Student Name: Seniha Serra Bozkurt
Student ID: 150190710
Class CRN: 21137

April 6, 2021
Istanbul Technical University
Department of Computer Engineering

BLG 202E - Numerical Methods in CE
Assignment 1

QUESTION 1:

a. To derive $f2(x_0, h)$, if we pick the variables in the following manner: $\phi = x_0 + h$, $\psi = x_0$ the equation will become:

$f2(x_0, h) = 2 * \cos(((2*x_0) + h) / 2) * \sin(h / 2)$
$= 2 * \cos(x_0 + h / 2) * \sin(h / 2)$

b. $f'(x) = (f(x_0 + h) - f(x_0)) / h$ ==>> The derivative formula
$= (\sin(x_0 + h) - \sin(x_0)) / h$
$= 2 * \cos(x_0 + h / 2) * \sin(h / 2) / h$ ==>> put the formula from part-a

To calculate the approximation $f'(1.2)$, the python script is as following:

```
import math

x0 = float(input()) # here 1.2 in this homework case

for i in range(-20, 0):
    h = 10**i
    y = 2 * math.cos(x0 + h/2) * math.sin(h/2) / h
    value = round(y, 7)
    print("When h = 1e" + str(i) + ", value:", value)
```

And the output of the code snippet above is:

```
When h = 1e-20, value: 0.3623578
When h = 1e-19, value: 0.3623578
When h = 1e-18, value: 0.3623578
When h = 1e-17, value: 0.3623578
When h = 1e-16, value: 0.3623578
When h = 1e-15, value: 0.3623578
When h = 1e-14, value: 0.3623578
When h = 1e-13, value: 0.3623578
When h = 1e-12, value: 0.3623578
When h = 1e-11, value: 0.3623578
When h = 1e-10, value: 0.3623578
When h = 1e-9, value: 0.3623578
When h = 1e-8, value: 0.3623577
When h = 1e-7, value: 0.3623577
When h = 1e-6, value: 0.3623573
When h = 1e-5, value: 0.3623531
When h = 1e-4, value: 0.3623112
When h = 1e-3, value: 0.3618917
When h = 1e-2, value: 0.3576916
When h = 1e-1, value: 0.315191
```

QUESTION 2:

3)

a. The numerical difficulty is the det(matrix) being very close to 0. So its condition number is very large, causing numerical instability.

b. After doing the matrix multiplication, we would obtain the following equations:
$ax + by = \mathbf{1}$
$bx + ay = \mathbf{0}$

If we add second equation to the first one:

$$ax + by = \mathbf{1}$$
$$\underline{+bx + ay = \mathbf{0}}$$
$$ax + ay + bx + by = \mathbf{1}$$

By doing some proper calculations:

$$a*(x + y) + b*(x + y) = \mathbf{1}$$
$$(x + y)*(a + b) = \mathbf{1}$$
$$(x + y) = \mathbf{1} / (a + b)$$

c. Determine whether the following statement is true or false, and explain why:
- "When $a \approx b$, the problem of solving the linear system is ill-conditioned but the problem of computing $x + y$ is not ill-conditioned."

We should obtain expressions for the variables to determine a condition of linear system solving, and we are given the following matrix, so we will use it:

$$\begin{bmatrix} a & b \\ b & a \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow ax + by = 1, \quad bx + ay = 0$$

After solving the equations with two unknown values, we obtain these equations below:

$$x + y = \frac{1}{a+b} \qquad x - y = \frac{1}{a-b} \qquad x = \frac{a}{a^2-b^2} \qquad y = \frac{-b}{a^2-b^2}$$

To understand the condition, we should determine how the small changes in a&b affects x&y. We will do that by giving random values to a&b and demonstrate the effects.

1. Pick **a = 2.001** and **b = 2.000** $\Rightarrow\Rightarrow\Rightarrow$ we obtain **x = 500.125** and **y = -499.875**

After making a small perturbation in the data:

2. Pick **a = 2.002** and **b = 2.000** $\Rightarrow\Rightarrow\Rightarrow$ we obtain **x = 250.125** and **y = -249.875**

As seen above, a **large** difference in the result is produced, so solving this linear problem is **ill-conditioned**, but in either case **z = x + y ≈ 0.25** no significant change happened! That means the calculation of x+y is **not ill-conditioned**.

SO THE STATEMENT IS TRUE.

QUESTION 3:

W/ exact rounding, each elementary operation has a relative error which is bounded by the rounding unit η: for two floating point numbers x and $y \Rightarrow fl(x + y) = (x + y) \cdot (1 + \epsilon), |\epsilon| \leq \eta$.
But is this true also for elementary functions such as sin, ln, and **exponentiation**?

1. Estimate the **relative error** in calculating x^y in floating point, everything else is exact.

   $fl(x + y) = (x + y) \cdot (1 + \epsilon), |\epsilon| \leq \eta$ ……(1)
   $fl(lnz) = (lnz) \cdot (1 + \epsilon), |\epsilon| \leq \eta$……(2)
   $x^y = e^{y.ln(x)}$ (*assuming* $x > 0$).....(3)

   For (1), we give $x = x^y$, $y = 0$. So (1) becomes:
   $fl(x^y + 0) = (x^y + 0) \cdot (1 + \epsilon), |\epsilon| \leq \eta$
   $\Rightarrow$ $fl(x^y) = x^y \cdot (1 + \epsilon), |\epsilon| \leq \eta$……(4)

2. Show that the sort of bound we have for elementary operations and for ln **does not hold** for exponentiation **when** x^y **is very large**.

   By the exponentiation equation (3) that is given to us from the question:
   $x^y = e^{y.ln(x)}$ $(x > 0)$
   $fl(x^y) = fl(e^{yln(x)})$
   $= e^{y.ln(x).(1+\epsilon)}$ $\Rightarrow$ Since $e^{y.ln(x)} = x^y$ is given us by question,
   $= x^{y \cdot (1+\epsilon)}$
   $fl(x^y) = (x^y).(x^y)^{\epsilon}$……(5)

   By using the equation-(5) above and the equation-(4) that is $fl(x^y) = x^y \cdot (1 + \epsilon), |\epsilon| \leq \eta$,
   $(x^y).(x^y)^{\epsilon} \stackrel{?}{=} x^y \cdot (1 + \epsilon)$ $|\epsilon| \leq \eta$,
   $(x^y)^{\epsilon} \stackrel{?}{=} (1 + \epsilon)$ $|\epsilon| \leq \eta$,
   In conclusion, for this equation to hold: $\mathbf{(x^y)^{\epsilon} \leq (1 + \epsilon)}$
   So the bound we have for elementary operations and for ln does not hold for exponentiation when x^y is very large.

QUESTION 4:

For a given floating point system, design a library function that returns cube root $y^{1/3}$ of $y > 0$.

$\{ \forall y : y > 0 \mid y = a*2^e \mid 0.5 \leq a < 1 \mid e \in Z \} \Rightarrow$ must be very efficient, should **always** work.

For efficiency $\Rightarrow$ store some useful constants ahead of computation time: $2^{1/3}$, 2/3, a/3.

a. I have **2** different approaches to obtain $y^{1/3}$, once $a^{1/3}$ has been calculated, w/ **flops ≤ 5.**

Approach - 1:

If we have $a^{1/3}$, ⅔ and $2^{1/3}$ as constants, we could approach this problem as: to find $y^{1/3}$, wee need to find $2^{e/3}$, since $y^{1/3} = (a*2^e)^{1/3} = (a^{1/3})*(2^{e/3})$ and we already have $(a^{1/3})$ factor, as stated in the question. So to do the calculation of $2^{e/3}$, we first change the e/3 part from an improper fraction to summation of an integer and a proper fraction. Like a mixed fraction. Click here for more info on fraction types. To demonstrate with an example: if e = 4 $\Rightarrow$ then $\Rightarrow$ e/3 = 4/3, but I'll represent it in the following manner: e = 1 + ⅓. The ease it does to the calculations is that 2^e becomes $2^{1+1/3}$. By applying some simple math: $2^{1+1/3} = (2^1).(2^{1/3})$ so finding (2^1) does not require any operations with floating points, we reduced our complexity here. And for the $(2^{1/3})$ part, we only should look for that (e mod 3) is, for the numerator and denominator would always be = 3. So after finding the required numerator, we would apply the $2^{1/3}$ or $2^{2/3}$ (I know that possible values of (e mod 3) = {0, 1, 2} and this could give me a value of $2^{0/3}$, but since $2^{0/3} = 2^0 = 1$, and multiplication with a number with 1 doesn't change it's value so I don't considered it) **Code visualisation of this:**

```
def cube_root_finder(y):
    a, e = y, 0
    while a >= 1:
        a /= 2
        e += 1

    const1, const2, const3 = 2/3, 2**(1/3), a**(1/3)
    x = 2 ** (e//3) #apply integer division to e
    #and then check get the power of remaining modulus part

    x *= (2 ** ((e % 3)/3))

    return x * const3

print("Cube root =", cube_root_finder(float(input("Input a float: "))))
```

Some outputs of the code snippet above:

```
Input a float: 8
Cube root = 2.0
Input a float: 4
Cube root = 1.5874010519681996
Input a float: 64
Cube root = 4.0
```

```
Input a float: 4252151
Cube root = 162.00791253330192
Input a float: 56
Cube root = 3.8258623655447783
Input a float: 5
Cube root = 1.709975946676697
```

Approach - 2: The newton iteration approach. Used in the following part-b:

b. Derived the corresponding Newton iteration.

So, give $y = x^3$ since $y = a*2^e$,

$x^3 = a*2^e$

$x^3 - a*2^e = 0 \Rightarrow$ obtained an $f(x) = 0$ equation, so our $f(x) = x^3 - a*2^e$

Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

By using Newton's method, with f(x) obtained above, our $f'(x) = 3.x^2$, so the generalised equation for this formula would become:

$x_{n+1} = x_n - (f(x_n) / f'(x_n)) \Rightarrow\Rightarrow\Rightarrow\Rightarrow\Rightarrow\Rightarrow x_{n+1} = x_n - (x_n^3 - a*2^e) / 3.x_n^2$

To simplify this a little, we will use some simple math tricks:

$x_{n+1} = x_n - (x_n^3 / 3.x_n^2 - a*2^e / 3.x_n^2)$

$= x_n - x_n^3 / 3.x_n^2 + a*2^e / 3.x_n^2$

$= x_n - x_n / 3 + a*2^e / 3.x_n^2$

$= 2*x_n / 3 + a*2^e / 3.x_n^2$

$= x_n*(⅔) + a*2^e / 3.x_n^2$

$= x_n*(⅔) + 2^e*(a/3) / x_n^2$……this will be the equation implemented to the Newton code.

Our pre-determined constants for efficiency purposes are: const-1 = ⅔ and const-2 = $(2^e)*(a/3)$.

The flop count per iteration = 4 :

- 1 from const-1 * x_n,
- 1 from x_n^2 (x_n*x_n),
- 1 from const-2 / x_n^2,
- 1 from final summation operation

This method implemented in a python function below:

```
def cube_root_finder_with_newton(y):
    a, e = y, 0

    while a >= 1:
        a /= 2
        e += 1

    const1, const2 = 2/3, (a/3)*(2**e)
    xn, xn1 = 0, const2

    while (abs(xn1 - xn) > 10**(-16)):# bc 2^(-52) ≈ 1.1*10^(-16)=> our
tolerance to error
        xn = xn1
        xn1 = const1*xn + const2/(xn**2)

    return xn1

y = float(input("Input a float number: "))
print("Cube root:", cube_root_finder_with_newton(y))
```

Some outputs of the code snippet above:

```
Input a float number: 3
Cube root: 1.4422495703074083
Input a float number: 8
Cube root: 1.9999999999999998
Input a float number: 64
Cube root: 3.9999999999999996
Input a float number: 45.36
Cube root: 3.5663531716578776
Input a float number: 43265327755247
Cube root: 35105.89094265875
Input a float number: 4365464274276426862
Cube root: 1634344.0244046878
```

c. How would you choose an initial approximation? Roughly how many iterations are needed? (The machine rounding unit is 2^{-52})

I would go with my approach-1 from above, because it requires *O(1)* operations, while the Newton method requires *O(n)* complexity. The usual floating point word in the standard floating point system, has 64 bits. Of these, 52 bits are devoted to the mantissa (or fraction) while the rest

store the sign and the exponent. Hence the rounding unit is $\eta = 2^{-53} \approx 1.1\times10^{-16}$. The decimal system is convenient for humans; but computers prefer binary. In binary the floating point representation (with number of digits = **t**) is:

$\mathbf{fl(x) = \pm (1.d_1d_2d_3 \cdot \cdot \cdot d_{t-1}d_t) \times 2^e}$ with binary digits $d_i = 0$ or 1 and exponent e.

Modern floating point systems, such as the celebrated IEEE standard, guarantee that the relative error is bounded by rounding unit $\eta = \frac{1}{2} \times 2^{-t}$ and here, in our case $\eta = 2^{-52}$.

$\eta = 2^{-t} \cdot 2^{-1} = 2^{-t-1} = 2^{-52}$,

$-t-1 = -52$

$t+1 = 52$

$t = 51 \Rightarrow$ digits of decimal part, in binary representation (base = 2). This corresponds to almost 10^{16} digits of decimal part, in decimal representation (base = 10). That's why I picked 10^{-16} as error accumulation between x_{n+1} and x_n.