MAT-INF 1100 - Mandatory Assignment 1

William Dugan

30. september 2021

Task 1

We are given the following difference equation

$$x_{n+2} - 4x_{n+1} - x_n = 0, \quad \text{where } x_0 = 1 \text{ og } x_1 = 1. \tag{1}$$

a) Before I write a program calculating $x_2, x_3, ..., x_{100}$, I rewrite equation (1) as follows:

$$\begin{aligned} x_{n+2} - 4x_{n+1} - x_n &= 0 \\ x_n - 4x_{n-1} - x_{n-2} &= 0 \\ 4x_{n-1} + x_{n-2} &= x_n \end{aligned}$$

This makes it easier for me to express x_n in terms of the two previous values. I place the initial values in a list, x, where n is the same as the list index. The for-loop is used to calculate the i-th value in the series. I add this value to my list, x, and I print the values.

```
x = [1, 1]
n = 100
for i in range(len(x), n+1):
        x.append(4*x[i-1] + x[i-2])
        print(i, x[i])
```

Running the code gives me the following result with format 'n x_n':

```
2 5
3 21
4 89
5 377
...
96 42654784246173937946014998000244228812489467885371395311443 3
97 18068856563237992497389336395866335131607925787813101397453 45
98 76540904677569363784158845383489763407680649939789545120958 13
99 32423247527351544763402471792982538876233052554697128188128597
100 13734708057716311543202577171027913184570027521276746726461020 1
```

It looks as the program is working and prints the correct values.

b) Here, I simply modify the program from (a) and change the value of $x[1]$.

```
import math
x = [1, 2-math.sqrt(5)]
n = 100
for i in range(len(x), n+1):
        x.append(4*x[i-1] + x[i-2])
        print(f"{i} {x[i]:10e}")
```

This code print the following:

```
2  5.572809e-02
3 -1.315562e-02
4  3.105620e-03
5 -7.331374e-04
6  1.730703e-04
7 -4.085635e-05
8  9.644873e-06
9 -2.276857e-06
10  5.374453e-07
11 -1.270758e-07
12  2.914229e-08
13 -1.050661e-08
14 -1.288417e-08
15 -6.204330e-08
16 -2.610574e-07
17 -1.106273e-06
18 -4.686149e-06
19 -1.985087e-05
20 -8.408962e-05
21 -3.562093e-04
22 -1.508927e-03
23 -6.391917e-03
24 -2.707660e-02
25 -1.146983e-01
26 -4.858698e-01
27 -2.058178e+00
28 -8.718580e+00
29 -3.693250e+01
30 -1.564486e+02
31 -6.627268e+02
32 -2.807356e+03
33 -1.189215e+04
34 -5.037595e+04
```

```
35 -2.133960e+05
36 -9.039598e+05
37 -3.829235e+06
38 -1.622090e+07
39 -6.871284e+07
40 -2.910722e+08
41 -1.233002e+09
42 -5.223080e+09
43 -2.212532e+10
44 -9.372436e+10
45 -3.970228e+11
46 -1.681815e+12
47 -7.124284e+12
48 -3.017895e+13
49 -1.278401e+14
50 -5.415393e+14
51 -2.293997e+15
52 -9.717529e+15
53 -4.116411e+16
54 -1.743740e+17
55 -7.386600e+17
56 -3.129014e+18
57 -1.325472e+19
58 -5.614788e+19
59 -2.378462e+20
60 -1.007533e+21
61 -4.267978e+21
62 -1.807944e+22
63 -7.658575e+22
64 -3.244224e+23
65 -1.374276e+24
66 -5.821525e+24
67 -2.466037e+25
68 -1.044630e+26
```

```
69 -4.425125e+26
70 -1.874513e+27
71 -7.940564e+27
72 -3.363677e+28
73 -1.424876e+29
74 -6.035873e+29
75 -2.556837e+30
76 -1.083093e+31
77 -4.588058e+31
78 -1.943532e+32
79 -8.232935e+32
80 -3.487527e+33
81 -1.477340e+34
82 -6.258114e+34
83 -2.650980e+35
84 -1.122973e+36
85 -4.756990e+36
86 -2.015093e+37
87 -8.536072e+37
88 -3.615938e+38
89 -1.531736e+39
90 -6.488538e+39
91 -2.748589e+40
92 -1.164321e+41
93 -4.932142e+41
94 -2.089289e+42
95 -8.850370e+42
96 -3.749077e+43
97 -1.588135e+44
98 -6.727446e+44
99 -2.849792e+45
100 -1.207191e+46
```

c) We have to use the characteristic equation for our second order difference equation in order to find the general solution.

$$x_{n+2} - 4x_{n+1} - x_n = 0 \tag{2}$$
$$r^2 - 4r - 1 = 0 \tag{3}$$

$$r = \frac{4 \pm \sqrt{16 - 4 * (-1)}}{2} = 2 \pm \sqrt{5}$$

Using the solutions to equation (3) we can write the general solution as

$$x_n = C(2 - \sqrt{5})^n + D(2 + \sqrt{5})^n \tag{4}$$

The two initial values are $x_0 = 1$ and $x_1 = 2 - \sqrt{5}$. We can make a set of equations with two unknowns, C and D.

$$C(2 - \sqrt{5})^0 + D(2 + \sqrt{5})^0 = 1 \tag{5}$$
$$C(2 - \sqrt{5})^1 + D(2 + \sqrt{5})^1 = 2 - \sqrt{5} \tag{6}$$

Solving (5) and (6) gives $C = 1$ and $D = 0$. Using these values we can write the specific solution as:

$$x_n = (2 - \sqrt{5})^n \tag{7}$$

d) To test if the solution in task c is the same as in b, I wrote the following python program:

```
1  import math
2
3  n = 100
4
5  x_b = [1, 2-math.sqrt(5)]       # Code from (b)
6  for i in range(len(x_b), n+1):
7     x_b.append(4*x_b[i-1] + x_b[i-2])
8
9  x_c = [0] * (n + 1)             # Using equation (7)
10 for i in range(n+1):
11    x_c[i] = (2 - math.sqrt(5))**i
12
13 for i in range(0, n+1, 5):
14    error = abs(x_b[i] - x_c[i])
15    print(f"{i:3}, {error:e}")
```

It calculates and prints the relative error between the analytical answer in c and the numerical answer in b. The error is only printed for a handful of n values.

```
Terminal> python oppgave1d.py
 0, 0.000000e+00
 1, 0.000000e+00
 2, 4.857226e-16
 3, 1.828399e-15
 4, 7.827940e-15
 5, 3.313419e-14
10, 4.519737e-11
15, 6.164925e-08
20, 8.408962e-05
25, 1.146983e-01
30, 1.564486e+02
35, 2.133960e+05
40, 2.910722e+08
45, 3.970228e+11
50, 5.415393e+14
55, 7.386600e+17
60, 1.007533e+21
65, 1.374276e+24
70, 1.874513e+27
75, 2.556837e+30
80, 3.487527e+33
85, 4.756990e+36
90, 6.488538e+39
95, 8.850370e+42
100, 1.207191e+46
```

For $n = 0$, the error is 0 since the initial values are the same. When $n = 1$, the error from representing $\sqrt{5}$ as a floating point number is the same for both approaches, therefore the relative error is still 0. Regardless, when $n > 1$, the numerical approach will have a accumulation of round-off error, since $\sqrt{5}$ will have an infinite number of decimal places, but the computer can only represent a handful of them. Generally, it looks as the error is increasing by an order of magnitude for every other value of x_n.

I observe that the numerical solution is approaching $-\infty$ while the analytical solution is approaching 0. This is seen by looking at the last line in the print in (b) and in (d). Since the error is approximately the same as the x_{100} value from (b), it means that x_{100} in (c) is ≈ 0. We can also see this by comparing equation (4) and (7). The error in C and D will be $1 + \epsilon_c$ and $0 + \epsilon_d$. Since $|2 - \sqrt{5}| < 1$, equation (7) will approach 0. This goes for the first part of (4) as well, but since D $= \epsilon_d$, $\epsilon_d(2 + \sqrt{5})^n$ will approach ∞ since $|2 + \sqrt{5}| > 1$. This applies even though ϵ_d is really small.

Task 2

a) To solve this task, we have to write a program that uses the formula

$$\binom{n}{i} = \prod_{j=1}^{i} \frac{n-j+1}{j}. \tag{8}$$

This gives me the following code:

```
1  import sys
2  import math
3
4  n, i = sys.argv[1:3]                    # input arguments for n, i
5  n = float(n)                            # converts string to float
6  i = float(i)
7
8
9  def coefficient(n, i):
10    if i > n / 2.0:                      # see task (c)
11           i = n - i
12    s = 1.0                              # start sum = 1
13    for j in range(1, int(i) + 1):
14           s *= float(n - j + 1) / float(j)
15    return s
16
17
18 print(f"{coefficient(n, i):.15e}")       # printing call of function
19 r = abs((math.comb(int(n), int(i)) - coefficient(n, i))) /
20     abs(math.comb(int(n), int(i)))      # calculating error
21 print(f"r = {r:e}")                      # printing error
```

I set $s = 1.0$ since when $j = 1$, the sum is 1. It is the lines 13 and 14 that implement the product notation from equation (1). The last three lines is to calculate the relative error. The expression math.comb is used to produce the same results as in the task. Running the code with n and i values given in the task gave the following results:

```
1  Terminal> python binomial_coefficient.py 5000 4
2  2.601042812375000e+13
3  r = 0.000000e+00
4
5  Terminal> python binomial_coefficient.py 1000 500
6  2.702882409454359e+299
7  r = 2.475719e-15
8
9  Terminal> python binomial_coefficient.py 100000 99940
10 1.180691979962567e+218
11 r = 9.957991e-16
```

Line 3 shows that when n and i is small, there is no round off error. Nevertheless, in line 7 and 11 we observe an error. Let's take the second trial as an example of the error:

$$Task: \binom{1000}{500} = 2.702882409454\underline{66} \cdot 10^{299},$$

$$Calculated: \binom{1000}{500} = 2.702882409454\underline{59} \cdot 10^{299}.$$

We observe that the last two digits are not the same. This is due to the large amount of operations being performed, leading to an accumulation of round off errors. We are using float numbers since it allows us to calculate lager values for both the numerator and denominator, as well as the coefficient itself.

b) With the code in (a) I do not get an 'Overflow error', but python returns the value 'inf' as in infinity. This is pythons way of dealing with arithmetics that lead to an overflow. To try to brute force my program to fail, I wrote the following code:

```
1 for n in range(2, 10000000):
2    i = int(n/2 - 1)
3    if coefficient(n, i) != math.inf:
4       print(coefficient(n, i))
5    elif coefficient(n, i) == math.inf:
6       print(f"inf reached. n = {n}, i = {i}")
7       exit()
```

From line 2: I use this value for i since it is the one that will create the largest number of operations being performed. The code runs until $n = 1030$ and $i = 514$. This means that the largest floating point number that my computer can represent is approximately $\binom{1029}{514} \approx 1.429821 * 10^{308}$

c) Since Pascal's triangle is symmetrical, we can write

$$\binom{n}{i} = \binom{n}{n-i}$$

Therefore, we add the following if-statement to the code (as seen in (a), line 10-11 in the first block of code).

```
1 if i > n / 2.0:
2    i = n - i
```

This let's us minimize the number of operations being performed, thus reducing the possibility for round-off errors.

Task 3

```
1  from random import random
2
3  antfeil = 0; N = 100000
4
5  for i in range(N):
6     x = random(); y = random(); z = random()
7     res1 = (x + y) * z
8     res2 = x*z + y*z
9     if res1 != res2:
10       antfeil += 1
11       x0 = x; y0 = y; z0 = z
12       ikkelik1 = res1
13       ikkelik2 = res2
14
15 print (100. * antfeil/N)
16 print (x0, y0, z0, ikkelik1 - ikkelik2)
```

```
30.859
0.6087077776638925  0.9204274878392227  0.06851310883531125  -1.3877787807814457e-17
```

a) Line 6 assigns random numerical values to x,y,z. Then you have two mathematical statements, res1 and res2. The value of these should be the same, but due to errors that is not always the case. Therefore, we test for this, which is what the if-statement does. It says that if res1 is not equal to res2, add 1 to the error-counter (antfeil), store the values for x,y,z as x0,y0,z0. Then we set ikkelik1 as res1 and ikkelik2 as res2. This will save the last occurrence of an error in the range 0, 100000.

The first print statement prints the percentage of calculations where our program has detected res1 $\neq$ res2, meaning that the if-statement is True. The values in the final line printed is the last x,y,z values in the range that made an error, and the rightmost value is the difference between res1 and res2.

b)

```
from random import random

antfeil = 0
N = 100000

for i in range(N):
   x = random()
   y = random()
   z = random()
   res1 = (x+y) * (y+z)
   res2 = x*y + x*z + y*y + y*z
   if res1 != res2:
      antfeil += 1
      x0 = x
      y0 = y
      z0 = z
      ikkelik1 = res1
      ikkelik2 = res2

print(100. * antfeil/N)
print(x0, y0, z0, ikkelik1 - ikkelik2)
```

```
41.642
0.7157340474556305 0.016437627194271553 0.491972627514226 5.551115123125783e-17
```

We observe that the percentage of calculations leading to an error is increased by ≈ 10 %. This is explained by the higher number of calculations being performed, where there is an accumulation of round-off errors. Note that the difference between 'ikkelik1' and 'ikkelik2' is the same order of magnitude in both (a) and (b), which tells me that even though there are more errors occurring, the error itself is not significantly larger.

As Mike has talked about in lectures, when multiplying two floating-point numbers a and b with error δa and δb, the error is less than or equal to $\delta a + \delta b$.