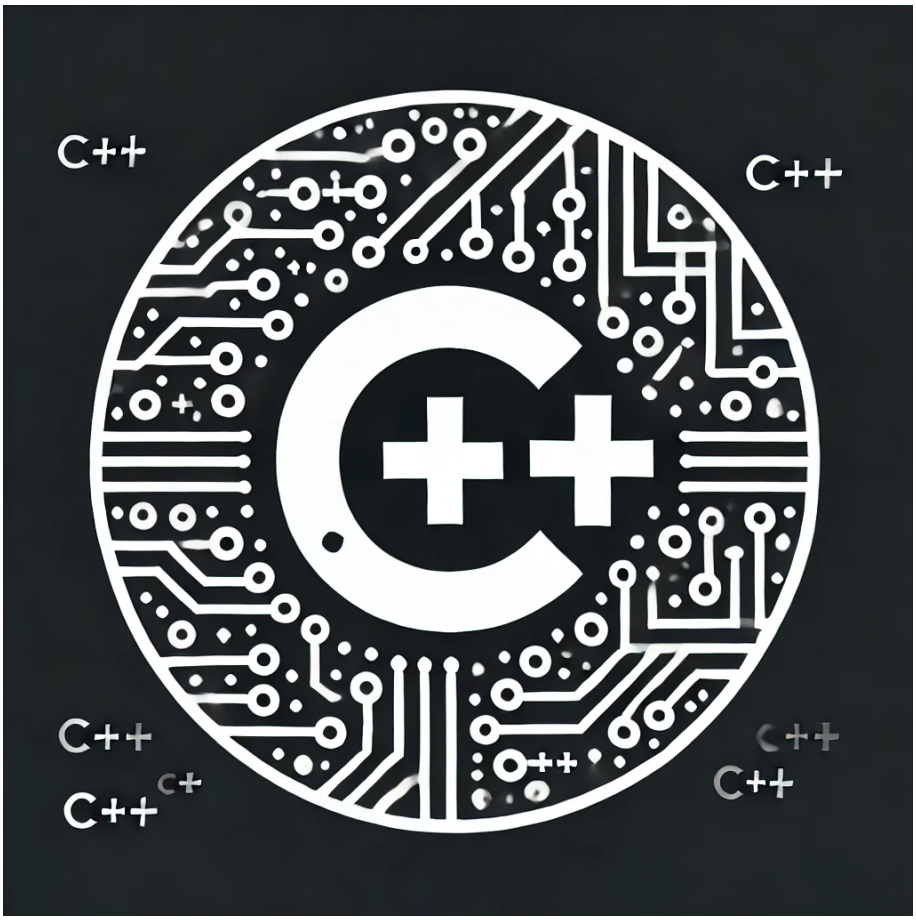
Korszerű számítástechnikai módszerek a fizikában 1

I. házifeladat

Beadás napja: 2025. 03. 02.

A dokumentáció végzője: Nemeskéri Dániel VS6E5A



*A logó Chat GPT-vel generálva.

A feladat megoldása:

Az 1. Listing. az első verzióban használt headereket tartalmazza. Először definiáltam a látott függvényt (2. Listing), figyelve a megfelelő változó típusra. Majd az 1 egyenlet szerinti Simpson-formulát választottam az integrál numerikus számolásához.

```
#include <iostream>
#include <cmath>
```

1. Listing. A használt Header fájlok

Először a 3. Listing szerint definiáltam az integrate függvényt. Itt egy for ciklust használtam, és "szó szerint" implementáltam a formulát. A C++ 0-tól indexszel, ennek megfelelően állítottam, minden lépésben kiszámolja a közelítő formulával az adott integrál szeletke értékét, és az x tengelyen dx-lépssel tovább megy, vagy mondhatom úgy is hogy $+= dx$-szel hozzáadok (xi változóhoz).

```
// Function
double func(double x_f) {
        double func_value = exp(-x_f*x_f)*cos(x_f);
        return func_value;
}
```

2. Listing. A használt függvény definíciója

A következő

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] \tag{1}$$

1. ábra. Simpson-formula

```
// Simpson's rule
double integrate(int n, double x0, double x1) {
        double int_value = 0.0;
        double dx = (x1-x0)/n;
        double xi = x0;

        for (int i=0; i<n; ++i) {
                double xi_1 = xi + 0.5 *dx;
                double xi_2 = xi_1 + 0.5 *dx;
                int_value += (dx/6) * (func(xi)+4*func(xi_1)+func(xi_2));
                xi += dx;
        }
        return int_value;
}
```

3. Listing. Első definíció a Simpson-módszerrel

A 4. Listing a kód végrehajtását adja meg. A feladat leírásának megfelelően 16 értékes jegyet íratok ki. A double kb. 15-16-ot tárol valójában!

```
int main() {
    std::cout.precision(16);
    std::cout << integrate(1000, -1.0, 3.0) << std::endl;
    return 0;
}
```

4. Listing. Main func

A fordítást a 'g++ -o *integrate*0 *first.cpp*' paranccsal végzetem. (Nincs bemenet, majd később látható, máshogy végzem a kiértékelést nem txt fájlba mentek ki for ciklussal, azt is lehetne, de speciál annyira nem szeretek cpp-szal be és kimenetet játszani, így máshogy csináltam.)

(Az angol kommentelés nem feltétlen jelez DeepSeek/Chat Gpt-t, azért én is tudok angolul... !)

A programot több különböző n értékekkel futattam, és 6-8 értékes jegyre sikerült jó konverziókat találni. Ezután DeepSeekkel leellenőriztettem a programomat, amely a következő módosítást javasolta 5. Listing. (Illetve az indexszeléssel is bénáztam, nem az 0-tól kezdés miatt hanem azzal kavarodtam eredetileg hogy lépés hogy nézve $dx/2$ vagy dx.) A módosított implamentáció jobb kb. 10-12 értékes jegy pontotságot eredményezett. Az implementáció két for ciklussal megy végig a páros és páratlan indexű x értékeken. Azaz az integrál szeletkék 3 tagját külön adja össze (két részben). A két szélső x értéknek megfelelő járulékokat az int_value definiálásakor már hozzáadja. A program itt ellentétben amit én eredetileg írtam $dx/2$ lépés helyett dx-et tesz az integrál szeletkék három x értékei között. Bár az eredeti Simpsons 1/3 nem az enyémmel egyezik / hasonlít, ez jól működött, így nem írtam vissza. Illetve a függvény elején ellenőrzi a program hogy páros számot adtunk-e meg, ha nem bővíti eggyel az n értéket, mivel a két for ciklusos összeadás esetén ugye rendre $1, 4, 2, 4, ..., 2, 4, 2, 1$ együtthatót kapnak az x értékek, azonban páratlan esetén ez sérülne, mivel itt ellentétben az előző első implementációval lényegében fél integrál szeletkékre osztjuk fel az összeadást.

```
// Simpson's rule
double integrate(int n, double x0, double x1) {
    if (n % 2 != 0) {  // Simpson's rule requires an even number of intervals
        ++n;
    }

    double dx = (x1 - x0) / n;
    double int_value = func(x0) + func(x1);

    for (int i = 1; i < n; i += 2) {
        int_value += 4 * func(x0 + i * dx);
    }

    for (int i = 2; i < n - 1; i += 2) {
        int_value += 2 * func(x0 + i * dx);
    }

    int_value *= dx / 3.0;
    return int_value;
}
```

5. Listing. DeepSeek által javasolt módosítás

```
#ifndef math_own
#define math_own

// Function to compute the sum of two numbers
double my_exp(double x_e);

// Function to integrate using Simpson's rule
double my_cos(double x_c);

#endif
```

6. Listing. Saját header fájl a függvények közelítésére (hogy cmath se kelljen)

A következő 7. Listing pedig a deklarált függvények definíciójait tartalmazza. Exponenciális esetben 3-3 Páde táblázattal, koszinusz esetén 4-4 táblázattal közelítettem.

```
#include "math_hw.h"

double my_exp(double x_e) {
    double z = x_e;
    return (1.0+0.5*z+0.1*z*z+(1.0/120.0)*z*z*z)/(1.0-0.5*z+0.1*z*z-(1.0/120.0)*z*z*z);
}

double my_cos(double x_c) {
    double w = x_c;
    return (1.0-w*w*(115.0/252.0)+w*w*w*w*(313.0/15120.0))/(1+w*w*(11.0/252.0)+w*w*w*w
        *(13.0/15120.0));
}
```

7. Listing. Megvalósítás a függvények közelítésére (hogy cmath se kelljen)

Ezután ezzel is megvalósítottam, illetve számoltam integrál értékeket, így a teljes cpp fájlt (mármint az előző kettőn kívül, így ez az eredeti módosítása) a 8. Listing adja meg, ez azonban csak max, 3-4 értékes jegyre adott pontos értéket. Itt nyilván a Pádé táblázatok közelítésének pontatlansága okozza. A továbbiakban sajnos ezt mégsem használtam sokat, mivel így nem lenne olyan érdekes a pontosságot vizsgálni. Ahhoz hogy így a saját header fájlokkal lefordúljon a szkript, a 9. Listring szerinti cmd / bash kód futtatása volt szükséges terminálban.

```cpp
#include <iostream>
#include "math_hw.h"

// Function
double func(double x_f) {
    return my_exp(-x_f * x_f) * my_cos(x_f);
}

// Simpson's rule
double integrate(int n, double x0, double x1) {
    if (n % 2 != 0) {  // Simpson's rule requires an even number of intervals
        ++n;
    }

    double dx = (x1 - x0) / n;
    double int_value = func(x0) + func(x1);

    for (int i = 1; i < n; i += 2) {
        int_value += 4 * func(x0 + i * dx);
    }

    for (int i = 2; i < n - 1; i += 2) {
        int_value += 2 * func(x0 + i * dx);
    }

    int_value *= dx / 3.0;
    return int_value;
}

int main() {
    std::cout.precision(16);
    std::cout << integrate(1000, -1.0, 3.0) << std::endl;
    return 0;
}
```

8. Listing. cmath nélküli megvalósítás, teljes

```
g++ -o -integrate third.cpp math_hw.cpp
```

9. Listing. cpp fájl fordítása

Ezután az n szerinti konvergencia vizsgálatához egy dinamikus könyvtárat készítettem, amely Pythonban is betölthető. A 10. Listing szerinti kulcsszóval a függvényeket C kompatibilis névkonvenció szerint exportáltam, biztosítva a C linker-konvenciót. Mivel a saját közelítéssel nem lenne értelmes sok értékes jegyre vizsgálni, így újra cmath-ot használok!

```cpp
#include <iostream>
#include <cmath>

extern "C" {
    // Function
    double func(double x_f) {
        return exp(-x_f * x_f) * cos(x_f);
    }

    // Simpson's rule
    double integrate(int n, double x0, double x1) {
        if (n % 2 != 0) {  // Simpson's rule requires an even number of intervals
            ++n;
        }

        double dx = (x1 - x0) / n;
        double int_value = func(x0) + func(x1);

        for (int i = 1; i < n; i += 2) {
            int_value += 4 * func(x0 + i * dx);
        }

        for (int i = 2; i < n - 1; i += 2) {
            int_value += 2 * func(x0 + i * dx);
        }

        int_value *= dx / 3.0;
        return int_value;
    }
}
```

10. Listing. C++ amelyet meg lehet híváni Pythonban

11. Listing szerint történt a fordítás. (dinamikus könyvtárrá)

```
g++ -shared -o -integrate2.so fifth.cpp
```

11. Listing. cpp fájl fordítása

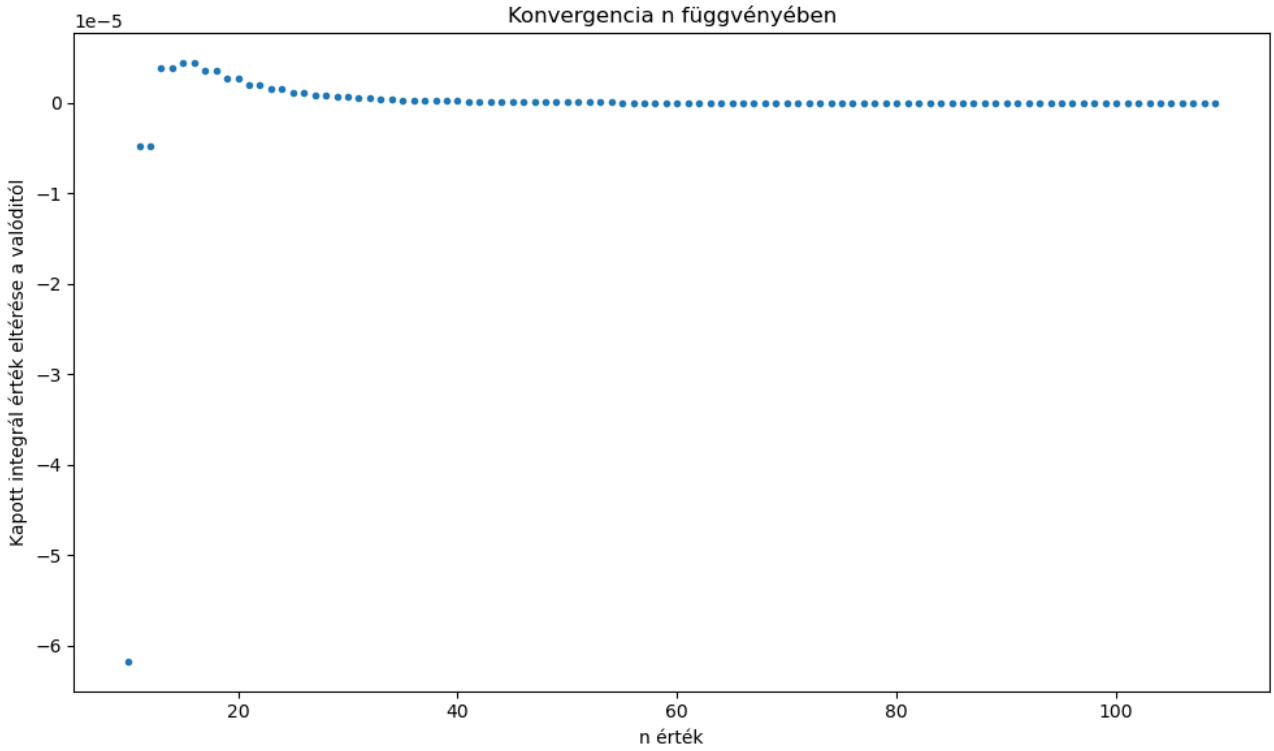
A 12. Python szkriptet pedig Jupyter-notebookban írtam, ctypes modulellal megfelelően betöltöttem az integrate.so dinamikus könyvtárból az integrate függvényt. (Valójában még generáltattam Chat GPT-fel egy függvényt hogy a plt.savefig()-et kicsit automatiáljam, és if and else-el mindig a következő indexű képnevet adja, de ez talán nem a feladathoz szűken tartozik.)

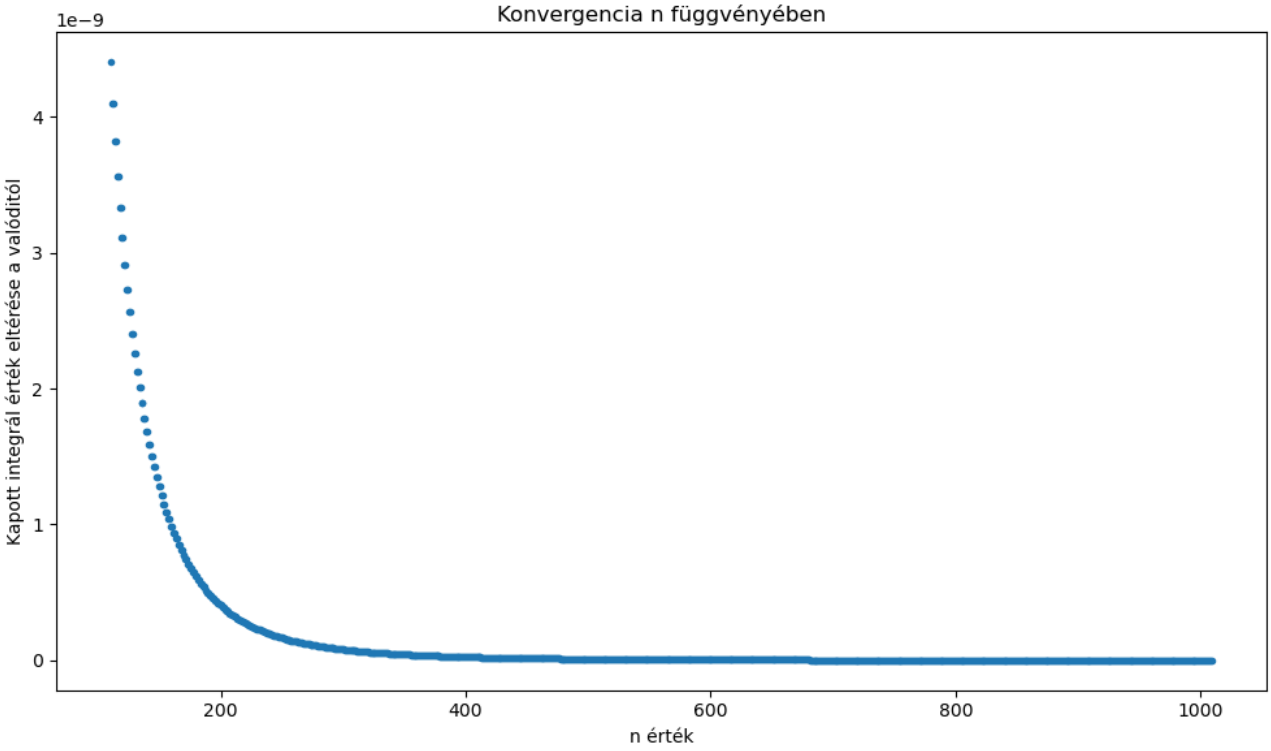
```python
import ctypes
import os
import numpy as np
import matplotlib.pyplot as plt

lib = ctypes.CDLL('./integrate2.so')

lib.integrate.argtypes = [ctypes.c_int, ctypes.c_double, ctypes.c_double]
lib.integrate.restype = ctypes.c_double

result = lib.integrate(1000, -1.0, 3.0)

n = np.linspace(10, 100, 91)
result_vector = [lib.integrate(int(x), -1.0, 3.0) for x in n]

plt.figure(figsize=(10,6))
plt.scatter(n, result_vector, marker=".")
plt.xlabel("n_ertek")
plt.ylabel("Kapott_integral_ertek")
plt.title("Konvergencia_n_fuggvenyeben")
plt.plot()
```

12. Listing. Pythonban számolás és ábrázolás

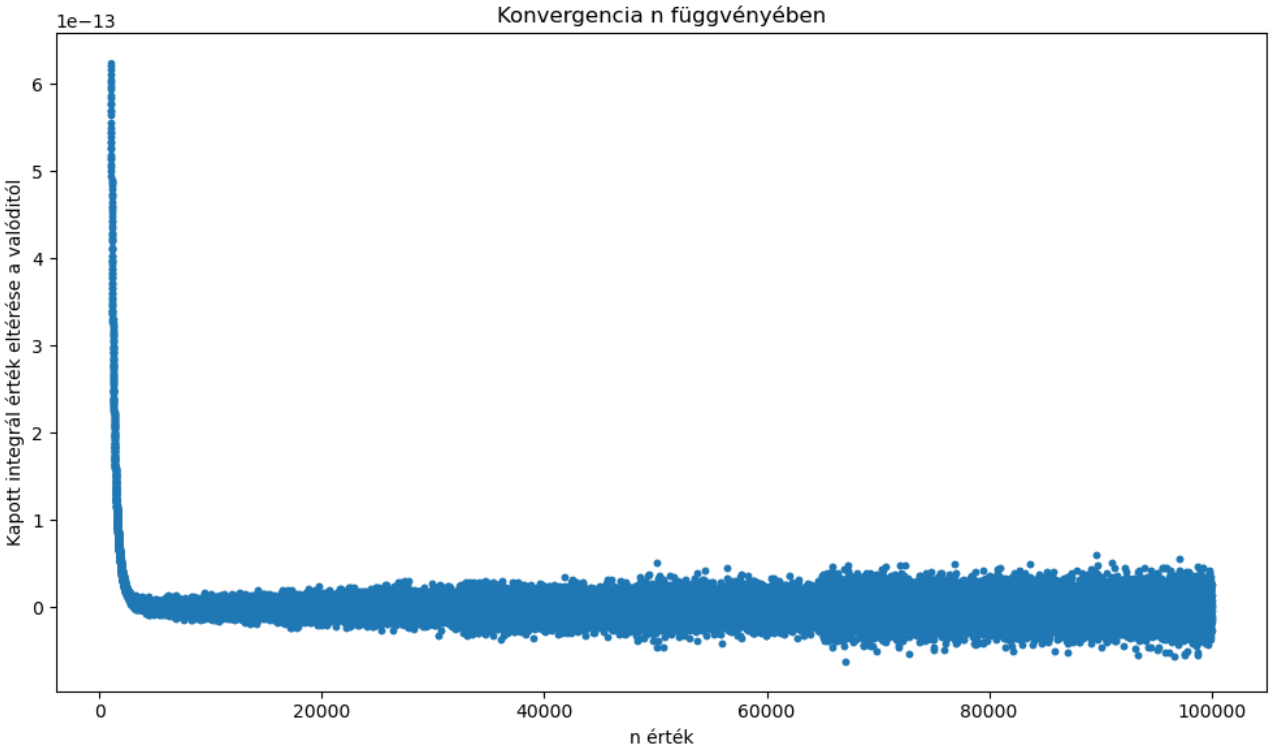
A kapott diagrammokat a következő pár oldal 2. ,3. , 4. és 5. ábrái mutatják! Ábrázoltam a pontos (Wolframalpha segítségével a feladat leírás / óra fóliák szerinti) értéktől való eltérést. Illetve magát a kapott értéket is. (Ilyen és olyan diagram is van.)

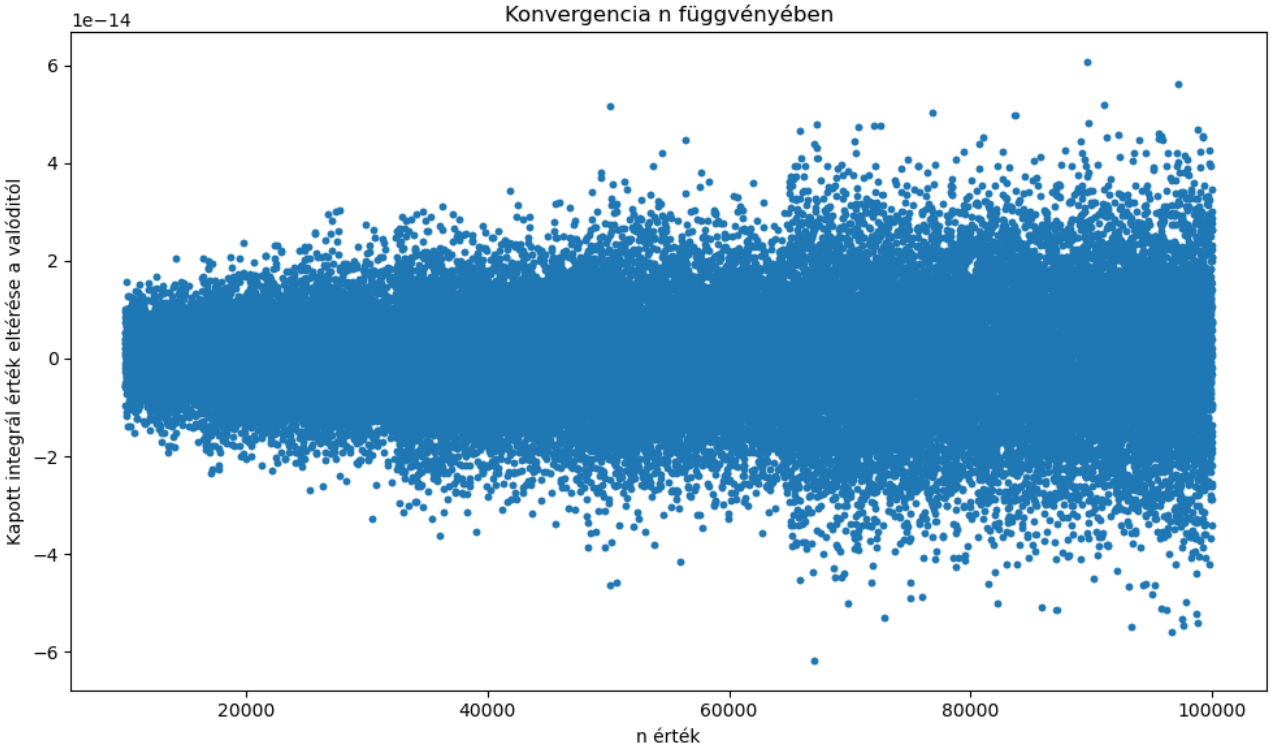
Összeségében elmondhat hogy kb. 12 értékes jegynél nagyobb pontosságot nem lehet elérni (vagy legalábbis nekem nem sikerült), ekkor már nincs csökkenés az eltérésben, nagy a szórás (még növekedik is) (80000-100000-es n értékek esetében, bár látható hogy látható szórás már pár ezres n értékek környékén megjelenik). Legjobban azt gondolom a kb. $n = 50$-es lépészámot érné meg használni általánosan, mivel már az is közel 5 értékes jegy pontosságot jelent(het), azonban az volt a tapasztalak hogy a sok ezres n-ek (10^5 is) viszonylag gyorsan lefut, de ha igen sok integrált kell kiszámítani akkor már lényegesen több idő a nagyobb pontosságú számítás lefutása. (Megjegyzés, van-e elméleti értelme a 80000-100000-as számolásoknak? Nem-e már az adatok pontossága a baj? NEM, itt még nem, a double azért inkább 15-16-17 értékes jegyre adja meg, így ez a 10^5 nagyságrendű lépésre bontás esetén még bőven bőven kisebb, tehát ezzel még nincs probléma.)



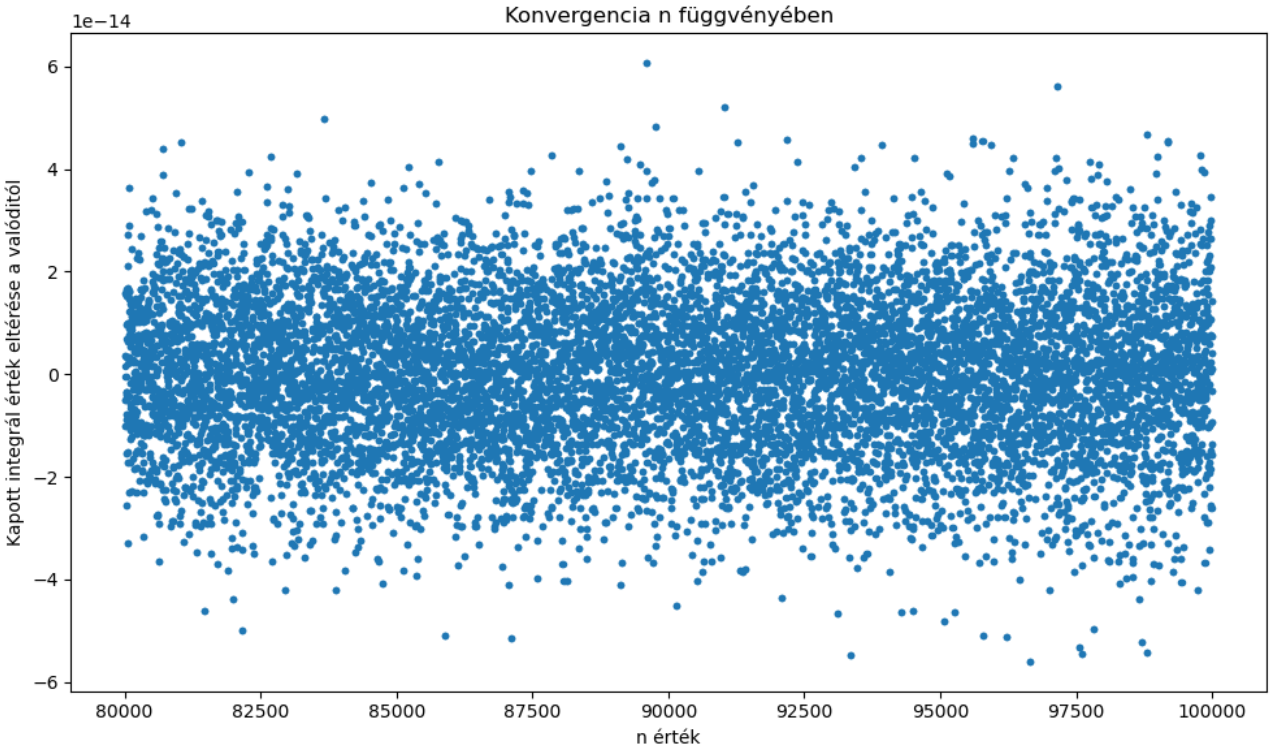


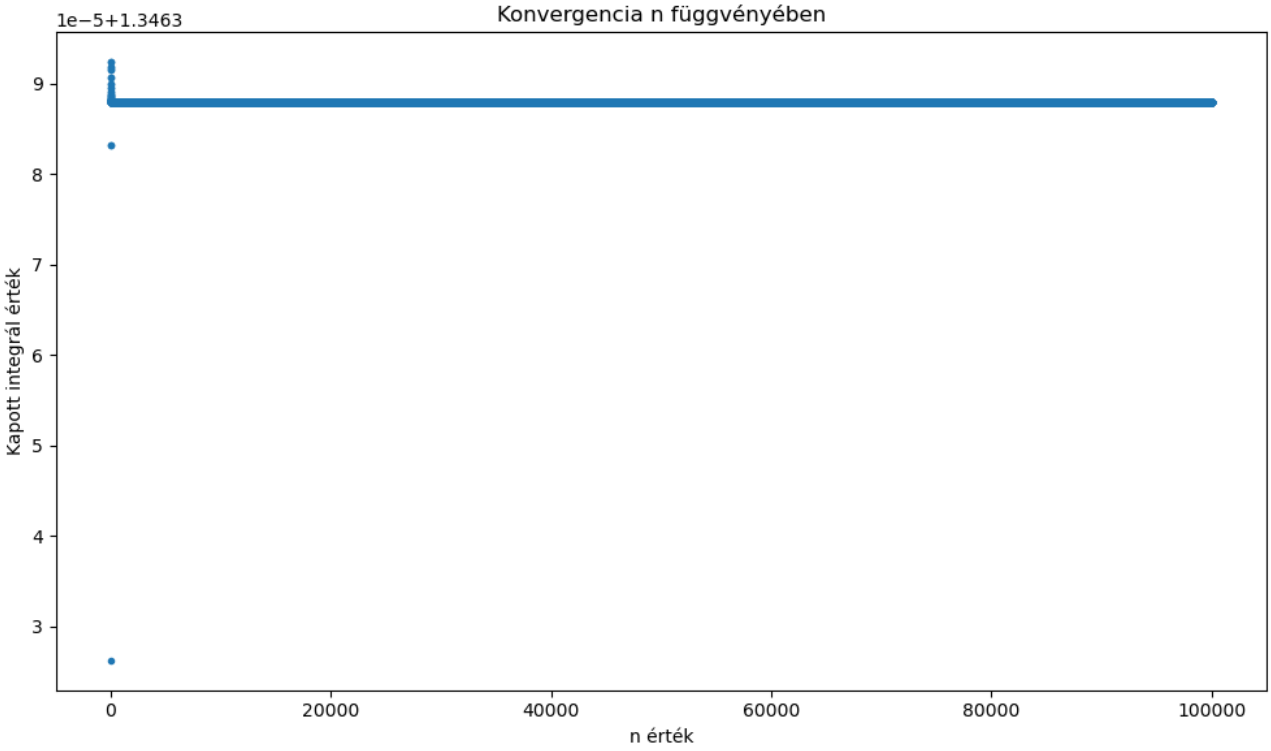
2. ábra. Konvergencia n függvényében



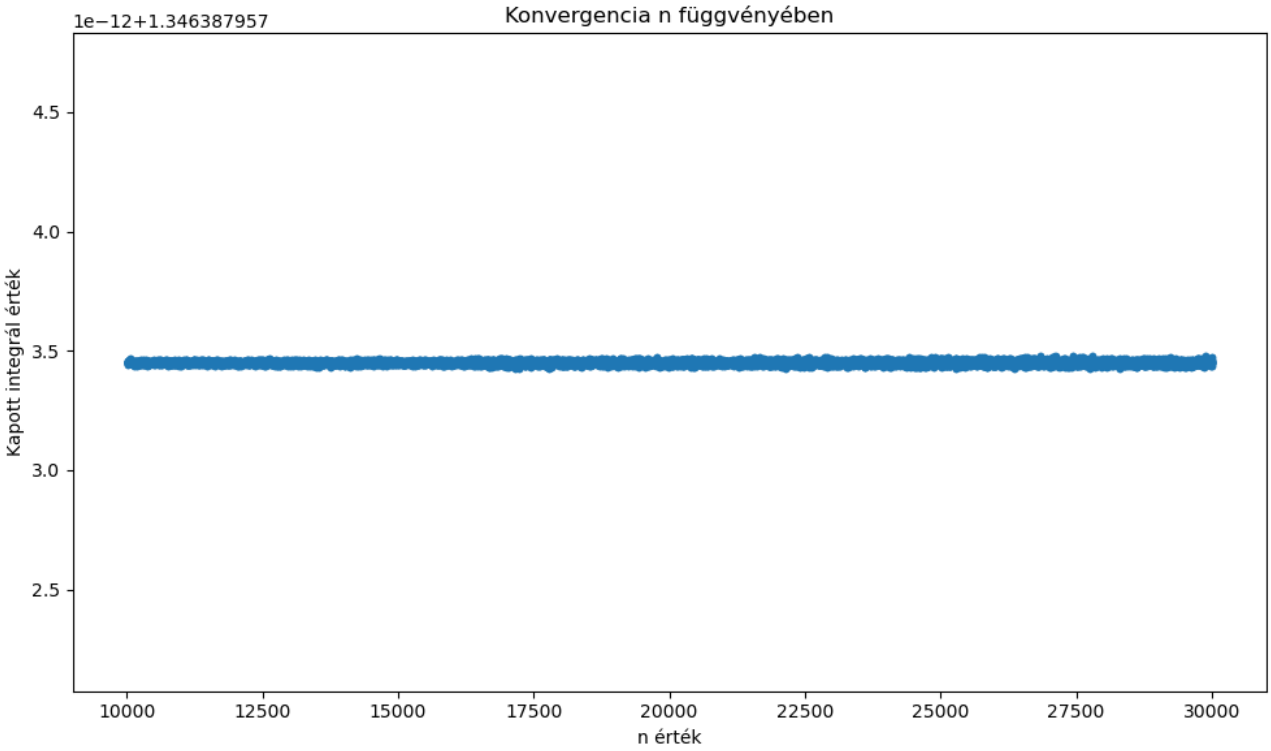


3. ábra. Konvergencia n függvényében





4. ábra. Konvergencia n függvényében



5. ábra. Konvergencia n függvényében