# Homework set 1

Please **submit this Jupyter notebook through Canvas** no later than **Mon Nov. 7, 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.

Run the following cell to import the necessary packages.

In [50]:
```
import numpy as np
import matplotlib.pyplot as plt
```

## NumPy in single-precision floating point numbers

Working with real numbers on a computer can sometimes be counter-intuitive. Not every real number cannot be represented exactly, because that would require an infinite amount of memory. Real numbers are in Python represented as "double-precision floating point numbers" that approximate the real numbers they represent. As such, the usual "rules of mathematics" no longer hold for very small or very large numbers:

In [51]:
```
print("very small numbers:")
print(1 - 1)          # Should be zero
print(1 - 1 + 1e-17)  # Should be 10 ** -17, i.e. a very small number
print(1 + 1e-17 - 1)  # Should *also* be 10**-17, but is it?

print("very large numbers:")
print(2.0**53)        # Some very large number
print(2.0**53 + 1.0)  # Some very large number + 1
```

```
very small numbers:
0
1e-17
0.0
very large numbers:
9007199254740992.0
9007199254740992.0
```

Usually, you don't have to worry about these rounding errors. But in scientific computing, these rounding errors sometimes become important. To reveal this problem more directly, we can decrease the precision of these approximations, using "single precision" instead of double precision floating point numbers, by employing `np.single` :

In [52]:
```
print(1.0 + 10**-9)              # Should be slightly above 1
print(np.single(1.0 + 10**-9))   # But in single precision, it is exactly 1.
```

```
1.000000001
1.0
```

Today we will practice with these single-precision floating point numbers. One thing to keep in mind is that Python will *really* try to work with double-precision floats:

In [53]:
```
a = 5.0
b = np.single(5.0)
print("a and b represent the same value:", a == b)
print("but they are of different types:", type(a), type(b))
print("If I add zero to a, its type does not change: ", type(a) == type(a + 0.0))
print("If I add zero to b, its type *does* change:   ", type(b) == type(b + 0.0))
```

```
a and b represent the same value: True
but they are of different types: <class 'float'> <class 'numpy.float32'>
If I add zero to a, its type does not change:  True
If I add zero to b, its type *does* change:    False
```

Any time Python encounters a number like `1` or `0` or `math.pi`, it will interpret this as double precision, unless you use `np.single`.

So we have to be extra careful when working with these single-precision numbers, to prevent these types changing. See the difference between `S` and `T` below.

In [54]:
```
S = 0.0
S += np.single(5.0)
print(type(S))

T = np.single(0.0)
T += np.single(5.0)
print(type(T))
```

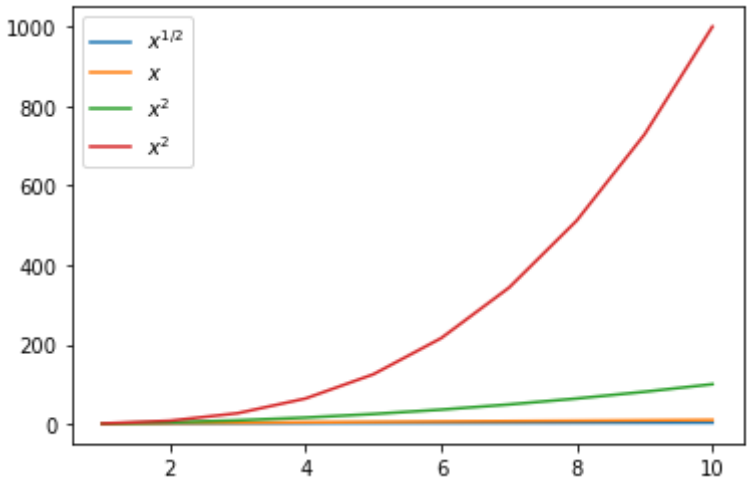
```
<class 'numpy.float64'>
<class 'numpy.float32'>
```

# 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 $x \in [1, 10]$.

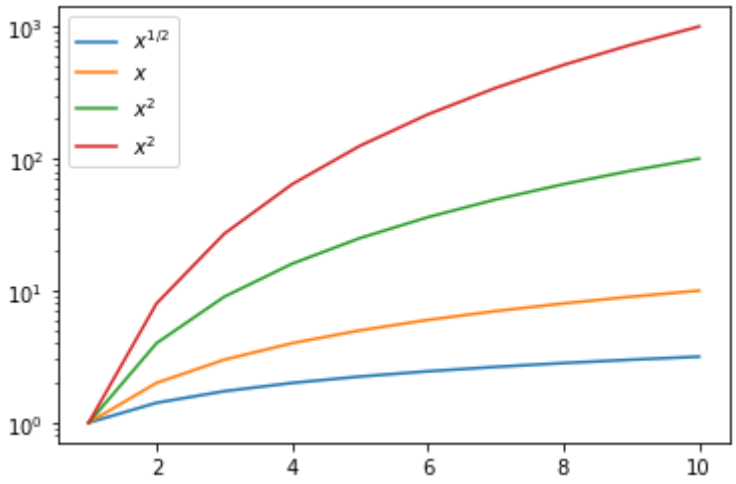
In [55]:
```
x = np.linspace(1, 10, 10)  # 10 points evenly between 1 and 10.
print(x)
plt.plot(x, x**0.5, label=r"$x^{1/2}$")
plt.plot(x, x**1, label=r"$x$")
plt.plot(x, x**2, label=r"$x^2$")
plt.plot(x, x**3, label=r"$x^2$")
plt.legend()
plt.show()
```

```
[ 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 [56]:
```
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"$x^2$")
plt.legend()
plt.show()
```

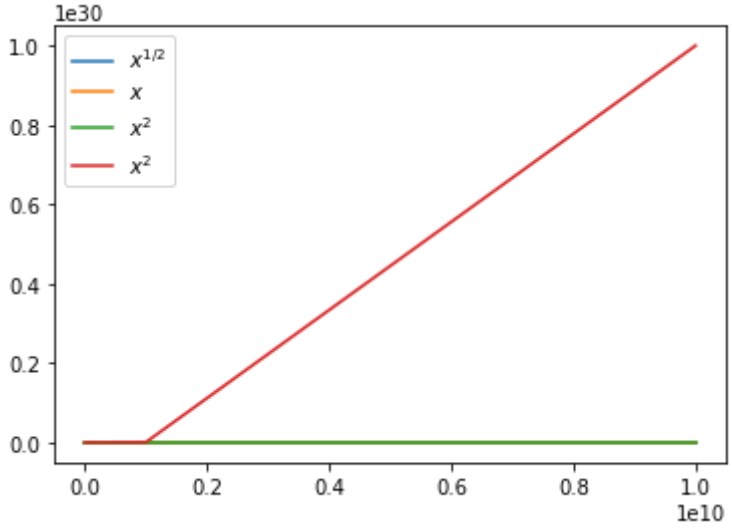


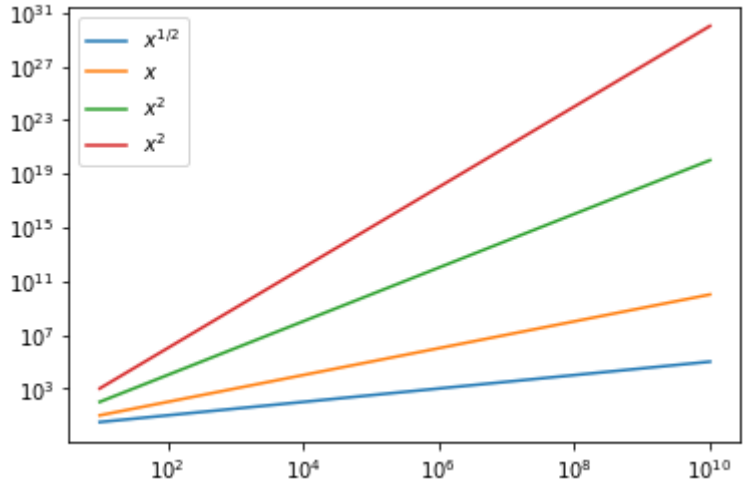

When also the $x$-axis contains many orders of magnitude, a log-log plot is most useful:

In [57]:
```
x = np.logspace(1, 10, 10, base=10)  # 10 points evenly between 10^1 and 10^10.
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"$x^2$")
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"$x^2$")
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]
```




---

# Exercise 1

This exercise is a variant of exercise 1.6 in the book.

## (a)

Lookup the Taylor series for $\cos(x)$ in the base point $0$. (You don't have to hand in the series expansion)

## (b) (0.5 pt)

What are the forward and backward errors if we approximate $\cos(x)$ by the first **two** nonzero terms in the Taylor series at $x = 0.2$, $x = 1.0$ and $x = 2.0$?

$x = 0.2$

$\Delta y = \hat{y} - y = 1 - \frac{x^2}{2} - y = 1 - \frac{0.2^2}{2} - 0.9800665778412416 = -6.657784124164401e - 05$

$\Delta x = \hat{x} - x = arccos(\hat{y}) - x = arccos(1 - \frac{x^2}{2}) - x = 0.20033484232311968 - 0.2 = 0.00033$

$x = 1.0$

$\Delta y = 1 - \frac{1.0^2}{2} - 0.5403023058681398 = -0.040302305868139765$

$\Delta x = arccos(1 - \frac{1.0^2}{2}) - 1.0 = 0.04719755119659785$

$x = 2.0$

$\Delta y = 1 - \frac{2.0^2}{2} - (-0.4161468365471424) = -0.5838531634528576$

$\Delta x = arccos(1 - \frac{2.0^2}{2}) - 2.0 = 1.1415926535897931$

In [58]:
```
import numpy as np
print(np.cos(0.2))
print(np.arccos(1 - 0.2**2/2))
print(np.cos(1.0))
print(np.arccos(1 - 1.0**2/2))
print(np.cos(2.0))
print(np.arccos(1 - 2.0**2/2))
```

```
0.9800665778412416
0.20033484232311968
0.5403023058681398
1.0471975511965976
-0.4161468365471424
3.141592653589793
```

In [59]:
```
print(1 - 0.2**2/2 - np.cos(0.2))
print(np.arccos(1 - 0.2**2/2) - 0.2)
print(1 - 1.0**2/2 - np.cos(1.0))
print(np.arccos(1 - 1.0**2/2) - 1.0)
print(1 - 2.0**2/2 - np.cos(2.0))
print(np.arccos(1 - 2.0**2/2) - 2.0)
```

```
-6.657784124164401e-05
0.00033484232311967177
-0.040302305868139765
0.04719755119659763
-0.5838531634528576
1.1415926535897931
```

# (c) (0.5 pt)

What are the forward and backward errors if we approximate $\cos(x)$ by the first **three** nonzero terms in the Taylor series at $x = 0.2$, $x = 1.0$ and $x = 2.0$?

$x = 0.2$

$\Delta y = 1 - \frac{0.2^2}{2} + \frac{0.2^4}{24} - 0.9800665778412416 = 8.882542501531532e - 08$

$\Delta x = arccos(1 - \frac{0.2^2}{2} + \frac{0.2^4}{24}) - 0.2 = -4.4710234142764094e - 07$

$x = 1.0$

$\Delta y = 1 - \frac{1.0^2}{2} + \frac{1.0^4}{24} - 0.5403023058681398 = 0.0013643607985268646$

$\Delta x = arccos(1 - \frac{1.0^2}{2} + \frac{1.0^4}{24}) - 1.0 = -0.0016222452979235413$

$x = 2.0$

$\Delta y = 1 - \frac{2.0^2}{2} + \frac{2.0^4}{24} - (-0.4161468365471424) = 0.08281350321380904$

$\Delta x = arccos(1 - \frac{2.0^2}{2} + \frac{2.0^4}{24}) - 2.0 = -0.0893667637509814$

In [60]:
```
import numpy as np
print(np.cos(0.2))
print(np.arccos(1 - 0.2**2/2 + 0.2**4/24))
print(np.cos(1.0))
print(np.arccos(1 - 1.0**2/2 + 1.0**4/24))
print(np.cos(2.0))
print(np.arccos(1 - 2.0**2/2 + 2.0**4/24))
```

```
0.9800665778412416
0.19999955289765858
0.5403023058681398
0.9983777547020765
-0.4161468365471424
1.9106332362490186
```

In [61]:
```
print(1 - 0.2**2/2 + 0.2**4/24 - np.cos(0.2))
print(np.arccos(1 - 0.2**2/2 + 0.2**4/24) - 0.2)
print(1 - 1.0**2/2 + 1.0**4/24 - np.cos(1.0))
print(np.arccos(1 - 1.0**2/2 + 1.0**4/24) - 1.0)
print(1 - 2.0**2/2 + 2.0**4/24 - np.cos(2.0))
print(np.arccos(1 - 2.0**2/2 + 2.0**4/24) - 2.0)
```

```
8.882542501531532e-08
-4.4710234142764094e-07
0.0013643607985268646
-0.0016222452979235413
0.08281350321380904
-0.0893667637509814
```

# (d) (1 pt)

Compute the relative condition of $x \mapsto \cos(x)$ at $x = 0.2$, $x = 1.0$ and $x = 2.0$.

$Two - terms\_approximation$

$x = 0.2$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{-6.657784124164401e-05}{0.00033484232311967177}| = 0.1988334109659407$

$x = 1.0$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{-0.040302305868139765}{0.04719755119659785}| = 0.853906714360318$

$x = 2.0$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{-0.5838531634528576}{1.1415926535897931}| = 0.5114373867218778$

$Three - terms\_approximation$

$x = 0.2$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{8.882542501531532e-08}{-4.4710234142764094e-07}| = 0.19866911171095003$

$x = 1.0$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{0.0013643607985268646}{-0.0016222452979235413}| = 0.8410323643860972$

$x = 2.0$

$C = |\frac{\Delta y}{\Delta x}| = |\frac{0.08281350321380904}{-0.0893667637509814}| = 0.926670047542139$

In [62]:
```
print(abs(-6.657784124164401e-05 / 0.00033484232311967177))
print(abs(-0.040302305868139765 / 0.04719755119659785))
print(abs(-0.5838531634528576 / 1.1415926535897931))
```

```
0.1988334109659407
0.853906714360318
0.5114373867218778
```

In [63]:
```
print(abs(8.882542501531532e-08 / -4.4710234142764094e-07))
print(abs(0.0013643607985268646 / -0.0016222452979235413))
print(abs(0.08281350321380904 / -0.0893667637509814))
```

```
0.19866911171095003
0.8410323643860972
0.926670047542139
```

# Exercise 2

This exercise is about computing the sum of a set of $n$ random numbers. You are asked to implement different ways to compute the sum. To be able to compare rounding errors for the different methods, all sums have to be executed in single precision (some hints are above), and implemented by yourself, unless specifically mentioned. The result of each sum can then be compared with a reference implementation that employs the standard double precision format.

Vary $n$ by choosing different powers of 10 at least up to, say, $10^7$.

## (a)

Create a function that returns an array of $n$ single precision random numbers (here denoted by $x_1, i = 1, \ldots, n$), uniformly distributed in the interval $[0, 1]$. You may use a suitable function from `numpy.random` .

Create a function to sum the numbers using double precision computations in the order they are generated.

In [64]:
```
import numpy as np
n_max = 7
n = [10**x for x in range(1, n_max + 1)]
#n = np.linspace(10, 1e7, 10, dtype='int')
sums_double = np.zeros(len(n))

np.random.seed(10)

for j in range(len(n)):
    arr = np.random.rand(n[j])

    for i in range(len(arr)):
        sums_double[j] += arr[i]

print(type(sums_double[1]))
print(sums_double)
```

```
<class 'numpy.float64'>
[4.11387260e+00 4.94232202e+01 4.93263079e+02 4.92227617e+03
 4.99268485e+04 5.00395700e+05 5.00094162e+06]
```

# (b) (a+b together 2 pts)

Create a function to sum the numbers in the order in which they were generated, this time using single-precision computations. Visualize the errors as a function of $n$ using a log-log plot.

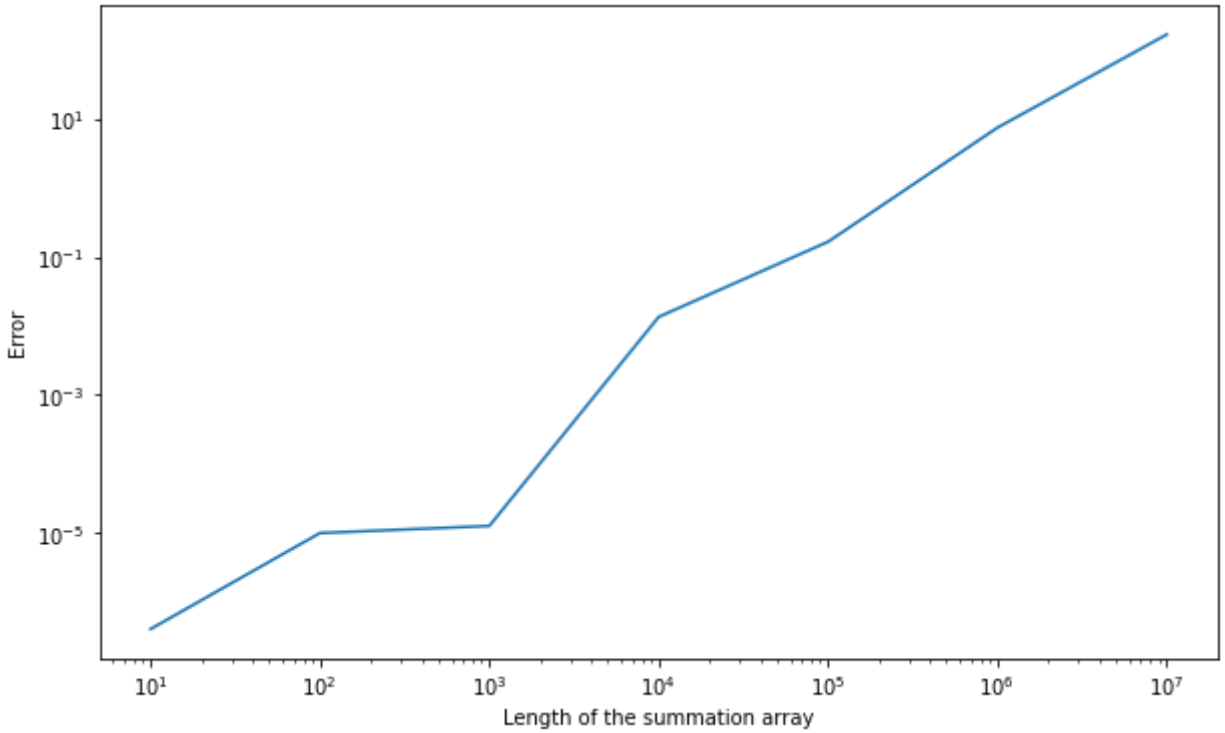
In [65]:
```
import numpy as np
n_max = 7
n = [10**x for x in range(1, n_max + 1)]
#n = np.linspace(10, 1e7, 10, dtype='int')
sums_single = np.zeros(len(n), dtype=np.single)

np.random.seed(10)

for j in range(len(n)):
    arr = np.single(np.random.rand(n[j]))

    for i in range(len(arr)):
        sums_single[j] += arr[i]

print(type(sums_single[1]))
print(sums_single)
```

```
<class 'numpy.float32'>
[4.1138730e+00 4.9423210e+01 4.9326309e+02 4.9222627e+03 4.9926684e+04
 5.0040309e+05 5.0007765e+06]
```

In [66]:
```
import matplotlib.pyplot as plt
```

```
err = []
for i in range(len(sums_double)):
    err.append(abs(sums_double[i] - sums_single[i]))

plt.figure(figsize=(10, 6))
plt.loglog(n, err)
plt.xlabel('Length of the summation array')
plt.ylabel('Error')
```

Out[66]: Text(0, 0.5, 'Error')




## (c) (1.5 pts)

Use the following compensated summation algorithm (due to Kahan), again using only single precision, to sum the numbers in the order in which they were generated:

(algorithm at https://canvas.uva.nl/files/7499123/download?download_frd=1)

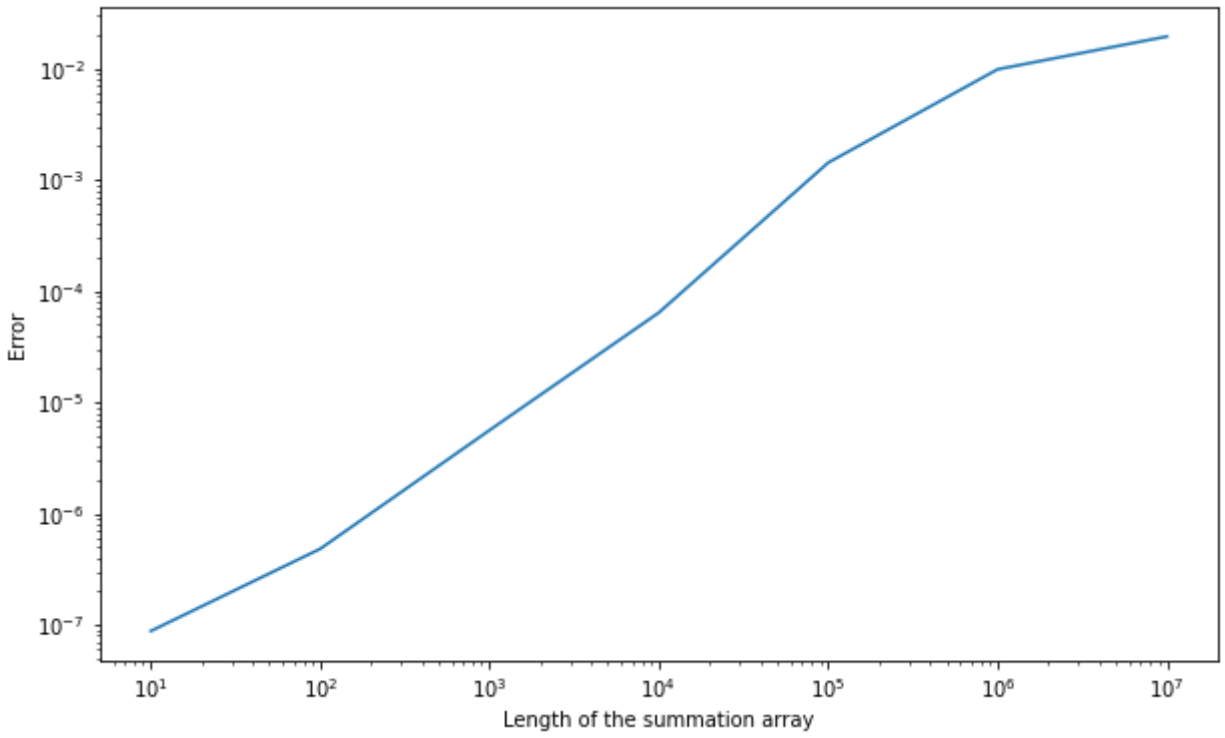
Plot the error as a function of $n$.

In [67]:

```
def sumation_double(rand_int,n):
    c = 0
    x_1 = rand_int[0,0]
    s = x_1
    for i in range(n):
        y = rand_int[0,i] - c
        t = s + y
        c = (t - s) - y
        s = t
    return s
```

```
def sumation_single(rand_int,n):
    c = np.single(0)
    x_1 = np.single(rand_int[0,0])
    s = x_1
    for i in range(n):
        y = np.single(rand_int[0,i] - c)
        t = np.single(s + y)
        c = np.single((t - s) - y)
        s = np.single(t)
    return s

N_pow = [10,10**2,10**3,10**4,10**5,10**6,10**7]
error = []
for i in N_pow:
    rand_int = np.random.rand(1,i)
    single   = sumation_single(rand_int,i)
    double   = sumation_double(rand_int,i)
    error.append(abs(single - double))
print(error)

plt.figure(figsize=(10, 6))
plt.loglog(N_pow, error)
plt.xlabel('Length of the summation array')
plt.ylabel('Error')
plt.show()
```

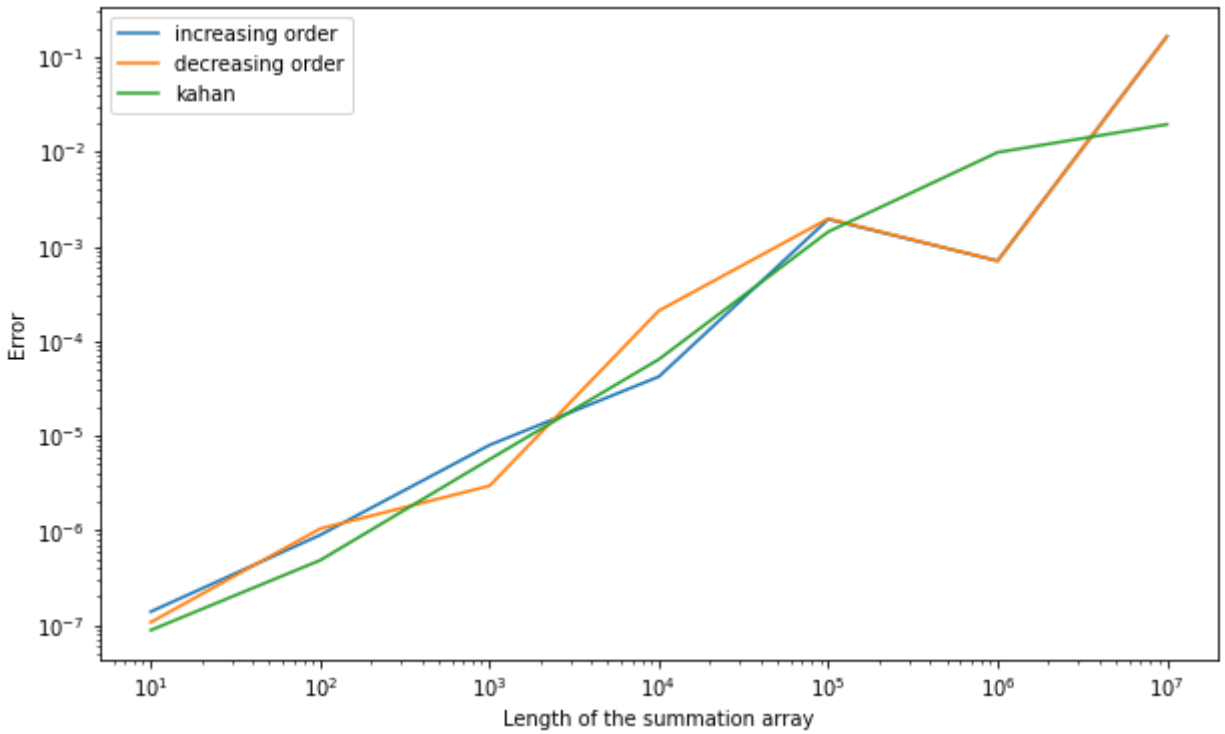
```
[8.809891660632729e-08, 4.823392814046201e-07, 5.619427952296974e-06, 6.40953230
6861365e-05, 0.001427238225005567, 0.009838833357207477, 0.019417935982346535]
```




# (d) (1.5 pts)

Sum the numbers in increasing order of magnitude and plot the error. Sum the numbers in decreasing order of magnitude and plot the error. You may use a `sort` function from NumPy or some other package. (You don't need to use the Kahan sums here.)

In [68]:
```
N_pow = [10,10**2,10**3,10**4,10**5,10**6,10**7]
error_dec = []
error_inc = []
for i in N_pow:
    rand_int = np.random.rand(1,i)
    inc = sumation_double(np.sort(rand_int),i)
    dec = sumation_double(-np.sort(-rand_int),i)
    inc_single = sumation_single(np.sort(rand_int),i)
    dec_single = sumation_single(-np.sort(-rand_int),i)
    error_dec.append(abs(dec - dec_single))
    error_inc.append(abs(inc - inc_single))

plt.figure(figsize=(10, 6))
plt.loglog(N_pow, error_inc, label="increasing order")
plt.loglog(N_pow, error_dec, label="decreasing order")
plt.loglog(N_pow, error, label="kahan")
plt.xlabel('Length of the summation array')
plt.ylabel('Error')
plt.legend()
plt.show()
```




# (e) (2 pts)

How do the methods rank in terms of accuracy? Can you explain the differences? Can you explain why the method of Kahan works? N.B.1 be precise in your explanations. Try to explain the size of any errors that are not incurred as well as of errors that are encurred. N.B.2 you are required to formulate an answer in text. You may also add computations if you feel this helps in the explanations.

Write your answer, using $\LaTeX$, in this box. We decided to use Kahan sums for both descending and ascending order. Kahn's sum algorithm is significantly more accurate than the basic method. In comparison to the Kahn sum without sorting, the sorting algorithms have

similar accuracy for both increasing and decreasing order, but for larger $n$ the error is slightly smaller for sorting algorithms (at least in our run).

The Kahan algorithm works because, it keeps a seperate variable (c in our example), which accumulates small errors to store the running time errors as the arithmetic operations are being performed. This improves accuracy by the precision of c.