MOwNiT – Arytmetyka Komputerowa

March 16, 2022

1 Treść zadania

W dokładnej arytmetyce ciąg $x_{k+1} = 111 - (1130 - 3000/x_{k-1})/x_k$, $x_0 = 11/2$, $x_1 = 61/11$ jest rosnący i zbieżny do 6. Obliczyć na swoim komputerze x_{34} (dla zmiennych typu *float*, *double*, *long double*) i spróbować wyjaśnić uzyskane wyniki. Dokładna wartość (zaokrąglona do 4 cyfr znaczących) jest równa $x_{34} = 5.998$.

2 Dane techniczne sprzętu

Obliczenia zostały wykonane na komputerze o następujących parametrach: - Procesor: AMD Ryzen 7 4700U (8 rdzeni, 8 wątków), - Pamięć RAM: 16 GB 3200 MHz (2×8GB), - Systemy operacyjne: - Windows 10 Education x64, - Ubuntu 20.04 x64 (do porównania wyników otrzymanych w języku C++ - były takie same dla obu systemów operacyjnych)

3 Przygotowanie

3.1 Przyjęte oznaczenia

W wykorzystywanym kodzie przyjmuję następujące oznaczenia:

x_{k+1} - `calc_next_x(x_curr, x_prev)` (x_{k+1} obliczam, przy pomocy funkcji `calc_next_x`)

x_k - `x_curr`

x_{k-1} - `x_prev`

3.2 Biblioteki

W celu uzyskania wysokiej precyzji obliczeń, z wykorzystaniem liczb zmiennoprzecinkowych, posłużę się dostarczoną wraz z Pythonem biblioteką `decimal` oraz biblioteką `fractions`.

Do stworzenia wykresów, wykorzystuję biblioteki `matplotlib` oraz `seaborn`. W celu łatwiejszego przetwarzania danych, korzystam z bibilioteki `pandas`.

[1]:
```
import decimal
from decimal import Decimal
from fractions import Fraction

import pandas as pd
from matplotlib import pyplot as plt
```

```
import matplotlib.ticker as mticker
import seaborn as sns
```

3.3 Obliczenia

[2]:
```
df = pd.DataFrame()
```

3.3.1 C++

Program `main.cpp`, przy pomocy którego dokonałem obliczeń w języku C++, dołączony jest wraz z plikami rozwiązania zadania.

Float (4 bajty) Znak: 1 bit

Mantysa: 23 bity

Wykładnik: 8 bitów (zakres: $[-127, 128]$)

Dokładność: $6 - 7$ cyfr znaczących (max: 8388608)

Wyniki obliczeń

[3]:
```
all_to_k = [
    5.5,5.545454502105713,5.590156555175781,5.633285522460938,5.672050476074219,
    5.667526245117188,4.941246032714844,-10.56215667724609,160.5036926269531,
    102.1900253295898,100.1250762939453,100.0073165893555,100.0004272460938,
    100.0000228881836,100,100,100,100,100,100,100,100,100,100,100,100,100,100,
    100,100,100,100,100,100,100
]

print('x_k =', all_to_k[-1])
print('\nAll subsequent numbers:')
print(*(f'x_{i} = {v:.6f}' for i, v in enumerate(all_to_k)), sep='\n')
```

```
x_k = 100

All subsequent numbers:
x_0 = 5.500000
x_1 = 5.545455
x_2 = 5.590157
x_3 = 5.633286
x_4 = 5.672050
x_5 = 5.667526
x_6 = 4.941246
x_7 = -10.562157
x_8 = 160.503693
x_9 = 102.190025
x_10 = 100.125076
x_11 = 100.007317
```

```
x_12 = 100.000427
x_13 = 100.000023
x_14 = 100.000000
x_15 = 100.000000
x_16 = 100.000000
x_17 = 100.000000
x_18 = 100.000000
x_19 = 100.000000
x_20 = 100.000000
x_21 = 100.000000
x_22 = 100.000000
x_23 = 100.000000
x_24 = 100.000000
x_25 = 100.000000
x_26 = 100.000000
x_27 = 100.000000
x_28 = 100.000000
x_29 = 100.000000
x_30 = 100.000000
x_31 = 100.000000
x_32 = 100.000000
x_33 = 100.000000
x_34 = 100.000000
```

Zapisywanie wartości do porównania

[4]:
```
df['C++ float'] = all_to_k
```

Double (8 bajtów) Znak: 1 bit

Mantysa: 52 bity

Wykładnik: 11 bitów (zakres: $[-1023, 1024]$)

Dokładność: $15 - 16$ cyfr znaczących (max: 4503599627370496)

[5]:
```
all_to_k = [
    5.5,5.545454545454546,5.590163934426243,5.633431085044251,5.674648620514802,
    5.713329052462441,5.74912092113604,5.781810945409518,5.81131466923334,
    5.83766396240722,5.861078484508624,5.883542934069212,5.935956716634138,
    6.534421641135182,15.41304318084583,67.47239836474625,97.13715118465481,
    99.82469414672073,99.98953968869486,99.9993761416421,99.99996275956511,
    99.99999777513808,99.99999986698653,99.9999999920431,99.99999999952378,
    99.99999999997149,99.99999999999829,99.9999999999999,100,100,100,100,100,
    100,100
]

print('x_k =', all_to_k[-1])
print('\nAll subsequent numbers:')
```

```
print(*(f'x_{i} = {v:.6f}' for i, v in enumerate(all_to_k)), sep='\n')
```

```
x_k = 100

All subsequent numbers:
x_0 = 5.500000
x_1 = 5.545455
x_2 = 5.590164
x_3 = 5.633431
x_4 = 5.674649
x_5 = 5.713329
x_6 = 5.749121
x_7 = 5.781811
x_8 = 5.811315
x_9 = 5.837664
x_10 = 5.861078
x_11 = 5.883543
x_12 = 5.935957
x_13 = 6.534422
x_14 = 15.413043
x_15 = 67.472398
x_16 = 97.137151
x_17 = 99.824694
x_18 = 99.989540
x_19 = 99.999376
x_20 = 99.999963
x_21 = 99.999998
x_22 = 100.000000
x_23 = 100.000000
x_24 = 100.000000
x_25 = 100.000000
x_26 = 100.000000
x_27 = 100.000000
x_28 = 100.000000
x_29 = 100.000000
x_30 = 100.000000
x_31 = 100.000000
x_32 = 100.000000
x_33 = 100.000000
x_34 = 100.000000
```

Zapisywanie wartości do porównania

[6]:
```
df['C++ double'] = all_to_k
```

Long double (16 bajtów) Znak: 1 bit

Mantysa: 64 bity

Dokładność: 19 – 20 cyfr znaczących (max: 18446744073709551616)

Dokładność powinna być większa niż w przypadku typu `double`, jednakże, mimo zajmowania przez zmienne typu `long double` 16 bajtów pamięci, nie zauważyłem zwiększonej precyzji obliczeń. Wyniki są identyczne do tych, które otrzymałem dla zmiennych typu `double`.

[7]:
```
all_to_k = [
    5.5,5.545454545454545,5.590163934426243,5.633431085044251,5.674648620514802,
    5.713329052462441,5.74912092113604,5.781810945409518,5.81131466923334,
    5.83766396240722,5.861078484508624,5.883542934069212,5.935956716634138,
    6.534421641135182,15.41304318084583,67.47239836474625,97.13715118465481,
    99.82469414672073,99.98953968869486,99.9993761416421,99.99996275956511,
    99.99999777513808,99.99999986698653,99.9999999920431,99.99999999952378,
    99.99999999997149,99.99999999999829,99.9999999999999,100,100,100,100,100,
    100,100
]

print('x_k =', all_to_k[-1])
print('\nAll subsequent numbers:')
print(*(f'x_{i} = {v:.6f}' for i, v in enumerate(all_to_k)), sep='\n')
```

```
x_k = 100

All subsequent numbers:
x_0 = 5.500000
x_1 = 5.545455
x_2 = 5.590164
x_3 = 5.633431
x_4 = 5.674649
x_5 = 5.713329
x_6 = 5.749121
x_7 = 5.781811
x_8 = 5.811315
x_9 = 5.837664
x_10 = 5.861078
x_11 = 5.883543
x_12 = 5.935957
x_13 = 6.534422
x_14 = 15.413043
x_15 = 67.472398
x_16 = 97.137151
x_17 = 99.824694
x_18 = 99.989540
x_19 = 99.999376
x_20 = 99.999963
x_21 = 99.999998
x_22 = 100.000000
x_23 = 100.000000
```

```
x_24 = 100.000000
x_25 = 100.000000
x_26 = 100.000000
x_27 = 100.000000
x_28 = 100.000000
x_29 = 100.000000
x_30 = 100.000000
x_31 = 100.000000
x_32 = 100.000000
x_33 = 100.000000
x_34 = 100.000000
```

Zapisywanie wartości do porównania

[8]:
```
df['C++ long double'] = all_to_k
```

3.3.2 Python

[9]:
```
def calc_next_x(x_prev, x_curr):
    return 111 - (1130 - 3000 / x_prev) / x_curr

def calc_x_k(k, x_0, x_1):
    if k == 0: return x_0
    if k == 1: return x_1

    x_prev = x_0
    x_curr = x_1

    for i in range(2, k + 1):
        x_prev, x_curr = x_curr, calc_next_x(x_prev, x_curr)

    return x_curr

def calc_all_to_k(k, x_0, x_1):
    if k == 0: return [x_0]
    if k == 1: return [x_0, x_1]

    res = [x_0, x_1]

    for i in range(2, k + 1):
        res.append(calc_next_x(res[-2], res[-1]))

    return res
```

Float (8 bajtów) Znak: 1 bit

Mantysa: 52 bity

Wykładnik: 11 bitów (zakres: $[-1022, 1023]$)

Dokładność: $15 - 16$ cyfr znaczących (max: 4503599627370496)

Wyniki obliczeń

[10]:
```
x_0 = 11/2
x_1 = 61/11
k = 34
print('x_k =', calc_x_k(k, x_0, x_1))
print('\nAll subsequent numbers:')
print(*(f'x_{i} = {v:.6f}' for i, v in enumerate(calc_all_to_k(k, x_0, x_1))),␣
 ↪sep='\n')
```

```
x_k = 100.0

All subsequent numbers:
x_0 = 5.500000
x_1 = 5.545455
x_2 = 5.590164
x_3 = 5.633431
x_4 = 5.674649
x_5 = 5.713329
x_6 = 5.749121
x_7 = 5.781811
x_8 = 5.811315
x_9 = 5.837664
x_10 = 5.861078
x_11 = 5.883543
x_12 = 5.935957
x_13 = 6.534422
x_14 = 15.413043
x_15 = 67.472398
x_16 = 97.137151
x_17 = 99.824694
x_18 = 99.989540
x_19 = 99.999376
x_20 = 99.999963
x_21 = 99.999998
x_22 = 100.000000
x_23 = 100.000000
x_24 = 100.000000
x_25 = 100.000000
x_26 = 100.000000
x_27 = 100.000000
x_28 = 100.000000
x_29 = 100.000000
x_30 = 100.000000
x_31 = 100.000000
```

```
x_32 = 100.000000
x_33 = 100.000000
x_34 = 100.000000
```

Zapisywanie wartości do porównania

[11]:
```python
df['Python float'] = calc_all_to_k(k, x_0, x_1)
```

Decimal Dokładność: zależna od ustawień użytkownika

Wiemy, że ciąg jest rosnący, przy czym różnica $x_i - x_{i-1}$ rośnie wraz ze wzrostem wartości i. Możemy więc, mając na uwadze ten fakt, wyznaczyć minimalną liczbę cyfr znaczących, przy której otrzymamy wynik bliski prawidłowemu.

Dekorator pozwalający na ustawienie precyzji

[12]:
```python
def precision(prec=15, rounding=decimal.ROUND_HALF_EVEN):
    def decorator(fn):
        def inner(*args, **kwargs):
            with decimal.localcontext() as ctx:
                ctx.prec = prec
                ctx.rounding = rounding
                return fn(*args, **kwargs)
        return inner
    return decorator
```

Funkcja wyznaczająca minimalną precyzję

[13]:
```python
def is_correct_seq(k):
    x_prev = Decimal(11) / Decimal(2)
    x_curr = Decimal(61) / Decimal(11)

    for i in range(2, k + 1):
        x_next = calc_next_x(x_prev, x_curr)

        if x_next < x_curr or x_next - x_curr > x_curr - x_prev:
            return False

        x_prev = x_curr
        x_curr = x_next

    return True


def find_precision(k, *, start_prec=15):
    prec = start_prec

    while not precision(prec)(is_correct_seq)(k):
        prec += 1
```

```
    return prec
```

Minimalna precyzja

[14]:
```
find_precision(34)
```

[14]:
```
46
```

Wyniki obliczeń

[15]:
```
@precision(46)
def calc():
    x_0 = Decimal(11)/Decimal(2)
    x_1 = Decimal(61)/Decimal(11)
    k = 34

    x_k = calc_x_k(k, x_0, x_1)
    all_to_k = calc_all_to_k(k, x_0, x_1)
    print('x_k =', x_k)
    print(f'x_k = {x_k:.3f} (truncated to 3 decimal numbers)')
    print('\nAll subsequent numbers:')
    print(*(f'x_{i} = {v:.6f}' for i, v in enumerate(all_to_k)), sep='\n')

    return all_to_k
```

[16]:
```
all_to_k = calc()
```

```
x_k = 5.998045651717498537174189164359380351479498 6
x_k = 5.998 (truncated to 3 decimal numbers)

All subsequent numbers:
x_0 = 5.500000
x_1 = 5.545455
x_2 = 5.590164
x_3 = 5.633431
x_4 = 5.674649
x_5 = 5.713329
x_6 = 5.749121
x_7 = 5.781811
x_8 = 5.811314
x_9 = 5.837657
x_10 = 5.860952
x_11 = 5.881377
x_12 = 5.899154
x_13 = 5.914525
x_14 = 5.927741
x_15 = 5.939050
```

```
x_16 = 5.948687
x_17 = 5.956871
x_18 = 5.963799
x_19 = 5.969649
x_20 = 5.974579
x_21 = 5.978726
x_22 = 5.982208
x_23 = 5.985130
x_24 = 5.987577
x_25 = 5.989626
x_26 = 5.991340
x_27 = 5.992773
x_28 = 5.993970
x_29 = 5.994970
x_30 = 5.995805
x_31 = 5.996502
x_32 = 5.997083
x_33 = 5.997572
x_34 = 5.998046
```

Zapisywanie wartości do porównania

[17]:
```
df['Python Decimal'] = [float(v) for v in all_to_k]
```

Fraction Brak utraty dokładności, obliczenia wykonywane są na ułamkach, w których licznik i mianownik są liczbami całkowitymi. Podczas działań, ułamki są sprowadzane do wspólnego mianownika.

Wyniki obliczeń

[18]:
```
def calc():
    x_0 = Fraction('11/2')
    x_1 = Fraction('61/11')
    k = 34

    x_k = calc_x_k(k, x_0, x_1)
    all_to_k = calc_all_to_k(k, x_0, x_1)
    print('x_k =', x_k)
    print(f'x_k = {float(x_k):.3f} (rounded to 3 decimal numbers)')
    print('\nAll subsequent numbers:')
    print(*(f'x_{i} = {float(v):.6f} ({v})' for i, v in enumerate(all_to_k)),␣
↪sep='\n')

    return all_to_k
```

[19]:
```
all_to_k = calc()
```

```
x_k = 172198118279409596138998630 1/28709387656720510591037532 1
x_k = 5.998 (rounded to 3 decimal numbers)
```

```
All subsequent numbers:
x_0 = 5.500000 (11/2)
x_1 = 5.545455 (61/11)
x_2 = 5.590164 (341/61)
x_3 = 5.633431 (1921/341)
x_4 = 5.674649 (10901/1921)
x_5 = 5.713329 (62281/10901)
x_6 = 5.749121 (358061/62281)
x_7 = 5.781811 (2070241/358061)
x_8 = 5.811314 (12030821/2070241)
x_9 = 5.837657 (70231801/12030821)
x_10 = 5.860952 (411625181/70231801)
x_11 = 5.881377 (2420922961/411625181)
x_12 = 5.899154 (14281397141/2420922961)
x_13 = 5.914525 (84467679721/14281397141)
x_14 = 5.927741 (500702562701/84467679721)
x_15 = 5.939050 (2973697798081/500702562701)
x_16 = 5.948687 (17689598897861/2973697798081)
x_17 = 5.956871 (105374653934041/17689598897861)
x_18 = 5.963799 (628433226338621/105374653934041)
x_19 = 5.969649 (3751525871703601/628433226338621)
x_20 = 5.974579 (22413787798580981/3751525871703601)
x_21 = 5.978726 (134005889633282761/22413787798580981)
x_22 = 5.982208 (801651152008680941/134005889633282761)
x_23 = 5.985130 (4797985983097007521/801651152008680941)
x_24 = 5.987577 (28728311253806654501/4797985983097007521)
x_25 = 5.989626 (172071844298962973881/28728311253806654501)
x_26 = 5.991340 (1030940949674393077661/172071844298962973881)
x_27 = 5.992773 (6178195117449434637841/1030940949674393077661)
x_28 = 5.993970 (3703191780171198868642l/6178195117449434637841)
x_29 = 5.994970 (2220052422953488364154 01/3703191780171198868642l)
x_30 = 5.995805 (1331100131197477539976781/222005242295348836415401)
x_31 = 5.996502 (7981944174311787847282561/1331100131197477539976781)
x_32 = 5.997083 (4786838198150534012080474l/7981944174311787847282561)
x_33 = 5.997568 (2870938765672051059103753 21/4786838198150534012080474l)
x_34 = 5.997973 (1721981182794095961389986301/287093876567205105910375321)
```

Zapisywanie wartości do porównania

[20]:
```
df['Python Fraction'] = [float(v) for v in all_to_k]
```

4 Opracowanie wyników

4.1 Porównanie otrzymanych wartości

[21]:
```
df
```

[21]:

	C++ float	C++ double	C++ long double	Python float	Python Decimal	Python Fraction
0	5.500000	5.500000	5.500000	5.500000	5.500000	5.500000
1	5.545455	5.545455	5.545455	5.545455	5.545455	5.545455
2	5.590157	5.590164	5.590164	5.590164	5.590164	5.590164
3	5.633286	5.633431	5.633431	5.633431	5.633431	
4	5.672050	5.674649	5.674649	5.674649	5.674649	
5	5.667526	5.713329	5.713329	5.713329	5.713329	
6	4.941246	5.749121	5.749121	5.749121	5.749121	
7	-10.562157	5.781811	5.781811	5.781811	5.781811	
8	160.503693	5.811315	5.811315	5.811315	5.811314	
9	102.190025	5.837664	5.837664	5.837664	5.837657	
10	100.125076	5.861078	5.861078	5.861078	5.860952	
11	100.007317	5.883543	5.883543	5.883543	5.881377	
12	100.000427	5.935957	5.935957	5.935957	5.899154	
13	100.000023	6.534422	6.534422	6.534422	5.914525	
14	100.000000	15.413043	15.413043	15.413043	5.927741	
15	100.000000	67.472398	67.472398	67.472398	5.939050	
16	100.000000	97.137151	97.137151	97.137151	5.948687	
17	100.000000	99.824694	99.824694	99.824694	5.956871	
18	100.000000	99.989540	99.989540	99.989540	5.963799	
19	100.000000	99.999376	99.999376	99.999376	5.969649	
20	100.000000	99.999963	99.999963	99.999963	5.974579	
21	100.000000	99.999998	99.999998	99.999998	5.978726	
22	100.000000	100.000000	100.000000	100.000000	5.982208	
23	100.000000	100.000000	100.000000	100.000000	5.985130	
24	100.000000	100.000000	100.000000	100.000000	5.987577	
25	100.000000	100.000000	100.000000	100.000000	5.989626	
26	100.000000	100.000000	100.000000	100.000000	5.991340	
27	100.000000	100.000000	100.000000	100.000000	5.992773	
28	100.000000	100.000000	100.000000	100.000000	5.993970	
29	100.000000	100.000000	100.000000	100.000000	5.994970	
30	100.000000	100.000000	100.000000	100.000000	5.995805	
31	100.000000	100.000000	100.000000	100.000000	5.996502	
32	100.000000	100.000000	100.000000	100.000000	5.997083	
33	100.000000	100.000000	100.000000	100.000000	5.997572	
34	100.000000	100.000000	100.000000	100.000000	5.998046	

3	5.633431
4	5.674649
5	5.713329
6	5.749121
7	5.781811
8	5.811314
9	5.837657
10	5.860952
11	5.881377
12	5.899154
13	5.914525
14	5.927741
15	5.939050
16	5.948687
17	5.956871
18	5.963799
19	5.969649
20	5.974579
21	5.978726
22	5.982208
23	5.985130
24	5.987577
25	5.989626
26	5.991340
27	5.992773
28	5.993970
29	5.994970
30	5.995805
31	5.996502
32	5.997083
33	5.997568
34	5.997973

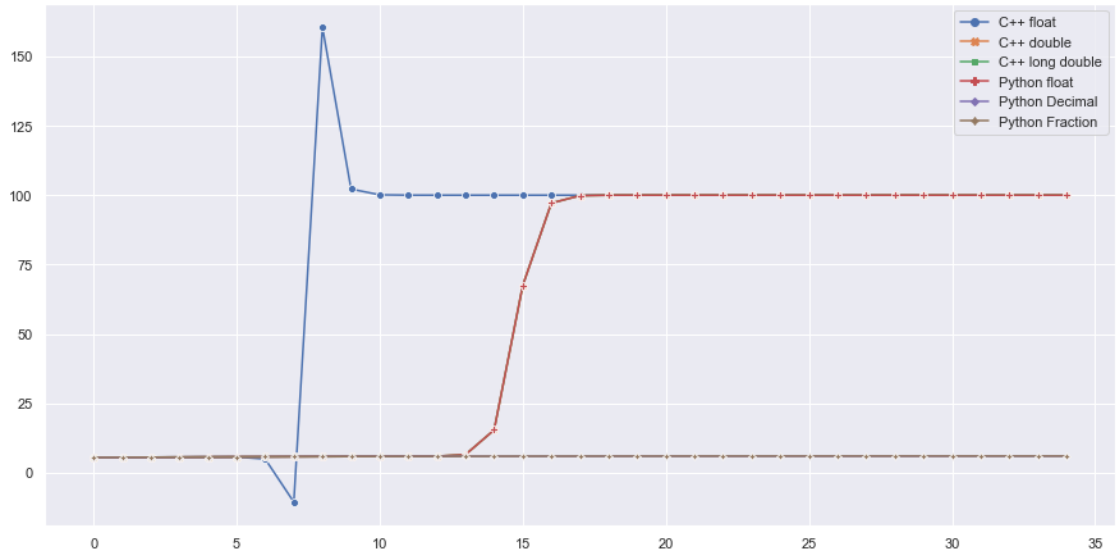
Tabela 1: Porównanie wartości kolejnych wyrazów dla wszystkich precyzji

4.2 Wykresy

[22]:
```
sns.set_theme()
```

4.2.1 Wszystkie wartości

[23]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df, markers=True, dashes=False)
pass
```

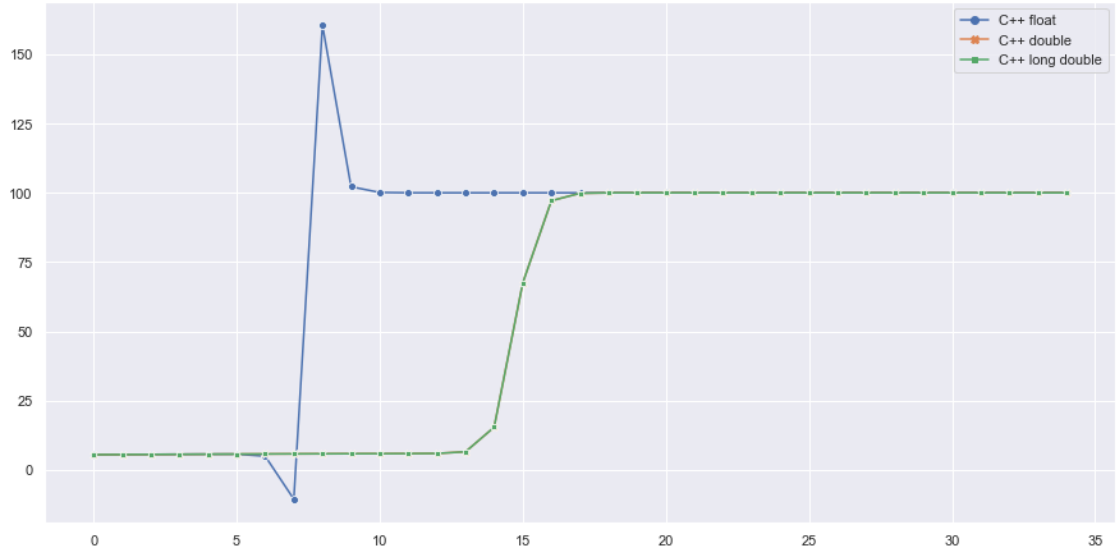


Wykres 1: Porównanie wartości kolejnych wyrazów dla wszystkich precyzji

4.2.2 C++

Wszystkie typy zmiennych

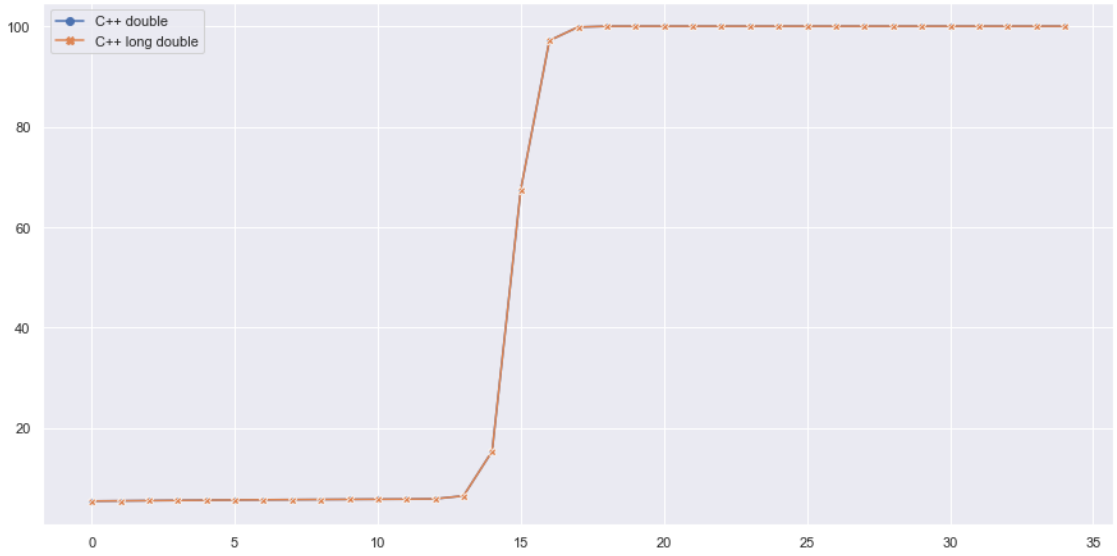
[24]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df[[c for c in df if c.startswith('C++')]], markers=True,␣
↪dashes=False)
pass
```



Wykres 2: Porównanie wszystkich wartości pomiarów wykonanych w języku C++

double i long double

[25]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df[[c for c in df if c.startswith('C++') and 'double' in c]],␣
 ↪markers=True, dashes=False)
pass
```



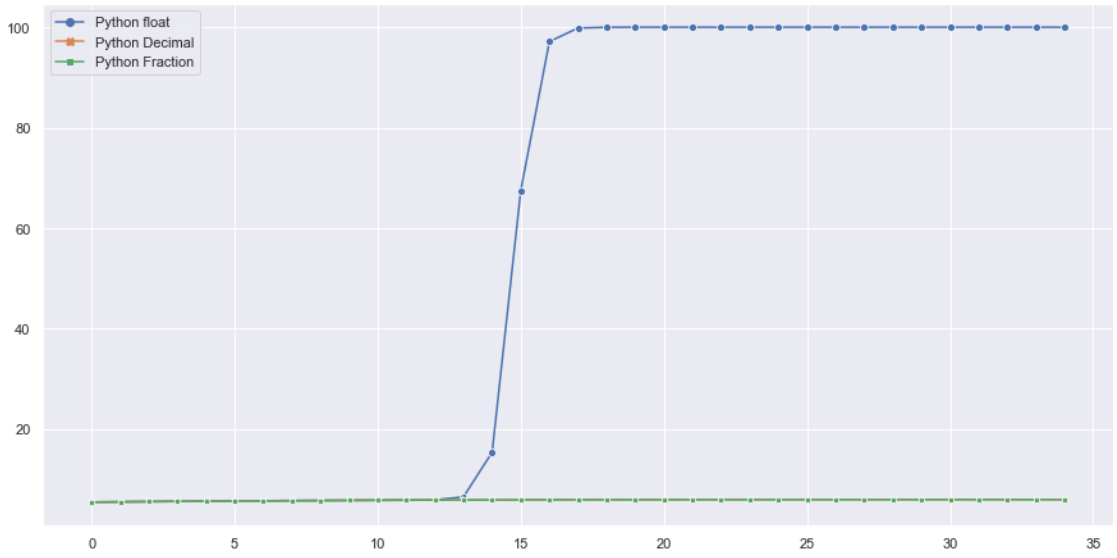
Wykres 3: Porównanie wartości dla zmiennych typu `double` i `long double` w języku C++

Jak widzimy, w przypadku zmienych typu `double` oraz `long double`, otrzymujemy identyczny wykres, ponieważ wartości kolejnych wyrazów ciągu są takie same.

4.2.3 Python

Wszystkie typy zmiennych

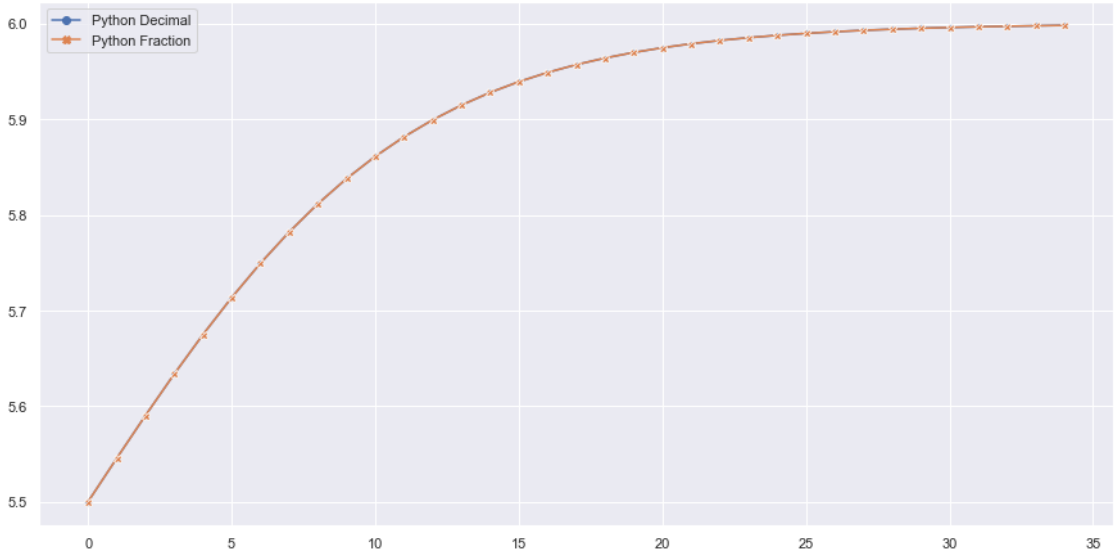
[26]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df[[c for c in df if c.startswith('Python')]], markers=True,␣
 ↪dashes=False)
pass
```



Wykres 4: Porównanie wszystkich wartości pomiarów wykonanych w języku Python

Decimal i Fraction

[27]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df[[c for c in df if c.startswith('Python') and 'float' not␣
 ↪in c]], markers=True, dashes=False)
pass
```



Wykres 5: Porównanie wartości wartości dla zmiennych typu Decimal i Fraction w języku Python

Możemy zaobserwować pokrywające się krzywe. W przypadku obu typów, otrzymane wyniki są bardzo dokładne i bliskie rzeczywistym wartościom.

4.3 Porównanie czasu obliczania wartości ciągu

4.3.1 Python

W celu porównania czasu obliczania wartości ciągu, w zależnosci od zastosowanego typu zmiennych, wykorzystam bibliotekę `timeit`. Aby jak najbardziej rzeczywisty czas wykonania funkcji, wykonam 10000 powtórzeń dla każdego z typów zmiennych i wyznaczę średni czas, jaki jest potrzebny na jedno wywołanie funkcji.

[28]:
```
from timeit import timeit

def measure_time(fn, k, x_0, x_1, *, reps=10_000):
    total = timeit(
        stmt=f'{fn.__name__}(k, x_0, x_1)',
        number=reps,
        globals={
            fn.__name__: fn,
            'k': k,
            'x_0': x_0,
            'x_1': x_1
        }
    )
    total_ms = 1000 * total
    total_ s = 1000 * total_ms
    print(f'Total time:   {total_ms:.6f} ms [10^-3s]')
    print(f'Average time: {total_ s / reps:.6f}  s [10^-6s]')
```

float

[29]:
```
measure_time(calc_x_k, 34, 11/2, 61/11)
```

```
Total time:   81.948400 ms [10^-3s]
Average time: 8.194840  s [10^-6s]
```

Decimal

[30]:
```
with decimal.localcontext() as ctx:
    ctx.prec = 46
    x_0 = Decimal(11) / Decimal(2)
    x_1 = Decimal(61) / Decimal(11)
    measure_time(calc_x_k, 34, x_0, x_1)
```

```
Total time:   321.337400 ms [10^-3s]
Average time: 32.133740  s [10^-6s]
```

Fraction

[31]:
```
measure_time(calc_x_k, 34, Fraction('11/2'), Fraction('61/11'))
```

```
Total time:   2599.239600 ms [10^-3s]
Average time: 259.923960  s [10^-6s]
```

Wnioski Możemy zauważyć, że czas potrzebny na wykonanie obliczeń znacząco wzrasta, gdy używamy typu `Decimal` (około 4-krotny wzrost względem typu `float`). W przypadku, gdy korzystamy z typu `Fraction`, czas się jeszcze bardziej wydłuża, co wynika z faktu, iż obliczenia dokonywane są na ułamkach, gdzie licznik i mianownik są liczbami całkowitymi. W tym przypadku obserwujemy aż 32-krotny wzrost czasu wykonania względem typu `float` i 8-krotny względem typu `Decimal`.

4.3.2 C++

Podobnie jak w przypadku Pythona, w języku C++ również wykonałem 10000 powtórzeń dla każdego z typów zmiennych. Do wyznaczenia czasu potrzebnego na pojedyncze wywołanie funkji, wykorzystałem bibliotekę `chrono`. Kod odpowiedzialny za obliczenia, znajduje się w pliku `main.cpp`.

float

```
Total time: 3.100699999999543 ms [10^-3s]
Average time: 0.3100699999999543  s [10^-6s]
```

double

```
Total time: 3.371799999999435 ms [10^-3s]
Average time: 0.3371799999999435  s [10^-6s]
```

long double

```
Total time: 3.325599999999536 ms [10^-3s]
Average time: 0.3325599999999536  s [10^-6s]
```

Wnioski Jak widać, wyniki się niewiele od siebie różnią. Po kilkukrotnym powtórzeniu testów, nie daje się zauważyć dużej zależnośći między użytym typem zmiennych a czasem potrzebnym na obliczenia.

4.4 Wyznaczanie błędu bezwzględnego i błędu względnego

Wyniki, które otrzymałem, korzystając z klasy `Fraction`, uznaję za wyniki dokładne i do tych wartości będę porównywał wyniki obliczeń wykonanych dla pozostałych typów zmiennych.

4.4.1 Błąd bezwzględny

Błąd bezwzględny obliczam, korzystając ze wzoru:

$|x_{i(z)} - x_i|$

gdzie: - $x_{i(z)}$ - wartość zmierzona i. wyrazu ciągu, - x_i - wartość rzeczywista i. wyrazu ciągu

Wartości błędów

[32]:
```
df_abs_err = df.sub(df['Python Fraction'], axis='rows').abs().
 ↪drop(columns='Python Fraction')
df_abs_err
```

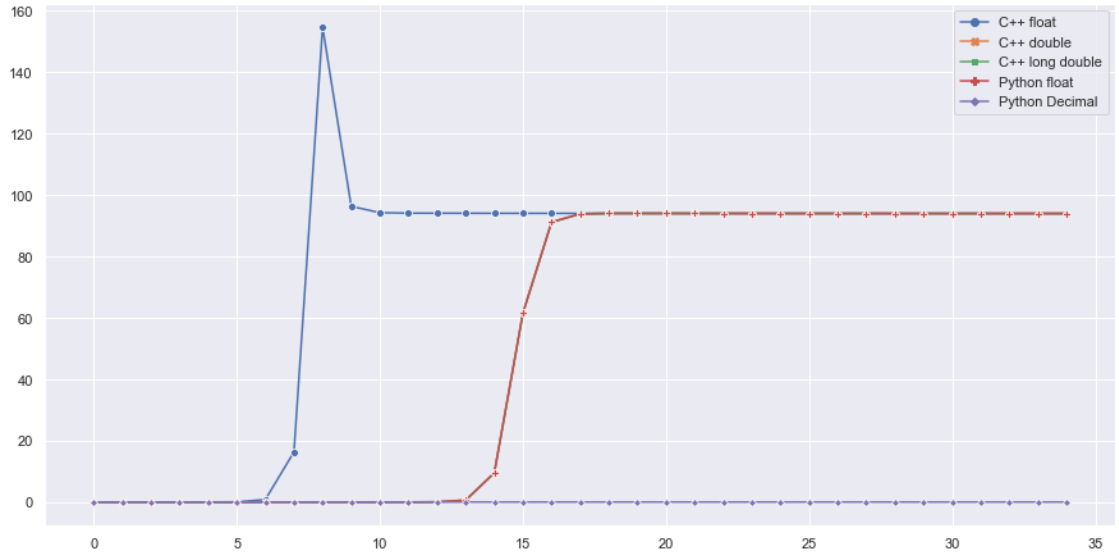
[32]:

	C++ float	C++ double	C++ long double	Python float	Python Decimal
0	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00
1	4.334883e-08	0.000000e+00	8.881784e-16	0.000000e+00	0.000000e+00
2	7.379250e-06	1.332268e-14	1.332268e-14	1.421085e-14	0.000000e+00
3	1.455626e-04	2.620126e-13	2.620126e-13	2.620126e-13	0.000000e+00
4	2.598144e-03	4.650502e-12	4.650502e-12	4.650502e-12	0.000000e+00
5	4.580281e-02	8.192558e-11	8.192558e-11	8.192558e-11	0.000000e+00
6	8.078749e-01	1.433402e-09	1.433402e-09	1.433402e-09	0.000000e+00
7	1.634397e+01	2.492390e-08	2.492390e-08	2.492390e-08	0.000000e+00
8	1.546924e+02	4.309393e-07	4.309393e-07	4.309393e-07	0.000000e+00
9	9.635237e+01	7.413449e-06	7.413449e-06	7.413449e-06	0.000000e+00
10	9.426412e+01	1.269620e-04	1.269620e-04	1.269620e-04	0.000000e+00
11	9.412594e+01	2.165718e-03	2.165718e-03	2.165718e-03	0.000000e+00
12	9.410127e+01	3.680281e-02	3.680281e-02	3.680281e-02	0.000000e+00
13	9.408550e+01	6.198967e-01	6.198967e-01	6.198967e-01	0.000000e+00
14	9.407226e+01	9.485302e+00	9.485302e+00	9.485302e+00	0.000000e+00
15	9.406095e+01	6.153335e+01	6.153335e+01	6.153335e+01	0.000000e+00
16	9.405131e+01	9.118846e+01	9.118846e+01	9.118846e+01	0.000000e+00
17	9.404313e+01	9.386782e+01	9.386782e+01	9.386782e+01	0.000000e+00
18	9.403620e+01	9.402574e+01	9.402574e+01	9.402574e+01	0.000000e+00
19	9.403035e+01	9.402973e+01	9.402973e+01	9.402973e+01	0.000000e+00
20	9.402542e+01	9.402538e+01	9.402538e+01	9.402538e+01	0.000000e+00
21	9.402127e+01	9.402127e+01	9.402127e+01	9.402127e+01	0.000000e+00
22	9.401779e+01	9.401779e+01	9.401779e+01	9.401779e+01	0.000000e+00
23	9.401487e+01	9.401487e+01	9.401487e+01	9.401487e+01	0.000000e+00
24	9.401242e+01	9.401242e+01	9.401242e+01	9.401242e+01	0.000000e+00
25	9.401037e+01	9.401037e+01	9.401037e+01	9.401037e+01	8.881784e-16
26	9.400866e+01	9.400866e+01	9.400866e+01	9.400866e+01	1.243450e-14
27	9.400723e+01	9.400723e+01	9.400723e+01	9.400723e+01	2.033929e-13
28	9.400603e+01	9.400603e+01	9.400603e+01	9.400603e+01	3.398171e-12
29	9.400503e+01	9.400503e+01	9.400503e+01	9.400503e+01	5.669598e-11
30	9.400420e+01	9.400420e+01	9.400420e+01	9.400420e+01	9.457288e-10
31	9.400350e+01	9.400350e+01	9.400350e+01	9.400350e+01	1.577307e-08
32	9.400292e+01	9.400292e+01	9.400292e+01	9.400292e+01	2.630362e-07
33	9.400243e+01	9.400243e+01	9.400243e+01	9.400243e+01	4.386046e-06
34	9.400203e+01	9.400203e+01	9.400203e+01	9.400203e+01	7.313003e-05

Tabela 2: Błąd bezwzględny obliczonych wartości

Wykres

[33]:
```
plt.figure(figsize=(15, 7.5))
sns.lineplot(data=df_abs_err, markers=True, dashes=False)
pass
```



Wykres 6: Porównanie błędów bezwzględnych dla wszystkich wartości

4.4.2 Błąd względny

Błąd względny obliczam, korzystając ze wzoru:

$\frac{|x_{i(z)} - x_i|}{x_i}$

gdzie: - $x_{i(z)}$ - wartość zmierzona i. wyrazu ciągu, - x_i - wartość rzeczywista i. wyrazu ciągu

Wartości błędów

[34]:
```
df_rel_err = df_abs_err.divide(df['Python Fraction'], axis='rows')
(df_rel_err * 100).applymap('{:.2f}%'.format)
```

[34]:

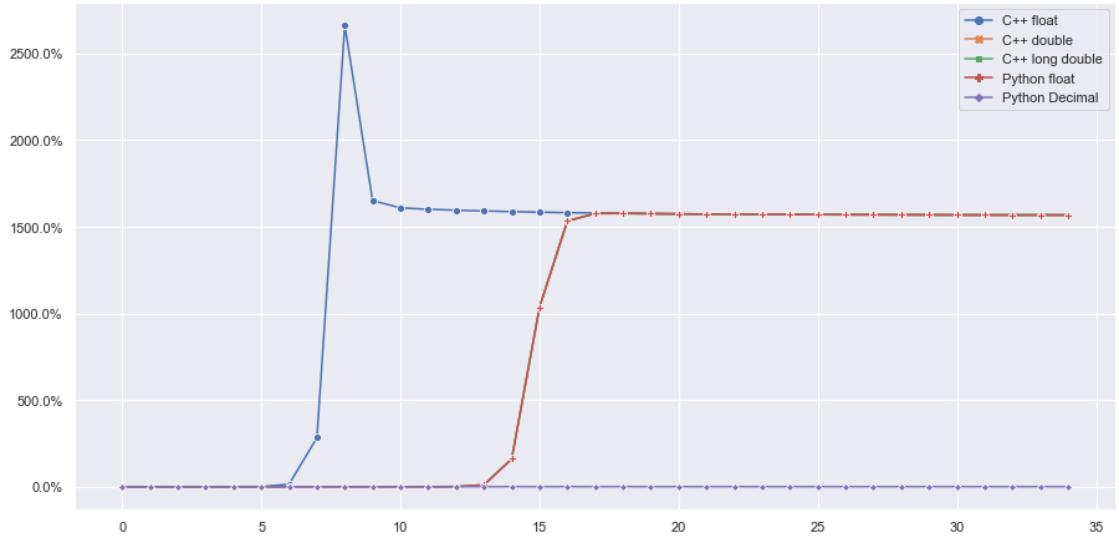
	C++ float	C++ double	C++ long double	Python float	Python Decimal
0	0.00%	0.00%	0.00%	0.00%	0.00%
1	0.00%	0.00%	0.00%	0.00%	0.00%
2	0.00%	0.00%	0.00%	0.00%	0.00%
3	0.00%	0.00%	0.00%	0.00%	0.00%
4	0.05%	0.00%	0.00%	0.00%	0.00%
5	0.80%	0.00%	0.00%	0.00%	0.00%
6	14.05%	0.00%	0.00%	0.00%	0.00%
7	282.68%	0.00%	0.00%	0.00%	0.00%
8	2661.92%	0.00%	0.00%	0.00%	0.00%
9	1650.53%	0.00%	0.00%	0.00%	0.00%
10	1608.34%	0.00%	0.00%	0.00%	0.00%
11	1600.41%	0.04%	0.04%	0.04%	0.00%
12	1595.17%	0.62%	0.62%	0.62%	0.00%
13	1590.75%	10.48%	10.48%	10.48%	0.00%

14	1586.98%	160.02%	160.02%	160.02%	0.00%
15	1583.77%	1036.08%	1036.08%	1036.08%	0.00%
16	1581.04%	1532.92%	1532.92%	1532.92%	0.00%
17	1578.73%	1575.79%	1575.79%	1575.79%	0.00%
18	1576.78%	1576.61%	1576.61%	1576.61%	0.00%
19	1575.14%	1575.13%	1575.13%	1575.13%	0.00%
20	1573.76%	1573.76%	1573.76%	1573.76%	0.00%
21	1572.60%	1572.60%	1572.60%	1572.60%	0.00%
22	1571.62%	1571.62%	1571.62%	1571.62%	0.00%
23	1570.81%	1570.81%	1570.81%	1570.81%	0.00%
24	1570.12%	1570.12%	1570.12%	1570.12%	0.00%
25	1569.55%	1569.55%	1569.55%	1569.55%	0.00%
26	1569.08%	1569.08%	1569.08%	1569.08%	0.00%
27	1568.68%	1568.68%	1568.68%	1568.68%	0.00%
28	1568.34%	1568.34%	1568.34%	1568.34%	0.00%
29	1568.07%	1568.07%	1568.07%	1568.07%	0.00%
30	1567.83%	1567.83%	1567.83%	1567.83%	0.00%
31	1567.64%	1567.64%	1567.64%	1567.64%	0.00%
32	1567.48%	1567.48%	1567.48%	1567.48%	0.00%
33	1567.34%	1567.34%	1567.34%	1567.34%	0.00%
34	1567.23%	1567.23%	1567.23%	1567.23%	0.00%

Tabela 3: Błąd względny obliczonych wartości

Wykres

[35]:
```
plt.figure(figsize=(15, 7.5))
g = sns.lineplot(data=(df_rel_err * 100), markers=True, dashes=False)
ticks_loc = g.get_yticks().tolist()
g.yaxis.set_major_locator(mticker.FixedLocator(ticks_loc))
g.set_yticklabels([f'{v}%' for v in g.get_yticks()])
pass
```



Wykres 7: Porównanie błędów względnych obliczonych wartości

5 Końcowe spostrzeżenia

5.1 Zbieżność ciągu

5.1.1 Dla $x_0 = 11/2$, $x_1 = 61/11$

Możemy pokazać, że ciąg jest zbieżny do 6, korzystając z obliczeń na typie `Fraction`, pozwalającym na obliczenia bez utraty precyzji. Aby wyniki obliczeń były bardziej czytelne, uzyskane ułamki skonwertuję na liczby zmiennoprzecinkowe typu `float` (w Pythonie jest to odpowiednik typu `double` z języka C++). Oczywiście nie jesteśmy w stanie obliczyć prawdziwej granicy ciągu, ponieważ konieczne jest zakończenie obliczeń na pewnym wyrazie ciągu, dlatego finalny wynik będzie również obarczony **błędem obcięcia**, ale błąd ten jest na tyle mały, że nie ma on dużego wpływu na wynik. Również konwersja do stratnego typu `float` wiąże się z wystąpieniem **błędu zaokrąglenia**, ale taka konwersja jest konieczna, żeby wynik był bardziej czytelny.

[36]:
```
x_arr = calc_all_to_k(10_000, Fraction('11/2'), Fraction('61/11'))
float(x_arr[-1])
```

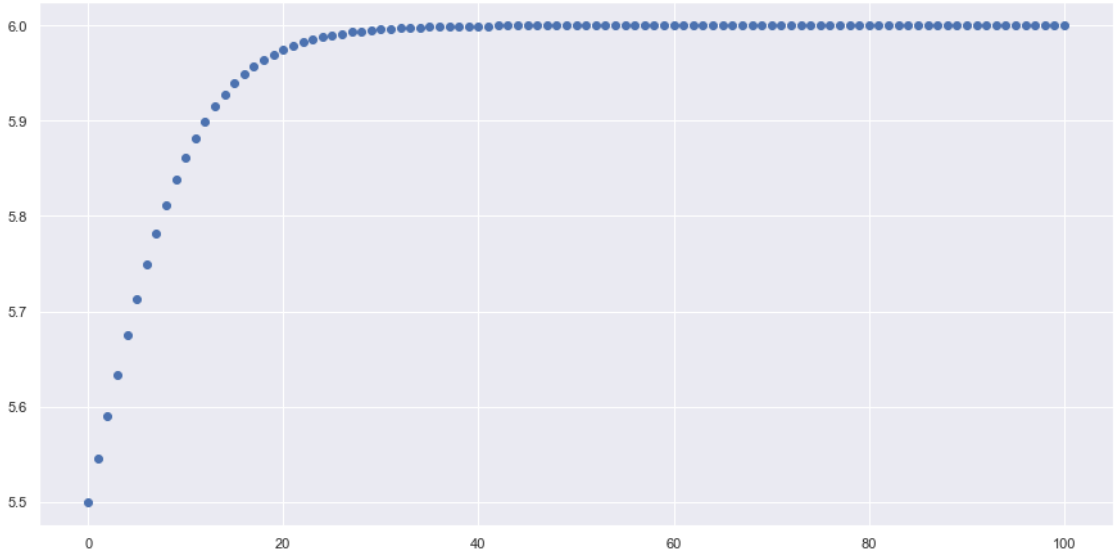
[36]:
```
6.0
```

Wykresy

[37]:
```
def show_seq_graph(k, x_0=Fraction('11/2'), x_1=Fraction('61/11')):
    x_arr = calc_all_to_k(k, x_0, x_1)
    plt.figure(figsize=(15, 7.5))
    plt.scatter(list(range(len(x_arr))), [float(v) for v in x_arr])
    pass
```

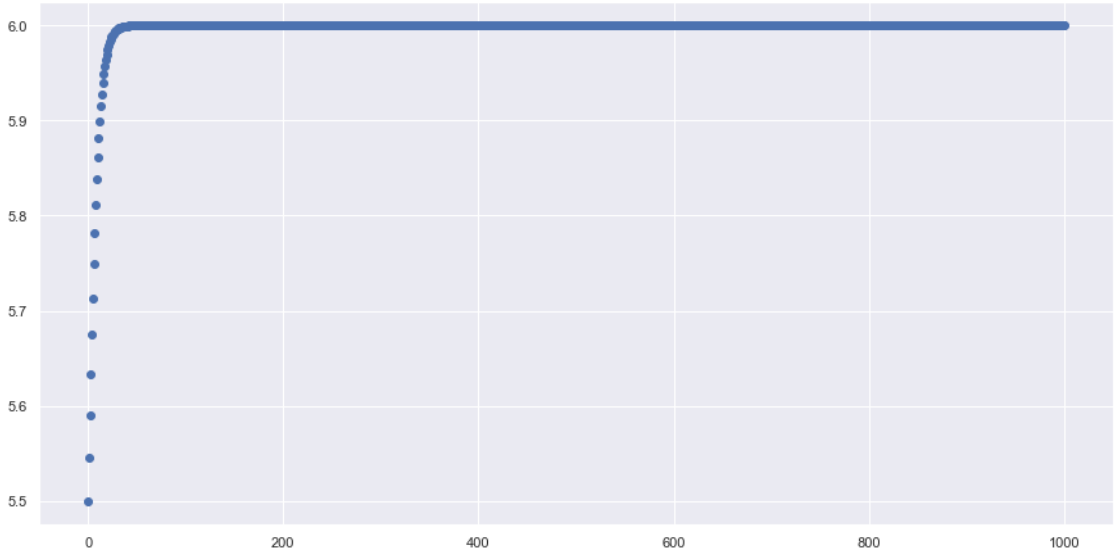
100 początkowych wyrazów

[38]:
```
show_seq_graph(100)
```



1000 początkowych wyrazów

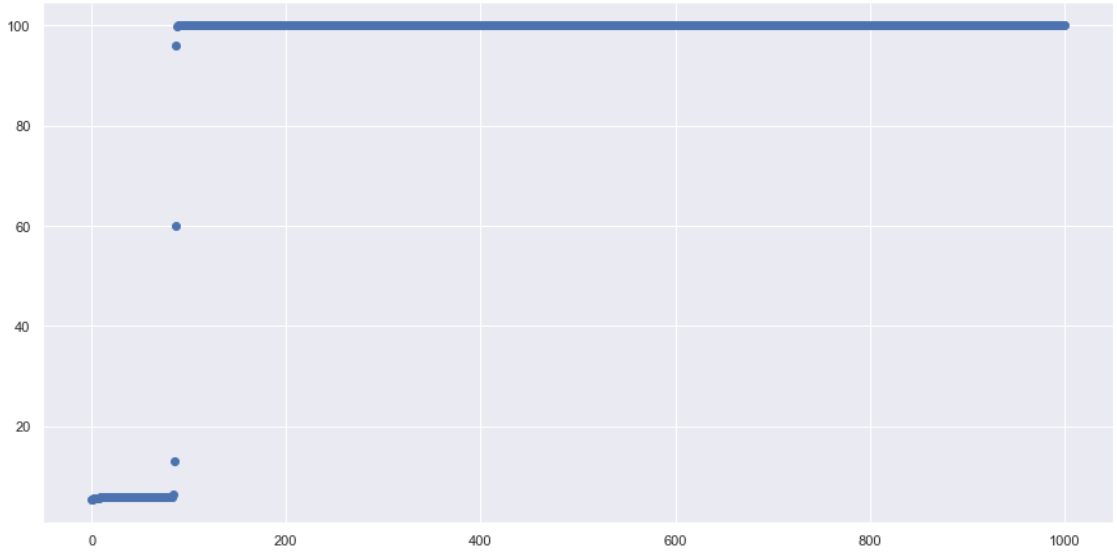
[39]:
```
show_seq_graph(1000)
```



5.1.2 Dla innych wyrazów początkowych

Możemy zauważyć, że nawet nieznaczna niedokładność danych wejściowych sprawia, że ciąg jest zbieżny do 100. Zbieżność do 100 uzyskałem dla wszystkich wyrazów początkowych różnych od tych, które zostały podane w treści zadania.

[44]:
```
n = 100
show_seq_graph(1000, x_1=Fraction(f'61{"0"*n}1/11{"0"*n}0'))
```



6 Wnioski

Przeprowadzona analiza pokazuje, że dokładność, z jaką reprezentowane są liczby zmiennoprzecinkowe, ma znaczący wpływ na dokładność wyników obliczeń. W celu uzyskania dokładnych wyników, potrzebna była precyzja równa przynajmniej 46 cyfr znaczących, dlatego wyniki, jakie otrzymałem dla zmiennych typu `float`, `double` oraz `long double` znacząco odbiegały od rzeczywistych wartości wyrazów ciągu.

Mogliśmy również zaobserwować, że typ `long double` niekoniecznie gwarantuje zwiększenie precyzji obliczeń względem typu `double`. Wynika to stąd, że precyzja dla typu `long double` musi być przynajmniej taka jak dla `double`, ale nie jest konieczne, żeby była ona większa. Program testowałem na komputerze z systemem operacyjnym Windows 10, a także na wirtualnej maszynie z systemem operacyjnym Ubuntu 20.04. W obu przypadkach otrzymałem takie same wyniki dla zmiennych typu `double` oraz `long double`.

Na niedokładność wyników obliczeń znaczący wpływ miał **błąd zaokrąglenia**.

Już przed rozpoczęciem obliczeń wyraz ciągu x_1 nie jest reprezentowany precyzyjnie, ponieważ jego wartość to ułamek $\frac{61}{11}$ o nieskończonym rozwinięciu okresowym $5.(54)$. W komputerze nie jest możliwe reprezentowanie nieskończonych liczb, więc zachowane zostaje jedynie kilka początkowych cyfr tego rozwinięcia, Z błędem zaokrąglenia mamy także do czynienia podczas obliczania każdego kolejnego wyrazu ciągu, ponieważ otrzymywane wartości muszą być zaokrąglane do ograniczonej liczby cyfr znaczących.