Fejlesztői dokumentáció

Motiváció

A kutatás, amiben részt veszek részben neutrondiffrakciós méréseken alapul. Ezeket a mérési adatokat (persze már nem nyersen) dolgozza fel az RMC, Reverse Monte Carlo nevű program. A program futása során előállít egy atomokból álló konfigurációt és abból kiszámítja a párkorrelációs (r-g(r)) függvényt. Ezt a függvényt Fourier-transzformálja és az így kapott, szimulált szerkezeti függvényt összehasonlítja a mért szerkezeti függvénnyel.

A Fourier-transzformáció viszont a program működésének csak egy része, önmagában nem futtatható így, ha az aktuális jelet és a járulékát meg akarom vizsgálni pl. más tartományon, akkor arra az RMC-n belül képtelen vagyok. Ezért írtam meg ezt a programot, ez a játszóterem.

Fordítás, futtatás

A fordításhoz nem igényel szabványos könyvtárakon kívül semmit. Azonban memory leak diagnosztika miatt van hivatkozás a crtdbg.h-ra, de ez nem szükséges a program működéséhez, kikommentezhető.

A futtatásról a Felhasználói dokumentációban lehet részletesen olvasni.

A program felépítése, főbb modulok

A függvények leírásánál csillaggal jelölöm a dinamikus memóriát allokáló függvényeket, azaz, amelyek kimenetét free()-vel fel kell szabadítani.

Belépési pont (main.c)

Itt hívjuk meg és integráljuk össze a következő fejezetekben leírt modulokat.

1. Összerakjuk a külső fájlok elérési útjait
2. Beolvassuk az input adatokat (r és gr)
3. Transzponáljuk őket sorvektorokká
4. Beolvassuk konfigurációt
5. Validáljuk a konfigurációt
6. Kiszámoljuk a számsűrűséget
7. Legeneráljuk a Q és rgr mátrixokat
8. Összeszorozzuk Q-t és r-t
9. Vesszük a szorzat szinuszát
10. Elvégezzük a diszkrét Fourier transzformációt
11. Az eredményt kiírjuk egy csv fáljba
12. Ha kikommentezzük a kódot, akkor megnyithatjuk excelben a kimenetet
13. Felszabadítjuk a lefoglalt memóriát
14. (Ha akarjuk, a _CrtDumpMemoryLeaks hívással debug során ellenőrizhetjük, hogy minden memóriát felszabadítottunk-e?)

A Fourier-Transform modulok (fourier.c és .h)

generate_q (*)

Generál egy számtani sorozatot, a min, max és dq paraméterek alapján, majd ezt egy dinamikusan allokált mátrixban adja vissza. A q értékek lesznek az illesztési pontok, amelyeken kiértékeljük a Fourier-transzformált eredményeket.

generate_rgr (*)

Bekéri az r és gr mátrixokat és kiszámítja elemenként az r(gr-1) szorzatot, majd az eredményt mátrixban visszaadja.

apply_sin_to_all_elements (*)

Bekér egy mátrixot, majd minden elemnek veszi a szinuszát. A program működéséhez csak ez a művelet fontos, így szükségtelen általánosan használhatóvá kibővíteni a függvényt.

discrete_fourier_transform (*)

Ez függvény végzi el magát diszkrét Fourier transzformációt, amely az alábbi képlet diszkrét verziója. Az integrálás helyett rgr és sin(qr) tagoknak a mátrix-szorzatát kell venni, valamint, ami a képleten nem látszik, le kell osztani a q mátrix megfelelő elemeivel, majd mindezt összegezni kell.

$$S(Q)-1=4\pi\rho\int_0^\infty r\left(g(r)-1\right)\sin(Qr)\,dr$$

Mátrix típus és műveletei (matrix.c és .h)

```c
// egy kétdimenziós mátrixot ír le
typedef struct {
    int nrow; //sorok száma
    int ncol; //oszlopok száma
    int len; // a sorok és oszlopok számának szorzata, memóriakezeléshez
    double val[]; //értékek egy dinamikus tömbben
} matrix;
```

A transzformáció elvégzéséhez mátrixszorzásra van szükség, érdemes az adatot mátrixokban tárolni. Emellett a bejövő és a kimenő adatot is kényelmesebb ilyen formában kezelni. A struct lényegében egy 2D-s tömb, ami ismeri a hosszát, valamint a sorainak és az oszlopainak számát. Az elemek egy folyamatos tömbben vannak eltárolva, így könnyen hozzáférni például mátrixszorzáskor, amikor több sorra vagy oszlopra mutató tömbbe kéne belecímezni, implementációtól függően. Azt viszont, hogy igazából itt egy vektorról és nem valódi mátrixról van szó, könnyen el lehet fedni. Ezt a célt szolgálja a set_value és a get_value függvények bevezetése.

Ahhoz, hogy a mátrixban eltárolt tömb dinamikus hosszú legyen, „struct hacket” használtam, nincs megadva a tömb hossza, így a sizeof(matrix) 16 byteot ad vissza, ami nem egyezik meg egy-egy mátrix valós méretével. A memóriaallokálás ezért így néz ki:

matrix* m = malloc(sizeof(matrix) + sizeof(double) * (nrow * ncol));

void set_value(matrix* m, int row, int col, double value)

Paraméterként kap egy mátrixot, azt, hogy az átírandó érték hányadik sorban és oszlopban van, valamint azt az értéket, ami a megadott helyre kerüljön. Ebből kiszámítja a tömbben a helyzetét. Mivel én az implementációmban soronként haladok, egy elem helyzetét a $\text{sorkoordináta} * \text{oszlopok száma} + \text{oszlopkoordináta}$ képlet adja meg.

double get_value(const matrix* m, int row, int col)

A set_value függvénnyel megegyező módon paraméterként kap egy mátrixot, valamint a keresett érték sor és oszlopszámát, és visszaadja az ott tárolt értéket.

matrix* create_matrix(int nrow, int ncol) (*)

Egy mátrixot létrehozó függvény, ahol megadjuk a mátrix sorainak és oszlopainak számát, és a függvény lefoglalja a mátrixnak a memóriát, de nem inicializálja az elemeit (szemét lesz benne).

Ez a három függvény egy intuitív és bővíthető vázat képez és implementálja a mátrixok logikáját C-ben. Ezek jelentik a többi mátrixokon értelmezett műveletek alapját.

matrix* transpose(const matrix* m) (*)

Paraméterként kap egy mátrixot és az elemeit átmásolja felcserélt sor és oszlopkoordinátákkal a transzponált mátrixba, ami majd vissza is ad. Maga a kód nagyon egyszerű, egy ciklussal bejárjuk a mátrix elemeit, majd meghívjuk a set_value és get_value függvényeket, az alábbi módon. Ebből is látszik a mátrix struct és rá épített függvényekre alapuló implementáció előnye, hogy nem a nyers memóriával kell dolgoznunk, hanem magasabb szintű absztrakciókat építettünk fel (a nyelv korlátjain belül).

```
set_value(transposed, c, r, get_value(m, r, c));
```

matrix* matrix_product(const matrix* m1, const matrix* m2) (*)

Paraméterként kap két mátrixot, és amennyiben az első oszlopainak száma megegyezik a második sorainak számával, kiszámolja a mátrixszorzatukat. A Fourier-transzformáció során a legköltségesebb művelet, n*n-es mátrixokra magának a szorzásnak köbös az erőforrásigénye (o(n^3)), léteznek rá hatékonyabb algoritmusok, de egyszerűsége miatt az alap algoritmusnál maradtam.

Egy alternatíva egy „Divide-and-conquer” algoritmus lett volna, ami szétbontotta volna a mátrixot kisebb mátrixokra, és azokat szorozta volna össze. A fejlettebb algoritmusok közelítenek a 2,5-es hatékonyághoz. A program által létrehozott mátrixok tipikusan az 500x500-as nagyságrendbe esnek, így ez egy jelentős optimalizáció lehetne (125.000.000 vs. 5.590.169 művelet).

void visualize(const matrix* m)

Diagnosztikai célú segédfüggvény, kiírja standard kimenetre a mátrixot, így könnyen leellenőrizhető az elemeinek értéke. Kisebb mátrixokkal érdemes meghívni, a konzol limitációi miatt.

matrix** matrix_to_vectors(const matrix* mat_in) (*)

Szétszed egy mátrixot oszlopvektorokra. Adatkezeléshez hasznos, a program a bemeneti file-t bontja fel vele.

Konfigurációkezelés (cfg.c és .h)

A program konfigurációs adatait az alábbi struktúra tárolja:

```
typedef struct {
    double rho;
    double molar_mass;
    double dq;
    double min;
    double max;
} config;
```

config read_config(const char* config_path)

A config file-t a függvény soronként beolvassa, majd egyenlőségjelek mentén szétszedi egy key és egy value stringre. A key stringet whitespace-ek mentén, mindkét végén trimmeli és kisbetűssé alakítja, hogy a konfig feldolgozás ne legyen kis-nagybetű érzékeny. A key alapján if utasításokkal beállítja a config struktúra megfelelő elemeit.

void validate_config(config cfg)

Validálja, hogy a betöltött konfig struct elemei megfelelnek-e a követelményeknek. Ha nem, akkor a program hibaüzenettel leáll.

Input és output fájlkezelés (io.c és .h)

A bemeneti és a kimeneti fájlok olvasási és írási műveleteit tartalmazza.

matrix* read_r_and_gr_from_csv(const char* path_to_csv) (*)

Ha a megadott elérési utú fájl nem létezik, akkor a függvény hibaüzetet ír ki, és a program leáll. Első körben végigmegy a sorokon, és megszámolja őket. Ezen adat ismeretében lefoglalja a kimeneti mátrixot, ami az adatokat fogja tartalmazni.

Aztán visszaállítja a file pointert a fájl kezdetére, és újra végigmegy rajta soronként. Ekkor beállítja a mátrix megfelelő elemeit a fájlból kiolvasott adatokkal.

void csv_out(const char* csv_out_file_path, const matrix* q_mat, const matrix* sq_mat)

Soronként kiírja két bemeneti mátrix elmeit egy tab szeparált fájlba.

density.c és .h

A számsűrűséget kiszámoló függvényt tartalmazza

mystring.c és .h

Apró segédfüggvények stringkezeléshez: string_tolower, trim, equal_string.

common.c és .h

Vegyes segédfüggvények.

void* safe_malloc(size_t _Size) (*)

Hogy ne kelljen minden malloc hívásnál lekezelni az elfogyó memóriát jelző NULL visszatérési értéket, ezért a program minden pontjáról ezt a saját „becsomagolt” mallocunkat hívjuk meg. Memória allokálás sikertelensége esetén kiír egy hibaüzenetet, majd leállítja a programot.

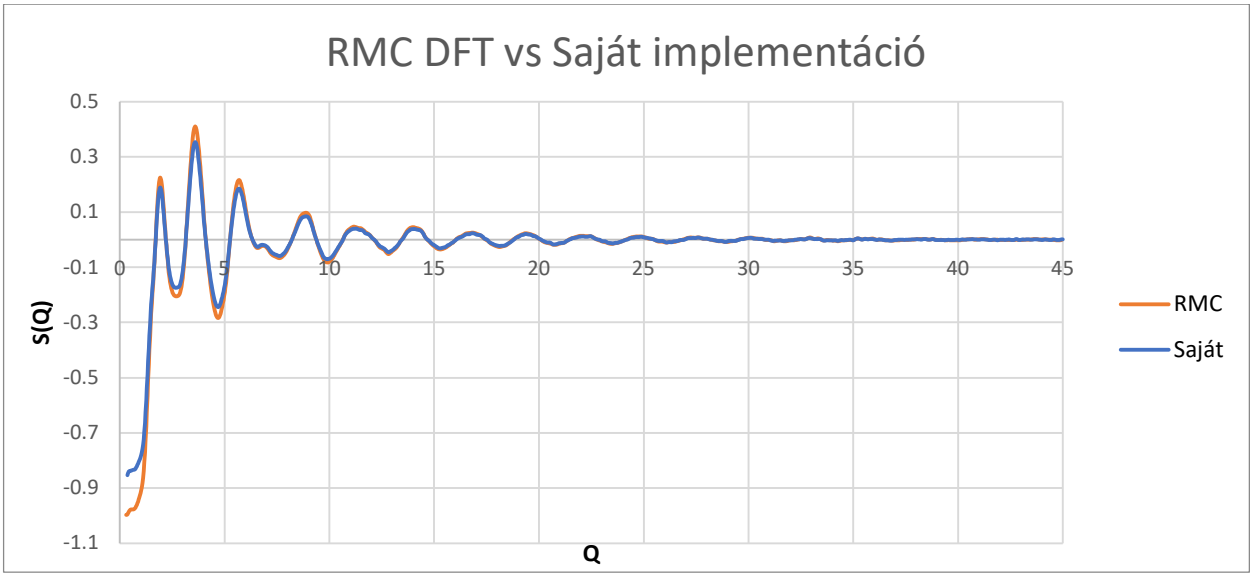
bool lazy_equal(const double a, const double b)

Két double-t hasonlít össze, Epsilon*10-es tűréssel.

bool is_zero(const double a)

Egy szám közel nulla-e?

Eredmény



Sikerült reprodukálni az RMC program által elvégzett műveletet, egyedül a súlyozásban van eltérés. A különbség oka, hogy az RMC dq értéke nem konstans, az spektrum elején kisebb felbontással dolgozik. Ez viszont nem jelent gondot, a programom diagnosztikai eszköznek tökéletesen használható.

Tapasztalatok

Tapasztalatok, manuális memória management nehézkes függvények egymásba ágyazott hívása során, mivel a részeredményeket valahogy fel kell szabadítani, és erről könnyű elfeledkezni. A _CrtDumpMemoryLeaks rá is mutatott két memory leakre, amit fejlesztés közben nem vettem észre.

A program hossza (kommentekkel és üres sorokkal) 580 sor. R-ban megírva ugyanez a kód 13 sor:

```
ppcf <- scan("D:/RMC/AsSeGeTeS/docs/R/input.dat", list(0,0))
r <- ppcf[[1]]
gr <- ppcf[[2]]
q <- seq(0.5,40,by=0.1)
sinqr <- function(q,r) sin(q*r)
sinqrmat <- outer(q,r,sinqr)
rgr <- r*gr-r
sq <- (0.0327*0.1*4*pi*(sinqrmat%*%rgr)/q)+0
print(r)
plot(q,sq)
df <- data.frame(q,sq)
print (df)
write.table(df,"D:/RMC/AsSeGeTeS/docs/R_out/text.dat", sep="\t",
            row.names = FALSE, col.names = FALSE, dec=".")
```

Ebből látszik, hogy a C nem feltétlen a legproduktívabb nyelv az ilyen feladatokra, de tanulásnak rendkívül érdekes és hasznos volt.