Homework set 1

Please **submit this Jupyter notebook through Canvas** no later than **Mon Nov. 6, 9:00**. **Submit the notebook file with your answers (as .ipynb file) and a pdf printout. The pdf version can be used by the teachers to provide feedback. A pdf version can be made using the save and export option in the Jupyter Lab file menu.**

Homework is in **groups of two**, and you are expected to hand in original work. Work that is copied from another group will not be accepted.

Exercise 0

Write down the names + student ID of the people in your group.

Zijian Zhang, 14851598

Lina Xiang, 14764369

Importing packages

Execute the following statement to import the packages `numpy` and `math` and the plotting package Matplotlib.

In [ ]:
```
import numpy as np
import math
from math import cos, sqrt
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
```

The `math` package contains functions such as $\tan$, $\exp$ and the factorial $n \mapsto n!$

In [ ]:
```
# example: the factorial function
math.factorial(5)
```

Out[ ]:
```
120
```

If you want to access `math.factorial` without typing `math.` each time you use it, use `import from`. Same for `math.exp`

In [ ]:
```
from math import factorial, exp, tan

factorial(5)
```

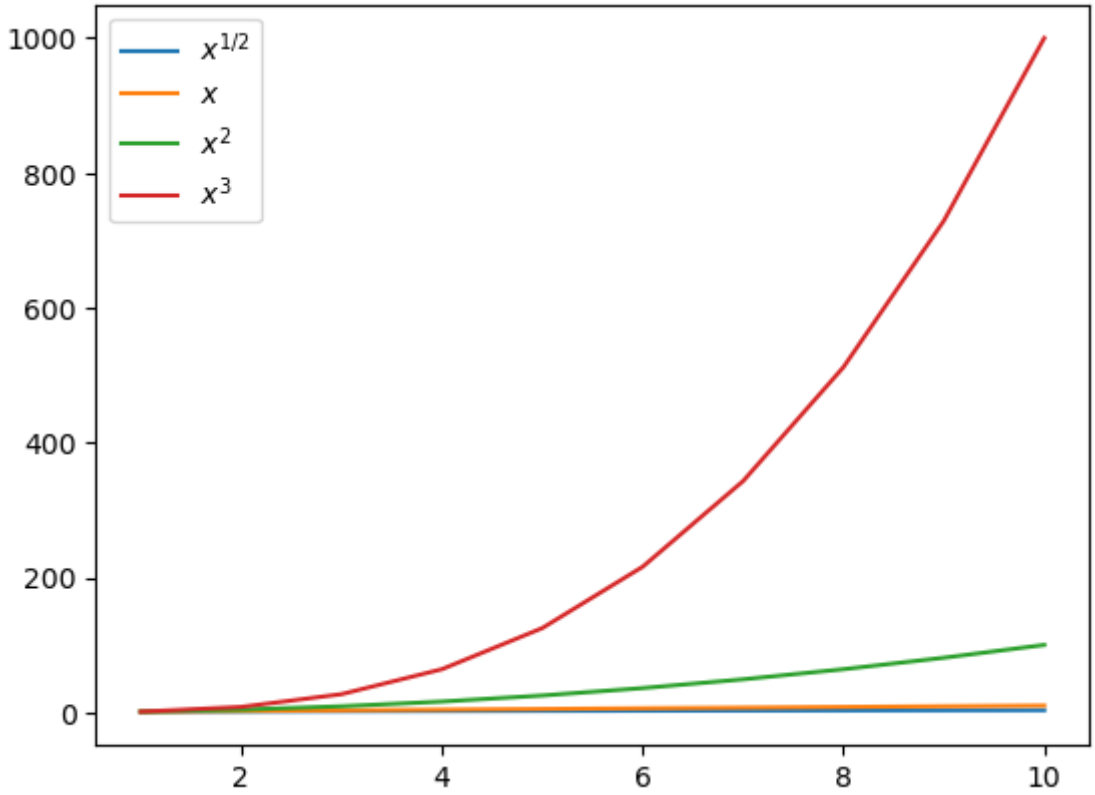
Out[ ]:
```
120
```

Very short introduction to Matplotlib

`matplotlib` is a useful package for visualizing data using Python. Run the first cell below to plot $\sqrt{x}, x, x^2, x^3$ for $x \in [1, 10]$.

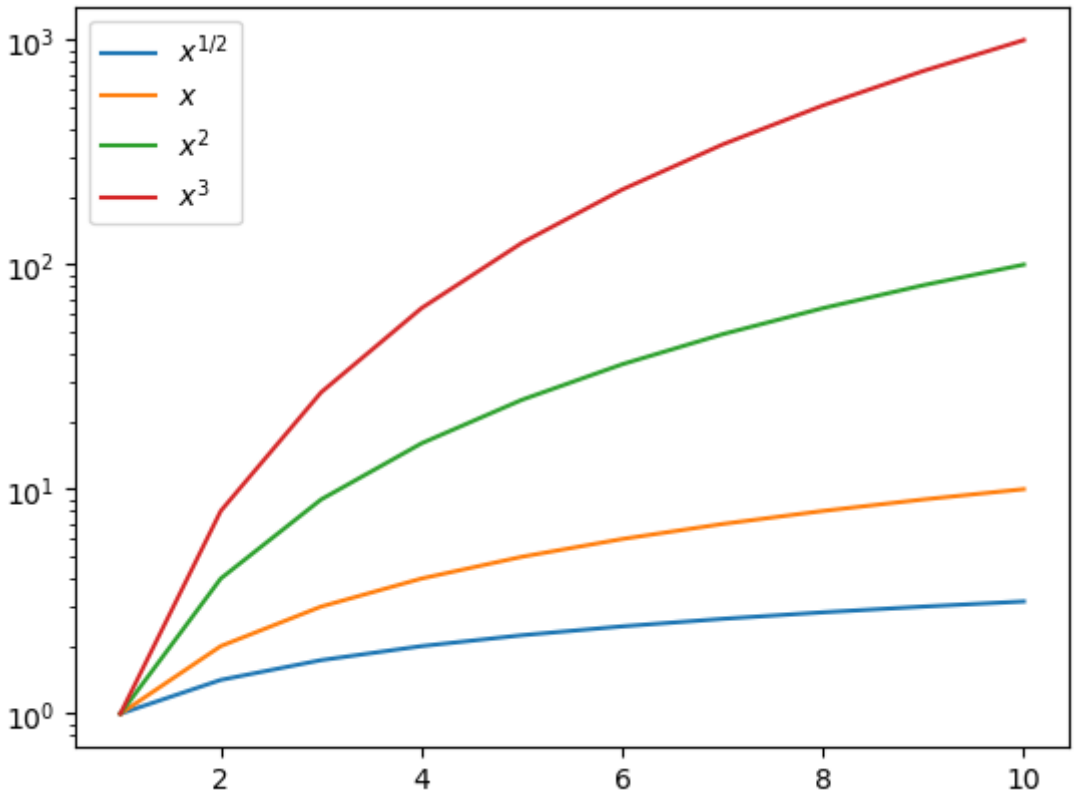
In [ ]:
```
x = np.linspace(1, 10, 10)  # 10 points evenly between 1 and 10.
print(x)
plt.plot(x, x**0.5, label=r"$x^{1/2}$")
plt.plot(x, x**1, label=r"$x$")
plt.plot(x, x**2, label=r"$x^2$")
plt.plot(x, x**3, label=r"$x^3$")
plt.legend()
plt.show()
```

```
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```



When visualizing functions where y has many different orders of magnitude, a logarithmic scale is useful:

In [ ]:
```
x = np.linspace(1, 10, 10)
plt.semilogy(x, x**0.5, label=r"$x^{1/2}$")
plt.semilogy(x, x**1, label=r"$x$")
plt.semilogy(x, x**2, label=r"$x^2$")
plt.semilogy(x, x**3, label=r"$x^3$")
plt.legend()
plt.show()
```



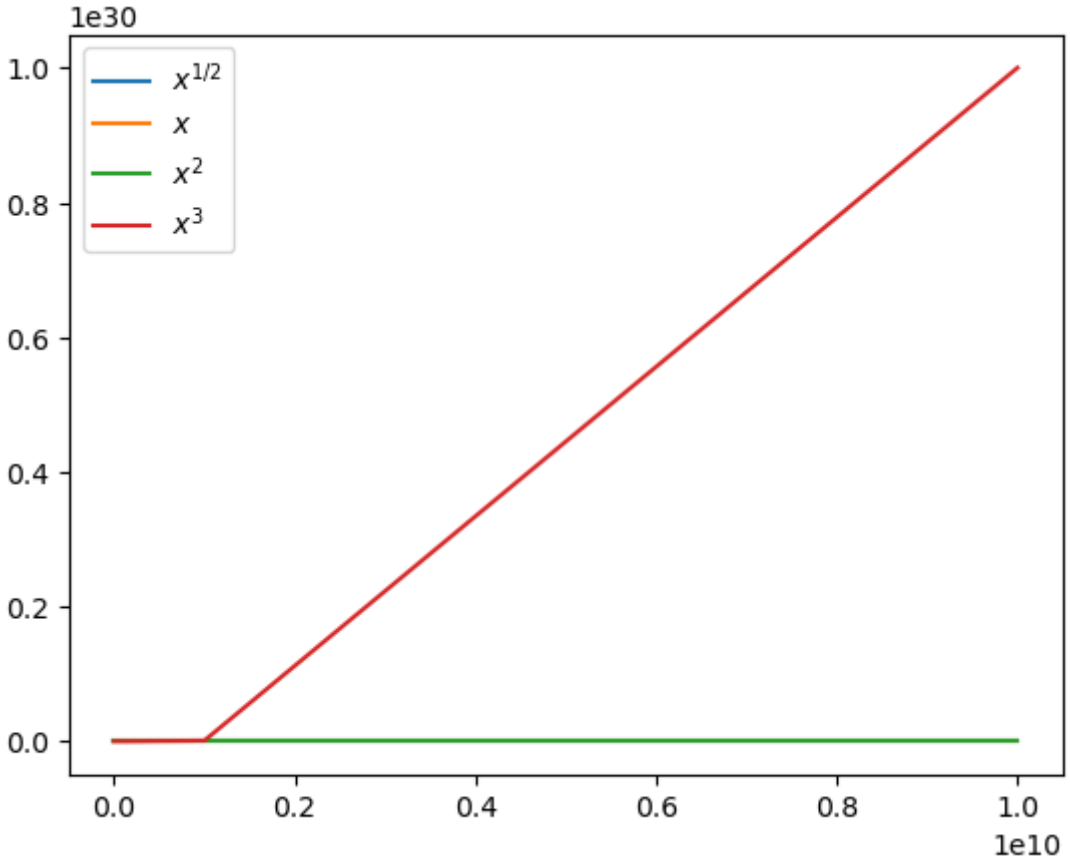
When also the x-axis contains many orders of magnitude, a log-log plot is most useful:

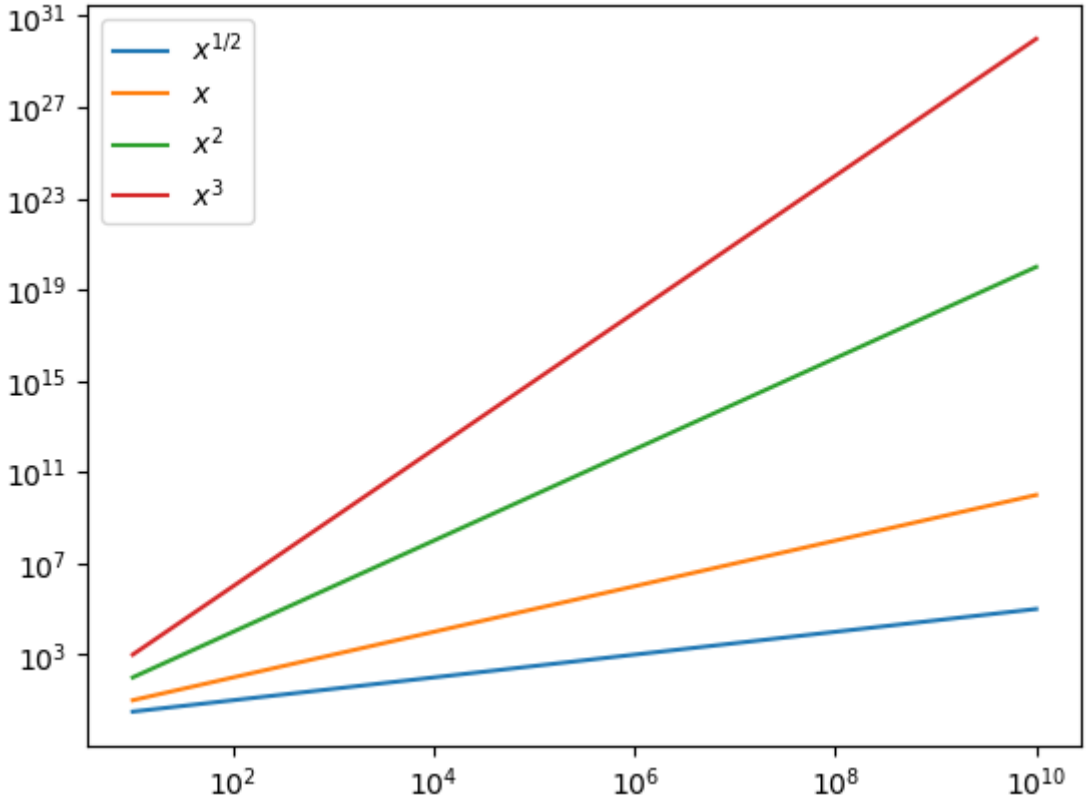
In [ ]:
```
x = np.logspace(1, 10, 10, base=10)  # 10 points evenly between 10^1 and 10^
print(x)

plt.plot(x, x**0.5, label=r"$x^{1/2}$")
plt.plot(x, x**1, label=r"$x$")
plt.plot(x, x**2, label=r"$x^2$")
plt.plot(x, x**3, label=r"$x^3$")
plt.legend()
plt.show()

plt.loglog(x, x**0.5, label=r"$x^{1/2}$")
plt.loglog(x, x**1, label=r"$x$")
plt.loglog(x, x**2, label=r"$x^2$")
plt.loglog(x, x**3, label=r"$x^3$")
plt.legend()
plt.show()
```

```
[1.e+01 1.e+02 1.e+03 1.e+04 1.e+05 1.e+06 1.e+07 1.e+08 1.e+09 1.e+10]
```





Python float types

Information about the Python `float` type is in `sys.float_info`.

In [ ]:
```
import sys
```

```
# printing float_info displays information about the python float type
print(sys.float_info)
```

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

In [ ]:
```
# the individual properties can be accessed as follows
print("epsilon for the python float type: ", sys.float_info.epsilon)
```

```
epsilon for the python float type:  2.220446049250313e-16
```

Exercise 1

(a)

Write a program to compute an approximate value for the derivative of a function using the finite difference formula

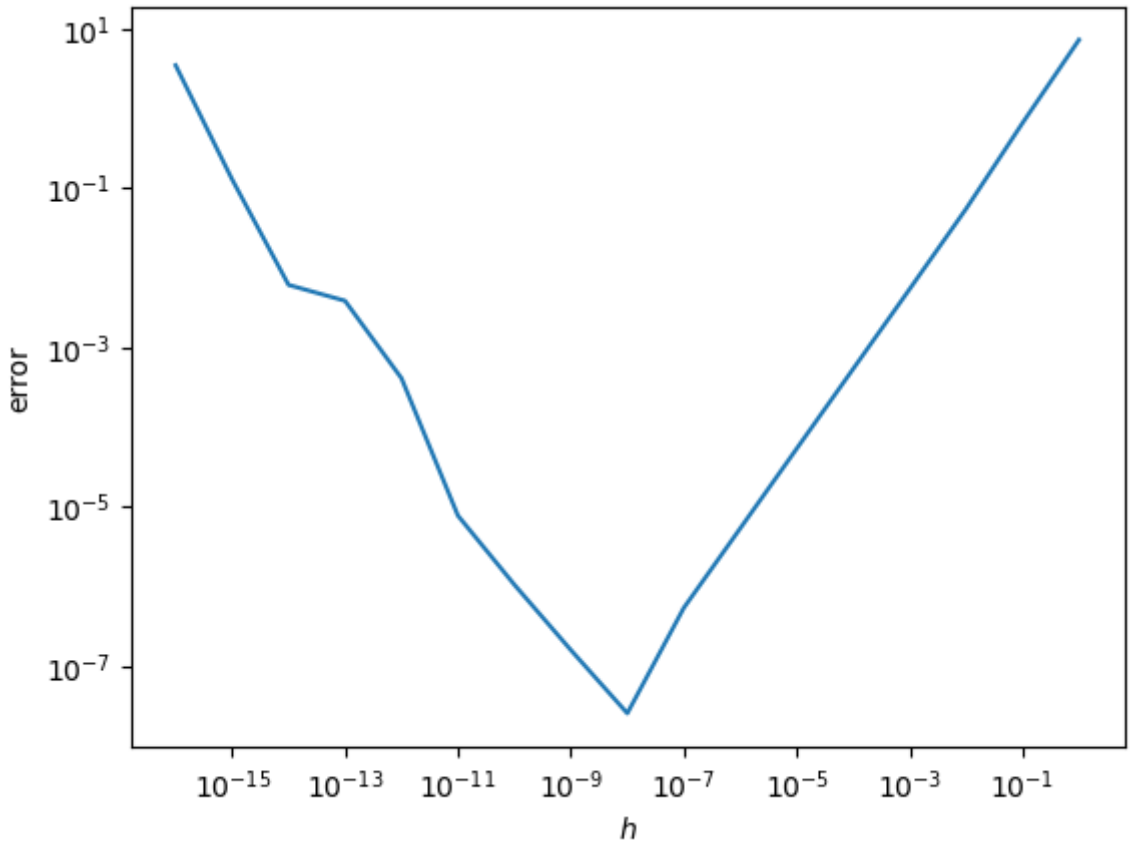
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Test your program using the function $\tan(x)$ for $x = 1$. Determine the error by comparing with the analytical derivative of $\tan(x)$. Plot the magnitude of the error as a function of h, for $h = 10^{-k}$, $k = 0, 1, 2, \ldots, 16$ using an appropriate type of plot. Is there a minimum value for the magnitude of the error? How does the corresponding value for h compare with the rule of thumb $h \approx \sqrt{\epsilon_{\text{mach}}}$ derived in Heath example 1.3?

In [ ]:
```
def fin_diff_form(f, x, h):
    dfdx = (f(x + h) - f(x)) / h
    return dfdx


epsilon = sys.float_info.epsilon
x = 1
y = 1 / cos(x)**2  # the analytical derivative of tan(x)
hs = np.logspace(0, -16, 17, base=10)
errors = []
for h in hs:
    y_hat = fin_diff_form(tan, x, h)
    errors.append(abs(y_hat - y))
plt.loglog(hs, errors)
plt.xlabel("$h$")
plt.ylabel("error")
plt.show()

print(f"The minimum value for the magnitude of the error: {min(errors)}")
print(f"The corresponding value for h: {hs[np.argmin(errors)]}")
print(f"sqrt(epsilon): {sqrt(epsilon)}")
```




```
The minimum value for the magnitude of the error: 2.554135347665465e-08
The corresponding value for h: 1e-08
sqrt(epsilon): 1.4901161193847656e-08
```

Answer:

Yes, there is a minimum value for the magnitude of the error: $2.554135347665465 \times 10^{-8}$.

The corresponding value for h is 10^{-8}, which is near the $\sqrt{\epsilon_{mach}} \approx 1.49 \times 10^{-8}$.

(b)

Repeat the exercise using the centered difference approximation

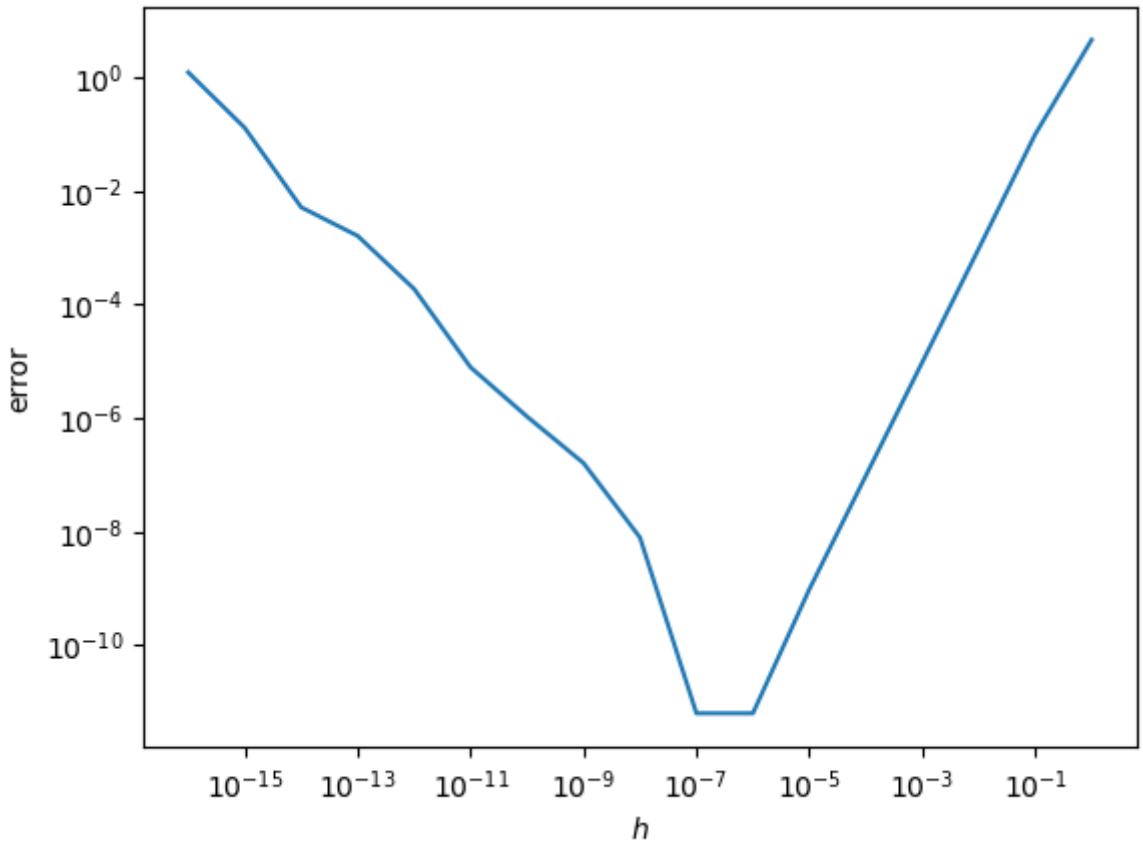
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}.$$

In [ ]:
```python
def fin_diff_form(f, x, h):
    dfdx = (f(x + h) - f(x - h)) / (2 * h)
    return dfdx


x = 1
y = 1 / cos(x)**2  # the analytical derivative of tan(x)
hs = np.logspace(0, -16, 17, base=10)
errors = []
for h in hs:
    y_hat = fin_diff_form(tan, x, h)
    errors.append(abs(y_hat - y))
plt.loglog(hs, errors)
plt.xlabel("$h$")
```

```
plt.ylabel("error")
plt.show()

print(f"The minimum value for the magnitude of the error: {min(errors)}")
print(f"The corresponding value for h: {hs[np.argmin(errors)]}")
print(f"epsilon**(1/3): {epsilon ** (1/3)}")
```




```
The minimum value for the magnitude of the error: 6.2239102760486276e-12
The corresponding value for h: 1e-06
epsilon**(1/3): 6.055454452393343e-06
```

Answer:

Yes, there is a minimum value for the magnitude of the error: $6.2239102760486276 \times 10^{-12}$. It is reduced by using a more accurate finite difference formula.

The corresponding value for h is 10^{-6}, which is near the $\sqrt[3]{\epsilon_{mach}} \approx 6.06 \times 10^{-6}$. The analysis of this result is as follows:

By Taylor's Theorem,

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(\theta_1)\frac{h^3}{6},$$
$$f(x-h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(\theta_2)\frac{h^3}{6},$$

for some $\theta_1 \in [x, x+h]$ and $\theta_2 \in [x-h, x]$, so the truncation error of the centered difference approximation is bounded by $\frac{Mh^2}{6}$, where M is a bound on $\left|f'''(t)\right|$ for t near x. The rounding error is still bounded by $\frac{2\epsilon_{mach}}{h}$. The total computational error is therefore bounded by the sum of two functions,

$$\frac{Mh^2}{6} + \frac{2\epsilon_{mach}}{h}.$$

Differentiating this function with respect to h and setting its derivative equal to zero, we see that the bound on the total computational error is minimized when

$$h = \sqrt[3]{\frac{6\epsilon_{mach}}{M}}.$$

Exercise 2

As you probably know, the exponential function e^x is given by an infinite series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots. \qquad (*)$$

(a)

Suppose you write a program to sum the series in the natural order, what stopping criterion should you use? Explain your answer.

Answer:

I would use the following stopping criterion:

- If $\frac{x^n}{n!} \leq \epsilon_{mach} \sum_{k=1}^{n-1} \frac{x^n}{n!}$, stop computation.

In this case, the partial sum ceases to change because $\frac{x^n}{n!}$ becomes negligible relative to the partial sum.

(b)

Write a program to sum the series in the natural order, using the stopping criterion you just described.

Test your program for

$$x = \pm 1, \pm 5, \pm 10, \pm 15, \pm 20,$$

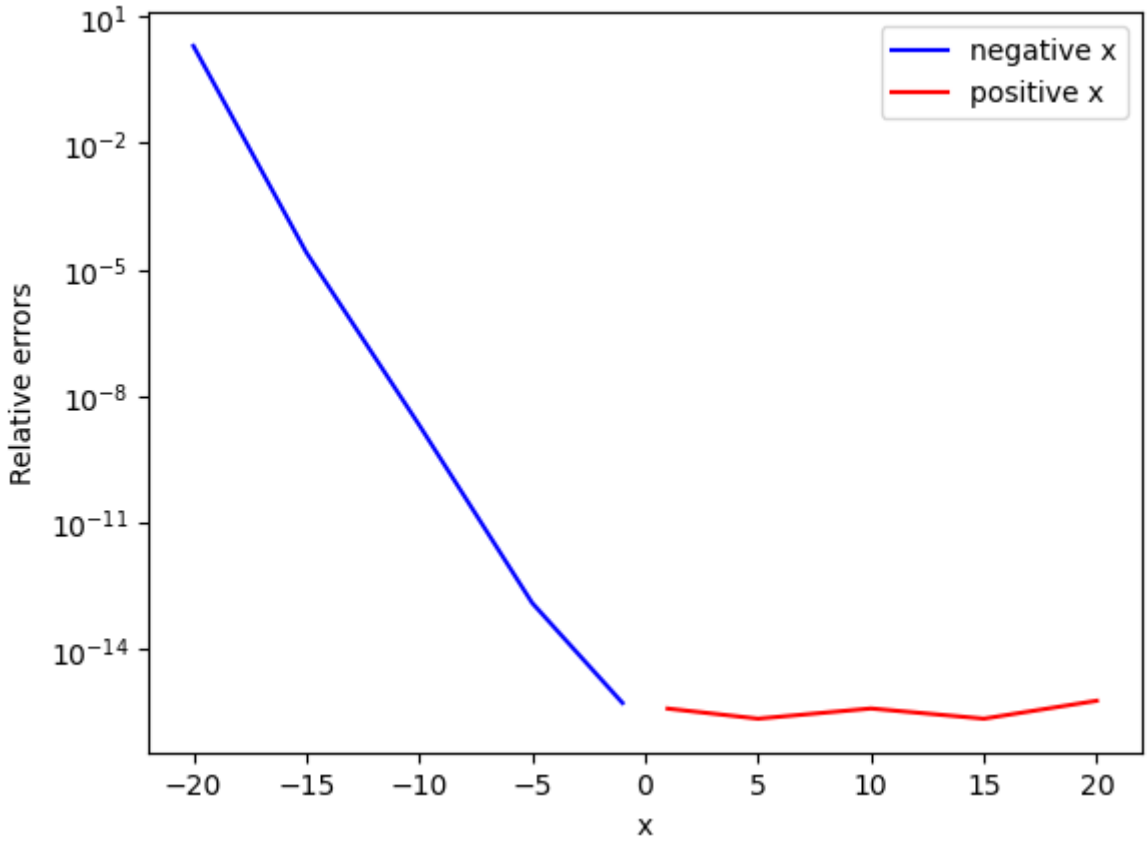
and compare your results with the built-in function $\exp(x)$. Explain any cases where the exponential function is not well approximated.

In [ ]:
```
def compute_exp_b(x):
    ex = 1.0
    xn = 1.0
    n = 1
    while abs(xn) > epsilon * abs(ex):
        xn *= x / n
        ex += xn
```

```
        n += 1
    return ex


xs = np.array([1, 5, 10, 15, 20])
xs = np.append(xs, -xs)
xs = np.sort(xs)
exs = []  # list of ex values computed by compute_exp_b()
exps = []  # list of ex values computed by math.exp()
for x in xs:
    exs.append(compute_exp_b(x))
    exps.append(math.exp(x))
exs = np.array(exs)
exps = np.array(exps)
rel_errors = np.abs((exs - exps) / exps)  # relative errors
ax = plt.subplot()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
# Add a small positive offset to ensure (x,0) can be displayed on the semilo
ax.semilogy(xs[:5], rel_errors[:5] + epsilon, c='b', label='negative x')
ax.semilogy(xs[5:], rel_errors[5:] + epsilon, c='r', label='positive x')
ax.legend()
ax.set_xlabel("x")
ax.set_ylabel("Relative errors")

plt.show()
```



Answer:

In scenarios where $x > 0$, all the observed relative errors remain below 10^{-13}, indicative of a markedly minimal deviation. In contrast, the application of the same computational methodology to cases where $x < 0$ results in relative errors of substantially greater magnitude, which escalate quickly as the absolute value of x increases. The disproportionate escalation of errors in the latter case is attributable to the phenomenon of numerical cancellation. Specifically, when x assumes negative values, the most

significant digits are the first to be relinquished due to cancellation, thereby precipitating an amplification in the accumulation of relative errors.

(c)

Can you use the series in this form to obtain accurate results for $x < 0$? (*Hint*: $e^{-x} = 1/e^x$.) If yes, write a second program that implements this and test it again on $x = -1, -5, -10, -15, -20$.

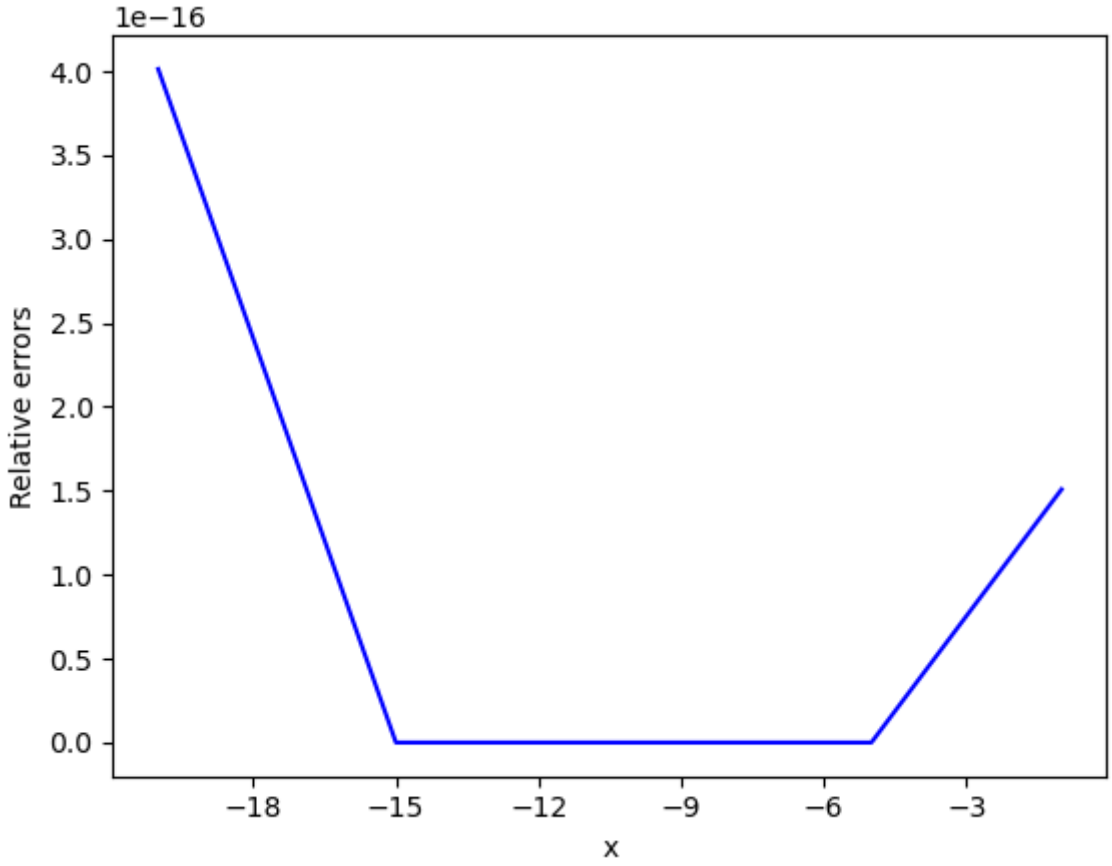
Answer:

When $x < 0$, the sign of the terms of the series (*) alternates between positive and negative, which tends to induce larger errors due to cancellation effects. To circumvent this issue, one may implement a change of variable, letting $X = -x$, thus ensuring $X > 0$. Following the method outlined in Question 2(b), the evaluation of e^X is achieved. The final result for e^x is then simply the reciprocal of e^X.

In [ ]:
```
def compute_exp_c(x):
    if x < 0:
        return 1 / compute_exp_c(-x)
    ex = 1.0
    xn = 1.0
    n = 1
    while xn > epsilon * ex:
        xn *= x / n
        ex += xn
        n += 1
    return ex


xs = np.array([1, 5, 10, 15, 20])
xs = np.sort(-xs)
exs = []  # list of ex values computed by compute_exp_c()
exps = []  # list of ex values computed by math.exp()
for x in xs:
    exs.append(compute_exp_c(x))
    exps.append(math.exp(x))
exs = np.array(exs)
exps = np.array(exps)
rel_errors = np.abs((exs - exps) / exps)  # relative errors
ax = plt.subplot()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
ax.plot(xs, rel_errors, c='b', label='negative x')
ax.set_xlabel("x")
ax.set_ylabel("Relative errors")

plt.show()
```



(d)

Can you rearrange the series of regroup the terms of the series (*) in any way to obtain more acccurate results for $x < 0$?

Answer:

We initially attempted to address the cancellation error that occurs when $x < 0$ by separately summing the odd and even terms of the series. This approach was implemented in the function `compute_exp_d1()` in the code block that follows. However, it became evident that this technique only delayed the occurrence of cancellation error, which resurfaced when the aggregated odd and even terms were combined.

To improve our strategy, we decided to prioritize the summation of odd terms first, followed by sequentially adding the even terms. This revised approach was implemented in the function `compute_exp_d2()` in the subsequent code block. To assess the effectiveness of this new method, we conducted a comparative analysis of the relative errors, as shown in the graph. The results indicated that there was negligible difference between the outcomes of `compute_exp_d1()` and `compute_exp_d2()`.

In conclusion, our exploration did not yield a methodology that effectively reduced the relative error by rearranging or grouping the terms of the series.

In [ ]:
```
def compute_exp_d1(x):
    ex_pos = 0.0
    ex_neg = 0.0
```

```
    xn_pos = 1
    xn_neg = x
    n_pos = 0
    n_neg = 1
    while abs(xn_pos) > epsilon * abs(ex_pos):
        ex_pos += xn_pos
        n_pos += 2
        xn_pos *= x * x / (n_pos * (n_pos - 1))
    while abs(xn_neg) > epsilon * abs(ex_neg):
        ex_neg += xn_neg
        n_neg += 2
        xn_neg *= x * x / (n_neg * (n_neg - 1))
    return ex_pos + ex_neg


def compute_exp_d2(x):
    ex = 0.0
    xn_pos = 1
    xn_neg = x
    n_pos = 0
    n_neg = 1
    while abs(xn_pos) > epsilon * abs(ex):
        ex += xn_pos
        n_pos += 2
        xn_pos *= x * x / (n_pos * (n_pos - 1))
    while abs(xn_neg) > epsilon * abs(ex):
        ex += xn_neg
        n_neg += 2
        xn_neg *= x * x / (n_neg * (n_neg - 1))
    return ex


xs = np.array([1, 5, 10, 15, 20])
xs = np.sort(-xs)
ax = plt.subplot()
ax.xaxis.set_major_locator(MaxNLocator(integer=True))

# Compute e^x by compute_exp_b()
exs = []
exps = []
for x in xs:
    exs.append(compute_exp_b(x))
    exps.append(math.exp(x))
exs = np.array(exs)
exps = np.array(exps)
rel_errors = np.abs((exs - exps) / exps)
ax.semilogy(xs, rel_errors, c='r', label='compute_exp_b(x)')
ax.set_xlabel("x")
ax.set_ylabel("Relative errors")

# Compute e^x by compute_exp_d1()
exs = []
exps = []
for x in xs:
    exs.append(compute_exp_d1(x))
    exps.append(math.exp(x))
exs = np.array(exs)
exps = np.array(exps)
rel_errors = np.abs((exs - exps) / exps)
ax.semilogy(xs, rel_errors, c='g', label='compute_exp_d1(x)')
ax.set_xlabel("x")
ax.set_ylabel("Relative errors")

# Compute e^x by compute_exp_d2()
```

```
exs = []
exps = []
for x in xs:
    exs.append(compute_exp_d2(x))
    exps.append(math.exp(x))
exs = np.array(exs)
exps = np.array(exps)
rel_errors = np.abs((exs - exps) / exps)
ax.semilogy(xs, rel_errors, c='b', label='compute_exp_d2(x)')
ax.set_xlabel("x")
ax.set_ylabel("Relative errors")

ax.legend()
plt.show()
```