1 Machine precision

Exercise 1.1. Machine epsilon or machine precision is an upper bound on the relative approximation error due to rounding in floating point arithmetic. Execute the following code

```
import sys
help(sys.float_info)
print(sys.float_info)
```

- understand the meaning of `max`, `max_exp` and `max_10_exp`.
- Write a code to compute the machine precision ϵ in (float) default precision with a `while` construct. Compute also the mantissa digits number.
- Use NumPy and exploit the functions `float16` and `float31` in the while statement and see the differences. Check the result of `np.finfo(float).eps`.

Da help(sys.float_info):

```
 |  max
 |      DBL_MAX -- maximum representable finite float
 |
 |  max_10_exp
 |      DBL_MAX_10_EXP -- maximum int e such that 10**e is representable
 |
 |  max_exp
 |      DBL_MAX_EXP -- maximum int e such that radix**(e-1) is representable
```

- **max**: Rappresenta il valore massimo che può essere rappresentato da un numero in virgola mobile sulla piattaforma corrente (approssimativamente 1.7976931348623157×10308$1.7976931348623157\times10^{308}$).
- **max_exp**: Indica il massimo esponente per le rappresentazioni in virgola mobile su questa piattaforma (1024).
- **max_10_exp**: Indica il massimo esponente base 10 per le rappresentazioni in virgola mobile su questa piattaforma (308).

```
print(sys.float_info)
```

sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)

```
mantiissa = 1   #t
eps = 1
B=2 #base
while 1+eps>1:
   eps/=B
   mantissa += 1
print("eps:", eps, "\nmantissa (t):", mantissa-1)
```

Output:

eps: 2.220446049250313e-16
mantissa (t): 52

```
import numpy as np
mantissa = 1   #t
eps = np.float16(1)
B=np.float16(2) #base

while np.float16(1)+eps/B>np.float16(1):
   eps = eps/B
   mantissa += 1
print("eps:", eps,"\nmantissa (t):", mantissa-1)
```

Output:

```
eps: 0.000977
mantissa (t): 10
```

```
import numpy as np
mantissa = 1   #t
eps = np.float32(1)

B=np.float32(2) #base

while np.float32(1)+eps/B>np.float32(1):
   eps = eps/B
   mantissa += 1
print("eps:", eps,"\nmantissa (t):", mantissa-1)
```

Output:

```
eps: 1.1920929e-07
mantissa (t): 23
```

```
print("Mantissa per float16:", np.finfo(np.float16).nmant)
print("Mantissa per float32:", np.finfo(np.float32).nmant)
print("Mantissa per float64:", np.finfo(np.float64).nmant)

print("Epsilon di macchina per float16:", np.finfo(np.float16).eps)
print("Epsilon di macchina per float32:", np.finfo(np.float32).eps)
print("Epsilon di macchina per float64:", np.finfo(np.float64).eps)
```

Output:

```
Mantissa per float16: 10
Mantissa per float32: 23
Mantissa per float64: 52
Epsilon di macchina per float16: 0.000977
Epsilon di macchina per float32: 1.1920929e-07
Epsilon di macchina per float64: 2.220446049250313e-16
```

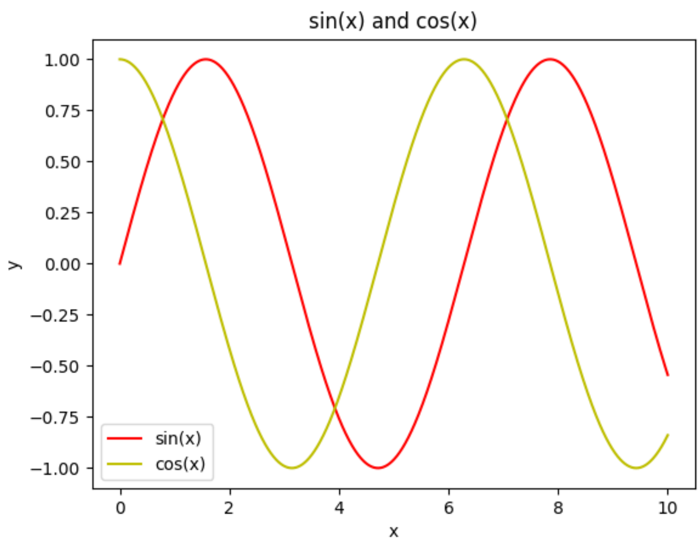
2 Plot of a function

Exercise 2.1. Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. Create a figure combining together the cosine and sine curves, on the domain $[0, 10]$:

- add a legend
- add a title
- change the default colors

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-0, 10, 1000)
y1= np.sin(x)
y2= np.cos(x)

plt.plot(x, y1, 'r')
plt.plot(x, y2, 'y')

plt.xlabel("x")
plt.ylabel("y")
plt.legend(['sin(x)', 'cos(x)'])
plt.title("sin(x) and cos(x)")

plt.show()
```

Output:



Exercise 2.2. The Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones and it is formally defined as:

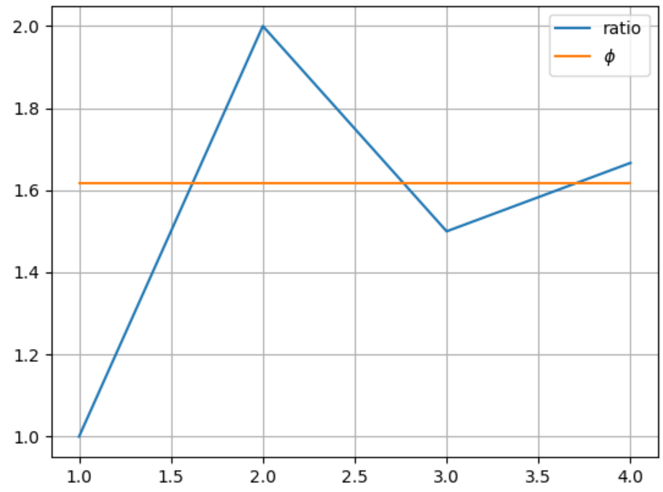
$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \qquad n > 2 \end{cases}$$

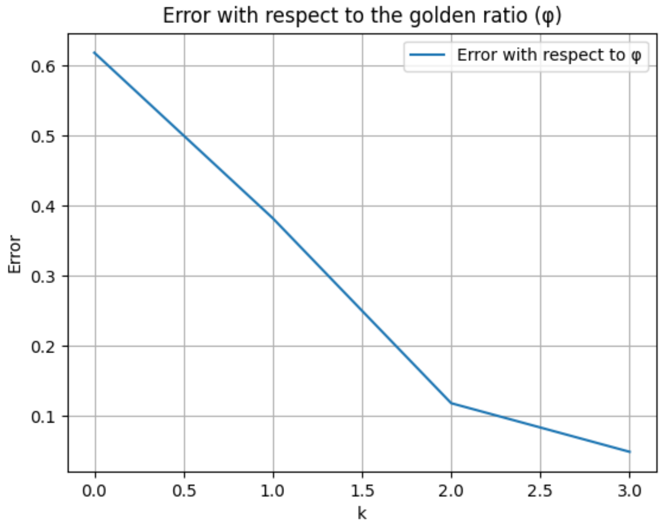
- Write a script that, given an input number n, computes the number F_n of the Fibonacci sequence.
- Write a code computing, for a natural number k, the ratio $r_k = \frac{F_{k+1}}{F_k}$, where F_k are the Fibonacci numbers.
- Verify that, for a large k, $\{r_k\}_k$ converges to the value $\varphi = \frac{1+\sqrt{5}}{2}$
- Create a plot of the error (with respect to φ)

```
def fibonacci(n):
    if(n == 0 or n == 1):
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1
        for _ in range(2, n + 1):
            a, b = b, a + b
        return b

def fibonacci_ratio(k):
    ratios = []
    for i in range(1, k):
        ratio = fibonacci(i + 1) / fibonacci(i)
        ratios.append(ratio)
    return ratios

k_values = 10
ratios = fibonacci_ratio(k_values)
phi = (1 + np.sqrt(5)) / 2
errors = [abs(phi - ratio) for ratio in ratios]

x = np.arange(1, k_values)
plt.plot(x, ratios)
plt.plot(x, phi*np.ones(np.shape(x)))
plt.grid(True)
plt.legend(['ratio', r'$\phi$'])
plt.show()

plt.plot(errors, label='Error with respect to φ')
plt.xlabel('k')
plt.ylabel('Error')
plt.title('Error with respect to the golden ratio (φ)')
plt.legend()
plt.grid(True)
plt.show()
```

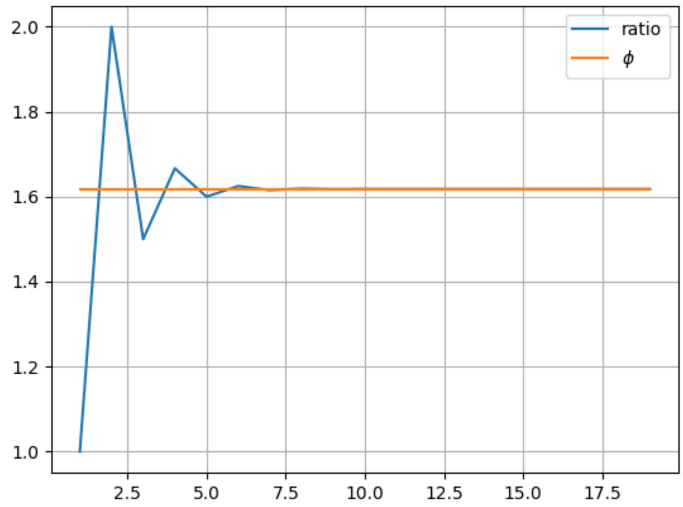
NOTA: Per questioni computazionali, per gli ultimi due casi ho usato un algoritmo che calcola Fibonacci iterativamente e non in maniera ricorsiva; tuttavia, la differenza risiede solo nelle tempistiche per il calcolo e non nel risultato.

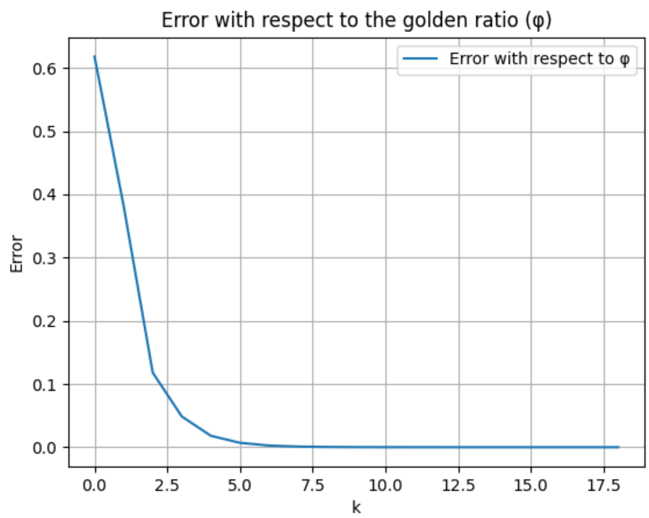
Output:
Per k = 5





Per k = 20





Per k = 200