LAB1 - ANALIZA BŁĘDÓW

Gosztyła Mikołaj, Smółka Antoni

Zadanie 1.

Oblicz przybliżoną wartość pochodnej funkcji, używając wzoru

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Sprawdź działanie programu dla funkcji $\tan(x)$ oraz $x = 1$. Wyznacz błąd, porównując otrzymaną wartość numerycznej pochodnej z prawdziwą wartością. Pomocna będzie tożsamość $\tan'(x) = 1 + \tan^2(x)$. Na wspólnym rysunku przedstaw wykresy wartości bezwzględnej błędu metody, błędu numerycznego oraz błędu obliczeniowego w zależności od h dla $h = 10^{-k}$, $k = 0, \ldots, 16$. Użyj skali logarytmicznej na obu osiach. Czy wykres wartości bezwzględnej błędu obliczeniowego posiada minimum? Porównaj wyznaczoną wartość $h_{\min}$ z wartością otrzymaną ze wzoru

$$h_{\min} \approx 2\sqrt{\frac{\varepsilon_{\text{mach}}}{M}}, \quad \text{gdzie} \quad M \approx |f''(x)|.$$

Powtórz ćwiczenie używając wzoru różnic centralnych

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

Porównaj wyznaczoną wartość $h_{\min}$ z wartością otrzymaną ze wzoru

$$h_{\min} \approx \sqrt[3]{\frac{3\varepsilon_{\text{mach}}}{M}}, \quad \text{gdzie} \quad M \approx |f'''(x)|.$$

In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize_scalar
```

In [2]:
```
def tan_f(x=1):
    return np.tan(x)

def tan_df2(x):
    return 2 * np.tan(x) / (np.cos(x) ** 2)

def tan_df3(x):
    return (4 * np.sin(x) ** 2 + 2) / np.cos(x) ** 4
```

```python
def d_f(f, h, x=1):
    return (f(x + h) - f(x)) / h

def d_f_central(f, h, x):
    return (f(x + h) - f(x - h))/(2 * h)

def real_d_f(f, x=1):
    return 1 + f(x) ** 2

def find_maximum(f, domain):
    res = minimize_scalar(lambda x: -f(x), bounds=domain, method='bounded')
    max_value = -res.fun
    return max_value
```

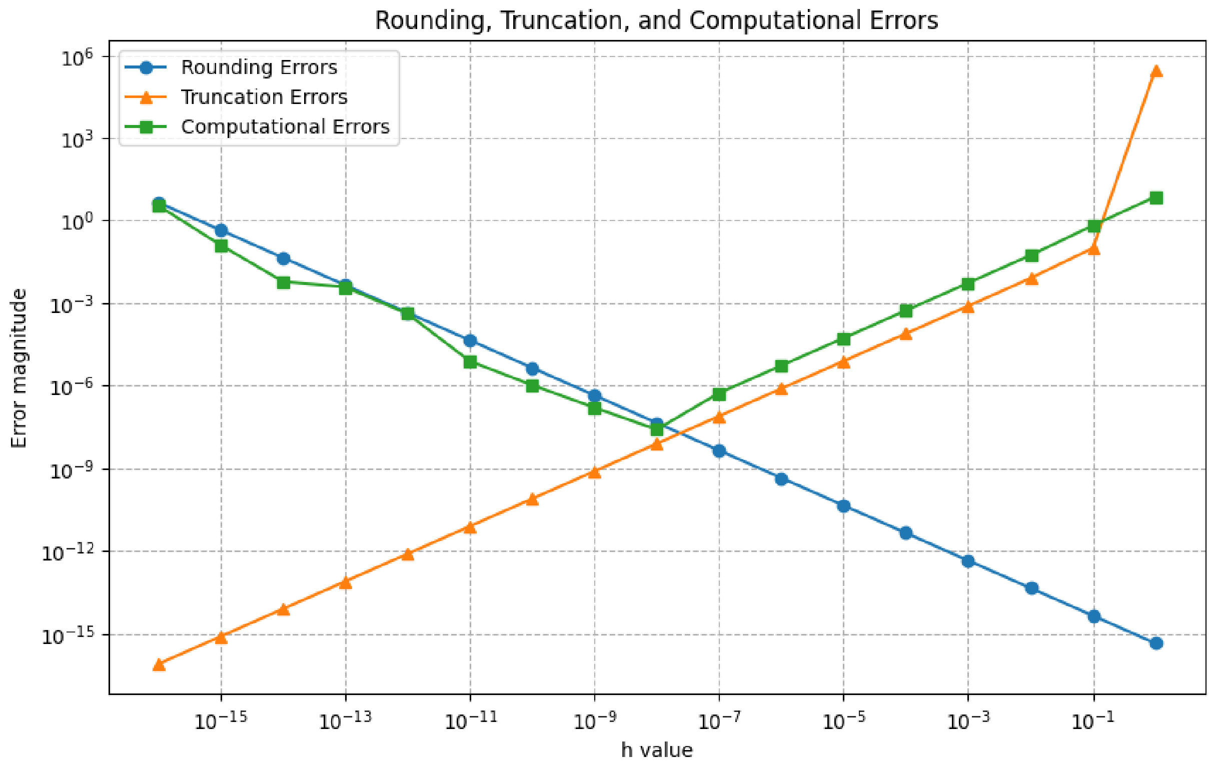
Idea jest taka, że zgodnie ze wzorem

$$|f(t)| \leq M \text{ for all } t \in [x-h, x+h]$$

przyjmujemy, że M jest równe maksimum badanej funkcji

Metoda Progresywna

In [3]:
```python
h_values = np.logspace(0, -16, num=17, base=10)
rounding_errors = []
truncation_errors = []
computational_errors = []
x0 = 1

for h in h_values:
    rounding_errors.append(2 * np.finfo(float).eps / h)
    truncation_errors.append(find_maximum(np.tan, (x0 - h, x0 + h)) * h / 2)
    computational_errors.append(abs(real_d_f(np.tan) - d_f(np.tan, h)))

plt.figure(figsize=(10, 6))
plt.loglog(h_values, rounding_errors, label='Rounding Errors', marker='o')
plt.loglog(h_values, truncation_errors, label='Truncation Errors', marker='^
plt.loglog(h_values, computational_errors, label='Computational Errors', mar
plt.xlabel('h value')
plt.ylabel('Error magnitude')
plt.title('Rounding, Truncation, and Computational Errors')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```



Zmiana wynika z tego że funkcja tangens w $\frac{\pi}{2}$ zmierza do nieskończoności

Porównujemy wyznaczoną wartość $h_{\min}$ z wartością otrzymaną ze wzoru

$$h_{\min} \approx 2\sqrt{\frac{\varepsilon_{\text{mach}}}{M}}, \quad \text{gdzie} \quad M \approx |f''(x)|.$$

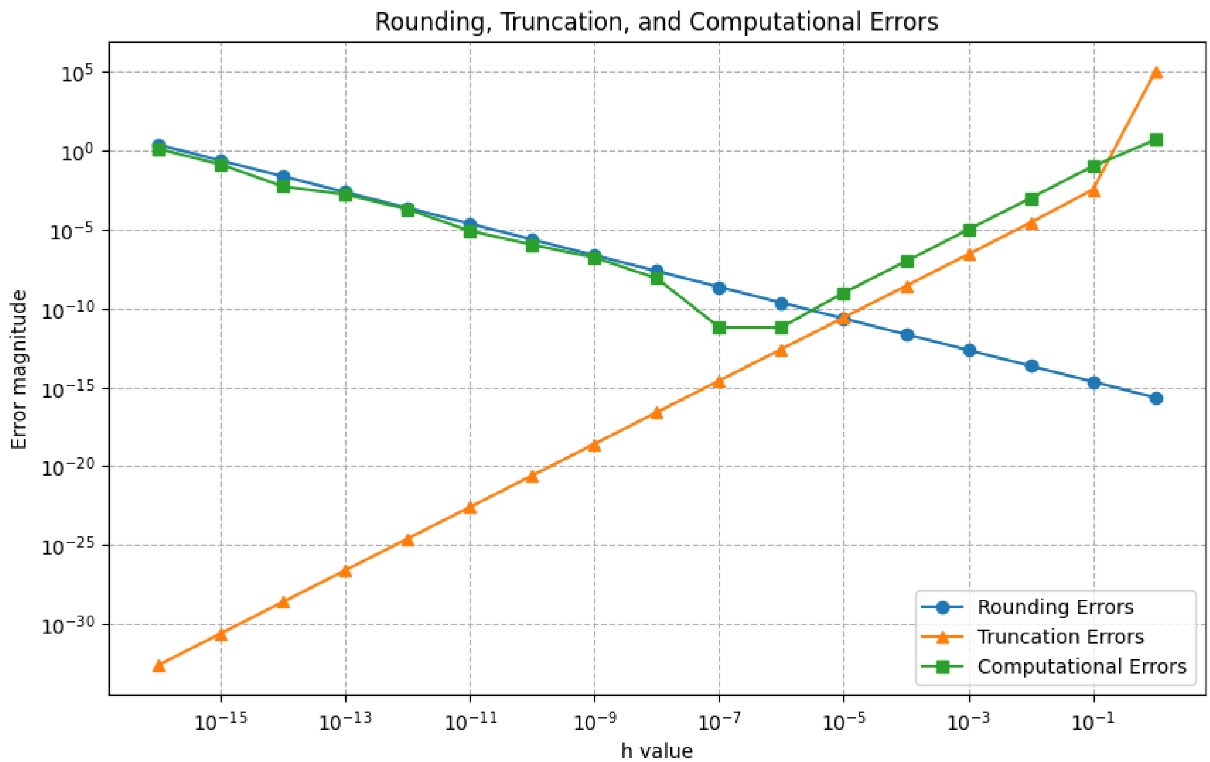
In [4]:
```
h_min = 2 * math.sqrt(np.finfo(float).eps / abs(tan_df2(x0)))
print(abs(h_min - h_values[8]))
print(h_min)
print(h_values[8])
```

```
8.763047748195454e-10
9.123695225180455e-09
1e-08
```

Metoda Centralna

In [5]:
```
rounding_errors = []
truncation_errors = []
computational_errors_2 = []
x0 = 1

for h in h_values:
    rounding_errors.append(np.finfo(float).eps / h)
    truncation_errors.append(find_maximum(np.tan, (x0 - h, x0 + h)) * h * h
    computational_errors_2.append(abs(real_d_f(np.tan) - d_f_central(np.tan,

plt.figure(figsize=(10, 6))
plt.loglog(h_values, rounding_errors, label='Rounding Errors', marker='o')
```

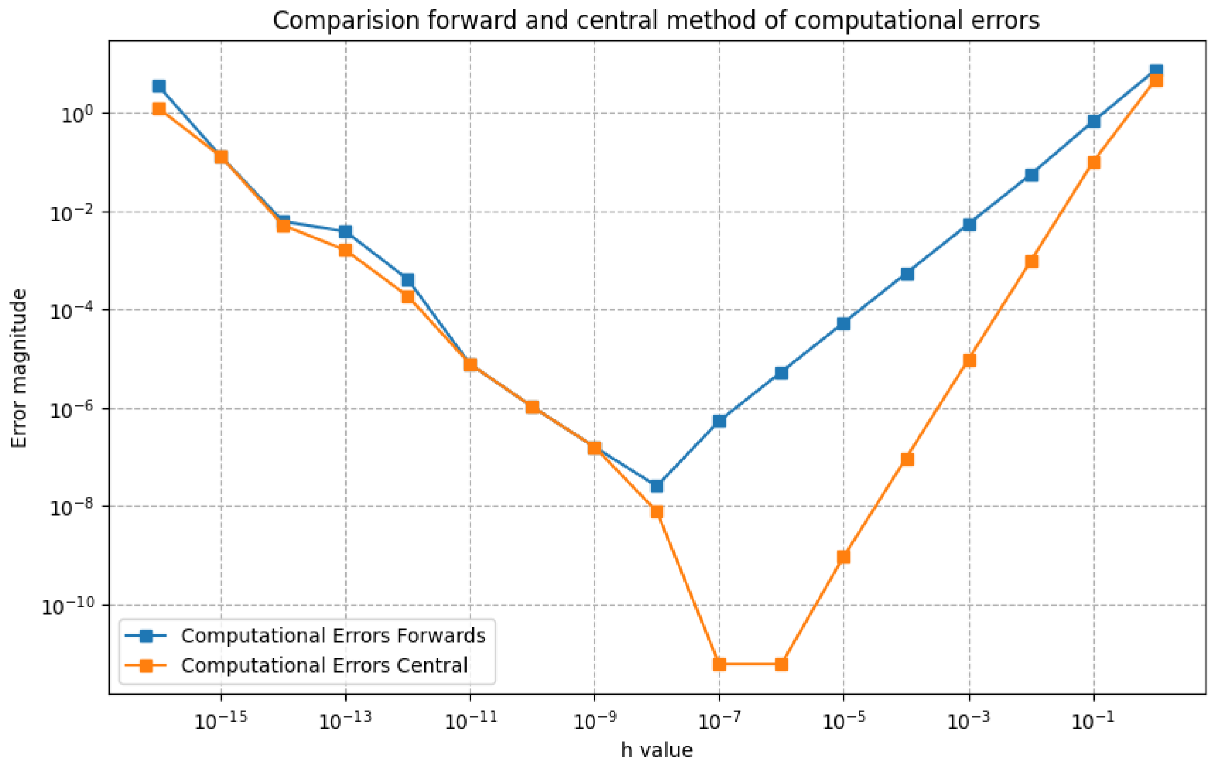
```
plt.loglog(h_values, truncation_errors, label='Truncation Errors', marker='^
plt.loglog(h_values, computational_errors_2, label='Computational Errors', m
plt.xlabel('h value')
plt.ylabel('Error magnitude')
plt.title('Rounding, Truncation, and Computational Errors')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```



Porównujemy wyznaczoną wartość h_{min} z wartością otrzymaną ze wzoru

$$h_{\text{min}} \approx \sqrt[3]{\frac{3\varepsilon_{\text{mach}}}{M}}, \quad \text{gdzie} \quad M \approx |f'''(x)|.$$

In [6]:
```
plt.figure(figsize=(10, 6))
plt.loglog(h_values, computational_errors, label='Computational Errors Forwa
plt.loglog(h_values, computational_errors_2, label='Computational Errors Cer
plt.xlabel('h value')
plt.ylabel('Error magnitude')
plt.title('Comparision forward and central method of computational errors')
plt.legend()
plt.grid(True, which="both", ls="--")
plt.show()
```



In [7]:
```
h2_min = math.pow(3 * np.finfo(float).eps / abs(tan_df3(x0)), 1/3)
print(abs(h_values[7] - h2_min))
print(h_values[7])
print(h2_min)
```

```
2.173274156839064e-06
1e-07
2.273274156839064e-06
```

Zadanie 2

Napisz program generujący pierwsze *n* wyrazów ciągu zdefiniowanego równaniem różnicowym:

$$x_{k+1} = 2.25x_k - 0.5x_{k-1}$$

z wyrazami początkowymi:

$$x_0 = \frac{1}{3}, \quad x_1 = \frac{1}{12}$$

Wykonaj obliczenia:

- używając pojedynczej precyzji oraz przyjmując $n = 225$
- używając podwójnej precyzji oraz przyjmując $n = 60$
- używając reprezentacji z biblioteki `fractions` oraz przyjmując $n = 225$.

Narysuj wykres wartości ciągu w zależności od *k*. Użyj skali logarytmicznej na osi y (pomocna będzie funkcja `semilogy`). Następnie narysuj wykres przedstawiający wartość bezwzględną błędu względnego w zależności od *k*.

Dokładne rozwiązanie równania różnicowego:

$$x_k = \frac{4^{-k}}{3}$$

maleje wraz ze wzrostem *k*. Czy otrzymany wykres zachowa się w ten sposób? Wyjaśnij otrzymane wyniki.

Unikaj pętli na rzecz kodu wektorowego oraz funkcji uniwersalnych (np. `np.tan` zamiast `math.tan`).

In [8]:

```
import numpy as np
from fractions import Fraction
import matplotlib.pyplot as plt
```

In [9]:

```
np.set_printoptions(precision=16)


def equation(x_k, x_k_1, a1, a2):
    return a1 * x_k - a2 * x_k_1


def generate_sequence(x0, x1, a1, a2, n):
    x = [x0, x1]
    for k in range(1, n - 1):
        x.append(equation(x[k], x[k - 1], a1, a2))
    return x


def xk_real_value(k):
    return 1 / (4**k * 3)
```

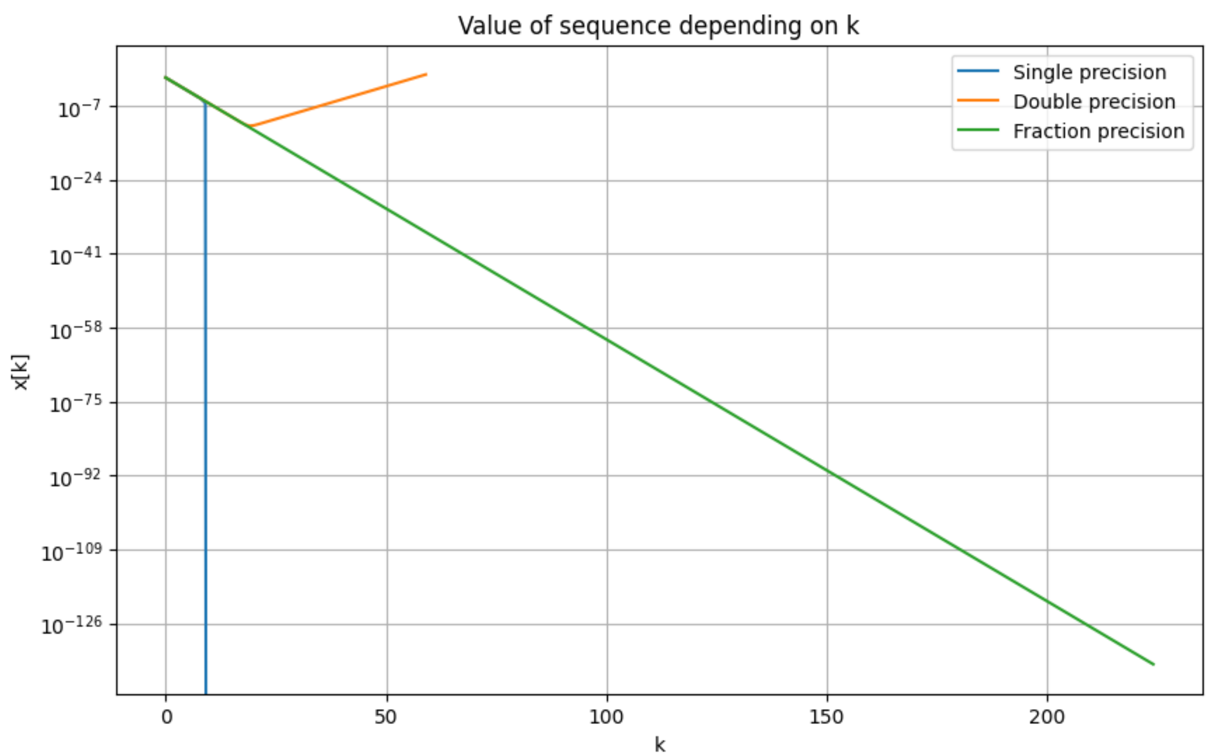
In [10]:

```
x0_single = np.float32(1 / 3)
x1_single = np.float32(1 / 12)
x0_double = np.float64(1 / 3)
x1_double = np.float64(1 / 12)
x0_fraction = Fraction(1, 3)
x1_fraction = Fraction(1, 12)
a1 = 2.25
s2 = 0.5

n_single = 225
x_single_tab = generate_sequence(x0_single, x1_single, np.float32(a1), np.fl

n_double = 60
x_double_tab = generate_sequence(x0_double, x1_double, np.float64(a1), np.fl

n_fraction = 225
x_fraction_tab = generate_sequence(x0_fraction, x1_fraction, Fraction(a1), F
```

```
/var/folders/60/sx7x5n3d3w591g26klnstcm00000gn/T/ipykernel_87162/2873410648.
py:5: RuntimeWarning: overflow encountered in scalar multiply
  return a1 * x_k - a2 * x_k_1
/var/folders/60/sx7x5n3d3w591g26klnstcm00000gn/T/ipykernel_87162/2873410648.
py:5: RuntimeWarning: invalid value encountered in scalar subtract
  return a1 * x_k - a2 * x_k_1
```

Te ostrzeżenia są spowodowane ograniczeniem pamięciowym reprezentacji zmiennych

In [11]:
```
plt.figure(figsize=(10, 6))
plt.semilogy(np.arange(n_single), x_single_tab, label='Single precision')
plt.semilogy(np.arange(n_double), x_double_tab, label='Double precision')
plt.semilogy(np.arange(n_fraction), x_fraction_tab, label='Fraction precisio
plt.xlabel('k')
plt.ylabel('x[k]')
plt.title('Value of sequence depending on k')
plt.legend()
plt.grid(True)
plt.show()
```



Wykres przedstawia wyniki zgodne z oczekiwaniami. Pojedyńcza precyzja okazała się tracić dokładność najszybciej, w przybliżeniu dwa razy szybciej niż podwójna precyzja. Reprezentacja ułamkowa nie wykazała żadnych niedokładności więc jest najbezpieczniejszym wyborem przy obliczeniach matematycznych.

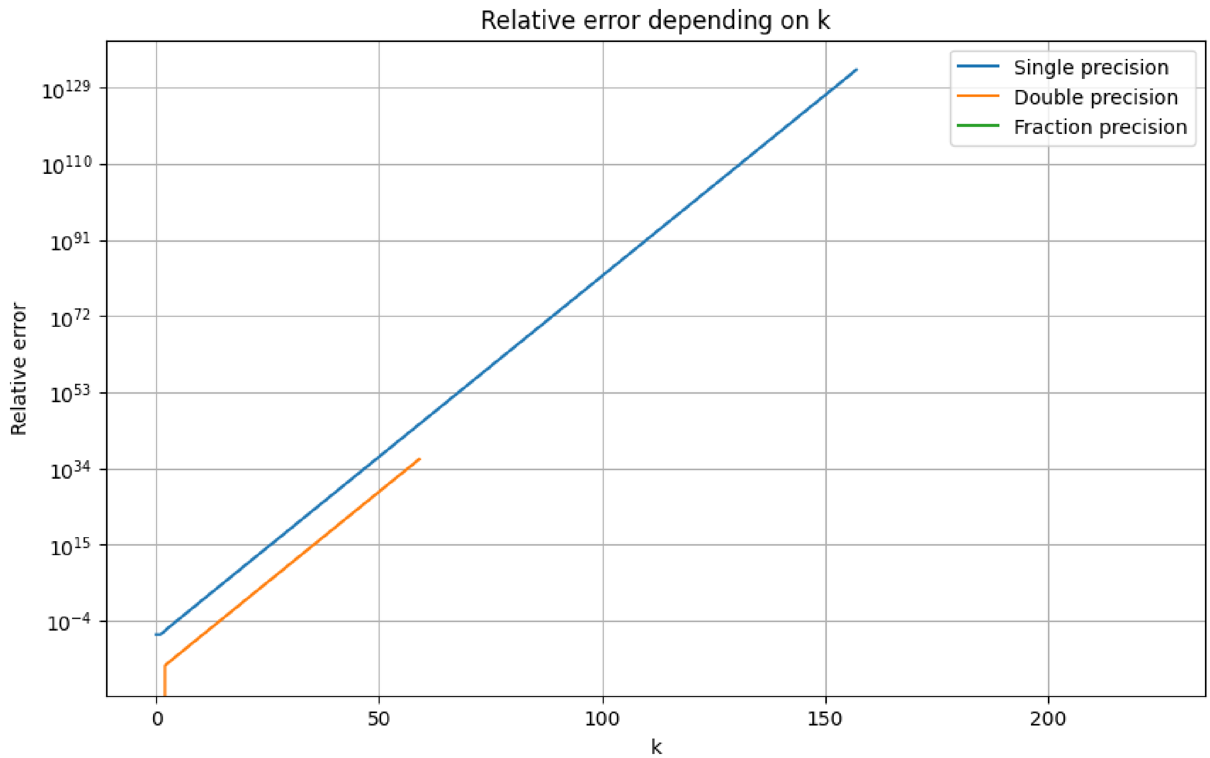
Pojedyncza precyzja: Używana, gdy prędkość obliczeń i zużycie pamięci są krytyczne, a aplikacja może tolerować niższą dokładność numeryczną. Jest wystarczająca dla wielu rzeczywistych zastosowań, ale może prowadzić do znaczących błędów w aplikacjach wymagających wysokiej precyzji.

Podwójna precyzja: Preferowana do większości obliczeń naukowych, gdzie wymagana jest wyższa dokładność numeryczna. Znacząco redukuje błędy zaokrąglenia w porównaniu do pojedynczej precyzji, co sprawia, że jest odpowiednia do obliczeń, takich jak te przedstawione tutaj, gdzie precyzja numerycznego różniczkowania jest kluczowa.

Precyzja ułamkowa (lub dowolna): Niezbędna do bardzo wrażliwych obliczeń matematycznych, gdzie nawet podwójna precyzja może wprowadzić nieakceptowalne błędy. Zastosowania obejmują kryptografię, pewne rodzaje symulacji numerycznych i obliczenia związane z bardzo dużymi lub bardzo małymi liczbami, gdzie błędy zaokrąglenia mogą znacząco wpłynąć na wyniki. Arytmetyka o dowolnej precyzji pozwala na kontrolę nad liczbą znaczących cyfr, kosztem prędkości obliczeń i zwiększonego zużycia pamięci.

In [12]:
```
single_relative_errors = []
double_relative_errors = []
fraction_relative_errors = []

for i in range(255):
    x_real = xk_real_value(i)
    if i < n_single: single_relative_errors.append(abs(x_single_tab[i] - x_r
    if i < n_double: double_relative_errors.append(abs(x_double_tab[i] - x_r
    if i < n_fraction: fraction_relative_errors.append(abs(x_fraction_tab[i]

print(fraction_relative_errors[:10])


plt.figure(figsize=(10, 6))
plt.semilogy(np.arange(n_single), single_relative_errors, label='Single prec
plt.semilogy(np.arange(n_double), double_relative_errors, label='Double prec
plt.semilogy(np.arange(n_fraction), fraction_relative_errors, label='Fractio
plt.xlabel('k')
plt.ylabel('Relative error')
plt.title('Relative error depending on k')
plt.legend()
plt.grid(True)
plt.show()
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```



Reprezentacja ułamkowa nie jest widoczna na wykresie ponieważ nie zawiera ona żadnego błędu(tablica z błędami reprezentacji ułamkowej zawiera same zera)