e-8]          RealMax: [6.55e4]
Float32: Macheps [1.1920929e-7]             NextFloat: [1.0e-45]         RealMax: [3.4028235e38]
Float64: Macheps [2.220446049250313e-16]    NextFloat: [5.0e-324]        RealMax: [1.7976931348623157e308]

julia>
```

Figure 1.1: Results for Task #1

Using that code the results matched with the values given by expressions build-in Julia language. The last thing to do is to check them with values in float.h from C lang.

C and Julia language comparison			
	Float16	Float32	Float64
(C) EPS	-	1.19209289550781e-7	2.220446049250313e-16
(J) EPS	0.000977	1.1920929e-7	2.220446049250313e-16
(C) MAX	-	3.402823466385288598e+38	1.797693134862315708e+308
(J) MAX	6.55e4	3.4028235e38	1.7976931348623157e308
(J) ETA	6.0e-8	1.0e-45	5.0e-324

By comparing all the gathered values I can assume that these floating point numbers are correct as they don't differ much from each other.

1.3 Conclusions

Real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers and that is the *EPS*. *ETA* is the next representable floating-point number to the argument. *Floatmax* and *Floatmin* gives us highest or lowest finite value representable by the given floating-point.

2 Task - Calculating Macheps

2.1 Problem - Kahan thoughts

Kahan said that by calculating $3 * (\frac{4}{3} - 1) - 1$ we can get macheps in floating point arithmetic. The task is to check if he is right in all available floating point numbers types in Julia.

2.2 Solution & Results

```
4   # Khan theory:
5   # It's true to get macheps by calculating
6   #           3 * ((4/3) - 1) - 1
7
8   function khanMacheps(Type)
9       eps = Type(3.0) * ((Type(4.0))/Type(3.0) - Type(1.0)) - Type(1.0)
10      return eps
11  end
```

```
Build-in vs. Khan:
Float16: Macheps [0.000977]                  Khan: [-0.000977]
Float32: Macheps [1.1920929e-7]              Khan: [1.1920929e-7]
Float64: Macheps [2.220446049250313e-16]     Khan: [-2.220446049250313e-16]

julia>
```

Figure 2.1: Code and it's compilation result for Task #2

2.3 Conclusions

The results matched if we would look only at the absolute values for the numbers. Turns out that if the number calculated before the last deduction $(3 * (\frac{4}{3} - 1))$ is in range from *0.0 to 0.(9)* the answer will have wrong sgn due to overflow.

```
22  # Trying to figure out why that happend
23  function testKhanForSomeKnowledge(Type)
24      t1 = Type(4.0)/Type(3.0)
25      t2 = ((Type(4.0))/Type(3.0) - Type(1.0))
26      t3 = Type(3.0) * ((Type(4.0))/Type(3.0) - Type(1.0))
27      t4 = Type(3.0) * ((Type(4.0))/Type(3.0) - Type(1.0)) - Type(1.0)
28      println(Type, ": t1 [", t1, "]\tt2: [", t2, "]", "\tt3: [", t3, "]", "\tt4: [", t4, "]")
29  end

Step by step calculations and the answer:
Float16: t1 [1.333]     t2: [0.333]     t3: [0.999]     t4: [-0.000977]
Float32: t1 [1.3333334] t2: [0.33333337]        t3: [1.0000001] t4: [1.1920929e-7]
Float64: t1 [1.3333333333333333]        t2: [0.33333333333333326]       t3: [0.9999999999999998]        t4: [-2.220446049250313e-16]

julia>
```

Figure 2.2: Closer look at the Task #2

3 Task - Even Steps in [1,2] (Float64)

3.1 Experiment that proves the task

Numbers in Float64 (double in IEEE 754 Standard) are evenly spread out in $[1,2]$ with the step $\delta = 2^{-52}$. In other words, every number in that range can be expressed by $x = 1 + k * \delta$, where $k = 1, 2, ..., 2^{52} - 1$ and $\delta = 2^{-52}$.

3.2 Solution

```
7   # Step testing function
8   function testRange(x, y, d)
9       println("\nRange: [$min - $max]")
10
11      # Testing for first 5 and last 5 numbers in k range
12      for k in [1, 2, 3, 4, 5, 2^52-4, 2^52-3, 2^52-2, 2^52-1, 2^52]
13          answ = (x + k * d)
14          println("$answ = $x + $k*d |-| Last five bin numbers: [$(bitstring(answ)[end-4:end])]")
15          if k == 5
16              println("...")
17          end
18      end
19  end
```

Figure 3.1: Range test with inclusion of custom delta

3.3 Results & Conclusion

The test for initial experiment succeeded assuring me by looking at changing bits on every step. The task asks about the arrangement of numbers in other ranges (0.5, 1.0) and (2.0, 4.0) so by putting said numbers into the function it turned out that I kept counting one unity number higher than minimum value instead of up to the max value.

```
Range: [1.0 - 2.0]
1.0000000000000002 = 1.0 + 1*d |-| Last five bin numbers: [00001]
1.0000000000000004 = 1.0 + 2*d |-| Last five bin numbers: [00010]
1.0000000000000007 = 1.0 + 3*d |-| Last five bin numbers: [00011]
1.0000000000000009 = 1.0 + 4*d |-| Last five bin numbers: [00100]
1.000000000000001 = 1.0 + 5*d |-| Last five bin numbers: [00101]
...
1.9999999999999991 = 1.0 + 4503599627370492*d |-| Last five bin numbers: [11100]
1.9999999999999993 = 1.0 + 4503599627370493*d |-| Last five bin numbers: [11101]
1.9999999999999996 = 1.0 + 4503599627370494*d |-| Last five bin numbers: [11110]
1.9999999999999998 = 1.0 + 4503599627370495*d |-| Last five bin numbers: [11111]
2.0 = 1.0 + 4503599627370496*d |-| Last five bin numbers: [00000]
```

Figure 3.2: Range test for [1.0 - 2.0]

```
0.5000000000000002 = 0.5 + 1*d |-| Last five bin numbers: [00010]
0.5000000000000004 = 0.5 + 2*d |-| Last five bin numbers: [00100]
1.4999999999999998 = 0.5 + 4503599627370495*d |-| Last five bin numbers: [11111]
1.5 = 0.5 + 4503599627370496*d |-| Last five bin numbers: [00000]

2.0000000000000004 = 2.0 + 2*d |-| Last five bin numbers: [00001]
2.000000000000001 = 2.0 + 3*d |-| Last five bin numbers: [00010]
2.9999999999999996 = 2.0 + 4503599627370494*d |-| Last five bin numbers: [11111]
3.0 = 2.0 + 4503599627370495*d |-| Last five bin numbers: [00000]
```

Figure 3.3: First range test for [0.5 - 1.0] and [2.0 - 4.0]

Based on the calculation result and how the bits had changed in those two different ranges I figured out that I have to change delta accordingly - double it or divide by 2 according to bits that had two times bigger/lesser step (and desired max value was 2 times further/closer away to present maximum number).

That gave me equal steps in all of desired ranges but size of the step differ in all of them ($\delta = 2^{-51}, 2^{-52}, 2^{-53}$ accordingly - the further from point 0.0 the further the step between numbers is).

4 Task - Seeking a number

4.1 Problem

Find a number in Float64 that is in range of $(1,2)$ and fulfills $x * (\frac{1}{x}) \neq 1$. Find lowest such number.

4.2 Code

```
4   # Define x as the lowest number that is not 1.0
5   x = nextfloat(Float64(1.0))
6
7   # Define upper bound as the higher number that is not 2.0
8   while x < prevfloat(Float64(2.0))
9       if x * (1/x) != 1
10          println(x)
11          break
12      end
13      global x = nextfloat(x)
14  end
15  # X is the number that satisfies both conditions in the Task
```

Figure 4.1: Solution to the Task

4.3 Results & Conclusions

The number is equal to 1.000000057228997 (expression gave answer: 0.9999999999999999). Computer dealt with math problem in order without "human" abstract checking for values (if maybe something could be reduced) and by rounding he lost precision giving "wrong" answer.

5 Task - different ways of adding up

5.1 Scalar Products

Write in Julia program that calculates scalar products.

$x = [2.718281828, 3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$
$y = [1486.2497, 878366.9879, 22.37492, 4773714.647, 0.000185049]]$

Implement these four algorithms and calculate if in four different ways for n = 5:

- Inwards (Code 4.)
- Backwards (Code 5.)
- From the biggest to the smallest (by adding positive numbers in order from the bioggest to the smallest and then add negative numbers from the smallest to the biggest and summing both results)
- From the smallest to the biggest (in reverse to the previous method)

Code 4: Inwards

```
S := 0
for i := 1 to n do
    S := S + xi * yi
end for
```

Code 5: Backwards

```
S := 0
for i := n downto 1 do
    S := S + xi * yi
end for
```

5.2 Solutions

Solution for first two methods is straightforward - just implemented the pseudocode into Julia language.

```
function sumForward(Type)
    sum = Type(0.0)
    for i = 1 : n
        sum += Type(Type(x[i]) * Type(y[i]))
    end
    println("[FRWR Sum] $Type: $sum")
end

function sumBackward(Type)
    sum = Type(0.0)
    for i = n:-1:1
        sum += Type(Type(x[i]) * Type(y[i]))
    end
    println("[BKWR Sum] $Type: $sum")
end
```

Figure 5.1: Implementation

Method 3. and 4. needed usage of sort function. I finally end up with merging two vector arrays into one that is calculated. Used sort!(vectors, rev=true) in order to make the values descend. After that I add positive values in valid order and after the loop finishes, it goes back to add negative in reverse (disclaimer: biggest negative is closer to 0, not further). Identical code is used for the ascending method but w/o reversing the sort.

```
25  function sumDescending(Type)
26      sum = Type(0.0)
27      sumPlus = Type(0.0)
28      sumMinus = Type(0.0)
29      vectors = Type[]
30
31      # Creating new vector array and sorting it
32      for i = 1 : n
33          push!(vectors, (x[i] * y[i]))
34      end
35      sort!(vectors, rev=true)
36
37      # Adding them from the biggest to the smallest for positive
38      #             and from smallest to the biggest for negative
39      for i = 1 : n
40          if vectors[i] > 0
41              sumPlus += vectors[i]
42          end
43          if i == n
44              for k = 0 : (n-1)
45                  if vectors[(i-k)] <= 0
46                      sumMinus += vectors[i-k]
47                  end
48              end
49          end
50      end
51      sum = sumPlus + sumMinus
52
53      println("[DESC Sum] $Type: $sum\t\t= ($sumPlus + $sumMinus)")
54  end
```

Figure 5.2: Descending Order

5.3 Results

	Float32	Float64
Forwards	-0.4999443	1.0251881368296672e-10
Backwards	-0.4543457	-1.5643308870494366e-10
Descending	-0.25	0.0
Ascending	-0.25	0.0

5.4 Conclusions

Keeping in mind that the correct expected result was $-1.00657107000000 * 10^{-11}$, the closest to that value was Float64 in Backward summing order. There's precision lost when adding sorted numbers to each other upon adding something bigger to something smaller.

6 Task - Various Results

6.1 Problem

Calculate in Julia language in Float64 these functions:

$f(x) = \sqrt{x^2+1} - 1$
$g(x) = \frac{x^2}{\sqrt{x^2+1}+1}$

with $x = 8^{-1}, 8^{-2}, 8^{-3}, ...$

6.2 Solution

Implementing said functions, iterating on the exponential of eight.

```
4   function fg(n)
5       for i = 1 : n
6           x = 8.0^(-i)
7           f = sqrt((x)^2.0 + 1.0) - 1.0
8           g = (x)^2 / (sqrt((x)^2.0 + 1.0) + 1.0)
9           println("[8^(-$i)] f = $f\tg = $g")
10      end
11  end
12
13  fg(200)
```

Figure 6.1: Functions implemented into Julia language

6.3 Results and Conclusion

Even though these two equations are equal to each other, both give various results after compilation.

	f(x) = sqrt(x^2+1) − 1	g(x) = $x^2/(sqrt(x^2+1)+1)$
8^{-1}	0.0077822185373186414	0.0077822185373187065
8^{-2}	0.00012206286282867573	0.00012206286282875901
8^{-3}	1.9073468138230965e-6	1.907346813826566e-6
...	...	...
8^{-8}	1.7763568394002505e-15	1.7763568394002489e-15
8^{-9}	0.0	2.7755575615628914e-17
8^{-10}	0.0	4.336808689942018e-19
...	...	...
8^{-178}	0.0	1.6e-322
8^{-179}	0.0	0.0

Function f loses information on -9th power and from that iteration onwards it will always be equal to 0.0. Same happens with function g but many iterations later - at -179th. Because of that it's safe to assume that the g(x) function is more accurate.

7 Task - Derivative of a function

7.1 Problem

Calculate derivative of a function in Julia language (in Float64) to approximate the value of the function $f(x) = sin(x) + cos3x$ in point $x_0 = 1$ and error $|f'(x_0 - f''(x_0)|$ for $h = 2^{-n}$, where $(n = 0, 1, 2, ..., 54)$.

7.2 Solution

```
4   # Calculating derivative:
5   #
6   # f(x) = sin(x) + cos(3*x)
7   #
8   # (sin(x))' = cos(x)
9   #
10  # (cos(3x))'
11  #       (cos(u))' = -sin(u)
12  #       (3x)'     = 3
13  #       ______________________
14  #                 = -3sin(3x)
```

Figure 7.1: Calculating Derivative

```
16  x0 = Float64(1.0)
17  f(x0) = sin(x0) + cos(3*x0)
18  g(x0) = cos(x0) - 3*sin(3*x0)
19
20  for i = Float64(0) : Float64(54.0)
21      h = Float64(2.0)^(-i)
22      point = ((f(x0 + h) - f(x0)) / h)
23      error = abs(g(x) - point)
24      println("[h^(-$i)]\tpoint = $point\terr = $error")
25  end
```

Figure 7.2: Calculating error and point approx

7.3 Results and Conclusion

iteration	f'(x)	error
0	2.0179892252685967	1.90104694358005855
1	1.8704413979316472	1.753499116243109
2	1.1077870952342974	0.9908448135457593
...	...	...
14	0.11718851362093119	0.0002462319323930373
15	0.11706539714577957	0.00012311545724141837
...	...	...
48	0.09375	0.023192281688538152
49	0.125	0.008057718311461848
50	0.0	0.11694228168853815
51	0.0	0.11694228168853815
52	-0.5	0.6169422816885382
53	0.0	0.11694228168853815

When the h is very low the expression $(h+1)$ starts rounding up to "1". Because of that issue it's not possible to make infinite approximation of the derivative.