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.

Karin Brinksma 13919938 Dominique Weltevreden 12161160

Run the following cell to import the necessary packages.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

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 [2]:
```
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 [3]:
```
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 [4]:
```
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 [5]:
```
S = 0.0
S += np.single(5.0)
print(type(S))

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

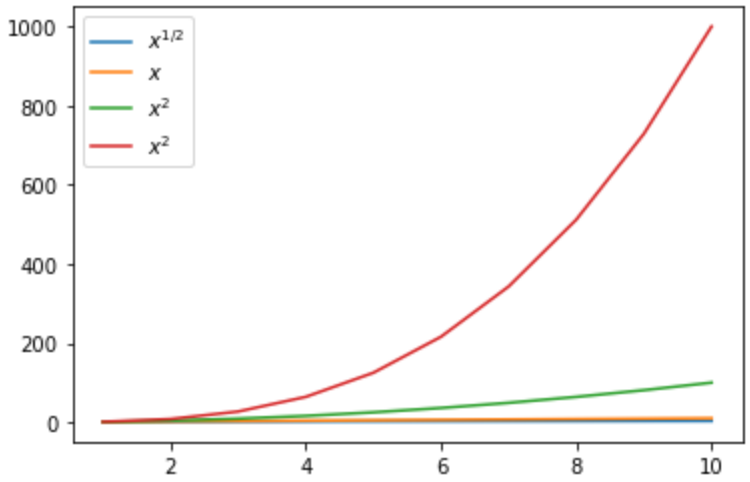
```
<class 'numpy.float64'>
<class 'numpy.float32'>
```

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 $x \in [1, 10]$.

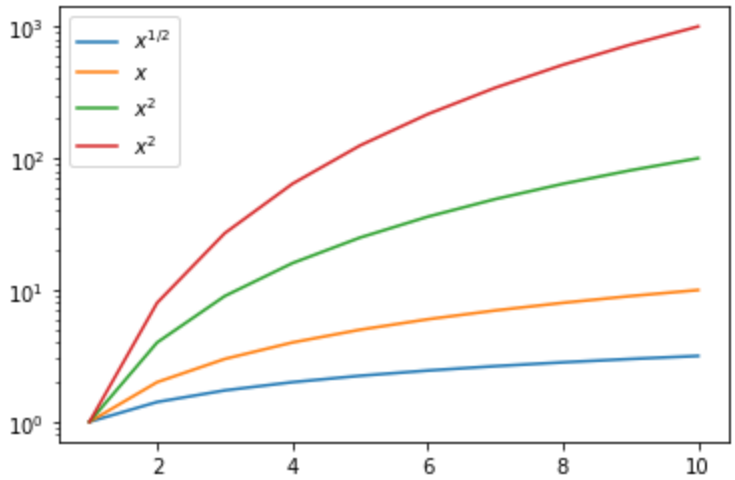
In [6]:
```
x = np.linspace(1, 10, 10)  # 10 points evenly between 1 and 10.
print(x)
plt.plot(x, x**0.5, label=r"$x^{1/2}$")
plt.plot(x, x**1, label=r"$x$")
plt.plot(x, x**2, label=r"$x^2$")
plt.plot(x, x**3, label=r"$x^2$")
plt.legend()
plt.show()
```

```
[ 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:

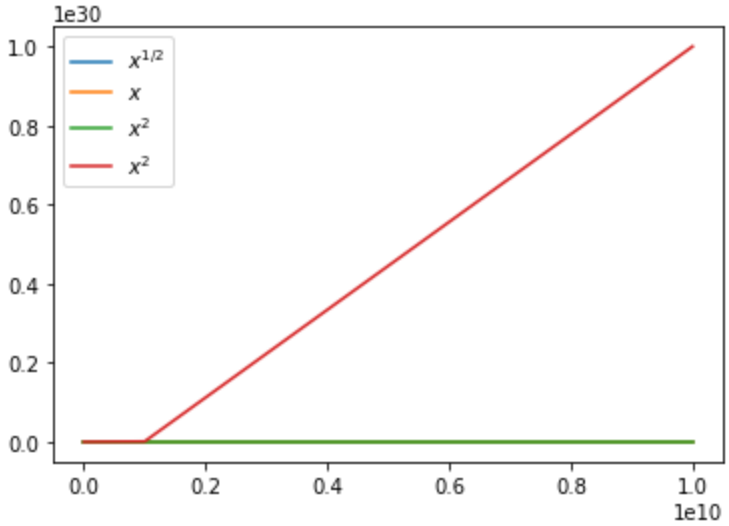
In [7]:
```
x = np.linspace(1, 10, 10)
plt.semilogy(x, x**0.5, label=r"$x^{1/2}$")
plt.semilogy(x, x**1, label=r"$x$")
plt.semilogy(x, x**2, label=r"$x^2$")
plt.semilogy(x, x**3, label=r"$x^2$")
plt.legend()
plt.show()
```

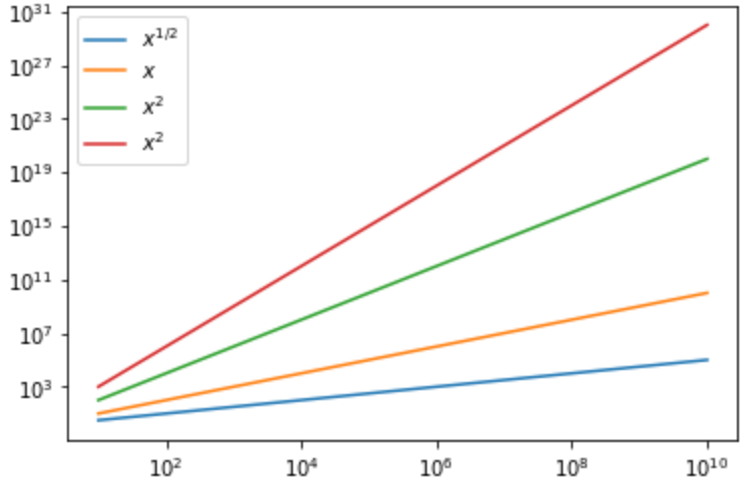


When also the x-axis contains many orders of magnitude, a log-log plot is most useful:

In [8]:
```
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 $x = 2.0$?

Using $1 - x^2/2 - \cos(x)$ for the forward error and $\arccos(\hat{y})$ for the backwards error.

In [9]:
```python
def approx_cos(x, n_terms = 2):

    if n_terms == 2:
        y_hat = 1 - (x ** 2 / 2)

    if n_terms == 3:
```

```
        y_hat = 1 - (x ** 2 / 2) + (x ** 4 / math.factorial(4))
    return y_hat


def forward_error(x, n = 2):
    y_hat = approx_cos(x, n)
    return y_hat - math.cos(x)

def backward_error(x, n = 2):
    y_hat = approx_cos(x, n)
    return math.acos(y_hat) - x

print(f"The forward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:\n"
      f"{forward_error(0.2)}\n{forward_error(1)} \n{forward_error(2)}")
print(f"The backward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:\n"
      f"{backward_error(0.2)}\n{backward_error(1)} \n{backward_error(2)}")
```

```
The forward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:
-6.657784124164401e-05
-0.040302305868139765
-0.5838531634528576
The backward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:
0.00033484232311967177
0.04719755119659785
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 $x = 2.0$?

In [10]:
```
print(f"The forward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:\n"
      f"{forward_error(0.2, 3)}\n{forward_error(1, 3)} \n{forward_error(2, 3)}")
print(f"The backward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:\n"
      f"{backward_error(0.2, 3)}\n{backward_error(1, 3)} \n{backward_error(2, 3)}")
```

```
The forward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:
8.882542501531532e-08
0.0013643607985268646
0.08281350321380904
The backward error for x = 0.2, x = 1.0 and x = 2.0, are respectively:
-4.4710234142764094e-07
-0.0016222452979235413
-0.0893667637509814
```

(d) (1 pt)

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

In [11]:
```
def cond(x):
    true_y = math.cos(x)
    rel_forward = forward_error(x, 3) / true_y
    rel_backward = backward_error(x, 3) / x
    return (abs(rel_forward) / abs(rel_backward))

print(f"The relative condition for x = 0.2, x = 1.0 and x = 2.0, is respectively:\n"
      f"{cond(0.2)}\n{cond(1)}\n{cond(2)}")
```

```
The relative condition for x = 0.2, x = 1.0 and x = 2.0, is respectively:
0.0405419623937287
1.55659591908414
4.4535724708658835
```

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 [12]:
```python
# Function that returns an array of  n  single precision random number uniformly distributed in 
def create_array_single(n):
    return np.single(np.random.rand(n))


# Sum the numbers using double precision computations in the order they are generated
def sum_array_double(array):
    summed = 0
    for idx in np.arange(0, len(array)):
        summed += np.double(array[idx])
    return summed
```

(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.

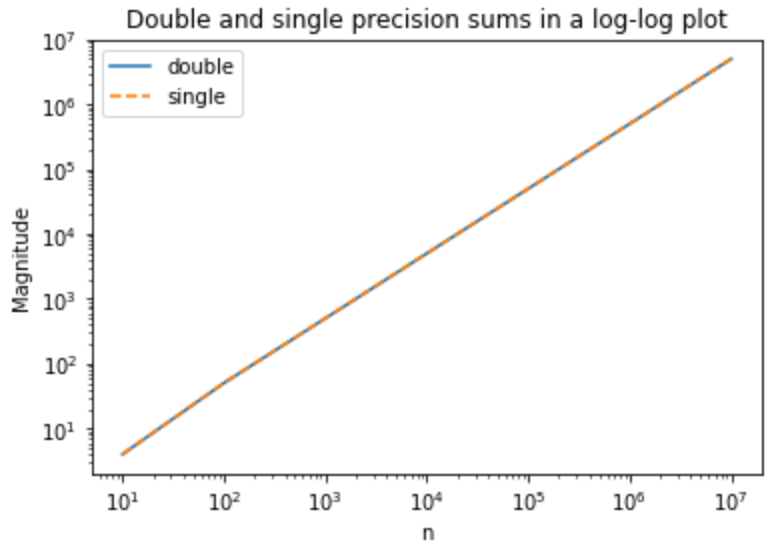
In [88]:
```python
# Function to sum the numbers in the order in which they were generated using single-precision co
def sum_array_single(array):
    summed = np.single(0)
    for idx in np.arange(0, len(array)):
        summed += np.single(array[idx])
    return summed
```

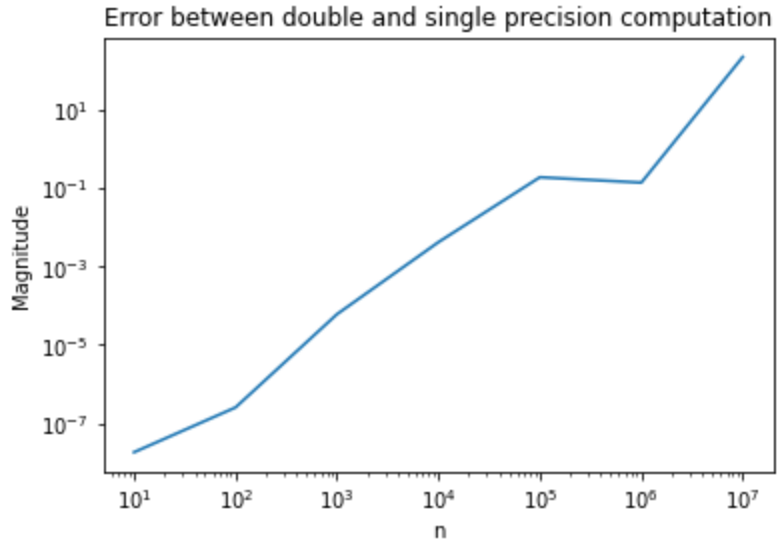
In [89]:
```python
def compute_error(kahan_summation = False, order = None):
    n_options = np.logspace(1, 7, 7, base=10)
    double_sums = []
    single_sums = []
    error = []
```

```
    for n in n_options:
        # Make n into integer
        n = int(n)

        # Generate n single precision random numbers
        array_single = create_array_single(n)

        # For 2d -> sort if specified in arguments
        if (order == "asc"):
            array_single = np.sort(array_single)
        elif (order == "desc"):
            array_single = np.sort(array_single)[::-1]


        # Sum these numbers with double precision computations
        double_summed = sum_array_double(array_single)

        # For 2c
        if kahan_summation:
            single_summed = comp_sum_alg(array_single)

        else:
            # Sum these numbers using single precision computations
            single_summed = sum_array_single(array_single)

        # Append to lists
        double_sums.append(double_summed)
        single_sums.append(single_summed)

        # Error should be absolute
        error.append(abs(double_summed - single_summed))

    return error, double_sums, single_sums
```

In [90]:
```
def plot_sums(double_sums, single_sums, title):
    n_options = np.logspace(1, 7, 7, base=10)
    plt.title(title)
    plt.loglog(n_options, double_sums, label=r"double")
    plt.loglog(n_options, single_sums, label=r"single", linestyle = 'dashed')
    plt.xlabel("n")
    plt.ylabel("Magnitude")
    plt.legend()
    plt.show()


def plot_error(error, title):
    n_options = np.logspace(1, 7, 7, base=10)
    plt.title(title)
    plt.loglog(n_options, error)
    plt.ylabel("Magnitude")
    plt.xlabel("n")
    plt.show()
```

In [91]:
```
error, double_sums, single_sums = compute_error()
plot_sums(double_sums, single_sums, "Double and single precision sums in a log-log plot")
plot_error(error, "Error between double and single precision computation")
```





(c) (1.5 pts)

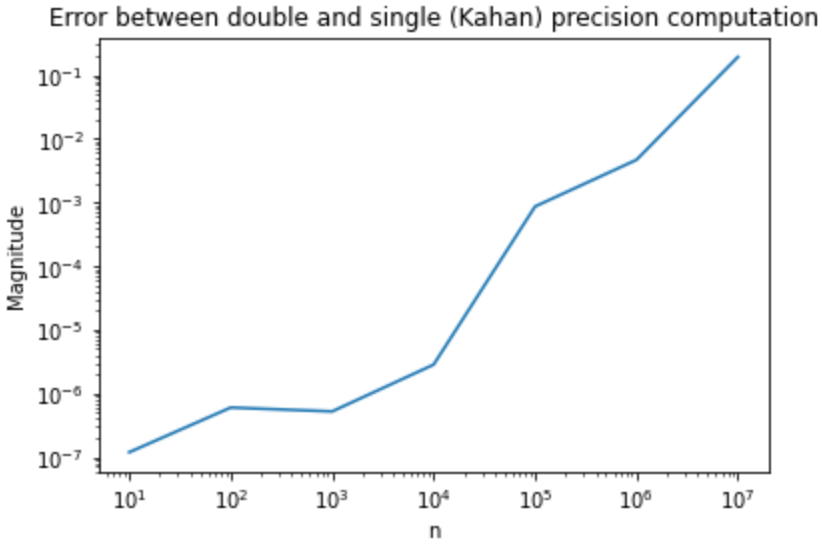
Use the following compensated summation algorithm (due to Kahan), again using only single precision, to sum the numbers in the order in which they were generated:

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

Plot the error as a function of n.

In [92]:
```
#Implementation compensated summation algorithm
def comp_sum_alg(x):
    s = np.single(x[0])
    c = np.single(0)
    for idx in np.arange(1, len(x)):
        y = np.single(x[idx]) - c
        t = s + y
        c = (t - s) - y
        s = t
    return s
```

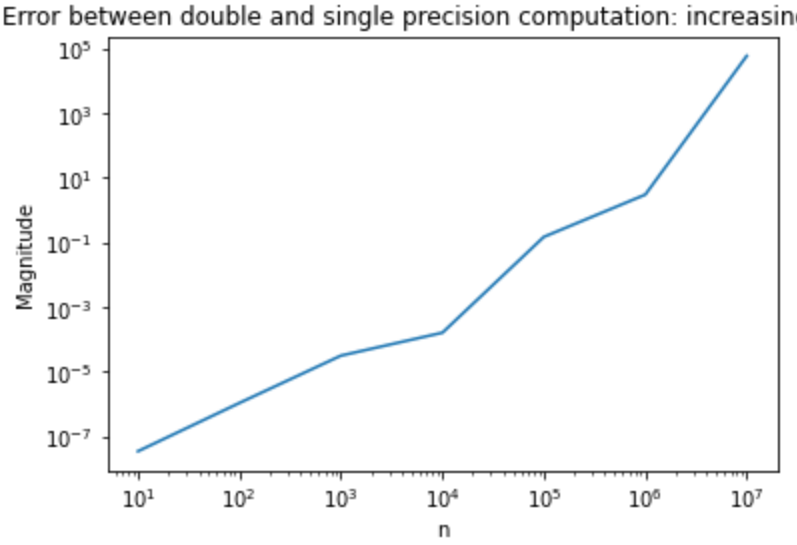
In [93]:
```
error, _, _ = compute_error(kahan_summation = True)
plot_error(error, "Error between double and single (Kahan) precision computation")
```

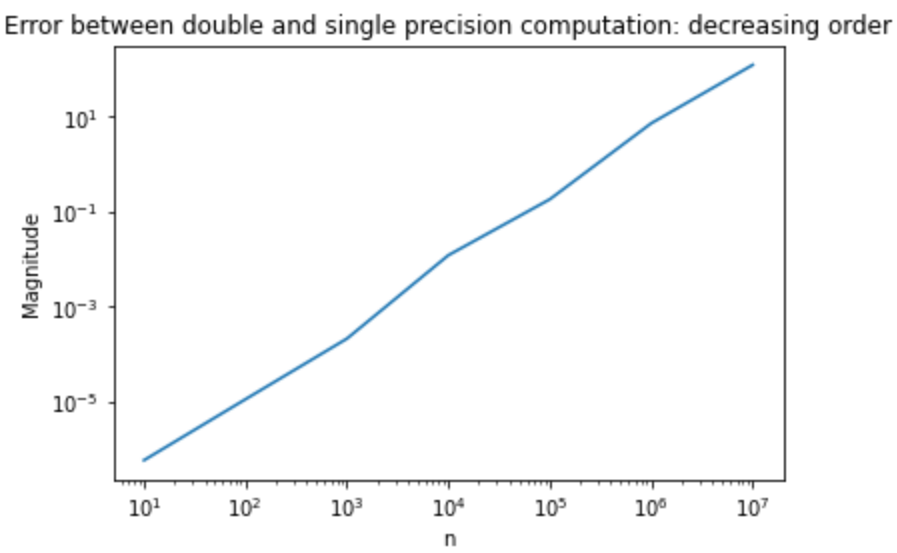


(d) (1.5 pts)

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 [94]:
```
# Increasing
error_asc, _, _ = compute_error(kahan_summation = False, order = "asc")
plot_error(error_asc, "Error between double and single precision computation: increasing order")


# Decreasing
error_desc, _, _ = compute_error(kahan_summation = False, order = "desc")
plot_error(error_desc, "Error between double and single precision computation: decreasing order"
```





(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.

> The accuracy was best for the method of Kahan for all magnitudes of N; order of generation and decreasing order of magnitude performed very similiarly, with decreasing slightly outperforming order of generation. Using increasing order of magnitude lead to the least accurate results.

> The Kahan algorithm aims to correct computation errors with the "c" variable; in the next loop of summing, the variable can adjust the value for this loop based on the rounding or truncation errors of the previous loop. This thus compensated for some of the computation error, where the other methods do nothing to compensate.

> The rest of the findings are not in line with our expectations. We would expect increasing order of magnitude to get a higher accuracy then random order and for random order to outperform decreasing order of magnitude. This is because the rounding error should be proportional to the size of the result, so by starting summing with the small numbers the intermediate results will be smaller and so the rounding order will be smaller than average. Thus we would also expect increasing order of magnitude to lead to a worse performance as the intermdiate summations produce large results. However, this is not reflected in our results.