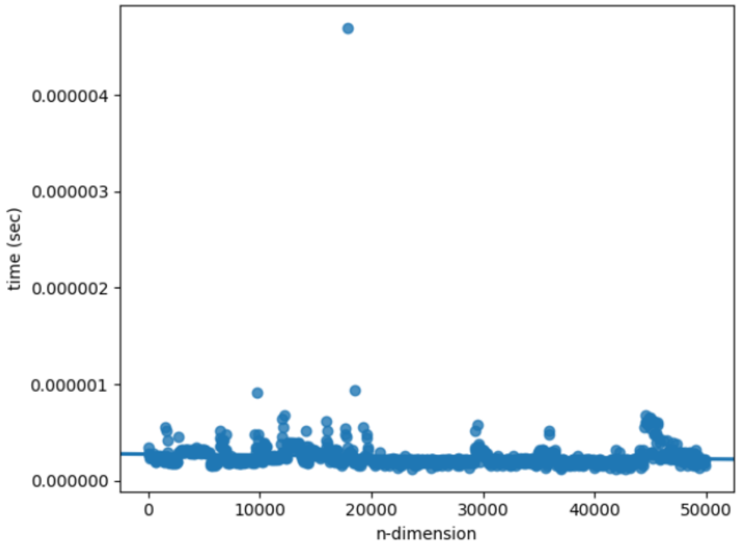
Task 1. Experimental time complexity analysis
Varvara Koshman, C4113, 24.09.2019

I.1) $f(v) = const$

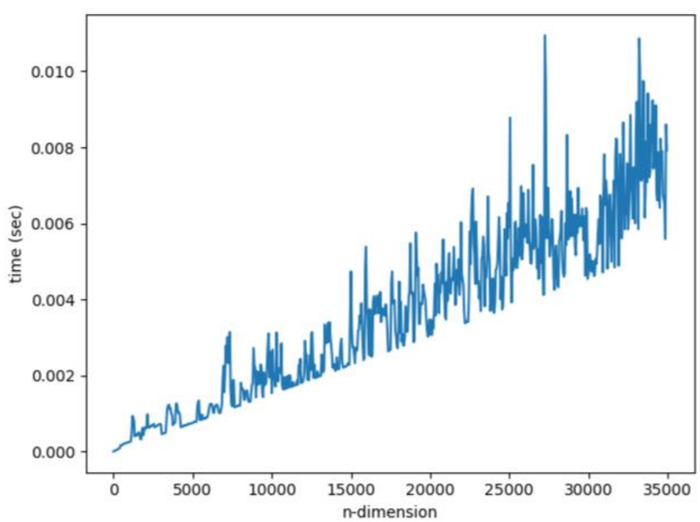
```
def f_const(v):
    return const
```

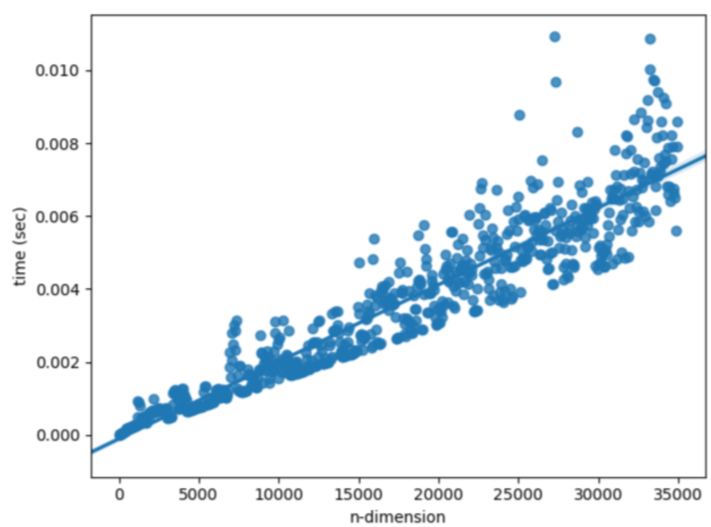


Function doesn't depend on n, so its theoretical complexity is a constant. Judging by empirical results, can be said that function can be approximated by a constant function and there is no dependence on n.
$T(n) = const$
$=> growth\ rate\ is\ O(1)$

2) $f(v) = \sum_{k=1}^{n} v_k$

```
def f_sum(v):
    sum = 0
    for i in range(len(v)):
        sum = sum + v[i]
    return sum
```





Plot showing dependence of n and average running time

Plot showing same dependence with regression curve (order = 1)

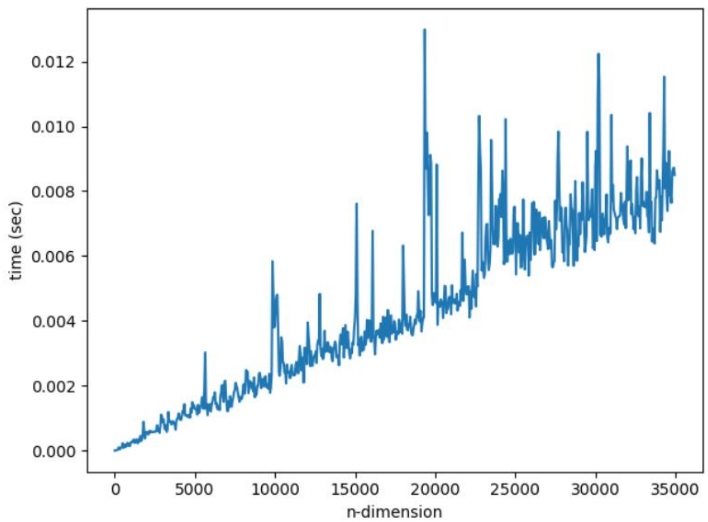
Sum of all coordinates of a vector is calculated by iterating over all n coordinates only once, so theoretical complexity is linear. Empirical results show that with growth of n, time spent on this calculation grows linearly.

$$T(n) = c_1 + c_2(n+1) + c_3 n, \qquad where\ c_1, c_2, c_3 = const$$

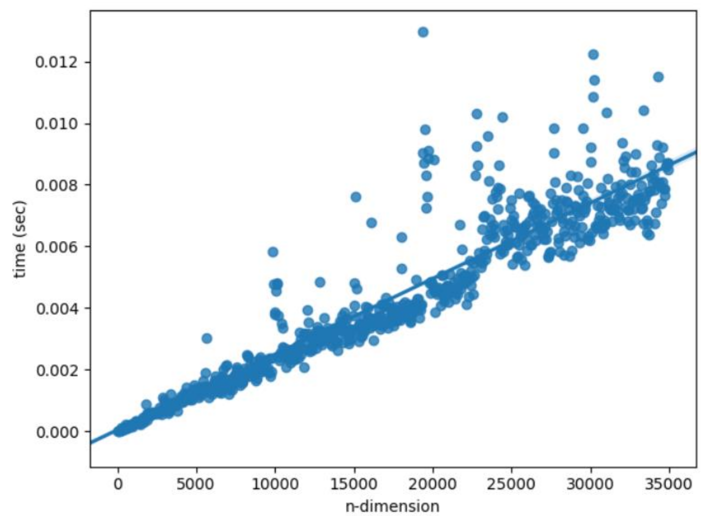
$$=> growth\ rate\ is\ O(n)$$

3) $f(v) = \prod_{k=1}^{n} v_k$

```
def f_product(v):
    product = 1
    for i in range(len(v)):
        product = product * v[i]
    return product
```



Plot showing dependence of n and average running time



Plot showing same dependence with regression curve (order = 1)

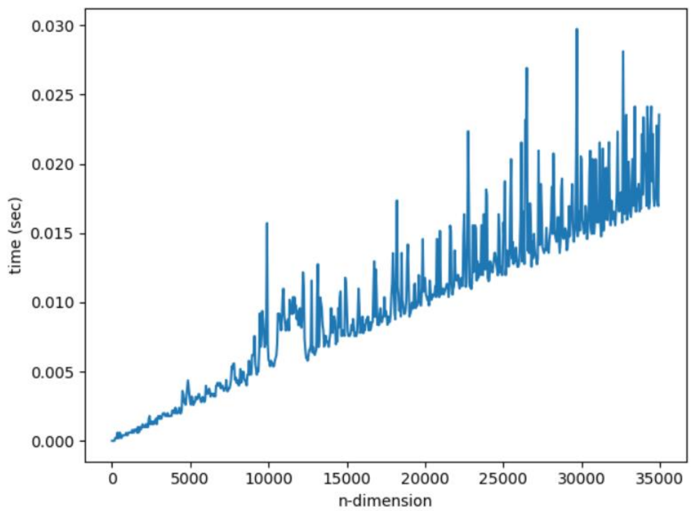
Product of all coordinates of a vector is calculated by iterating over all n coordinates only once, theoretical complexity is linear. Empirical results show that with growth of n, time spent on this calculation grows linearly.

$$T(n) = c_1 + c_2(n+1) + c_3 n, \qquad where\ c_1, c_2, c_3 = const$$

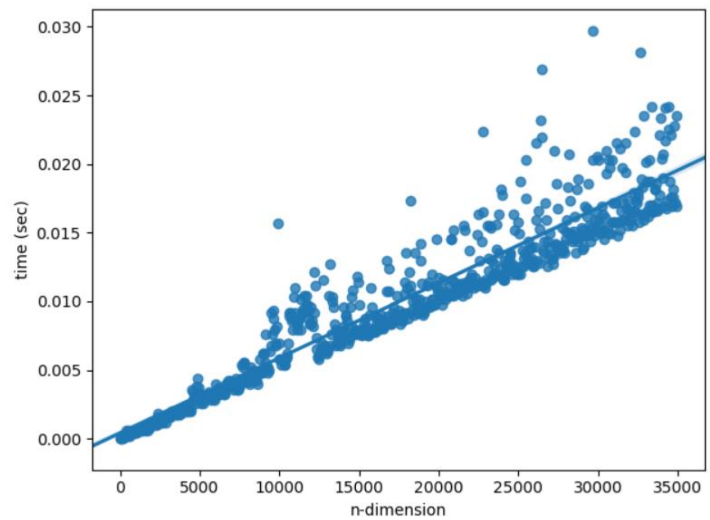
$$=> growth\ rate\ is\ O(n)$$

4) $f(v) = \sqrt{\sum_{k=1}^{n} {v^2}_k}$

```
def f_norm(v):
    sum = 0
    for i in range(len(v)):
        sum = sum + v[i] ** 2
    norm = np.sqrt(sum)
    return norm
```



Plot showing dependence of n and average running time



Plot showing same dependence with regression curve (order = 1)

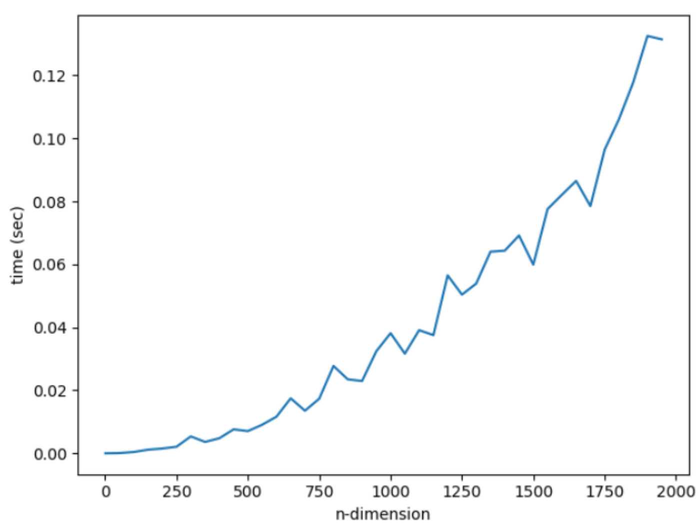
The Euclidian norm of a vector is calculated by iterating over all n coordinates only once, theoretical complexity is linear. Empirical results show that with growth of n, time spent on this calculation grows linearly.

$$T(n) = c_1 + c_2(n+1) + c_3 n + c_4, \qquad where\ c_1, c_2, c_3, c_4 = const$$

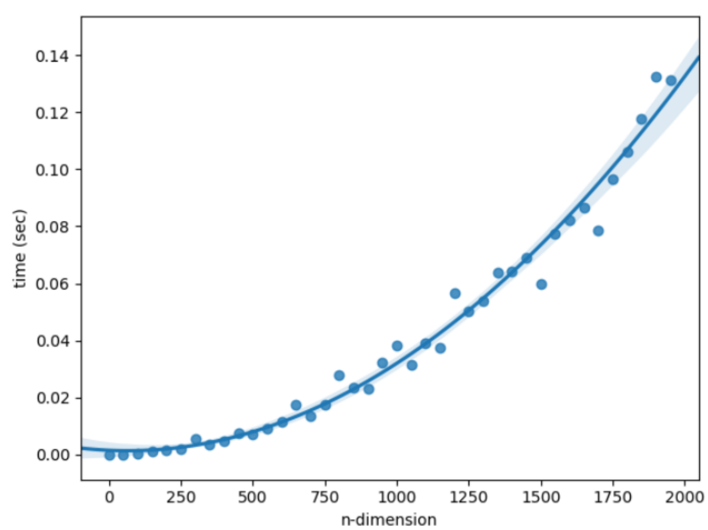
$$=> growth\ rate\ is\ O(n)$$

5) Direct calculation of polynomial P of degree $n-1$: $P(x) = \sum_{k=1}^{n} v_k x^{k-1}$ for $x = 1.5$

```
def f_polynomial_direct(v):
    p_x = 0
    for k in range(1, len(v) + 1):
        power_of_x = 1
        for _ in range(1, k):
            power_of_x = power_of_x * x
        p_x += v[k - 1] * power_of_x
    return p_x
```



Plot showing dependence of n and average running time



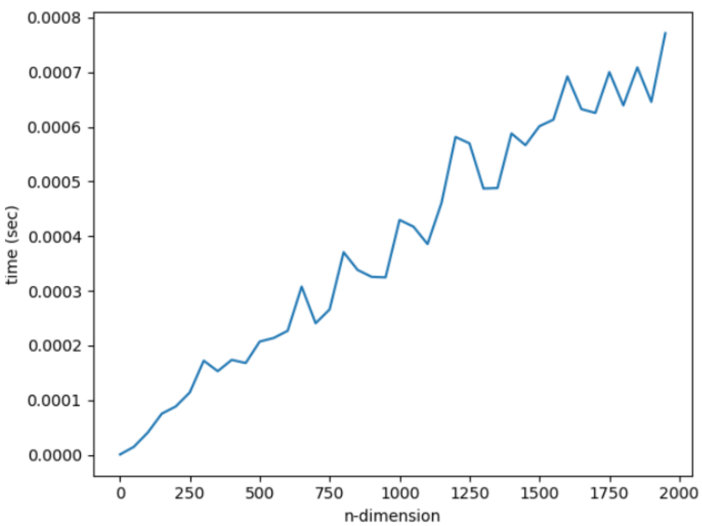
Plot showing same dependence with regression curve (order = 2)

$$T(n) = c_1 + c_2(n+1) + n(c_3 + c_4 n) + c_5 n, \qquad where\ c_1, c_2, c_3, c_4, c_5 = const$$

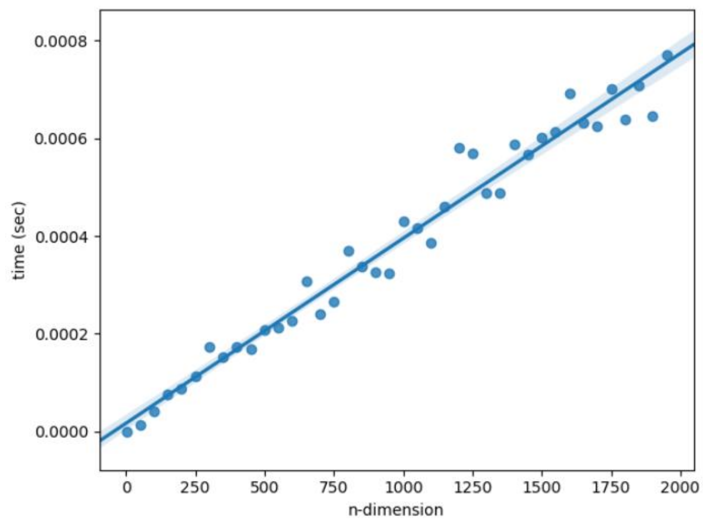
$$=> growth\ rate\ is\ O(n^2)$$

Horner's method: $P(x) = v_1 + x(v_2 + x(v_3 + \cdots))$ for $x = 1.5$

```
def f_polynomial_horner(v):
    p_x = v[0]
    for i in range(1, len(v)):
        p_x = p_x * x + v[i]
    return p_x
```



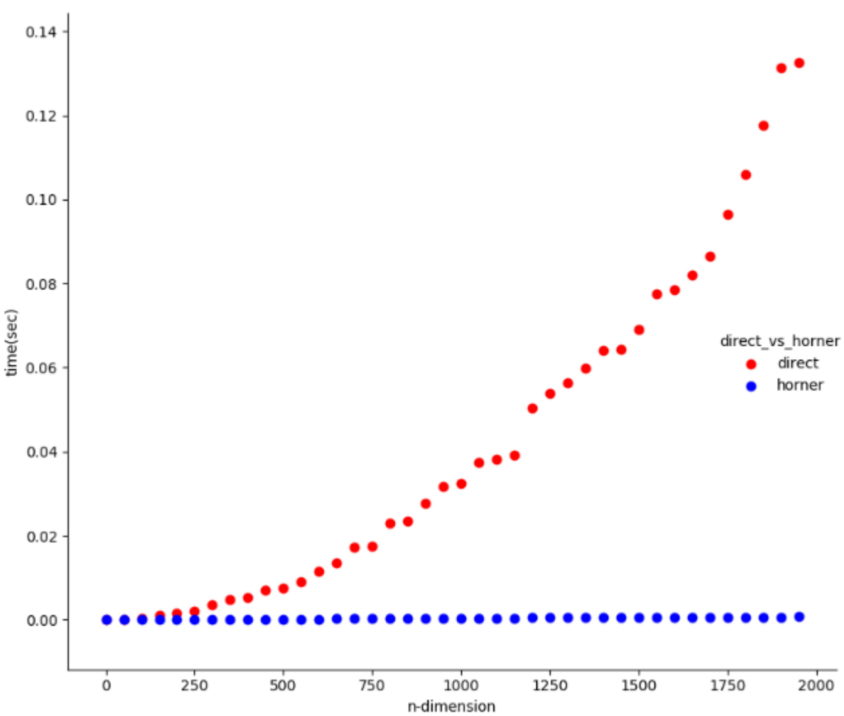
Plot showing dependence of n and average running time



Plot showing same dependence with regression curve (order = 1)

$$T(n) = c_1 + c_2(n+1), \qquad where\ c_1, c_2 = const$$
$$=> growth\ rate\ is\ O(n)$$

Comparison of two methods:



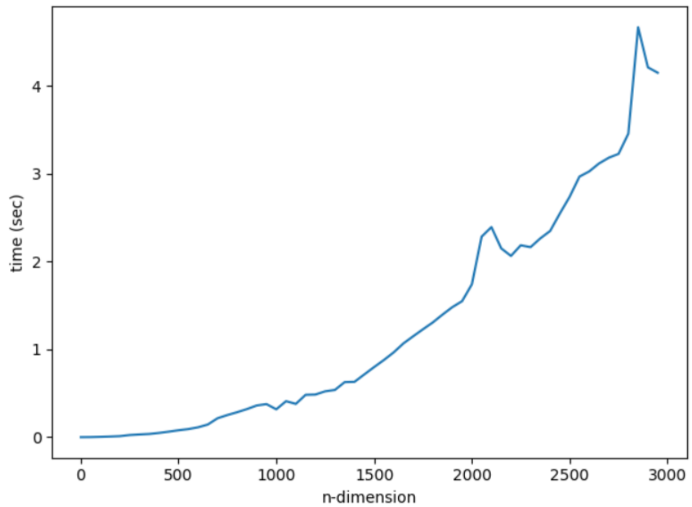
A direct way to evaluate a polynomial is n times repeatedly multiplying coordinate on evaluated x^k (evaluated in a k-loop every time). Plots above show that growth rate is $O(n^2)$, as it is expected theoretically.

Horner's method on the contrary evaluates a polynomial of degree n with n multiplications and additions – its theoretical order of growth is linear, which is empirically visible.

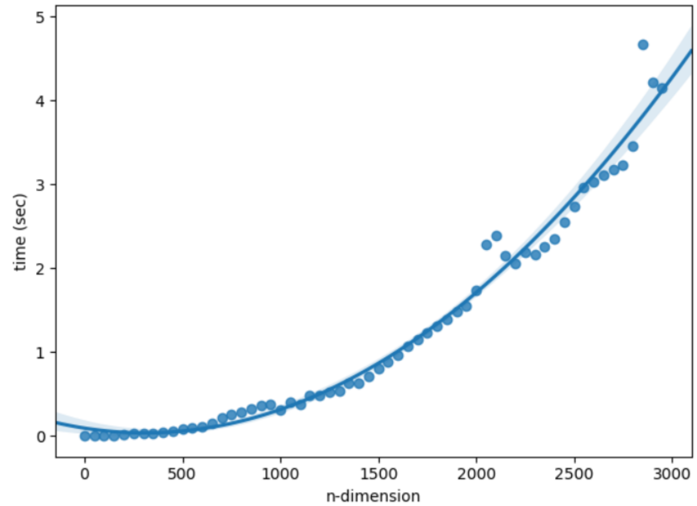
Looking at both at once it is obvious how much Horner's method is preferable.

6) The bubble sort of vector's coordinates

```
def f_bubblesort(v):
    for i in range(len(v) - 2):
        for j in range(len(v) - 2):
            if v[j] > v[j + 1]:
                v[j], v[j + 1] = v[j + 1], v[j]
    return v
```



Plot showing dependence of n and average running time



Plot showing same dependence with regression curve (order = 2)

$T(n) = n^2(c_1 + c_2), \qquad where\ c_1, c_2 = const$

$=> growth\ rate\ is\ O(n^2)$,

The bubble sort of a vector's coordinates is calculated by iterating over all n coordinates and comparing each coordinate with $n - 1$ left, so theoretical complexity is $O(n^2)$. Empirical results show that with growth of n, time spent on this calculation grows as expected.

II. $A(n \times n)$ and $B(n \times n)$ matrix multiplication (naive approach)

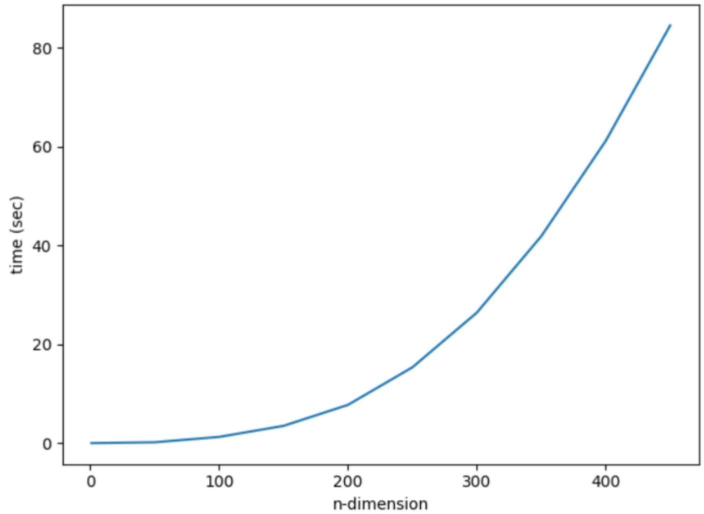
```
def matrix_product(n):
    A = [[random.randrange(10) for _ in range(n)] for _ in range(n)]
    B = [[random.randrange(10) for _ in range(n)] for _ in range(n)]
    product = [[0 for _ in range(len(A))] for _ in range(len(A))]
    for i in range(len(B)):
        for j in range(len(B)):
            for k in range(len(B)):
                product[i][j] += A[i][k] * B[k][j]
    # product = np.matmul(A, B)
    return product
```

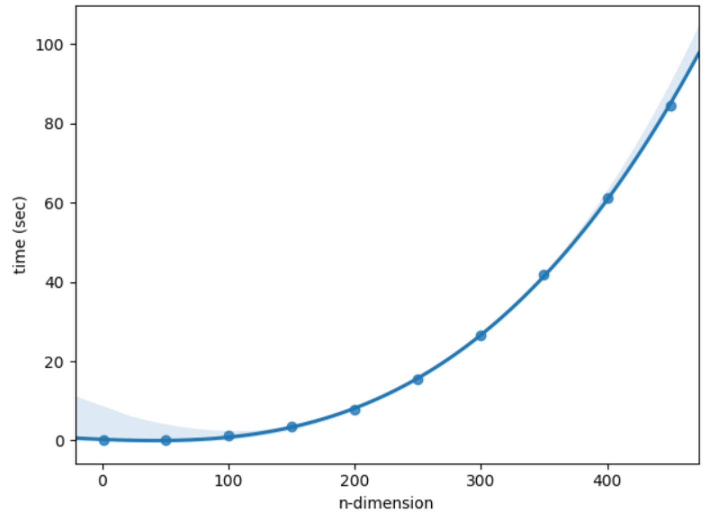
Naive approach to matrix multiplication is a result of multiplying each element of each row of first matrix (A) on each element of each column of second matrix (B). This approach requires 3 iterations over all n dimensions, so theoretical time complexity is cubic.

$T(n) = c_1 + c_2 + c_3 + n^3c_4, \qquad where\ c_1, c_2, c_3, c_4 = const$

$=> growth\ rate\ is\ O(n^3)$

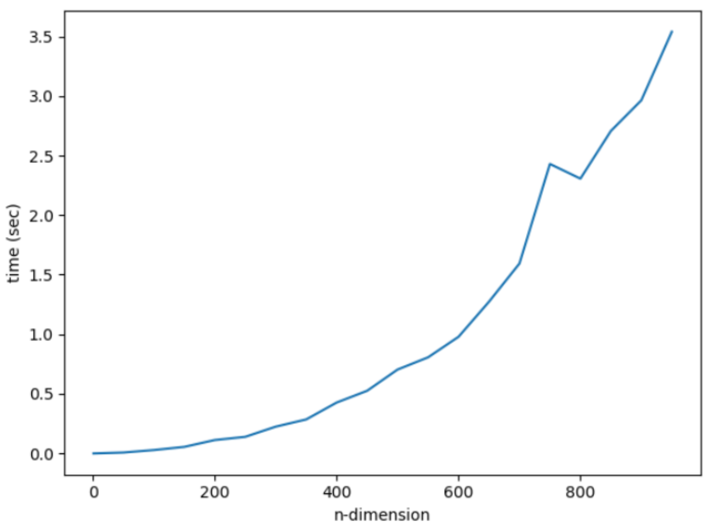
Results of empirical experiment show cubic order of growth.

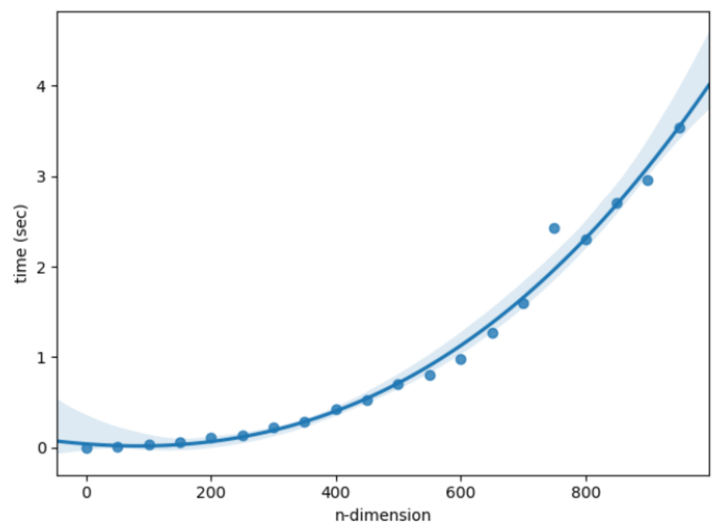




Plots showing dependence of n and average running time

Plots showing same dependence with regression curve (order = 3)





numpy's implementation (above) however appears to have a much better performance on [1, 450].

Utility code:

```
import numpy as np
from timeit import default_timer as timer
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import random
from scipy import stats

const = np.random.randint(0, 1, 1)  # const for const function
n_bound = 550  # dimension of a vector
k_iterations = 5  # number of runs for each experiment
right_upper_bound = 10  # right upper bound for sampling from uniform distribution
x = 1.5  # value of parameter x for polynomial evaluation
step = 50  # step for choosing n-dimension

# initialization of arrays containing average time for each n (n given with step 50)
time_const = np.zeros(n_bound // step)
```

```python
time_sum = np.zeros(n_bound // step)
time_product = np.zeros(n_bound // step)
time_norm = np.zeros(n_bound // step)
time_polynomial_direct = np.zeros(n_bound // step)
time_polynomial_horner = np.zeros(n_bound // step)
time_bubblesort = np.zeros(n_bound // step)
time_matrix_product = np.zeros(n_bound // step)

n_list = [i for i in range(0, n_bound, step)]  # list of all dimensions
n_list[0] = 1


def get_time(function, vec):
    start_time = timer()
    _ = function(vec)
    end_time = timer() - start_time
    return end_time

def plot_graph(empirical):
    plt.plot(n_list, empirical)  # plot just empirical
    plt.xlabel('n-dimension')
    plt.ylabel('time (sec)')
    plt.show()
    # order parameter was varied for each example
    sns.regplot(x=n_list, y=empirical, order=1)  # plot empirical approximated by theoretical
    plt.xlabel('n-dimension')
    plt.ylabel('time (sec)')
    plt.show()


def quantile_plot(x, y, **kwargs):
    _, xr = stats.probplot(x, fit=False)
    _, yr = stats.probplot(y, fit=False)
    plt.scatter(xr, yr, **kwargs)


def plot_comparing_polynomials(time_polynomial_direct, time_polynomial_horner):
    direct = pd.DataFrame({'x1': n_list, 'y1': time_polynomial_direct})
    horner = pd.DataFrame({'x2': n_list, 'y2': time_polynomial_horner})
    direct_vs_horner = pd.concat([direct.rename(columns={'x1': 'n-dimension', 'y1': 'time(sec)'})
                             .join(pd.Series(['direct'] * len(direct),
name='direct_vs_horner')),
                              horner.rename(columns={'x2': 'n-dimension', 'y2': 'time(sec)'})
                             .join(pd.Series(['horner'] * len(horner),
name='direct_vs_horner'))],
                             ignore_index=True)
    pal = dict(direct="red", horner="blue")
    g = sns.FacetGrid(direct_vs_horner, hue='direct_vs_horner', palette=pal, size=7)
    g.map(quantile_plot, "n-dimension", "time(sec)")
    g.add_legend()
    plt.show()


def main():
    for n in n_list:
        # generate vector of size n taken from uniform distribution [1,right_upper_bound)
        vector = np.random.uniform(1, right_upper_bound, n)
        # initialize arrays storing time for each run for a particular n
        time_const_temp = np.zeros(k_iterations)
        time_sum_temp = np.zeros(k_iterations)
        time_product_temp = np.zeros(k_iterations)
```

```python
        time_norm_temp = np.zeros(k_iterations)
        time_polynomial_direct_temp = np.zeros(k_iterations)
        time_polynomial_horner_temp = np.zeros(k_iterations)
        time_bubblesort_temp = np.zeros(k_iterations)
        time_matrix_product_temp = np.zeros(k_iterations)
        for k in range(k_iterations):
            time_const_temp[k] = get_time(f_const, vector)
            time_sum_temp[k] = get_time(f_sum, vector)
            time_product_temp[k] = get_time(f_product, vector)
            time_norm_temp[k] = get_time(f_norm, vector)
            time_polynomial_direct_temp[k] = get_time(f_polynomial_direct, vector)
            time_polynomial_horner_temp[k] = get_time(f_polynomial_horner, vector)
            time_bubblesort_temp[k] = get_time(f_bubblesort, vector)
            time_matrix_product_temp[k] = get_time(matrix_product, n)
        time_const[n // step] = np.average(time_const_temp)
        time_sum[n // step] = np.average(time_sum_temp)
        time_product[n // step] = np.average(time_product_temp)
        time_norm[n // step] = np.average(time_norm_temp)
        time_polynomial_direct[n // step] = np.average(time_polynomial_direct_temp)
        time_polynomial_horner[n // step] = np.average(time_polynomial_horner_temp)
        time_bubblesort[n // step] = np.average(time_bubblesort_temp)
        time_matrix_product[n // step] = np.average(time_matrix_product_temp)
    plot_graph(time_const)
    plot_graph(time_sum)
    plot_graph(time_product)
    plot_graph(time_norm)
    plot_graph(time_polynomial_horner)
    plot_comparing_polynomials(time_polynomial_direct, time_polynomial_horner)
    plot_graph(time_bubblesort)
    plot_graph(time_matrix_product)


if __name__ == '__main__':
    main()
```