דף סיכום בחינה

מזהה סטודנט: 314811290

מזהה קורס: 0366210504 - אנליזה נומרית0366210504 - Analysis Numerical   שם קורס: מטלת אמצע

מספר שאלה	ניקוד מירבי	ציון
1.1	7.00	7.00
1.2	3.00	3.00
1.3	7.00	7.00
1.4	10.00	10.00
1.5	8.00	8.00
2.1	10.00	10.00
2.2	10.00	10.00
2.3	10.00	10.00
3.1	4.00	4.00
3.2	4.00	4.00
3.3	14.00	9.00
3.4	3.00	3.00
3.5	10.00	6.00

ציון בחינה סופי : 91.00

הבחינה הבדוקה בעמודים הבאים

אנליזה נומרית – מטלת אמצע

שם: מוחמד דגש

ת"ז: 314811290

שאלה 1

קטע קוד:

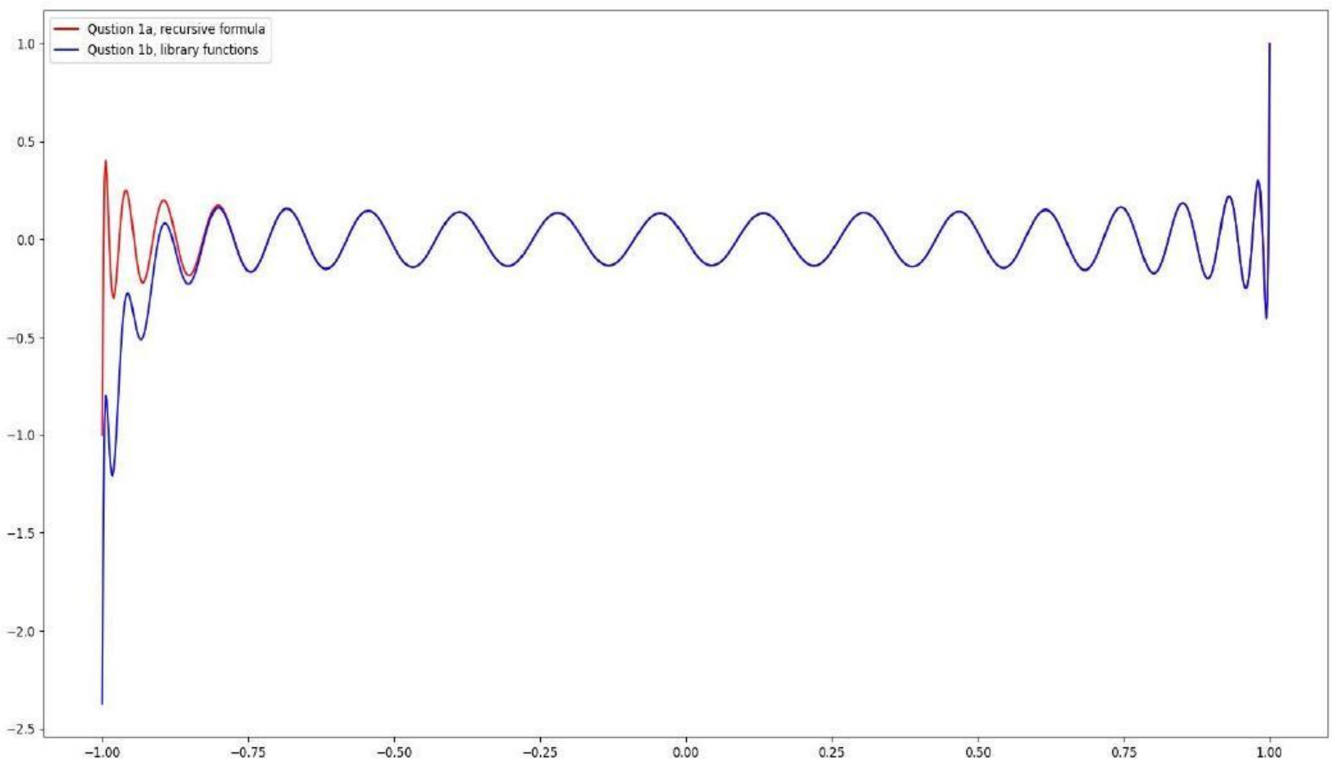
```
import numpy as np
from scipy.special import legendre
import matplotlib.pyplot as plt

#Question 1a, auxiliary function
def recursive_legendre(x, p_curr, p_prev, n):
    p_next = ((2*n+1)/(n+1))*x*p_curr - (n/(n+1))*p_prev
    return p_next

#Question 1a, auxiliary function
#normalized=1 to test part 1c
def recursive_pk(k, x, normalized=0):

    p0 = 1
    p1 = x

    p_prev = p0
    p_curr = p1
    for i in range(1, k):
        p_curr_saved = p_curr
        if normalized:
            p_curr = recursive_legendre_normalized(x, p_curr, p_prev,
i)
        else:
            p_curr = recursive_legendre(x, p_curr, p_prev, i)
        p_prev = p_curr_saved

    return p_curr




#Question 1b
def direct_pk(k, x):
    legendre_coeffs = legendre(k).c
    return np.polyval(legendre_coeffs, x)




#Question 1a+1b, main
def question1_main():
    k = 35
```

```python
    n_pts = 1000
    x = np.linspace(-1, 1, num=n_pts)

    p35_recursive = recursive_pk(k, x)
    p35_direct = direct_pk(k, x)

    do_plot = 1
    if do_plot:
        plt.plot(x, p35_recursive, '-r', label='Qustion 1a, recursive
formula')
        plt.plot(x, p35_direct, '-b', label='Qustion 1b, library
functions')
        plt.legend(loc="upper left")
        plt.show()




#Question 1c, auxiliary function to test
def recursive_legendre_normalized(x, p_curr, p_prev, n):
    p_next = x*p_curr - (n**2/(4*(n**2)-1))*p_prev
    return p_next

#Question 1c, test
def question_1c_test():
    k = 4
    n_pts = 1000
    x = np.linspace(-1, 1, num=n_pts)
    if k == 4:
        ptilda = ((35*(x**4) - 30*(x**2) + 3)/8)*(8/35)
    elif k == 2:
        ptilda = ((3*(x**2)-1)/2)*(2/3)
    else:
        ptilda = ((5*(x**3)-3*x)/2)*(2/5)
    p4_tilda_recursive = recursive_pk(k, x, normalized=1)
    diff = np.amax(np.abs(ptilda - p4_tilda_recursive))

    return diff




#question 1d
def find_weights_and_partition(n):
    n_range = np.arange(n)
    gammas = n_range / np.sqrt(4*(n_range**2)-1) #according to 1c
    coeff_mat = np.zeros((n, n))
    for i in range(n-1):
        coeff_mat[i, i+1] = gammas[i+1]
        coeff_mat[i+1, i] = gammas[i+1]

    xs, vs = np.linalg.eig(coeff_mat) #eigenvalues, i.e. the roots of
the Legendre polynomials; eigenvectors, to compute the weights
    v_norms = np.linalg.norm(vs, axis=0)
    vs_normalized = vs/v_norms
    lambda0_squared = 2
    vi1s = vs_normalized[0, :]
    ws = lambda0_squared * vi1s**2

    return xs, ws
```

```python
def question_1d_main():
    k = 35
    xs, ws = find_weights_and_partition(k)
    p35_gauss = recursive_pk(k, xs)
    sum_ws = np.sum(ws)

    return p35_gauss, sum_ws




#Question 1e auxiliary
def general_interval(a, b, ws, xs, f):
    half_len = (b-a)/2
    center = (b+a)/2
    xs_new = half_len*xs+center
    ys = f(xs_new)
    return half_len * np.sum(ws * ys)

#Question 1e auxiliary
def question_1e_per_n_gauss(n, a, b, f):
    xs, ws = find_weights_and_partition(n)

    gauss_sum = general_interval(a, b, ws, xs, f)
    return gauss_sum

#Question 1e auxiliary
#assumes n is odd and n>1
def question_1e_per_n_simpson(n, a, b, f):

    xs = np.linspace(a, b, num=n)
    edges = f(a) + f(b)
    xs_no_edges = xs[1:-1]
    odd_pts = xs_no_edges[1::2]
    odd_sum = np.sum(f(odd_pts))
    even_pts = xs_no_edges[0::2]
    even_sum = np.sum(f(even_pts))
    h = (b-a)/(n-1)
    sum_simp = (edges + 2*odd_sum + 4*even_sum)*(h/3)

    return sum_simp

def question_1e_main():

    a = 1
    b = 1.2

    f = lambda x: np.cos(x**24)
    exact_integral = -0.01521513770698
    n_list = [11, 15, 21, 31, 51, 101, 201, 301, 401, 601, 801, 999]
    error_gauss, error_simp = [0]*len(n_list), [0]*len(n_list)

    for i, n in enumerate(n_list):
        integral_approx_gauss = question_1e_per_n_gauss(n, a, b, f)
        error_gauss[i] = integral_approx_gauss - exact_integral
        integral_approx_simp = question_1e_per_n_simpson(n, a, b, f)
        error_simp[i] = integral_approx_simp - exact_integral

    do_plot = 1
    if do_plot:
```

```
        plt.plot(n_list, np.abs(error_gauss), 'r', label='Gauss
Error')
        plt.plot(n_list, np.abs(error_simp), 'b', label='Simpson
Error')
        plt.legend(loc="upper left")
        plt.show()

    #checking Gaussian error
    integral_approx_gauss = question_1e_per_n_gauss(66, a, b, f)
    error_gauss_minimal = integral_approx_gauss - exact_integral

    #checking Simpson error
    hs = (b-a)/(np.array(n_list)-1)
    error_normalized = np.array(error_simp)/(hs**4)
    plt.plot(n_list, np.abs(error_normalized), 'r')
    plt.show()


if __name__ == '__main__':
    part = 'd'
    if part == 'c':
        question_1c_test()
    elif part == 'd':
        question_1d_main()
    elif part == 'e':
        question_1e_main()
    else:
        question1_main()
```

<u>סעיף א' + ב' – ציור גרפים:</u>



סעיף ב':

השוואות בין התוצאות:

התוצאות מאוד דומות בקטע [1 , 0.75-].

הנוסחה הרקורסיבית נותנת ערכים גדולים יותר בקטע [1- , 0.75].

הסיבה להבדל הוא שבשיטה הרקורסיבית נצברת שגיאה בכל איטרציה.

סעיף ג':

עבור פולינומי לג'נדר לא מנורמליים ידוע כי:

$$(*)\ P_{n+1}(x) = \left(\frac{2n+1}{n+1}\right) x P_n(x) - \left(\frac{n}{n+1}\right) P_{n-1}(x).$$

$$(n = 1,2,\dots)$$

P_n פולינום ממעלה n, P_{n-1} פולינום ממעלה n-1 .

לכן ברור שאם נסמן ב- a_k את המקדם המוביל של P_k, נקבל:

$$a_{n+1} = \left(\frac{2n+1}{n+1}\right)a_n$$

ובאופן דומה :

$$a_n = \left(\frac{2n-1}{n}\right)a_{n-1}$$

(נתקף גם עבור n=1 שכן a1=a0=1 , P0=1 , P1=x) לכן:

$$(\#)\ a_n = \left(\frac{n+1}{2n+1}\right)a_{n+1},\ a_n = \left(\frac{2n-1}{n}\right)a_{n-1}$$

נחלק את (*) ב- a_n:

$$(**)\ \left(\frac{P_{n+1}(x)}{a_n}\right) = \left(\frac{2n+1}{n+1}\right) x \acute{P}n(x) - \left(\frac{n}{n+1}\right)\left(\frac{P_{n-1}(x)}{a_n}\right)$$

נציב את a_n מ- (#):

$$\left(\frac{P_{n+1}(x)}{\left(\frac{n+1}{2n+1}\right)a_{n+1}}\right) = \left(\frac{2n+1}{n+1}\right)x\acute{P}_n(x) - \left(\frac{n}{n+1}\right)\left(\frac{P_{n-1}(x)}{\left(\frac{2n-1}{n}\right)a_{n-1}}\right)$$

$$\left(\frac{2n+1}{n+1}\right)\acute{P}_{n+1}(x) = \left(\frac{2n+1}{n+1}\right)x\acute{P}_n(x) - \left(\frac{n^2}{(n+1)(2n-1)}\right)\acute{P}_{n-1}(x)$$

נחלק ב- $\left(\frac{2n+1}{n+1}\right)$:

$$\acute{P}_{n+1}(x) = x\acute{P}_n(x) - \left(\frac{n^2}{4n^2-1}\right)\acute{P}_{n-1}(x)$$

לכל n=1,2,...

וסיימנו, כי ניקח $\gamma_n=\frac{n}{\sqrt{4n^2-1}}$

, $\beta_n=0$.

<u>סעיף ד':</u>

הפונקציה מצורפת, זה נעשה דרך:

$$J_n = \begin{bmatrix} \beta_0 & \gamma_1 & & & \\ \gamma_1 & \beta_1 & \gamma_2 & & \\ & \gamma_2 & \ddots & \ddots & \\ & & \ddots & \beta_{n-2} & \gamma_{n-1} \\ & & & \gamma_{n-1} & \beta_{n-1} \end{bmatrix}$$

שנק' החלוקה $Z_{i,n}$ הן הערכים העצמיים שלו והמשקלים נתונים על ידי :

$$W_i = (\chi_0)^2 \left(\tilde{V}_{i,1}\right)^2$$

כאשר :

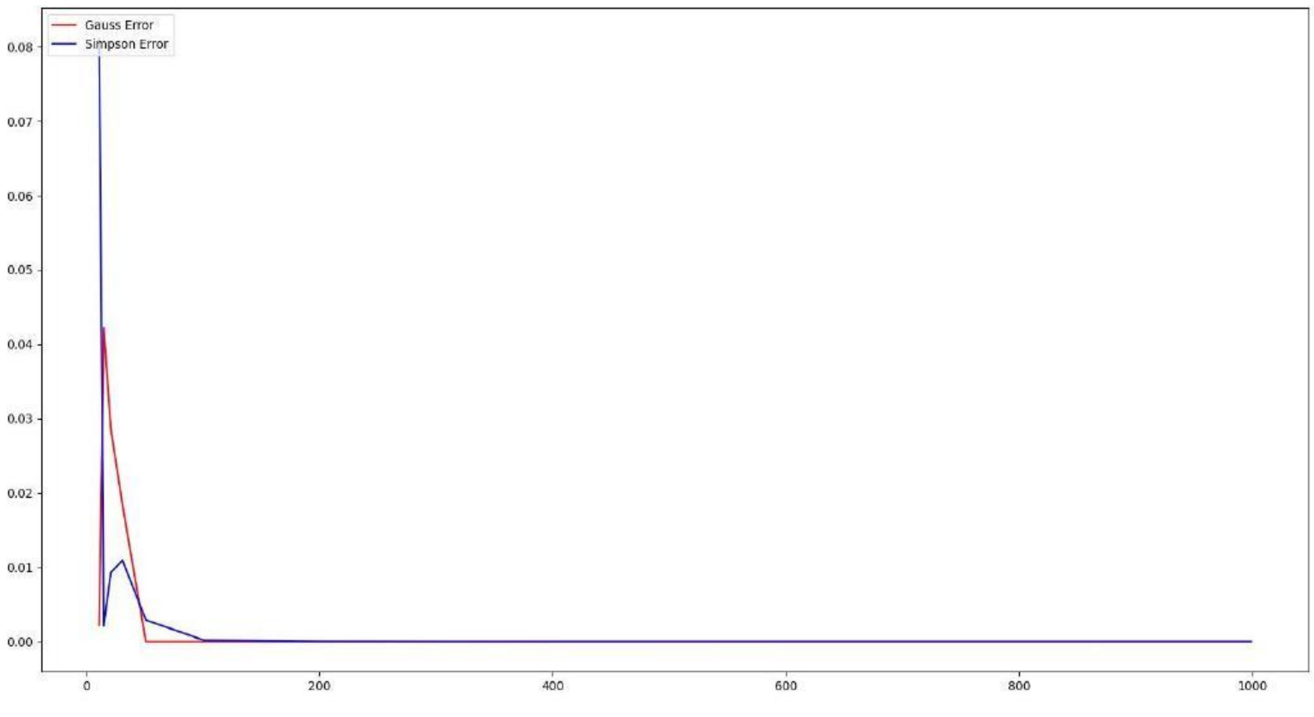
$$(\chi_0)^2 = < \acute{P}_0 , 1 > = \int_{-1}^{1} dx = 2$$

ו- $\tilde{V}_{1,1} \ldots \tilde{V}_{n,1}$ הם האיברים הראשונים של הווקטורים העצמיים המנורמלים.

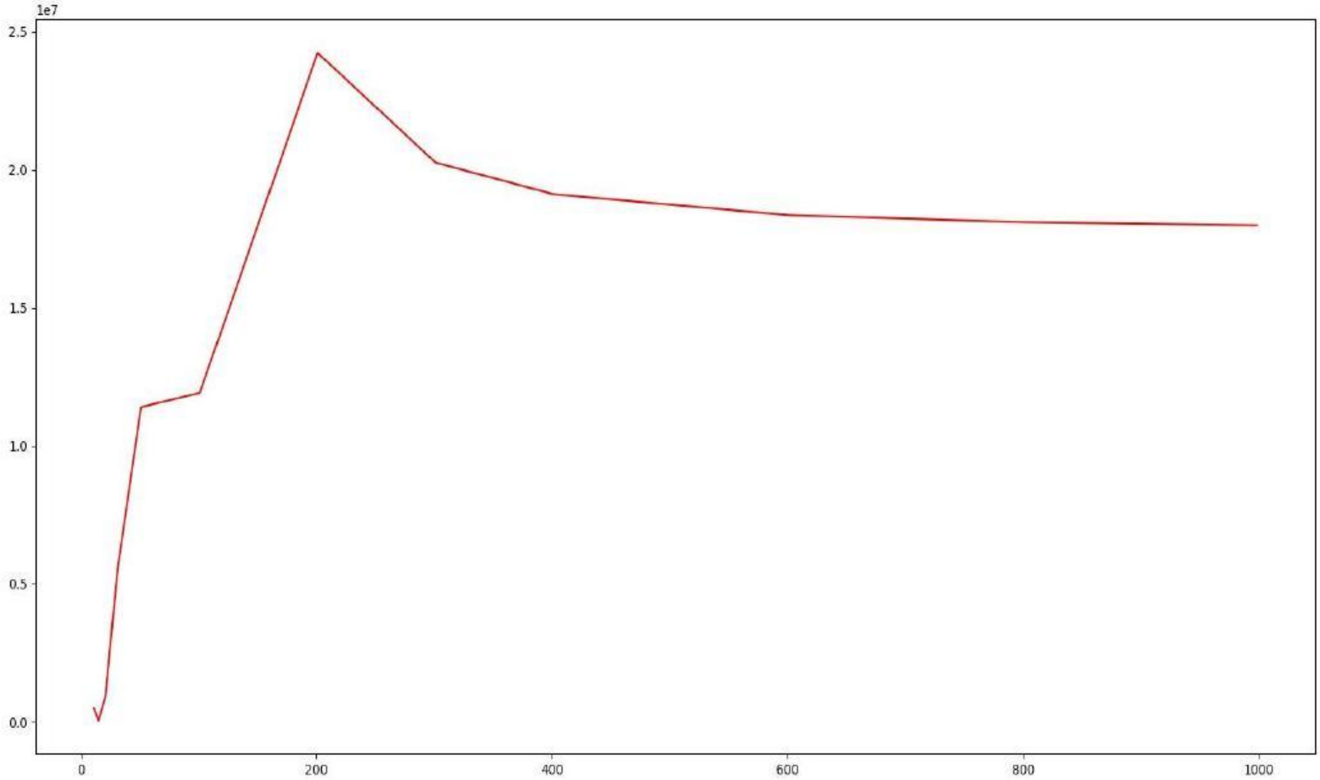
<u>סעיף ה':</u>

הפונקציה מצורפת.

הגרף :



לגבי שגיאת Simpson, מצורף תרשים של $(\frac{error-simp}{h^4})$



רואים שהגרף שואף לאסימפטוטה, מה שמוכיח שהשגיאה אכן דועכת בקצב שנלמד בכיתה.

לגבי שגיאת גאוס, רואים שעבור n=65 השגיאה בסדר גודל של 10^{-14} אם גדול היותר, אבל עבור n=66 השגיאה כבר בסדר גודל של 10^{-16}.

שאלה 2

<u>קטע הקוד:</u>

```
from scipy.special import factorial


#Question 2a, auxiliary function
def f(n):
    return n**n/factorial(n)

#Question 2a
def question2a():
    k = 1001
    ns = list(range(1, k+1))
    for n in ns:
```

```python
        print('(', n, ',', f(n), ')')



#Question 2b, auxiliary function
def newf(n):
    newf_n = 1
    for k in reversed(range(1, n)):
        newf_n *= (n/k)
    return newf_n

#Question 2b
def question2b():
    k = 1001
    ns = list(range(1, k+1))
    for n in ns:
        print('(', n, ',', newf(n), ')')


#Question 2b, auxiliary for log2 computations
def log_factorial(n):
    import math
    log_sum = 0
    for i in range(1, n+1):
        log_sum += math.log2(i)
    return log_sum


if __name__ == '__main__':
    part_a = 0
    if part_a:
        question2a()
    else:
        question2b()
```

<u>סעיף א':</u>

n^n מסוג int , הפונקציה שלנו f מחזירה float .

במהלך הדפסה השגיאה מתקבלת ב-144

"Overflow Error : int too large to convert to float"

הסיבה לשגיאה: בעת החישוב נדרש לחלק int n^n שהוא ענק בשלב מסוים , ב- float n! .

לצורך כך נדרש להמיר את n^n ל- float .

כשממירים int ל- float , נדרש לייצג את המספר:

$(-1)^s\ 2^{c-1023}\ (1+f)$, c < 2047 , ולכן אם ה- int גדול מדי ההמרה לא מתאפשרת ומקבלים שגיאה.

אם נדייק: $144\ \log_2(144)=1032+\varepsilon_1$

$$log_2(144^{144}) = 144\log_2(144) = 1032 + \varepsilon_1$$

$0 < \varepsilon_1 < 1$

$$log_2(143^{143}) = 143\log_2(143) = 1024 + \varepsilon_2$$

$0 < \varepsilon_2 < 1$

ולכן ברור שההמרה הראשונה שלא מתאפשרת היא עבור *n=144* , כי המעריך של 2 נדרש להיות בין 1023- ל 1024 .

<u>סעיף ב':</u>

אם *f* מקבלת *int* , היא תחזיר *float* , במהלך ההדפסה הפעם אין שגיאות, והערכים מודפסים עד 713 החל מ- 714 הערך שמודפס הוא *int* , וזה קורה שכן:

$$log_2\left(\frac{714^{714}}{714!}\right) = 714\log_2(714) - \log_2(714!)$$

$$714\log2(714) = 6768 + \varepsilon_1$$

$0 < \varepsilon_1 < 1$

$$\log_2(714!) = \sum_{k=0}^{714}\log_2(k) = [\text{צירפתי פונקציה}] = 5744 + \varepsilon_2$$

$0 < \varepsilon_2 < \varepsilon_1 < 1$

לכן:

$$log_2\left(\frac{714^{714}}{714!}\right) = 6768 - 5744 = 1024 + \varepsilon_3$$

$0 < \varepsilon_3 < 1$

באופן דומה נחשב :

$$log_2\left(\frac{713^{713}}{713!}\right) = 1022 + \varepsilon_4$$

$0 < \ \varepsilon 4 < 1$

ולכן כשמציגים מספר כזה כי $(-1)^s\ 2^{c-1023}\ (1+f)$ מקבלים שעבור 713 אין בעיה כי המעריך של 2 בטווח 1023- עד 1024 , עבור 714 אנחנו חורגים.

סעיף ג':

ההבדל הוא ממתי מתבצעת ההמרה מ-int ל- float . (ההמרה נדרשת כשמחלקים int ב- float).

בשיטה הראשונה ההמרה מתבצעת על כל המספר $(\frac{n^n}{n!})$ שהחל מ-n מסוים נעשה ענק(כי $\lim_{n\to\infty}(\frac{n^n}{n!}) = inf$) .

ולכן עצם ההמרה היא בעייתית החל משלב מסוים והקוד מפסיק לרוץ.

בשיטה השנייה ההמרה מתבצעת בכל שלב על מספרים קטנים k,n $(1 \le k \le n)$ ומשם כבר הפעולות מבוצעות ב-float-ים.

לכן אין כאן שגיאת המרה.

מה שכן קורה הוא שבאיזשהו שלב המספר $(\frac{n^n}{n!})$ חוצה את הטווח המותר לאחסון בשיטת הדיוק הכפול $(-1)^s\ 2^{c-1023}\ (1+f)$ ולכן מאותו שלב ואילך מוגדר כ-inf .

שאלה 3

סעיפים א'+ב'+ג'+ד'- קטע הקוד:

```
import math
import matplotlib.pyplot as plt
import numpy as np

#Question 3a, auxiliary
def get_max_div_diff(xs, ys):

    n = len(xs)
    if n <= 1:
        return ys

    ys_head = ys[:-1]
    ys_tail = ys[1:]
    xs_head = xs[:-1]
    xs_tail = xs[1:]
```

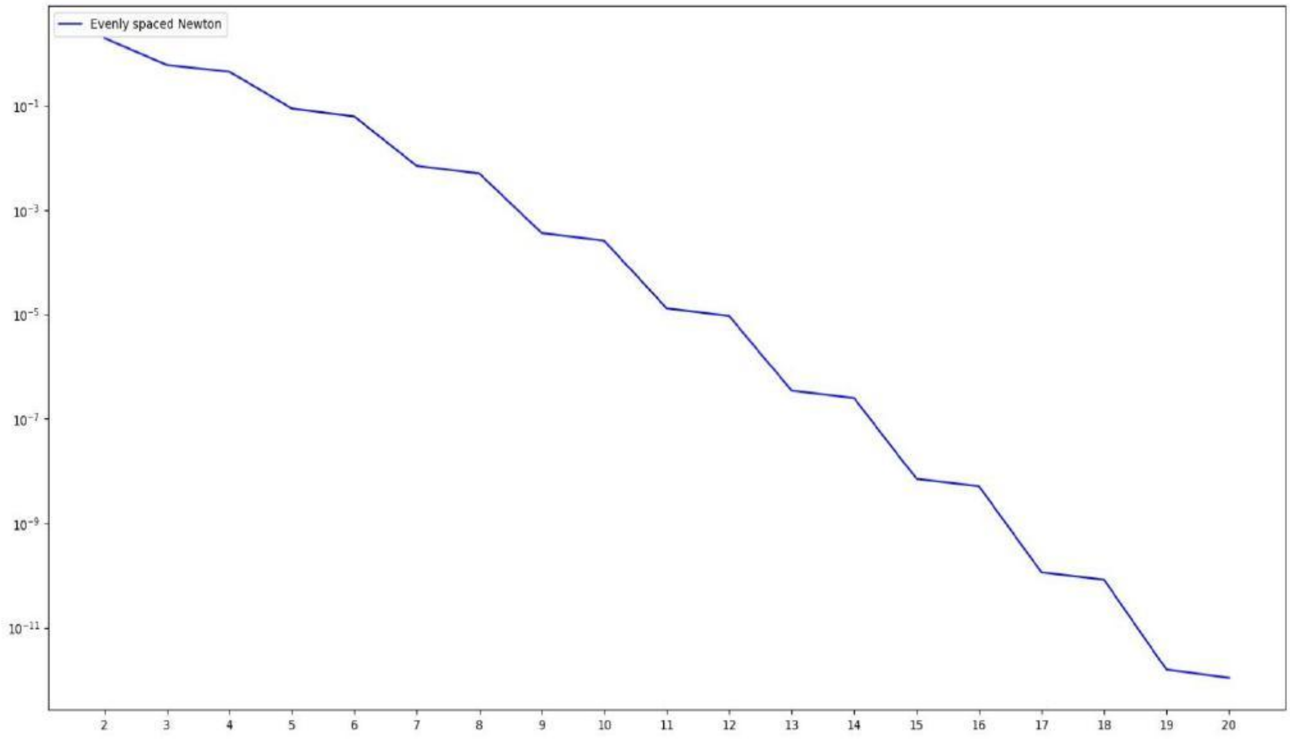
```python
    x_diff = xs[-1] - xs[0]
    head_diff = get_max_div_diff(xs_head, ys_head)
    tail_diff = get_max_div_diff(xs_tail, ys_tail)

    return (tail_diff - head_diff) / x_diff

#Question 3a.
def get_divided_differences(xs, ys):
    n = xs.size
    div_diff_list = np.zeros(n)
    for i in range(n):
        x_head = xs[:i+1]
        y_head = ys[:i+1]
        div_diff_list[i] = get_max_div_diff(x_head, y_head)
    return div_diff_list




#Question 3b.
def compute_newton_interpolation(x, y, x_new):
    c = get_divided_differences(x, y)
    interp_pol = c[-1]
    for i in reversed(range(len(c)-1)):
        interp_pol *= (x_new - x[i])
        interp_pol += c[i]

    return interp_pol




#Question 3d and 3d, auxiliary
def choose_ns(base=None):
    if base is not None:
        p_max = 9
        base = 1.5
        powers = np.arange(1, p_max+1)
        ns = np.rint(base ** powers).astype(int)
    else:
        ns = list(range(2, 51))
    return ns

#Question 3c, auxiliary
def get_chebyshev_pts(n):
    ks = np.arange(n)
    pi_coeffs = (2*ks+1)/(2*n)
    return np.pi * np.cos(pi_coeffs*np.pi)

#Question 3c and 3d, auxiliary
def function_error(ys, y_approx):
    return np.amax(np.abs(ys-y_approx))

def question_3c_per_n(n, x_mode='even', interp_mode='newton'):

    from scipy.interpolate import barycentric_interpolate as bar_int
    x_new = np.linspace(-np.pi, np.pi, num=100)
    y_new = np.cos(x_new)
```

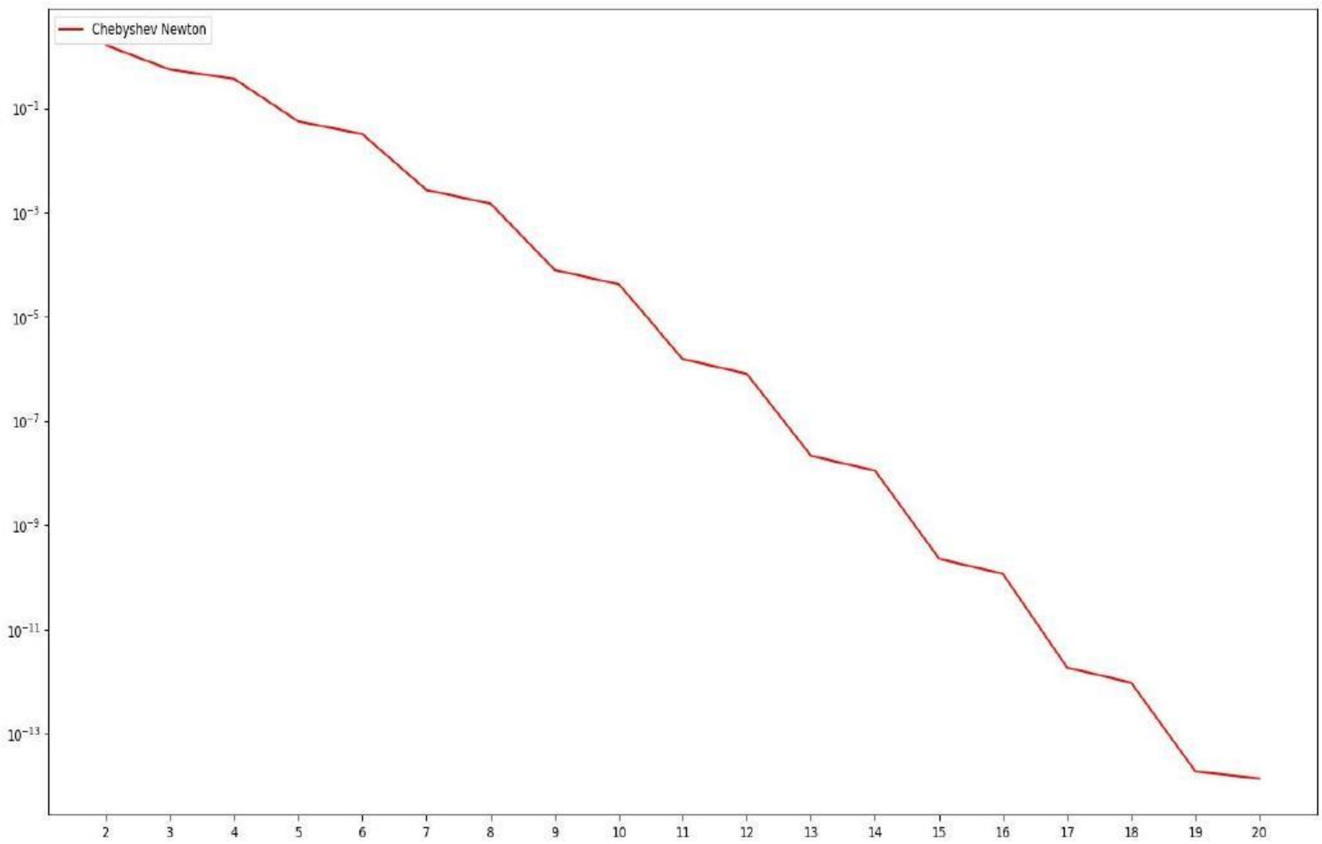
```
    if x_mode == 'even':
        x = np.linspace(-np.pi, np.pi, num=n)
    else:
        x_chebyshev = get_chebyshev_pts(n)
        x = x_chebyshev
        if x_mode =='cheb_random':
            np.random.shuffle(x)

    y = np.cos(x)

    if interp_mode == 'newton':
        y_interp = compute_newton_interpolation(x, y, x_new)
    else:
        y_interp = bar_int(x, y, x_new)

    err_interp = function_error(y_interp, y_new)

    return err_interp

def question_3c():

    ns = choose_ns()
    num_ns = len(ns)
    x_mode = 'cheb_random' #'cheb_random' #'even', 'cheb'
    interp_mode = 'bary' #'bary'
    err = np.zeros(num_ns)
    for p, n in enumerate(ns):
        err[p] = question_3c_per_n(n, x_mode=x_mode,
interp_mode=interp_mode)

    debug_plot = 1
    if debug_plot:
        ns = np.array(ns)
        if interp_mode == 'newton':
            if x_mode == 'even':
                plt.plot(ns, err, 'b', label='Evenly spaced Newton')
            elif x_mode == 'cheb':
                plt.plot(ns, err, 'r', label='Chebyshev Newton')
            else:
                plt.plot(ns, err, 'g', label='Chebyshev rand order
Newton')
        else:
            if x_mode == 'even':
                plt.plot(ns, err, 'k', label='Evenly spaced
Lagrange')
            elif x_mode == 'cheb':
                plt.plot(ns, err, 'm', label='Chebyshev Lagrange')
            else:
                plt.plot(ns, err, 'c', label='Chebyshev rand order
Lagrange')
        plt.legend(loc="upper left")
        plt.xscale('linear')
        plt.xticks(ns, ns)
        plt.yscale('log', base=10)
        plt.show()




##Question 3d, auxiliary
```

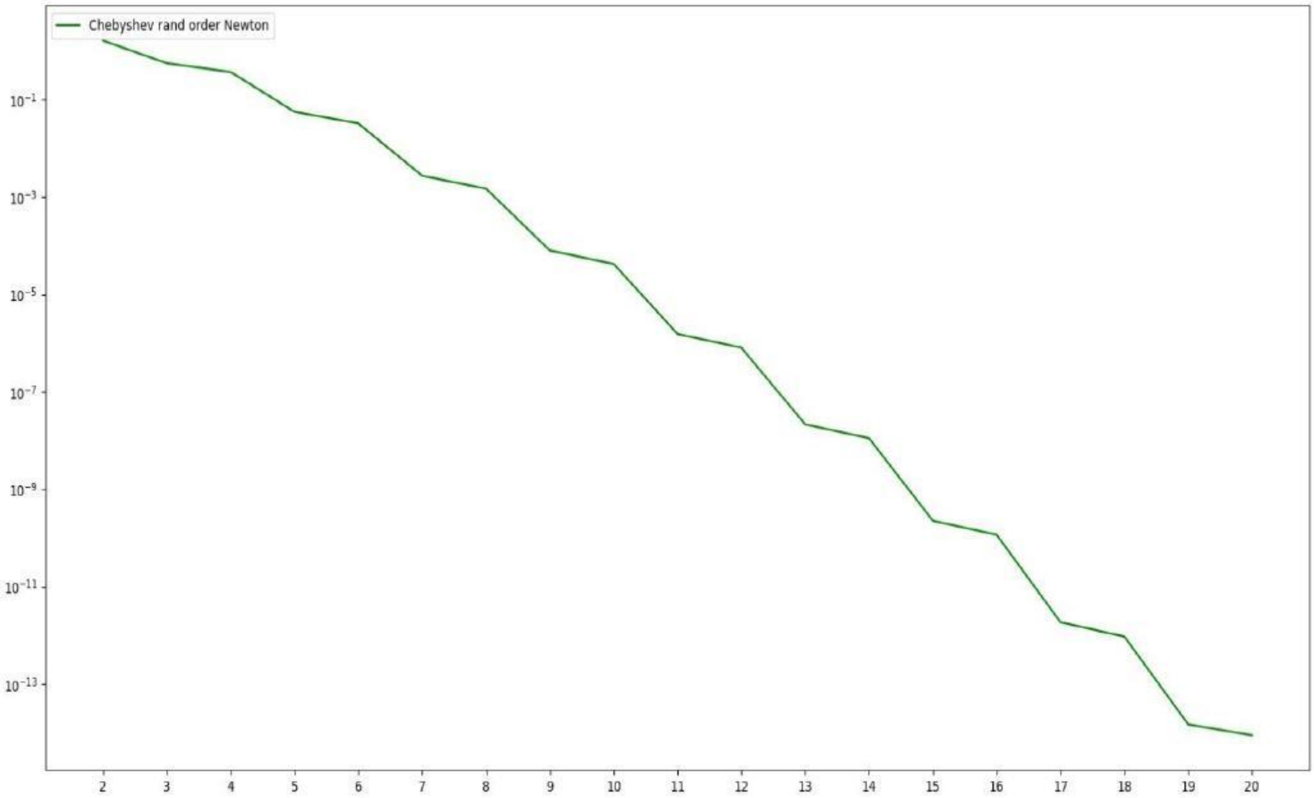
```python
def cos_taylor(n, x_new):
    from scipy.special import factorial
    y_taylor = 0
    #taylor series
    for i_double in range(0, n+1, 2):
        i = round(i_double/2)
        taylor_coeff = (-1)**i / factorial(i_double)
        new_term = taylor_coeff*(x_new**i_double)
        y_taylor += new_term
    cosx = np.cos(x_new)

    taylor_error = function_error(cosx, y_taylor)

    return y_taylor, taylor_error

#Question 3d.
def taylor_question():
    from scipy.special import factorial

    ns = choose_ns()

    taylor_errors = np.zeros(len(ns))
    for p, n in enumerate(ns):
        x_new = np.linspace(-np.pi, np.pi, num=n)
        y_taylor, taylor_errors[p] = cos_taylor(n, x_new)

    ns_no_overflow = np.array(ns[:-2])
    factorials = [factorial(n) for n in ns_no_overflow.tolist()]
    taylor_theor_max_err =
(np.pi**ns_no_overflow)/np.array(factorials)
    debug_plot = 1
    if debug_plot:
        plt.plot(ns, taylor_errors, 'b', label='Empirical Error')
        plt.plot(ns_no_overflow, taylor_theor_max_err, 'r',
label='Taylor Theoretical Max Error')
        plt.xticks(ns, ns)
        plt.yscale('log', base=10)
        plt.legend(loc="upper left")
        plt.show()


if __name__ == '__main__':
    part = 'c'
    if part == 'd':
        taylor_question()
    else:
        question_3c()
```

המשך סעיף ג'- הגרפים:

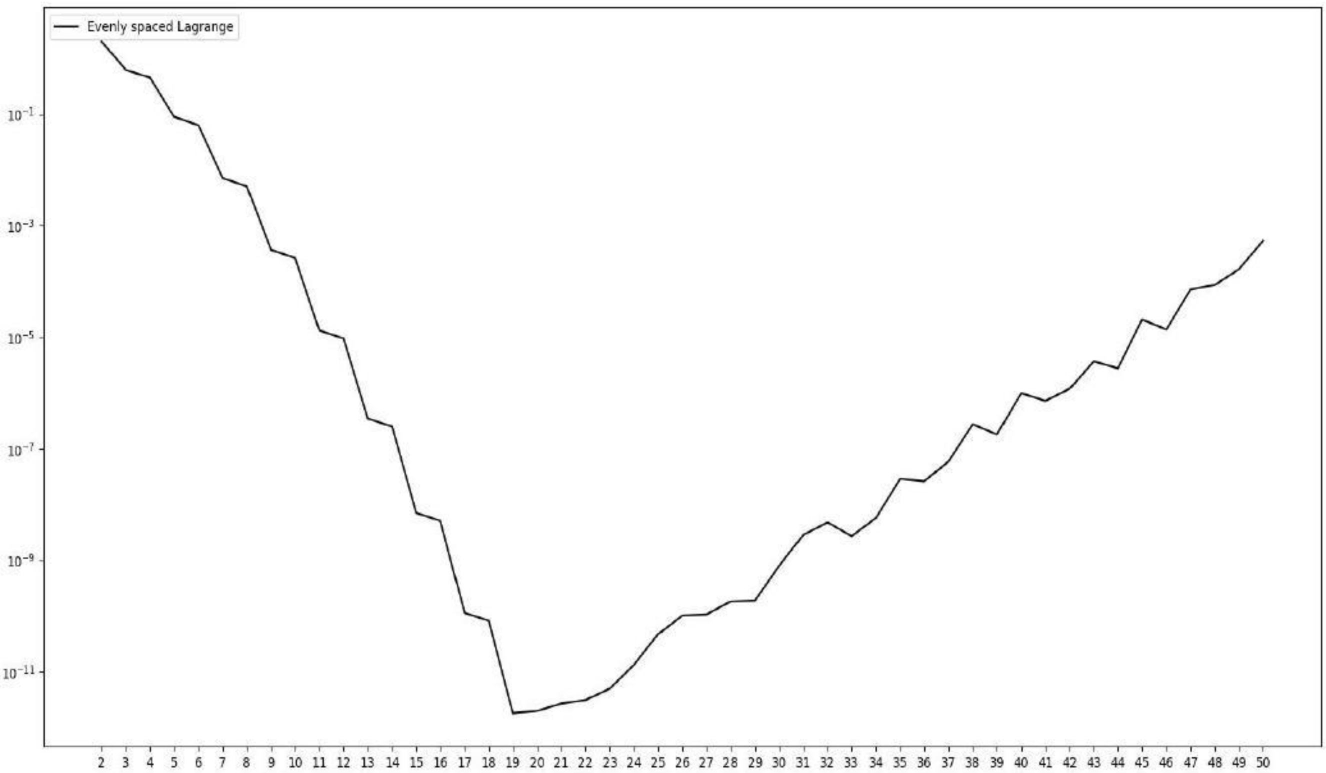
למה הגרפים לא באותה
מערכת צירים וחסר פירוט
בסעיף ד


Evenly spaced Newton
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

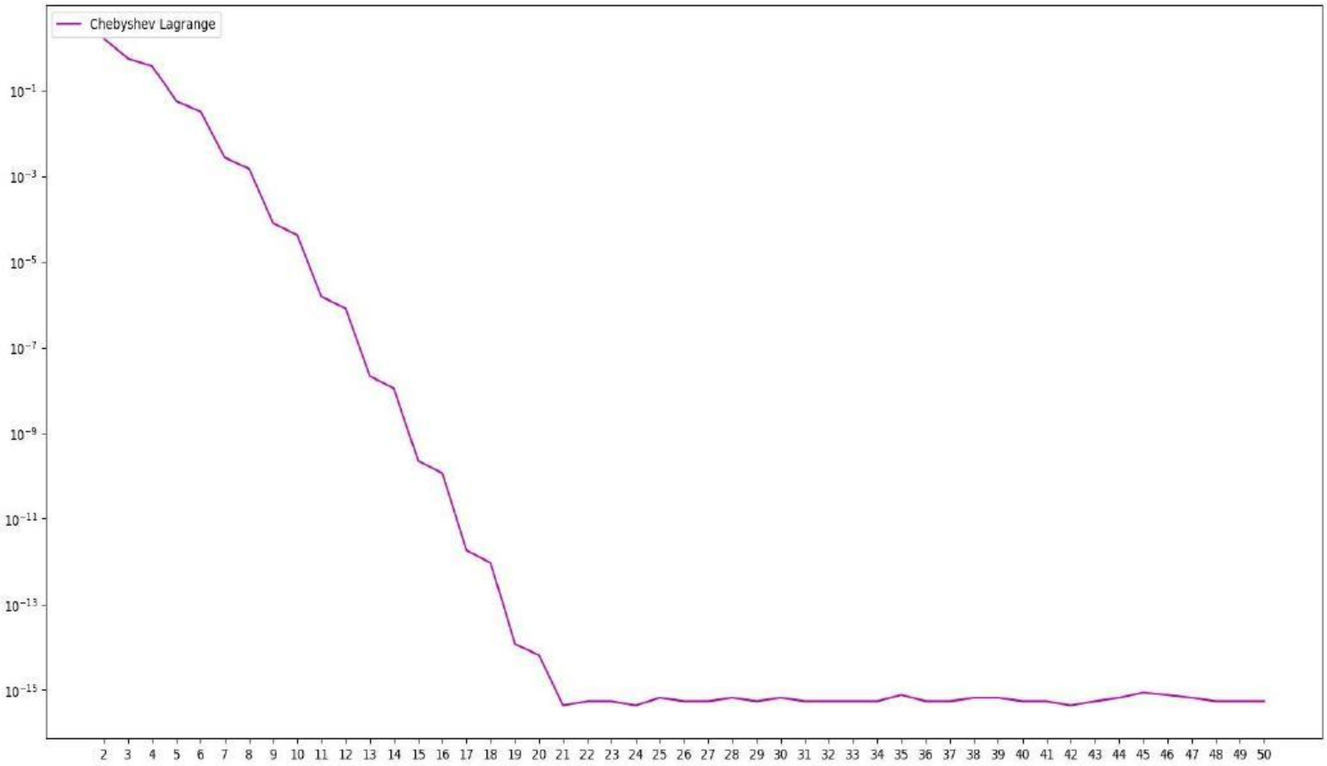

Chebyshev Newton
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

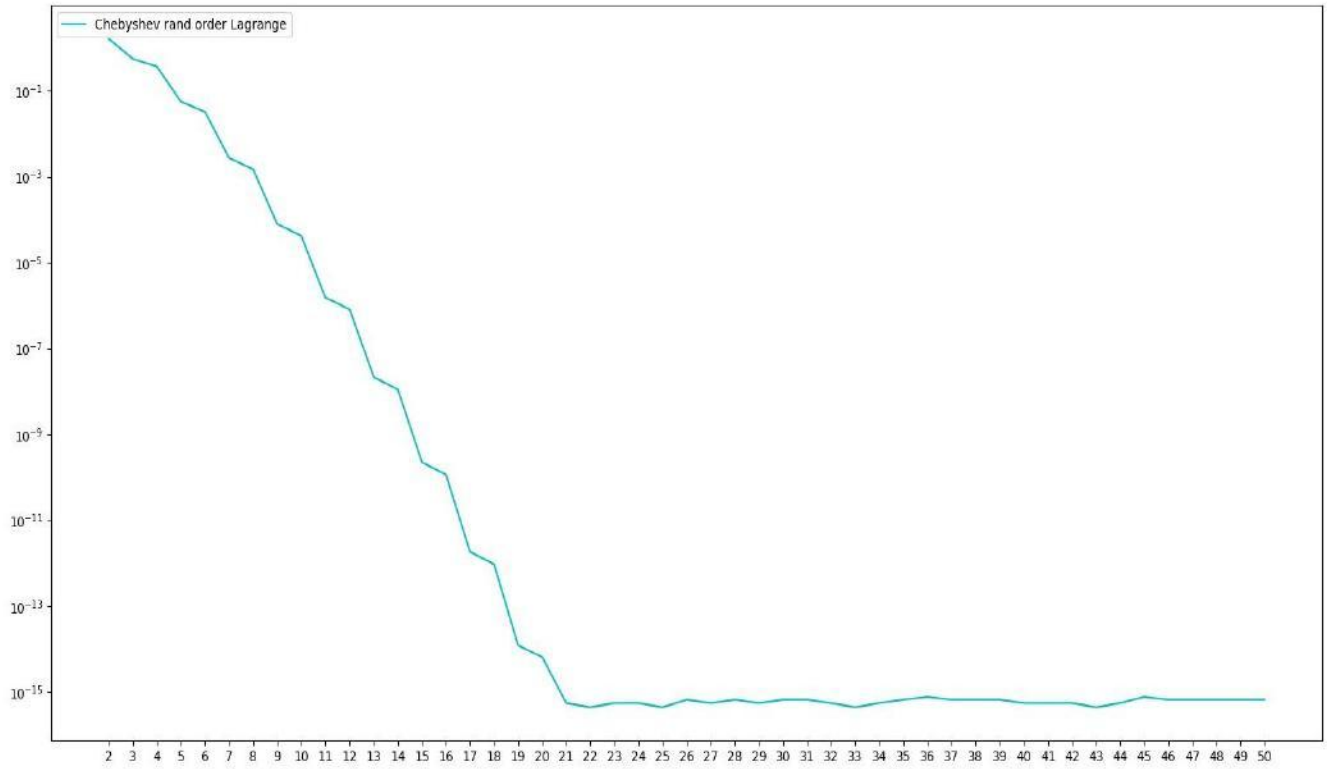


Chebyshev rand order Newton
10^-1
10^-3
10^-5
10^-7
10^-9
10^-11
10^-13
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20



Evenly spaced Lagrange
10^-1
10^-3
10^-5
10^-7
10^-9
10^-11
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50





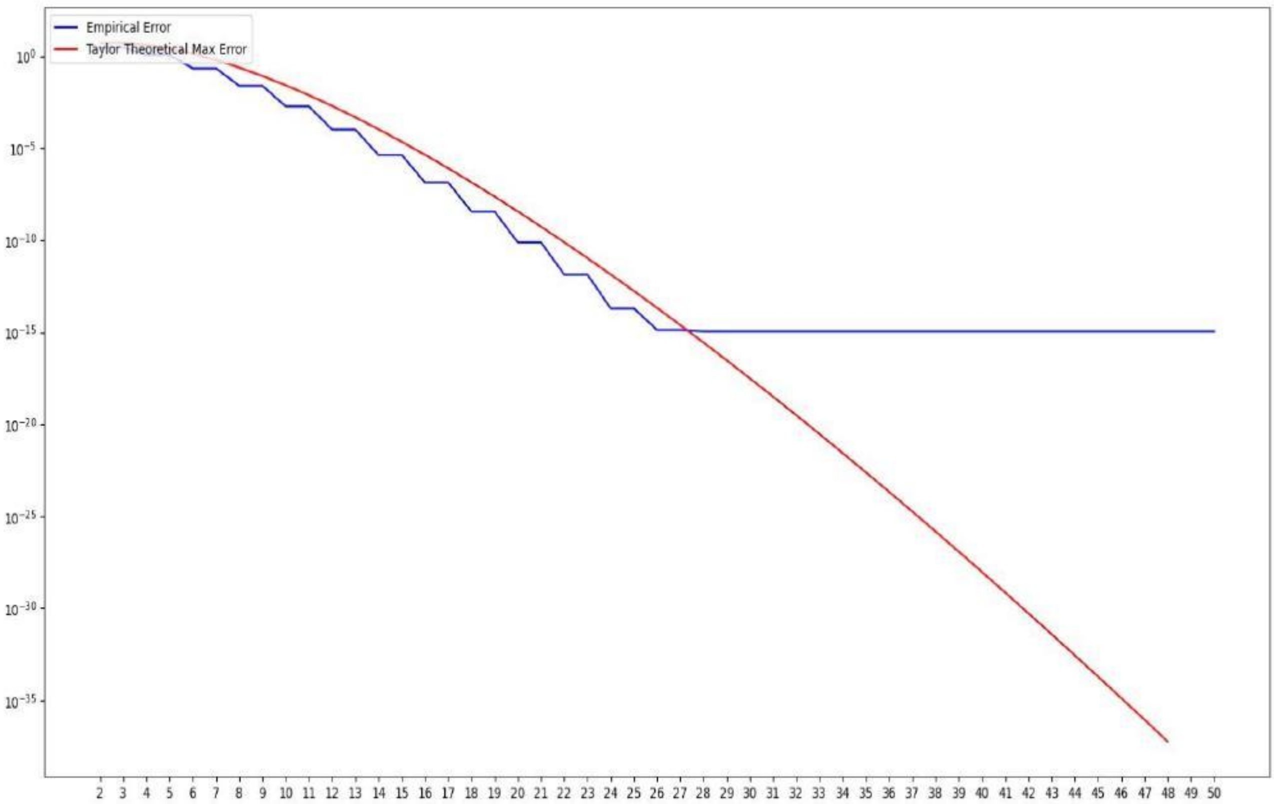
<u>המשך סעיף ג':</u>

הבעיה: בשיטת ניוטון קשה מאוד להריץ מעבר ל- n=20 .

בשיטת לגראנז' רואים ש-chebyshev (בשתי השיטות) נותן שגיאה יציבה יותר כאשר n גדל, מאשר בחירת x שנלקחים במרחקים שווים. בנוסף, גם בניוטון וגם בלגרנז' הערכת השגיאה ב-chebyshev קטנה יותר.

לגבי היציבות של ניוטון, עד n=20 השיטה נראית יציבה. ייתכן שבהמשך השגיאה הייתה גדלה, אבל כאמור לא הצלחתי להריץ עבור n גדולים יותר.

<u>סעיף ד'- הגרף:</u>



חסר

<u>סעיף ה':</u>

ראשית, רואים שבגרף של שגיאת טיילור, הגרף מתייצב באיזשהו שלב, אך זה בגלל העיגול, לא בגלל הקיטוע. מבדיקה עולה שהשגיאה המקסימלית מתקבלת מקצוות הקטע (במקרה שלנו $-\pi, +\pi$).

השגיאה יציבה במובן זה שכאשר n גדל השגיאה לא גדלה בשלב מסויים.

זה שונה מהמקרה של אינטרופולציה כאשר לוקחים נקודות ב- even spacing.

בנוסף, פולינומי טיילור מתכנסים במידה שווה בקטע לקוסינוס ומכאן הירידה המתמדת בשגיאות. נשים לב שיש קפיצות ענקיות איפה שאין אינטרפולציות, זה בגלל שפולינומי האינטרפולציה מתכנסים עבור כל נקודה אבל לא במידה שווה.

הסיום