ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

**NEUMANN JÁNOS
INFORMATIKAI KAR**

5. sz. melléklet

DIPLOMAMUNKA

OE-NIK Hallgató neve: **Galgóczi Bálint**
2022 Hallgató törzskönyvi száma: **T/005368/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

DIPLOMAMUNKA FELADATLAP

Hallgató neve:	**Galgóczi Bálint**
Törzskönyvi száma:	T/005368/FI12904/N
Neptun kódja:	FHZZFG

A diplomamunka címe:

Gépi tanulási módszerek alkalmazása az optikai spektroszkópiában előforduló nemlineáris modellek illesztésénél
Application of machine learning methods in nonlinear regression probelms encountered in optical spectroscopy

Intézményi konzulens:	Sipos Miklós László
Külső konzulens:	Somogyi Bálint

Beadási határidő: 2022. december 15.

A feladat

Több különböző, az optikai spektroszkópia családjába tartozó mérési módszer is fontos szerepet tölt be az anyagtudományi vizsgálatokban, mind ipari, mind pedig alapkutatási környezetben. A kiértékelés során jellemzően modell függvényeket illesztenek a mért spektrumokra, és az illesztett paraméterekből következtetnek a vizsgált anyag tulajdonságaira. A modellek egydimenziós nemlineáris függvények, paramétereik száma jellemzően magas (6≤). A mért spektrumokra rakódó zaj bizonyos esetekben jelentős lehet. Ezen okokból kifolyólag a hagyományos, gradiens alapú illesztési módszerek nem jól teljesítenek, mert a nemlineáris illesztésnél alkalmazott hibafüggvénynek sok lokális minimuma van. A jelölt feladata alternatív illesztési módszerek vizsgálata. A gépi tanulási módszerek rohamos fejlődésének köszönhetően a nemlineáris regressziós problémákban is történt előrelépés az elmúlt 15 évben. A jelölt modern, bayesi statisztikán alapuló módszereket (például Szekvenciális Monte Carlo) fog vizsgálni optikai méréstechikák (fotolumineszcens spektroszkópia, Raman-spektroszkópia) szemszögéből. Feladata, hogy megismerkedjen a szakirodalommal, elsajátítsa a módszerekhez tartozó elméleti alapokat és tapasztalatot szerezzen az alkalmazásukban. A munkája során elemezni fogja a módszerek teljesítményét megbízhatóság, pontosság és számítási igény szempontjából, valamint kísérletet tesz a módszerek finomhangolására.

A diplomamunkának tartalmaznia kell:

- spektroszkópiai anyagvizsgálatok áttekintése és a kapcsolódó nemlineáris illesztési problémák ismertetése (Fotolumineszens spektroszkópia és Raman-spektroszkópia, a hibafüggvény lokális minimumainak problémája),
- alternatív illesztési módszerek vizsgálata (hagyományos (gradiens alapú) illesztési módszer és néhány alternatívájának bemutatása),
- vizsgált módszerek összehasonlítása teljesítmény, megbízhatóság, pontosság, számítási igény alapján,
- fejlesztési feladatok elvégzése, tesztelése, elemzése,
- konklúzió megtétele.

Ph.

Dr. Vámossy Zoltán
intézetigazgató

A diplomamunka elévülésének határideje: **2024. december 15.**
(OE TVSz 55.§ szerint)

A diplomamunkát beadásra alkalmasnak tartom:

külső konzulens

intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült diplomamunkámban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. 05. 10.

.....
hallgató aláírása

Tartalomjegyzék

1. ABSZTRAKT

A diplomamunkám elkészítése során a kitűzött célom az volt, hogy implementáljam a szekvenciális Monte-Carlo (SMC) algoritmust C++ programozási nyelvet használva. A motivációt az optikai spektroszkópián alapuló metrológiákban felhasznált nemlineáris matematikai modellek illesztése során felmerülő kihívások adták, ahol a hagyományos illesztési módszerek gyakran nem kielégítőek.
A munkát alapos irodalomkutatással kezdtem, ahol feltérképeztem a nemlineáris függvények illesztésére felhasználható módszereket. Ezek a módszerek egy hibafüggvény definiálásán alapulnak, amely számszerűsíti a függvényillesztés sikerességét. Ennek a hibafüggvénynek a minimumát keresve, a feladat a minimumkeresési problémára vezethető vissza. Megvizsgáltam a determinisztikus (gradiens alapú) és nem determinisztikus módszereket is (Monte Carlo algoritmusokat, és heurisztikus módszereket).
Az implementált SMC algoritmus a Markov-lánc Monte Carlo módszer és az evolúciós algoritmusok keresztezésének tekinthető. Az algoritmus generációkkal dolgozik, amely generációkat a paramétertérben elhelyezkedő pontok halmaza alkot. Ezek a "részecskék" rövid Monte-Carlo láncok mentén mozognak a paramétertérben, és a láncok végén új populációkat generálok súlyozott mintavételezéssel. A módszer előnye hogy nem heurisztikus, valamint nem hajlamos beragadni lokális minimumokban.
A program elkészítése során törekedtem a modularitásra, újrafelhasználhatóságra és arra, hogy a modern C++ adta lehetőségeket kiaknázzam. A fejlesztés során erősen támaszkodtam az Eigen template könyvtárra. Az eredmény egy jól struktúrált, fenntartható szoftver lett, ahol jól elkülönül az optikai spektrumok modellezése és az SMC algoritmus.
Az elkészült programot teszteltem. Megbizonyosodtam, hogy jól működik, valóban képes a globális minimum megtalálására a vizsgált illesztési problémák esetében. Készítettem egy összehasonlító tesztet a leggyakrabban használt nemlineáris illesztési módszerrel, a Levenberg-Marquardt (LM) algoritmussal. A teszt eredménye az lett, hogy az SMC módszer sokkal robusztusabb, és nem hajlamos lokális minimumokban megakadni az LM módszerrel szemben. Ennek az előnyös tulajdonságnak viszont meg kell fizetnünk az árát, az SMC módszer futási ideje kb. 3 nagyságrenddel nagyobb az LM módszerhez képest.

2. ABSTRACT

The main goal of this thesis was to implement the sequential Monte Carlo algorithm utilizing the C++ language. The motivation for this work was provided by the many difficulties encountered in optical spectroscopy-based metrology applications, where fitting non-linear functions using convential methods often did not provide satisfactory results.
I carried out a thorough literature survey regarding the methods applicable to the fitting of nonlinear functions. These methods are based on the definition of an error function, which quantifies the goodnes of the fit. By searching for the minimum of the error function, the task becomes a optimization problem. I investigated deterministic (gradient-based) and non-deterministic (Monte Carlo based algorithms and heuristic methods) methods too.
The implemented SMC algorithm can be considered as a crossover between the Markov-chain method and the evolutionary algorithm. It utilizes generations, which are made up from points in the parameter space. These "particles" are moved along short Markov-chains, and new populations are generated at the end of the chains by weighted resampling.
I made efforts to make the resulting code modular and reusable, and tried to use the features of modern C++. During the development, I strongly relied on the Eigen template library. The resulting software is well structured, maintainable, and the modelling of optical spectra and the SMC algorithm are well separated.
I tested the resulting program. I confirmed that the software works well, it is indeed capable of finding the global minimum for the fitting problems I investigated. I created a test to compare the SMC method with the most often used algorithm for non-linear fitting, the Levenberg-Marquardt (LM) method. The result of the test indicated that the SMC method is much more robust in comparison, it is not prone to get stuck in local minima. However, this advantage comes at a cost of 1000$\times$ increased execution time compared to the LM algorithm.

3. BEVEZETÉS

Az utóbbi 50 év egyik legdinamikusabban fejlődő iparága a félvezetőipar. Az integrált áramkörökben található MOSFET-ek száma exponenciális fejlődést mutat, míg karakterisztikus méretük drasztikusan csökkent: 1971-ben az Intel 8008 processzornak a legkisebb struktúrái 10 mikrométer nagyságúak voltak. Csupán pár évvel később, 1977-ben az Intel 8086 processzor (az első, x86 utasításkészlettel rendelkező processzor) már 3 mikrométeres technológiával készült [1]. Napjainkban a modern processzorok már 10 nanométer alatti gyártástechnológiával készülnek. Ennek a drasztikus fejlődésnek a következménye, hogy az integrált áramkörök gyártása egy rendkívül komplikált és költséges folyamattá vált. Egy modern chip elkészítéséhez akár több száz gyártási folyamat szükséges, és akár egyetlen rosszul sikerült gyártási lépés is használhatatlan terméket eredményezhet. Éppen ezért a gyártási folyamatot folyamatosan ellenőrizni kell szúrópróba szerűen a gyártási folyamat kifejezetten kritikus pontjain akár minden szilícium szelet (wafer) tulajdonságait meg kell mérni. Nyilvánvaló tehát, hogy a különböző, félvezetőiparban használt metrológiáknak tartaniuk kell a lépést a gyártás fejlődésével, és egyre nagyobb teljesítményű (egyre pontosabb, megbízhatóbb, gyorsabb stb.) mérőeszközök kifejlesztése a cél. Ezen mérőeszközöket gyakran részben, vagy teljesen automatizált környezetben alkalmazzák, tehát a műszernek nem elég a nyers mérési adatokat meghatároznia, hanem azokból következtetni kell a minta tulajdonságaira.

Egy fontos szempont, hogy a mérési módszer roncsolásmentes legyen (legalábbis ahol ez megoldható), így mérés nem befolyásolja a wafer tulajdonságait. A legtöbb ilyen roncsolásmentes (non-kontakt) módszer optikai elven működik, azaz a fény segítségével vizsgáljuk a minta tulajdonságait. Ezen belül az optikai spektroszkópia a mérési módszerek egy fontos családja. Az optikai spektroszkópia során egy atomokat és molekulákat magában foglaló szerkezet és az elektromágneses hullámok közötti kölcsönhatásokat vizsgálják. Az elektromágneses hullámok képesek az anyagon áthatolni vagy belépve az anyagba kölcsönhatást előidézni a részecskéivel. Ezzel gerjesztett állapotokat hozva létre. A gerjesztett állapotok aztán fotonok kibocsátásával újra alapállapotba kerülnek. A kibocsájtott fény spektrumából következtetni tudnak az anyag tulajdonságaira. Az összefüggés a spektrum és az anyagi tulajdonságok között azonban általában igen bonyolult. Ahhoz, hogy a minta tulajdonságait kvantitatívan jellemezni tudják, a mért spektrumra jellemzően matematikai függvényeket illesztenek. A függvényeket megadó képleteket fizika elméletek alapján választják ki, és ez illesztett függvények paramétereiből származtatják a mérési eredményt: valamilyen szennyező koncentrációja, a mintában fellépő mechani-

kai feszültség vagy egy vegyület félvezető pontos összetétele.
Az illesztendő modellek egydimenziós, jellemzően nem lineáris függvények, gyakran igen sok (5-20) paraméterük van. emiatt igen nagy kihívást jelent a függvényillesztés. A mérés során a spektrumra rakódó zajok és a sokdimenziós nemlineáris függvényeknek köszönhetően az illesztésnél alkalmazott hibafüggvény nagyon sok minimummal rendelkezik, így a „hagyományos", determinisztikus illesztési algoritmusok (gradiens módszerek) nem minden esetben kielégítők.
A dolgozat célja egy olyan illesztési algoritmus implementálása, amely képes kiküszöbölni a gradiens módszerek hiányosságait. A nehéz illesztési feladatokra számos módszert fejlesztettek ki. Ebben a dolgozatban a Monte-Carlo algoritmusok családjába tartozó szekvenciális Monte-Carlo (SMC) módszert fogom vizsgálni, mivel ez a módszer már bizonyított különböző jelfeldolgozási és bayesiánus következtetési problémákban [1] [2] [3].
Az elvégzendő feladat a módszer implementálása C++ programozási nyelvben, és részletes vizsgálata az optikai spektroszkópiában előforduló nehéz függvényillesztési problémák szempontjából. Célom, hogy az SMC algoritmusok által meghatározott illesztések pontossága és megbízhatósága mellett a számítási erőforrás igényt is vizsgáljam, és meghatározzam a paramétereit az algoritmusnak, amelyek ideálisak az alkalmazásra jellemző függvények illesztésére.

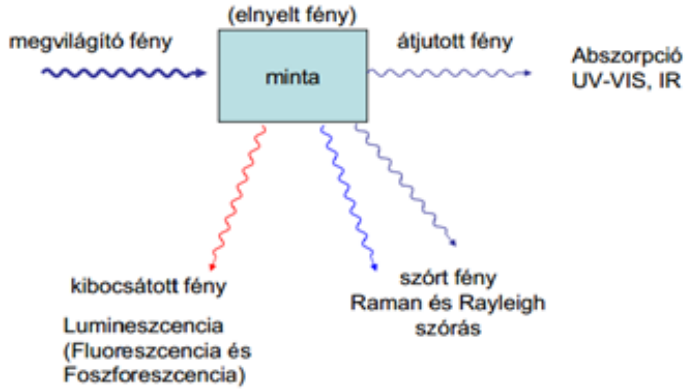
4. IRODALOMKUTATÁS

4.1. Spektroszkópiai összefoglaló

Optikai spektroszkópiai vizsgálatok a mintára érkező fény és az anyag kölcsönhatását vizsgálják. A beérkező fény

- Visszaverődhet a minta felületéről (Reflexió)
- Elnyelődhet a mintában (Abszorpció)
- Kiléphet a minta túlsó oldalán (Transzmisszió)
- Szóródhat a mintán (Rayleigh és Raman)
- Az elnyelt fénnyel fotolumineszcenciát okozhat

A felsorolásban szereplő kölcsönhatásokat az 4.1 ábra szemlélteti.

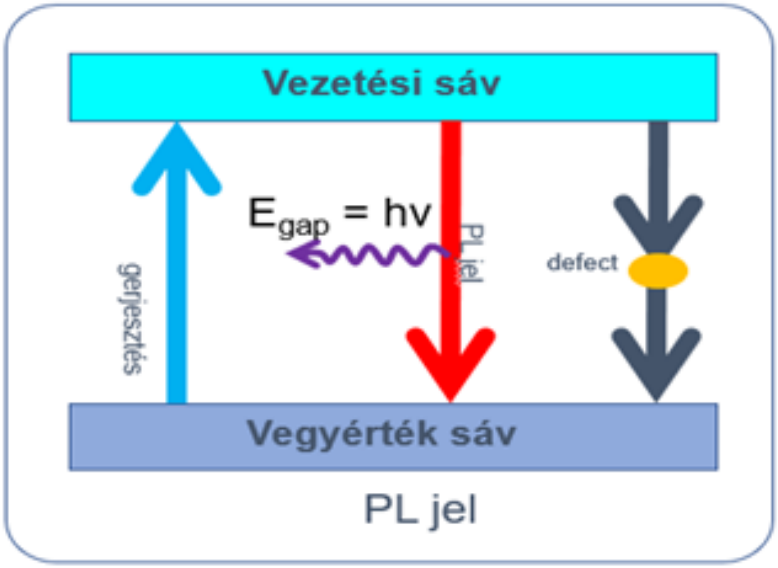


4.1. ábra. Optikai spektroszkópiai módszerek

4.1.1. Fotolumineszcencia

A fotolumineszcencens (PL) spektroszkópia olyan vizsgálati módszer, mely az anyagban levő elektronállapotokról ad információt. A folyamat sematikus ábrája a 4.2 ábrán szemléltetem. Az anyag lézerrel való megvilágítása során fotont nyel el, így az gerjesztett állapotba kerül. Gyors termalizációs folyamatot követően foton kibocsátása révén ismét alapállapotba kerül. A kibocsátott fotonok jellemzik az anyag elektronállapotainak energia szerinti eloszlását. A félvezető anyagokban a sáv-sáv átmenet mellett a tilossávban

levő lokalizált állapotokon keresztül is lejátszódhat sugárzó rekombináció. Ezek a lokalizált állapotok hibahelyekkel, szennyező atomok jelenlétével kapcsolatosak. Így a PL spektroszkópia kiválóan alkalmazható félvezető vékonyrétegek minőségének jellemzésére [4]. PL spektroszkópiai vizsgálatok során általában monokromatikus fénnyel (lézer) vagy lámpával történik a gerjesztés. Az emittált fény monokromátorral komponenseire bontható majd megfelelő detektorral a komponensek intenzitása mérhető.

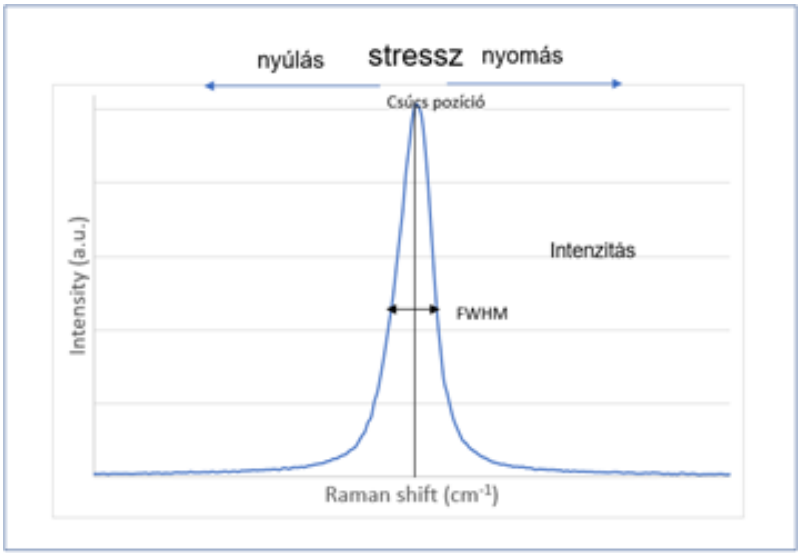


4.2. ábra. Félvezetőben történő fotolumineszcencia sematikus ábrája

4.1.2. Raman-spektroszkópia

A Raman-spektroszkópia a rezgési spektroszkópiai módszerek egyike. A Raman-szórás a fény rugalmatlan szóródása egyes kötések, illetve atomcsoportok rezgőmozgásán. A rezgési spektroszkópiai módszerekkel a molekulákban vagy kristályos anyagokban lévő kötések (kötött atomcsoportok) rezgései, periodikus oszcillációi vizsgálhatók, így áttételesen egyszerre kémiai és lokális szerkezeti információt képesek megmutatni [4] [5]. A beeső fotonok veszítenek vagy növelik energiájukat miközben kölcsönhatásba lépnek az anyag rezgő rácsszerkezetével. A rezgés frekvenciája a kötés erősségétől és az atom tömegétől, míg a spektrumban lévő sávok (csúcsok) száma a kristály szimmetriájától függ. Fő előnye, hogy a vizsgálat a mért anyagot nem roncsolja, a legegyszerűbb esetben még minta előkészítésre sincs szükség. Azonosítani lehet az anyagokat vagy a kötéstípusokat, a kristályban kialakult stresszt, a kristályosságot, az összetételt és az adalékolást. Továbbá a kristálytani orientációt is meg lehet vele találni. A csúcspozícióból meghatározható

az anyag és annak struktúrája. Amennyiben a csúcs eltolódik akkor bizonyos mértékű nyúlást vagy nyomást szenvedett el a kristályszerkezet. Továbbá ebből lehet következtetni arra, hogy milyen anyaggal lett dopeolva, azaz adalékolva. A félvezetőgyártásban az adalékolás a szennyezések szándékos bevezetése a belső félvezetőbe annak elektromos, optikai és szerkezeti tulajdonságainak módosítása céljából. Például a stresszmentes szilícium Raman csúcsának energiája 520 cm-1. Ha a mérés ettől eltérő értéket ad, akkor ebből a szilíciumkristályban ébredő mechanikai feszültségekre lehet következtetni. Ezeken kívül a félértékszélesség (FWHM) értékéből a kristályosságra, valamint az adalékolás és defektek koncentrációjára lehet következtetni [4]. A raman csúcsot a 4.3 ábra szemlélteti.



4.3. ábra. Egy jellegzetes Raman spektrum

4.2. Modell függvények a spektroszkópiában

Mind a PL spektroszkópiában mind pedig Raman spektroszkópiában a mért spektrum csúcsokból épül fel. Ezeket a csúcsokat jellemzően 3-4 paraméterrel lehet leírni: a csúcs energiája (a csúcs maximumához tartozó energia vagy hullámhossz), a csúcs intenzitása („magassága") és a csúcs félértékszélessége (full width at half maximum - $FWHM$). A leggyakrabban alkalmazott csúcs függvények egyike a Lorentz függvény:

$$\mathrm{f}\left(\mathrm{x}\right) = \frac{\mathrm{I}}{\pi\gamma}\left[\frac{\gamma^2}{\left(\mathrm{x}-\mathrm{x}_0\right)^2+\gamma^2}\right] \tag{4.1}$$

ahol I az intenzitás, x_0 a csúcs energiája, γ pedig a félértékszélesség fele: $FWHM = 2$

γ. Egy másik gyakori csúcs függvény a Gauss-függvény:

$$f(x) = \frac{I}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(x-x_0)^2} \tag{4.2}$$

ahol I az intenzitás, x_0 a csúcs energiája és $FWHM = 2\sqrt{2ln2}\sigma$.

4.3. Minimalizálási probléma

Egy mért spektrumot a N_s darab (x, y) pont határoz meg: (x_i, y_i) ahol N_s a spektroszkóp kamera pixeleinek száma, az x_i értékek a mért hullámhosszokat míg az y_i értékek az ezekhez tartozó spektrum intenzitásokat jelölik. Az illesztett matematikai modell az x_i pontokban:

$$\widetilde{y_i} = f(x_i, \theta_1, \theta_2, \ldots, \theta_n) \tag{4.3}$$

ahol $\theta_1, \theta_2, \ldots, \theta_n$ a modell paraméterei. Cél az optimális paramétereket $\theta_1^*, \ldots, \theta_n^*$ megtalálása, amelynél a legjobb egyezés kapható a mért spektrum y_i és a modellezett spektrum $\widetilde{y_i}$ között. Ezt az egyezést számszerűsíteni lehet egy távolság függvénnyel $d(y, \widetilde{y})$, amely lehet az abszolút hiba átlaga:

$$d_{abs}(y, \widetilde{y}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - \widetilde{y_i}| \tag{4.4}$$

vagy az átlagos négyzetes hiba:

$$d_2(y, \widetilde{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \widetilde{y_i})^2 \tag{4.5}$$

$d(y, \widetilde{y})$ a hibafüggvény, amelynek minimuma adja a legjobb egyezést a szimulált és mért spektrumok között. Így könnyen belátható, hogy a nemlineáris modell illesztési probléma matematikailag egyenértékű egy általános minimalizálási problémával. A nemlineáris illesztési problémákra számtalan algoritmust fejlesztettek ki [6]. Ebben a dolgozatban nincs lehetőség az összes bemutatására, de a következőkben illusztrálom a hagyományos, determinisztikus algoritmusok hiányosságát egy minimum keresési probléma vizsgálatával. A determinisztikus minimumkeresési módszerek jellemzően egy kezdőpontból indulnak ki és innen lépkednek olyan irányokba, amerre a függvény értéke csökken. Ezen módszerekre két családra oszthatók: a gradiens alapú módszerekre, ahol a lépés irányának meghatározásához ki kell számítani a függvény gradiensét az adott pontban, illetve a gradiens nélküli módszerek, ahol erre nincs szükség (ezek a módszereket akkor alkalmazzák, amikor a függvény nem deriválható, vagy nagyon költséges kiszámolni a deriváltját).

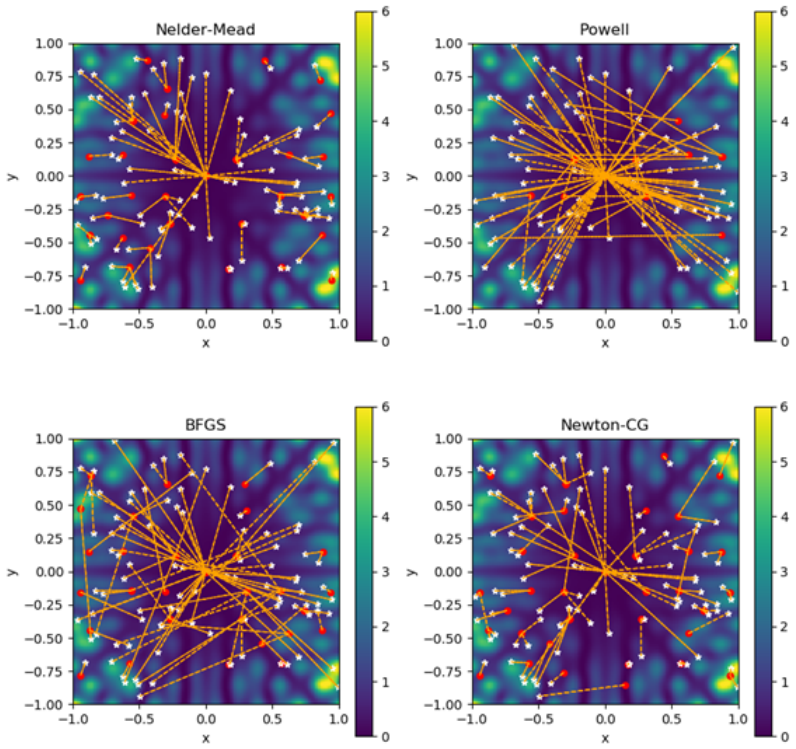
A gradiens alapú módszerekre példa a konjugált gradiens módszer és a BFGS (Broyden-Fletcher-Goldfarb-Shanno) módszer, míg a Nelder-Mead algoritmus és Powell módszere deriváltakat nem igénylő algoritmusok [6] [7]. Adott a következő függvény [8], amelynek a minimuma az alábbi függvénnyel határozható meg:

$$f(x,y) = (xsinsin(20y) + ysinsin(20x))^2 coshcosh\ (sinsin\ (10x)\ x)\ + ((10y)\ - ysinsin\ (10x) \tag{4.6}$$

Ez a függvény ugyan csak 2 dimenziós, de így is sok lokális minimummal rendelkezik. A globális minimuma a $0, 0$ pontban van. Kérdés, hogy az említett módszerek milyen kiindulási pontokból képesek megtalálni a helyes minimumot. Ehhez generáltam 100 véletlen pontot az $x, y\epsilon\,[-1, 1]$ intervallumban, és minden pontra végrehajtottam az illesztést a négy módszerrel a `scipy` Python csomag `optimize()` függvényével. Az eredményeket a 4. ábrán ábrázoltam. Látható, hogy az algoritmusok a 100 kiindulási pont egy részében megtalálják a helyes minimumot, de sok esetben a minimumkeresés megakad egy lokális minimumban. Ez a példa egy mesterségesen generált függvény rengeteg lokális minimummal, amely jól illusztrálja a lokális minimumok problémáját. Valódi alkalmazások esetében a determinisztikus algoritmusok gyakran megbízhatóan működnek, és viszonylag rossz kiindulási pont esetében is megtalálják a lokális minimumot. A spektroszkópiában előforduló csúcs függvények illesztésének esetében a tapasztalat az, hogy determinisztikus algoritmusok jól működnek, ha az egyes spektrum csúcsok közötti átlapolás minimális, és az illesztett csúcs a kezdeti paramétereivel valamennyire átfed a mért spektrum csúcsaival.

4.4. Hasonló rendszerek

Számos regressziós és optimalizációs algoritmus létezik, ezért ebben a fejezetben szeretnék bemutatni néhányat ezek közül. A bemutatott módszerek három csoportba sorolhatók. Gradiens alapú módszerek, ahol a hibafüggvény gradiensével ellentétes irányba lépkedve próbáljuk annak minimumát meghatározni. A gradiens alapú módszereket két családba sorolhatók: első és másodrendű módszerek, annak függvényében, hogy a hibafüggvény minimuma körüli Taylor-sorfejtésében első, vagy másodrending megyünk el. Első rendű módszerek esetében (például a gradient descent módszer) csak a gradiens vektor kiszámítására van szükség, míg másodrendű módszerek esetében (konjugált gradiens, BFGS, Newton módszer, Levenberg-Marquardt algoritmus) meg kell határoznunk közelítőleg a Hesse mátrix inverzét is. Jellemző, hogy a másodrendű módszerek kevesebb lépés-



4.4. ábra. Minimum keresés 100 véletlen pontból kiindulva. A fehér csillagok jelölik a kiindulási pontokat, a piros pontok pedig a megtalált minimumokat. A narancssárga vonalak jelzik, hogy mely kiindulási pontok melyik (lokális) minimumba találnak be.

ből konvergálnak [9] az első rendűekhez képest, viszont egy-egy lépés számítási költsége jóval nagyobb. Részletesebben csak a gradiens descent módszert és annak sztochasztikus változatát tárgyaljuk (mivel ez talán a legfontosabb gradiens alapú algoritmus), illetve a Levenberg-Marquardt algoritmust mutatjuk be, mivel az a de facto sztenderd módszer nemlineáris függvényillesztési problémák megoldására. A következő kategória a heurisztikus módszerek csoportja. Ezek a módszerek gyakran a természetből merítenek ihletet. A genetikus algoritmusok a természetes kiválasztódást utánozzák, míg a részecske raj optimalizációt a madárrajok viselkedésének leírására kifejlesztett algoritmusból származtatták [10]. Ezeknek az algoritmusoknak rengeteg változatuk létezik, és gyakran kiválóan teljesítenek nehéz optimalizációs problémák esetében is. A harmadik csoportba a Monte-Carlo alapú algoritmusok sorolthatók. Mivel ezekről az algoritmusokról sok szó esik a dolgozatban, ebben a fejezetben csak az általunk vizsgált SMC algoritmusnak pár fontos alternatívája kerül ismertetésre.

4.4.1. Első rendű gradiens módszer (Gradient descent)

Alapvető nem lineáris optimalizációs módszer. Lokális minimumok keresésére szolgáló első rendű iteratív algoritmus, amelyet differenciálható függvényen lehet végrehajtani. Az alap gondolta az, hogy a függvény gradiensével ellentétes irányba lépkedve a lokális minimum helye meghatározható. Ha egy többváltozós függvény differenciálható egy adott pont környezetében, akkor a többváltozós függvény süllyedése akkor a leggyorsabb, ha a függvény negatív gradiense felé történnek lépések:

$$a_{n+1} = a_n - \gamma \nabla F\left(a_n\right) \tag{4.7}$$

Ahol a egy következő pont helyét mutatja, míg γ a lépés nagyságát reprezentálja. A függvény gradiensét azért szükséges kivonni, mivel a gradienssel szembe van a haladás iránya, hogy a lokális minimumot találjuk meg. A módszer hátrányai, hogy csak lokális minimumhoz történő konvergencia garantált. Valamint a konvergencia sebessége sem optimális. Gyorsabb konvergáló algoritmusok például a Newton módszer és a konjugált gradiens módszer. [11]

4.4.2. Levenberg–Marquardt módszer

Másodrendű gradiens alapú módszer, amelyet a leggyakrabban alkalmazzák függvényillesztésre. A legkisebb négyzetek elvén alapul, az algoritmus kihasználja, hogy a hibafüggvény négyzetes hibák összege, így a Hesse-mátrix meghatározható a második deriváltak kiszámítása nélkül. Iteratív módszer. A Gauss-Newton metódus és a gradient descent módszer egyesítése, ezek hiányosságainak kiküszöbölésére. Ha egy függvény minimumának keresésének megoldása távol van a valós minimumtól, az algoritmus a gradient descent módszert használja. Ez lassú, de végeredményben jó megoldást fog adni. Amikor az aktuális állapot közel áll a helyes megoldáshoz, akkor a Gauss-Newton módszer érvényesül. A módszer előnye, hogy nagyon jó a lokális minimum keresése. Az adattudományokban széleskörűen alkalmazott numpy csomagban az alapértelmezett módszer görbeillesztésre, de a jelenleg a Semilabon belül is ezt a módszert használjuk az optikai spektrumok elemzésére. Hátránya, hogy a többi determinisztikus módszerhez hasonlóan nem alkalmas nehéz függvényillesztési problémák globális optimális megoldását megtalálni, ha az messze van a kiindulási ponttól. [6]

4.4.3. Sztochasztikus gradiens módszer (Stochastic gradient descent)

A determinisztikus gradiens alapú módszerek alternatívája. A gradient descent módszerrel szemben, ahol a teljes rendelkezésre álló adathalmaz felhasználásra kerül a hibafüggvény gradiensének kiszámítása során, a sztochasztikus gradient descent módszer a teljes adathalmaz véletlenszerűen kiválasztott részhalmazaira (úgynevezett „batch”-ek) számolja ki a hibafüggvény gradiensét. Mivel minden lépés előtt új batch-et választunk ki véletlenszerűen, képes bizonyos lokális minimumok elkerülésére. Az algoritmus talán legnagyobb előnye, hogy gyorsabb, mint a gradient descent metódus. Neurális hálózatok tanításánál az egyik leggyakrabban alkalmazott módszer. Hátránya, hogy tanulási rátát kell választani, aminek rossz megválasztása lassú konvergenciához vezethet. [11]

4.4.4. Genetikus algoritmusok

Heurisztikus függvény optimumkeresési algoritmus és azok csoportja. Előnye, hogy képes egy adott tulajdonságú elem megtalálására is. Az algoritmusok ötlet az evolúcióbiológiából származik, a genetikus algoritmusok az evolúciós algoritmusok csoportjába tartozik. Széleskörben alkalmazott és ismert algoritmus. Ahogy az evolúció is működik, a genetikus algoritmusnak van egy populációja, amely a megfelelő jelöltekből, egyedekből áll, amelyek folyamatos evolválása egy jobb megoldáshoz fog vezetni. Minden phenotípus rendelkezik paraméterekkel, amelyek képesek mutálódni és változni, ezeket genotípusnak hívjuk. Az evolúció egy véletlenszerűen kiválasztott egyeddel kezdődik, majd folyamatos iteráció során, melyet generációnak neveznek, fejlődik a fitness folyamat által. Fitness folyamat során az elvégzett célfüggvény értéke kerül kiszámításra. A legjobban egyező jelölt sztochasztikusan kerül meghatározásra, ami után rekombinálódik és véletlenszerűen mutálódik, módosul a következő generáció számára. Az új generációban lévő egyedekkel indul az algoritmus következő generációja. Általában akkor ér véget az algoritmus, amikor a generációk maximális számát elérte, vagy a populáció elétre a maximális fittség szintjét. Kezdeti lépésként inicializálni kell a populációt meghatározva annak méretét, ami a megoldandó feladat méretével függ össze. Akár több ezer egyed is lehet egy ilyen populációban. Továbbá meghatározásra kerülhet az is, hogy melyik az a terület, ahol számíthatunk a megfelelő egyedekre, a terület kijelölésével. Ezt követően történik a kiválasztási folyamat, ahol az egy generációban lévő azon egyedeket választják ki, ahol a fitnessz alapján értékelt egyedek megfelelnek, vagy az általunk elvárt végeredménynek megfelelnek. A továbbiakban a kiválasztott egyedekből új generációt képeznek a rekombináció és a mutáció által. A folyamat addig zajlik amíg el nem éri a

leállás feltételeket, amelyek lehet az, hogy megtalálásra került a minimum, vagy elérte a maximális generációt, vagy manuális lett leállítva. [12] Genetikus algoritmusok költségesek, azonban rendkívül jól párhuzamosíthatóak. Amennyiben nincs jól meghatározott leállítási feltétel, akkor nem tudható biztosra, az, hogy mikor konvergál.

4.4.5. Részecske raj optimalizáció (Particle swarm optimization)

Hasonlóan a genetikus algoritmusokhoz a particle swarm optimalizáció (PSO) is az evolúciós algoritmusok csoportjába tartozik. A decentralizált, önszerveződő természetes vagy mesterséges rendszerek kollektív viselkedését kutatja. A tipikus PSO módszerek egyszerű ágensek vagy boidok populációjából állnak, amelyek lokálisan kölcsönhatásba lépnek egymással és környezetükkel. Az inspiráció gyakran a természetből, különösen a biológiai rendszerekből származik. Az SI-rendszerben az ágensek nagyon egyszerű szabályokat követnek. Nincs központi irányítási struktúra, amely megszabná, hogy hogy viselkedjenek lokálisan, amely bizonyos fokig véletlen. Azonban az ágensek közötti kölcsönhatások olyan "intelligens" globális viselkedés kialakulásához vezetnek, amely az egyes ágensek számára ismeretlen. A természetből ismert madárraj minta alapján lehet értelmezni a PSO módszert. A PSO fő jellemzője az önszerveződés. Ez egy olyan spontán folyamat, amelyben a globális rend vagy koordináció egy kezdetben rendezetlen rendszer összetevői közötti helyi kölcsönhatásokból alakul ki. Ezt nem irányítja semmilyen a rendszeren kívüli, vagy belüli ágens. Ehhez három fontos összetevőnek kell teljesülnie. Az erős dinamikus nem linearitás pozitív vagy negatív visszajelzéssel: a pozitív visszajelzés segíti elő az alkalmas szerkezetek kialakulását, míg a negatív visszacsatolás ellensúlyozza a pozitív visszacsatolást és segít stabilizálni a kollektív mintát. A kihasználás és felfedezés egyensúlya biztosítja az értékes átlagos kreativitását a rendszernek. Folyamatos interakciók a rajban lévő egyedek közötti adatcsere által, amely elősegíti az információ szétszórását a rajban. Ezen felül öt alapelvnek teljesülnie kell ahhoz, hogy megfelelő legyen az raj intelligenciája. Ezek a közelség, a minőség elve, a különféle válaszelv, a stabilitás és az alkalmazkodóképesség elve. A PSO részecskék raján keresztül végez keresést, amely iterációról iterációra frissül. Az optimális megoldás keresése érdekében minden részecske a raj legutóbbi (pbest) és a globális legjobb (gbest) pozíciója felé halad.

$$\mathrm{pbest}(i,t) = \arg\min_{k=1,\ldots,t} \left[f\left(P_i(k)\right) \right], i \in 1,2,\ldots,N_p \tag{4.8}$$

$$\mathrm{gbest}(t) = \arg\min_{\substack{k=1,\ldots,t \\ i=1,\ldots,N_p}} \left[f\left(P_i(k)\right) \right] \tag{4.9}$$

Ahol i jelöli a részecske indexét, N_p az összes részecske száma, t a jelenlegi iteráció száma, f a fitness függvény, míg P a pozíció. Ezen felül az algoritmushoz tartozik a részecske sebességének V meghatározása is, valamint az inercia súly ω, amelyet a globális kutatás és a helyi hasznosítás egyensúlyára használnak. Továbbá egységesen elosztott véletlen változók r is bevezetésre kerültek gyorsulási konstans c hányadossal együtt.

$$V_i\left(t+1\right) = \omega V_i\left(t\right) + c_1 r_1(pbest(i,t)\ - P_i(t))\ +\ c_2 r_2(gbest(t)\ - P_i(t))) \tag{4.10}$$

$$P_i\left(t+1\right) = P_i\left(t\right) + V_i\left(t+1\right) \tag{4.11}$$

A 4.10 egyenlet első része az úgynevezett inercia reprezentálja a sebességet, amely biztosítja a részecskék szükséges lendületét a keresési térben való barangoláshoz. A második rész a kognitív komponens, ami az egyéni részecskék gondolkodását reprezentálja. Arra ösztönzi a részecskéket, hogy mozogjanak a saját eddigi legjobb helyzetük felé. A 4.11 egyenlet az együttműködést komponens, amely képviseli a részecskék együttes hatását a globális optimum megtalálására. [13] Előnye, hogy egyszerű és költséghatékony. Ezzel szemben a paramétereinek az optimális megválasztása nem triviális.

4.4.6. Monte Carlo algoritmusok

Több fajtája ismert és alkalmazott az iparban is. Egy alapvető Monte Carlo algoritmus a Markov lánc Monte Carlo (MCMC) melyet széleskörűen alkalmaznak, a bayesianus statisztikában is. Erőssége, hogy bonyolult többdimenziós feladatok is megoldhatóak vele. Folyamatos véletlen változókból hoz létre mintákat valószínűségi sűrűség függvénnyel arányos függvényekkel. Ezek a minták felhasználhatók az adott változó feletti integrál kiértékelésére, mint annak várható értéke vagy szórása. Általában láncok sorozatát alakítják ki, melynek kiindulási pontja egy egymástól kellő távolságra lévő és tetszőlegesen kiválasztott ponthalmaz. A láncon keresztül a lépések sztochasztikus folyamatok, melyekhez valószínűségek vannak rendelve. Ezek az algoritmusok úgy hoznak létre Markov-láncokat, hogy egyensúlyi eloszlásuk legyen, amely arányos a megadott funkcióval. [14] A Hamiltoni Monte Carlo módszer a hagyományos MCMC algoritmus egy változata. Az MCMC algoritmus hátrányát, azaz a véletlen bolyongást küszöböli ki azzal, hogy részben véletlenszerűsített hamiltoni dinamikának megfelelő trajektórián mozog. Ez azt jelenti, hogy a részecskéknek nem csak pozíciót, de impulzust is rendelnek így lehetőség van arra, hogy a klasszikus mechanika alapján szimulálni lehessen. A Metropolis-Hastings algoritmus egy változata. A Gauss-féle véletlen bolyongás javaslateloszlással összehasonlítva a

Hamiltonian Monte Carlo csökkenti az egymást követő mintavételezett állapotok közötti korrelációt azáltal, hogy olyan távoli állapotokba történő mozgásokat javasol, amelyek a szimulált hamiltoni dinamika közelítő energiatartó tulajdonságai miatt nagy valószínűséggel elfogadhatóak. A csökkentett korreláció azt jelenti, hogy kevesebb Markov-láncmintára van szükség az integrálok közelítéséhez a célvalószínűség-eloszlás tekintetében egy adott Monte Carlo-hiba esetén. [14] Előnye, hogy a véletlen bolyongáshoz képest sokkal gyorsabban konvergál, viszont sokkal költségesebb is, mivel sokkal több paraméterrel rendelkezik és a potenciális energia gradiensét is ki kell számítani. Szekvenciális Monte Carlo módszer egyesíti több Monte Carlo algoritmus előnyét. A rendszerben sok részecske van, mely a genetikus algoritmushoz hasonlóan mozognak, viszont a mozgásokat a MCMC algoritmus alapján teszik. A részecske halmazoknak több generációja is van, de mutáció és rekombináció helyett Monte Carlo módszert használnak. Előnye, hogy a genetikus algoritmussal és a részecske raj optimalizációval szemben nem heurisztikus, könnyebb a konvergencia tulajdonságokat vizsgálni. Továbbá a sok generált részecske felhasználása miatt nem hajlamos beragadni a lokális minimumokba. Valamint jól párhuzamosítható a nagy számú részecskének köszönhetően. Ezen előnyök figyelembe vételéve azonban elmondható, hogy költséges módszer. Az SMC módszert egy későbbi fejezetben mélyebben bemutatom.

4.5. Matematikai alapok és jelölések

Ebben a fejezetben röviden összefoglalom azokat a matematikai alapokat és jelöléseket, amelyek az algoritmus megértéséhez nélkülözhetetlenek. A cél az olvasó segítése a későbbiekben használt jelölések bemutatásával. Ha X egy valószínűségi változó, amelynek sűrűségfüggvénye f, akkor $x_1,\ x_2, \ldots,\ x_n \quad f$ jelöli, hogy X-ből n darab mintát veszünk. A valószínűségi sűrűség függvény meghatározza annak valószínűségét, hogy a véletlen változó megegyezik-e egy bizonyos értékkel:

$$P\left(X = x\right) = f\left(x\right) \tag{4.12}$$

$E_f\left[h\left(X\right)\right]$ jelöli a $h\left(X\right)$ függvény várható értékét. A numerikus szimulációkban ezt a várható értéket n darab véletlenül generált minta átlagolásával számoljuk $(x_i \quad f)$:

$$\mu_n = \frac{1}{n}\sum_{i\ =1}^{n} h\left(x_i\right) \tag{4.13}$$

Ha n elég nagy, akkor $E_f\left[h\left(X\right)\right] \ \approx \ \mu_n$. Abban az esetben, ha a modell több valószínűségi változóval (például X és Y) bír, akkor együttes valószínűségi sűrűségfüggvényről

beszélünk:

$$P\left(X = x,\ Y\ =\ y\right) = \varphi\left(x, y\right) \tag{4.14}$$

Ha X és Y nem korrelál egymással, akkor az együttes valószínűségi sűrűség függvényt szét lehet választani az X és Y valószínűségi változó sűrűségfüggvényeinek szorzatára:

$$P\left(X = x,\ Y\ =\ y\right) = f\left(x\right) g\left(y\right) \tag{4.15}$$

Végül bevezetésre kerül a többváltozós normál eloszlás sűrűségfüggvény (általános, korrelált esetben), amely meghatározó szerepet tölt be az SMC algoritmusban. Legyen $x\ =\ (x_1, \ldots, x_k)$ egy k-dimenziós vektor, amely egy, a $X_1,\ \ldots,\ X_k$ véletlen változókkal generált mintát jelöl. Jelölje a $\mu\ =\ (\mu_1, \ldots, \mu_k)$ vektor a k-dimenziós várható értéket ($E\left[X_1\right], \ldots, E\left[X_k\right]$). Ha az $X_1,\ \ldots,\ X_k$ változóknak normál eloszlásuk van, akkor az együttes valószínűségi sűrűség függvény a következő:

$$\varphi\left(x|y,\ \Sigma\right)\ =\ \frac{1}{\sqrt{2\pi}^k \sqrt{det\ \Sigma}}\ e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} \tag{4.16}$$

ahol Σ a korrelációs mátrix, amely mátrix elemeit a következő képlet írja le

$$\Sigma_{ij} = E\left[\left(X_i - E\left[X_i\right]\right)\left(X_j - E\left[X_j\right]\right)\right] \tag{4.17}$$

A továbbiakban nem különböztetem meg az egydimenziós (X) és a többdimenziós ($X = (X_1,\ \ldots,\ X_k)$) véletlen változókat. X-el jelölöm a k dimenziós véletlen vektort, ahol k a paraméterek száma a modellünkben, amíg $x_1,\ x_2, \ldots,\ x_n$ áll n darab k dimenziós véletlen minta, amelyet az X valószínűségi változóból generáltunk. Végezetül bevezetésre kerül a *likelihood* függvény, amellyel mérni lehet a modell illesztésének jóságát a mérési adatokhoz az ismeretlen paraméterek adott értékeinek függvényében. Ha X egy valószínűségi változó egy f sűrűségfüggvénnyel, amely a θ paramétertől függ, akkor

$$L\left(x\right) = f_\theta\left(x\right) = P_\theta\left(X = x\right) \tag{4.18}$$

a *likelihood* függvény, amely megadja annak a valószínűségét, hogy a paraméter értéke θ egy adott szám, ha az X valószínűségi x változó értéket vesz fel.

4.6. Közelítő Bayes-számítás (Approximate Bayesian Computation)

A bayesiánus következtetés az egyik legfontosabb statisztikai következtetési módszer, amely nevét a Bayes tételről kapta, amelyen a módszer alapszik. Bayesiánus statisztikai

érvelés során egy statisztikai modell formájában megfogalmazott hipotézis valószínűsége (előzetes hiedelem-mértéke) kerül frissítésre a beérkezett új információk, bizonyítékok (például események, mérési eredmények, közvélemény kutatások stb.) alapján. Az *utólagos valószínűség*-et határozza meg két előzetes mennyiség figyelembe vételével. A két előzetes mennyiség az előzetes valószínűség és a feltételes valószínűségként is értelmezhető *likelihood* függvény (valószínűség függvény). Az *utólagos valószínűség* a Bayes tétel alapján számolható:

$$P\left(H|B\right) = \frac{P\left(B|H\right)P\left(H\right)}{P\left(B\right)} \tag{4.19}$$

ahol H és B jelöli a hipotézist és a bizonyítékot, $P\left(H|B\right)$ az utólagos valószínűséget, $P\left(H\right)$ az előzetes valószínűséget, $P\left(B|H\right)$ a *likelihood* függvény, és $P\left(B\right)$ a marginális *likelihood* függvény, amely minden hipotézis esetében egyenlő, tehát a különböző hipotézisek relatív valószínűségeit nem befolyásolja. A bayesiánus következtetés a következőképpen zajlik: különböző hipotézisek vannak egy jelenség leírására, és minden egyes hipotézishez egy $P\left(H\right)$ valószínűséget rendelünk. Értelemszerűen annak a hipotézisnek a legnagyobb a valószínűsége, amelyről úgy gondoljuk, hogy a legjobb eséllyel ad magyarázatot a jelenség leírására. Ezután új bizonyíték (B) érkezik, amelynek hatására az egyes hipotézisekhez tartozó valószínűségek frissülnek, tehát az utólagos valószínűség $P\left(H|B\right)$ kerül meghatározásra a beérkező tények függvényében. Értelemszerűen ahhoz, hogy a bayesiánus következtetés végrehajtható legyen, szükség van a *likelihood* függvény (amely meghatározza, hogy egyes H hipotézisek milyen valószínűséggel eredményeznek B bizonyítékokat) ismeretére. Klasszikus példa a pénzérme, amelyet feldobva p valószínűséggel fej $(1-p)$ valószínűséggel írás az eredmény, és a hipotézis p értékére vonatkozik. Ha a kezdeti hiedelmünk p=0.4, akkor a pénzérme feldobás *likelihood* függvényének értékei a bizonyíték két lehetséges értéke esetében P(B=fej | H) = 0.4 illetve P(B=írás | H) = 0.6. A gyakorlatban a *likelihood* függvény gyakran nem ismert, vagy – komplikált statisztikai modellek esetében – rendkívül sok számítási erőforrást igényel a meghatározása. Ezekben az esetekben a Bayes-tétel közvetlenül nem alkalmazható, és közelítő megoldásra van szükség. Innen ered a motiváció a közelítő bayesiánusi számítási (Approximate Bayesian Computation- ABC) módszerek alkalmazására. Az ABC módszerek közös tulajdonsága, hogy megkerülik a *likelihood* függvény kiértékelését, ezért szokás őket *likelihood*-mentes következtetésként is emlegetni. A legalapvetőbb ilyen módszer az ABC elutasításos mintavételezés (rejection sampling) [15]. Ez a módszer szimuláció segítségével közelíti a *likelihood* függvényt, mégpedig úgy, hogy a szimuláció eredményeit (S) összehasonlítja a mért eredményekkel (bizonyítékokkal, B). Ehhez az összehasonlításhoz egy távolság

függvény kerül bevezetésre ($d(x, y)$), amely képes jól jellemezi a különböző paraméterértékekkel számolt statisztikai modellek egyezését az adatokkal. Gyakran alkalmazott távolságfüggvény a négyzetes hibák összege. Egy adott paraméterérték akkor kerül elfogadásra, ha $d(S, B) < \varepsilon$, ahol ε a tűrésparaméter. Az elfogadott eredményekhez tartozó paraméterekből statisztikát készítve az utólagos valószínűségi eloszlás lesz az eredmény. Az ABC megközelítést más statisztikai módszerrel is kombinálni lehet, mint például a Markov-lánc Monte-Carlo módszerrel illetve a szekvenciális Monte-Carlo módszerrel, ez utóbbi képezi e dolgozat témáját. Ezeknek a variációknak a lényegük, hogy statisztikailag hatásosabbá tegyék az ABC szimulációt, a mért adatoktól nagyon eltérő eredményeket adó paraméterértékek elkerülésével [15].

4.7. A szekvenciális Monte-Carlo algoritmus (Sequential Monte Carlo)

A szekvenciális Monte-Carlo algoritmus több fontos, a statisztikai következtetésben és gépi tanulásban gyakran használt koncepciót kombinál. A Monte-Carlo illesztési módszer egy véletlen mintavételezésen alapuló metódus, amely alkalmazása során a cél, hogy meghatározásra kerüljön az adott modell paraméterértékeinek együttes valószínűségi sűrűségfüggvénye, egy mérési adatra (az esetünkben mért spektrumra). Ez úgy érhető el, hogy ebből a céleloszlásból mintákat veszek. A minták átlagának és szórásának kiszámításával megbecsülhető a modell paramétereink értékei és bizonytalanságai. Talán szokatlannak tűnhet, hogy a mérés eredményét sztochasztikus algoritmus segítségével próbálom meghatározni, annak ellenére, hogy a mérés a determinisztikus tulajdonságokra vonatkozik (mint például mechanikai feszültség vagy defektek koncentrációja a félvezetőkben). Két érv motiválja ezt a megközelítést. Először is, a mérések során mindig valamennyi véletlen zaj ül a mért spektrumra. A legfontosabb zajforrások a CCD kamera zaja és a fotonkibocsátás sztochasztikus jellege. Emiatt ugyanazt a mérést többször végrehajtva enyhén eltérő spektrumok jönnek létre, még tökéletesen stabil minta és mérőeszköz esetén is. Így még a determinisztikus algoritmus esetén is a mérési folyamatnak van némi véletlenszerűsége. Másodszor, az összes determinisztikus nemlineáris illesztési módszer alkalmazása során meg kell adni a paraméterek kiindulópontját. Amint azt az előző szakaszokban tárgyaltam, a determinisztikus módszereknek az eredménye gyakran függ a kiindulási ponttól, különösen a sokdimenziós hibafüggvényeknél (sok paraméteres modell esetében). Ha a kiindulási pont közel van a globális minimumokhoz, akkor a determinisztikus módszerek jól teljesítenek, de ha a kezdeti paraméterek pontatlanabbak, nő az

esélye annak, hogy az algoritmus egy lokális minimumban megakad. Ez egy további látszólagos véletlenszerűséget hoz be mérésekbe: még akkor is, ha képes vagyok elkészíteni egy illesztési modellt egy adott mintához, lehetséges, hogy nem vezet sikerre a különböző tulajdonságokkal rendelkező minták esetében, ha a modell hibafüggvényeinek sok lokális minimuma van. Célom egy olyan módszer megvalósítása, amely nem hajlamos elakadni a lokális minimumokon, így sztochasztikus jellege ellenére megbízhatóbb eredményeket nyújt. A *fontossági mintavételezés* (importance sampling) a Monte-Carlo szimulációkban általános fogalma, és valószínűségi következtetés, amelyet egy adott eloszlás becsült tulajdonságainak szórásának csökkentésére használnak. Tegyük fel, hogy mintákat akarunk venni egy f valószínűségi sűrűség függvényből $x_1,\ x_2, \ldots,\ x_n\ \ f$ és a minták segítségével meghatározni az egyes mennyiségeket, például a h függvény várható értékét $E_f\left[h\left(X\right)\right]$:

$$\mu_n = \frac{1}{n}\sum_{i\ =1}^{n} h\left(x_i\right) \tag{4.20}$$

A Nagy Számok Törvényéből következik, hogy $\mu_n\ \rightarrow\ E_f\left[h\left(X\right)\right]$ ha $n \rightarrow \infty$. A *fontossági mintavételezés* mögötti ötlet az, hogy az f eloszlás paramétereinek bizonyos értékei sokkal nagyobb befolyással vannak a μ_n-re, mint más értékek. Célom, hogy nagyobb hangsúlyt fektessek ezekre a fontos paraméter tartományokra úgy, hogy gyakrabban veszek mintákat azokban a tartományokban, ahol μ_n gyorsan változik. Ahhoz, hogy ezt elérjük, a véletlen mintákat egy másik, g eloszlásból generálom. Ezzel a mintavételezés torzul, így a paraméterek becslését az úgynevezett fontossági súlyok bevezetésével kell korrigálni $\omega_i = f\left(x_i\right)/g\left(x_i\right)$.

$$E_f\left[\frac{f\left(X\right)}{g\left(X\right)}h\left(X\right)\right]\ \approx\ \mu_n = \frac{1}{n}\sum_{i\ =1}^{n}\frac{f\left(x_i\right)}{g\left(x_i\right)}h\left(x_i\right)\ =\ \frac{1}{n}\sum_{i\ =1}^{n}\omega_i h\left(x_i\right)\ \approx E_f\left[h\left(X\right)\right] \tag{4.21}$$

A következőkben a Markov-lánc Monte-Carlo (MCMC) módszert írom le dióhéjban. Egy "bonyolult" π valószínűségi sűrűségfüggvényből szeretnék mintát venni. Tegyük fel, hogy hatékonyan kiértékelésre képes, de közvetlenül mintákat venni belőle nem lehet. Ebben segítséget nyújtanak bizonyos sztochasztikus folyamatok (olyan folyamatok, ahol véletlenszerű a mozgás egy paramétertérben), amiket Markov-láncoknak neveznek. Gyakran hívják ezeket memória nélküli folyamatoknak, ahol a következő lépés $x_n \rightarrow x_{n+1}$ csak a pillanatnyi pozíciótól függ (x_n) és teljesen független a korábban bejárt úttól $(x_0, \ldots, x_{n-1})$. Ezek a folyamatok egy stacionárius valószínűségi eloszláshoz konvergálnak. Cél, hogy olyan Markov láncot generáljak, amelynek π a stacionárius eloszlása. Ha ebből a Markov-láncból veszek mintákat, akkor a minták eloszlása a π-hez fog

konvergálni. A Markov-láncokon alapuló Monte-Carlo módszereknek sok változata van, én a Metropolis-Hastings algoritmust mutatom be. Legyen $q\left(X = x\right)$ a Markov-lánc átmenet valószínűsége, amely meghatározza az x pontból y pontba lépés valószínűségét. Ha jelenleg az x pontban vagyok, akkor a következő pontot szimulálhatom az $y\ q\left(x\right)$ átmenet valószínűség alapján. A Metropolis-Hastings algoritmusokban bevezetjük az úgynevezett elfogadási rátát, ahol π a valószínűségi eloszlás, amit nem tudunk mintavételezni:

$$\alpha\left(x\right) = min\left\{\frac{\pi\left(y\right)q\left(y\right)}{\pi\left(x\right)q\left(x\right)},\ 1\right\} \tag{4.22}$$

és generálok egy egyenletes eloszlású r számot a $(0,1)$ intervallumon. Ha $r \leq \alpha\left(x\right)$, akkor elfogadható a lépés és y-ba lépünk. Ha $r > \alpha\left(x\right)$ akkor a javasolt lépés elvetésre kerül és x-ben marad. Ha az algoritmust elegendő ideig futtatom, akkor a véletlen mintáik eloszlása π-hez fog konvergálni. *Temperálás/hőkezelés* is egy fontos, gyakran alkalmazott koncepció a Monte Carlo módszereknél. A temperálás (simulated annealing) lényege, hogy bevezetésre kerül egy T hőmérséklet paramétert, amivel hangolni lehet egy Markov-lánc lépéseinek elfogadását vagy elutasítását. Tegyük fel, hogy egy részecske véletlenszerű mozgását szimuláljuk egy potenciálban. Nagy hőmérsékleten a részecskék mozgási energiája dominál, így a mozgásuk szinte teljesen véletlenszerű (független a potenciáltól). Ahogy a hőmérséklet csökken, a helyzeti energia kezd dominálni, és a részecske egyre nagyobb valószínűséggel fog olyan irányba mozogni amerre a helyzeti energiája csökken. A $T \to \infty$ határesetben a részecske mozgása teljesen véletlenszerű, míg a $T \to 0$ határesetben csak azok a lépések lesznek elfogadva, ahol a helyzeti energia csökken. A minimalizálás szempontjából belátható, hogy egyik véghelyzet sem megfelelő. A $T \to \infty$ esetben a részecskék (MCMC-láncok) nem tudnak konvergálni a minimumba. Másrészről, ha $T \to 0$, a részecske beragad az első lokális minimum helyre, amely természetesen nem elfogadható annál a függvénynél, ahol több lokális minimum van. Optimalizációs problémákra a legjobb stratégia az, hogy nagyon magas hőmérsékletről indulva fokozatosan „hűtöm" az algoritmust, amíg be nem konvergál a globális minimumba. A Metropolis algoritmusban a temperálás úgy történik, hogy „bekapcsolom" a *likelihood* függvényt az elfogadási arány kiszámítása során:

$$\alpha\left(x\right)_{\beta} = min\left\{\frac{\pi\left(y\right)q\left(y\right)^{\beta}}{\pi\left(x\right)q\left(x\right)^{\beta}},\ 1\right\} \tag{4.23}$$

ahol $\beta = 1/T$ azaz, az inverz hőmérséklet. Az algoritmus elején $\beta = 0$, ahol a *likelihood* függvénynek nincs járuléka az elfogadási rátában. Ez azt eredményezi, hogy a Markov-lánc egy véletlenszerű mozgást fog végezni a paraméter térben és a hibafüggvény értéke

(annak a függvénynek, amelyet próbálunk minimalizálni) nem befolyásolja a javasolt lépés elfogadásának valószínűségét. A hőmérsékletet addig csökkentem (β növelésével), amíg a $\beta = 1$. Ezen a ponton a *likelihood* függvény teljes mértékben figyelembe vehető és visszakapom a fent szereplő 4.15 egyenletet.

4.8. Megvalósítási lehetőségek

Érdemes megvizsgálni azt, hogy az algoritmus inspirációjául szolgáló PyMC3, python nyelven íródott algoritmus helyett, olyan programozási nyelvet találjak, amely hatékonyan és újra felhasználhatóan képes helyettesíteni a python nyelvet. Manapság számos programozás nyelv létezik, mindegyik rendelkezik előnyökkel és hátrányokkal attól függően, hogy milyen problémákat szeretnénk megoldani. Általában a programozási nyelvek néhány típusba sorolhatók, azonban ezek a nyelvek többféle programozási stílust támogatnak. Jelenleg a programozónak számos választási lehetősége van a nyelv kiválasztására, de a programozási nyelvek között sok különbség van.

4.8.1. Programozási nyelvek összehasonlítása

A programozási nyelv olyan jelek összesége, amelyet utasítások géphez vagy számítógéphez történő csatlakoztatására terveztek. [16] A programozási nyelveket elsősorban a gép teljesítményének ellenőrzésére vagy algoritmusok kifejezésére használják. A programozási nyelveket imperatív formában vagy deklaratív formában használnak. [17] Továbbá számos programozási nyelvre vezettek be standardokat. A C++ nyelv egy nagyon sokoldalú eszköz, képes figyelemre méltó számítási teljesítményre, lehetővé teszi az információ egyszerű cseréjét a fizikai helytől függetlenül, leegyszerűsít számos mindennapi feladatot és sok olyan folyamat automatizálását, amelyek elvégzése egyébként fárasztó vagy unalmas lenne. Valamint egy standardizáld nyelv, az ISO bizottság szabványosította. Közvetlenül gépi natív kódra fordítja a fordítója. Ennek köszönheti a figyelemre méltó számítási teljesítményét. Erősen típusos nyelv, mely biztosítja, hogy fordítási időben ellenőrizze a típusok megfelelőségét, elkerülve azt, hogy futás időben történjenek hibák és kivételek. Figyelmeztet a hibás változó hozzárendelésre, a rossz függvény visszatérési értékekre, a hiányos eljárás argumentumokra és a függvényhívásokra. [18] A nyelv különböző programozási paradigmák megvalósítására alkalmas, úgymint procedurális, generikus, objektum-orientált. A procedurális programozási paradigma azt jelenti, hogy a programozási nyelv támogatja az eljárások és az alprogramok koncepcióját. A generikus programozás a vázalgoritmusok olyan típusok alapján történő megírására össz-

pontosít, amelyek az algoritmus tényleges használatakor kerülnek meghatározásra, így némi engedékenységet biztosítva a programozók számára, akik el akarják kerülni a szigorú, erős tipizálási szabályokat. [19] Míg az objektum-orientált paradigma lehetőséget biztosít arra, hogy programozás során objektumok formájában lehessen kifejezni a típusokat. Ez a paradigma lehetővé teszi azt, hogy kódunk újrefelhasználható és könnyen érthetővé váljon. A C++ rendelkezik minden platformra elérhető fordítóval, így a megírt kód hordozhatóvá válik. Ezeken felül érdemes figyelembe venni azt is, hogy számos könyvtár elérhető, amelyek szabadon felhasználhatóak. Ez nagymértékben segíti a speciális programozási feladatok gyors implementálását. [19] A C# nyelv hasonlóképpen a C++-hoz erősen típusos, viszont a programozási paradigmák közül az objektum-orientált programozást és a komponens-orientált programozást támogatja. Komponens-orientált paradigma lehetővé teszi a logika elkülönítését és beillesztését. A C# fordító a kódot először egy úgynevezett intermediate language (IL) kóddá fordítja, amely tartalmazza a futtatáshoz szükséges forrásokat, amik lehetnek képek, fájlok, majd ezeket egy .dll kiterjesztésű fájlba menti. A program futtatásakor az IL kód betöltődik a Common Language Runtime (CLR) rendszerébe, ami a Just In Time (JIT) fordító segítségével az IL kódot natív gépi kóddá fordítja. A CLR ezen felül lehetőséget biztosít további szolgáltatások használatára, amihez hozzá tartozik az garbage collection, kivételkezelés, erőforrás management. A CLR által végrehajtott kódot "menedzselt kódnak" nevezik, ellentétben a "nem menedzselt kóddal", amelyet egy adott platformot megcélzó natív gépnyelvre fordítanak. A futási idejű szolgáltatások mellett a .NET kiterjedt könyvtárakat is tartalmaz. Ezek a könyvtárak sokféle megvalósítási típust támogatnak, úgymint fájlkezelés, kivételkezelés, matematikai könyvtárak. [20] Jelenleg a C# .NET Core verziója képes arra, hogy minden platformon használható legyen. Továbbá a .NET keretrendszeren kívül érdemes megemlíteni a NuGet csomag rendszert is, mely az interneten elérhető C# nyelven íródott könyvtárak használatának megkönnyítését szolgálja. A Python egy interpretált, objektum-orientált, magas szintű programozási nyelv, dinamikus szemantikával. Magas szintű beépített adatszerkezetekkel, dinamikus tipizálással és dinamikus kötéssel kombinálva. Nagyon vonzóvá teszik a Rapid Application Development számára, valamint szkriptként vagy ragasztónyelvként használják a meglévő összetevők összekapcsolására. Könnyen elsajátítható szintaktikája és olvashatósága miatt csökkenteni képes a karbantartási költségeket. Python támogatja a modulokat és csomagokat, ami moduláris programozásra és újrafelhasználhatóságra készteti a programozót. Valamint a python nyelv nem erősen típusos. A python interpreter és a szabványosított könyvtár források minden platformra elérhetőek. Fontos megemlíteni, hogy nincs fordítási lépése. Python kódot

bytecode formátumba konvertálás után, amely egy alacsony szintű instrukciós lépésekből álló leírás, az interpreter feldolgozza azt, amely, ahelyett, hogy CPU-n közvetlenül futtatná, virtuális gépen futtatja a kódot. Különösebb összehasonlító benchmarkok használata nélkül is belátható az, hogy a C++ teljesítményben gyorsabb kód futtatást eredményez az által, hogy közvetlenül szólítja meg a CPU-t a natív gépi kódra forduló kód segítségével. Az algoritmus létrehozása előtt azonban fontos információkat hordozhat a memória fogyasztásról és az algoritmus végrehajtási sebességéről egy benchmark elemzés. Ehhez a különböző nyelveken el kell készíteni az algoritmust, majd ugyan azon a számítógépen több alkalommal futtatni, miközben mérések folynak a felhasznált erőforrásokról és eltelt időről. Ezek figyelembevételével készült egy elemzés mátrixok allokálásának és szorzásának mérésére. Az 4.1 táblázat tartalmazza a mért eredményeket. [21]

4.1. táblázat. Programozási nyelvek összehasonlítása mátrix allokáció és szorzás figyelembevételével

Programozási nyelv	Idő [s]	Memória használat [MiB]	Felhasznált energia [J]
Python (NumPy)	0.094±0.001	27.66±00.09 + 57.68±00.03	5.36±00.09
C++/g++ (Eigen)	0.186±0.003	3.54±00.03 + 85.25±00.00	3.76±00.03
C#/.NET Core	7.288±0.069	33.94±00.09 + 69.14±00.03	127.95±02.59
Python	228.084±1.992	10.49±00.02 + 68.58±00.00	5019.49±156.86

A méréseket egy Intel(R) Core(TM) i7-10710U CPU-n végezték. A táblázatban a második oszlopa jelöli az időt, amely a program indulásától a befejezéséig eltelt idő. Itt szembetűnik az, hogy a NumPy könyvtár használatával a leggyorsabb a mátrixműveletek elvégzése. Egyéb könyvtár használata nélkül viszont a python nyelv bizonyult a leglassabbnak. A harmadik oszlop a benchmark folyamat memóriafogyasztása, ahol alap és megnövekedett értékek jelennek meg. Az alap az RSS referenciaérték előtt, a megnövekedett pedig az RSS csúcskeresése a referenciaérték alatt. Az utolsó oszlop szemlélteti a futtatáskor mért CPU energiafelhasználást. Tapasztalatok szerint viszont C++ és C#-ban megírt kódok ráfordított ideje egy nagyságrenddel nagyobb, mint ugyan az az algoritmus elkészítése python nyelven. [22] A nyelvfilozófiák gyakran magyarázzák a nyelvek relatív kifejezőképessége és olvashatósága közötti különbségeket. A Python filozófiája például az, hogy egyszerű, világos és lényegre törő megközelítést kell alkalmazni a program megírásához és el kell fogadni az ebből adódó teljesítménybeli hátrányt.

4.9. Összegzés

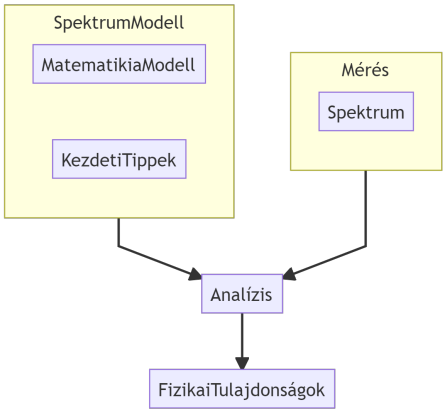
A fentieket összegezve belátható, hogy az algoritmus nagy számítási teljesítményt igényel a Metropolis láncok elindítása után. Ez az iteráció n_{steps} lépést hajt végre, miközben $logP$ és $logL$ értékeket számítja ki minden részecskére. A futási sebesség gyorsítására ezen a ponton szükséges lesz majd párhuzamosítást alkalmazni.

5. KÖVETELMÉNY SPECIFIKÁIÓ

A dolgozatomban C++ nyelven valósítom meg a fent tárgyalt SMC algoritmust. Választásom ok az, hogy az állandó teljesítmény fenntartható legyen, valamint a modern nyelvi környezet és karbantarthatóság is fontos szerepet játszott. Továbbá nem elhanyagolható tény, hogy python nyelv teljesítményben alulmarad a C++-al szemben. Valamint a párhuzamosítás megoldása sem triviális. Azonban elsősorban a teljesítményt vettem figyelembe a választásomkor. Az ipari alkalmazásokban elengedhetetlen, hogy a kapott eredményeket kvázi valós időben megkapjuk. Valamint, a lehető legpontosabb eredményeket adja az algoritmus, azért, hogy az eredmények kiértékelése is pontos maradjon. Továbbá, hogy a mért eredményeket real-time kell feldolgozni. Az előző fejezetből láthatóvá vált, hogy szükséges használnom különböző C++ könyvtárakat, mint például matematikai könyvtárak és párhuzamosításra alkalmas könyvtárak. C++ környezetben nagy népszerűségnek örvendő matematikai könyvtárak például az *Eigen* valamint a *SciMath*. Az *Eigen* könyvtár segítségével fogok elsősorban véletlen mintákat generálni, valamint mátrix és vektor műveleteket megvalósítani. Választásom azért esett az *Eigen*-re mivel dokumentációja nagymértékben kiterjedt, valamint használata is egyszerűbb és logikusabb. Mivel algoritmus számos egymástól független Metropolis láncot indít, ezért kiváló lehetőség adódik a párhuzamosításra. Figyelembe véve a módszer magas számítási igényét, az effektíven párhuzamosított implementáció szinte követelmény a gyakorlati alkalmazhatóság szempontjából. A párhuzamosított implementáció megvalósítására kiváló eszközök lehetnek a megosztott memória elérésű *OpenACC* és az *OpenMP* sztenderdek, amelyeket sikeresen alkalmaztak különböző numerikus C++ programok párhuzamosítására. Ezek a módszerek kifejezetten ígéretesek, mert egyszerű a szintaxisuk, és a CPU mellett a GPU párhuzamosításra is alkalmazhatók.

6. TERVEZÉS

A tervezés elkezdésénél a megoldandó feladatot vettem alapul. A feladat az, hogy olyan algoritmust készítsek gépi tanulási módszerekkel, amely hatékonyan képes a bemenetként adott spektrum, matematikai modell és kezdeti tippek segítségével meghatározni a csúcs pozícióját az x tengelyen, a csúcs intenzitását az y tengelyen, valamint a csúcs félérték szélességét. Az algoritmus futása után, a kapott eredményekből következtetések vonhatóak le arról, hogy a vizsgált anyag milyen fizikai és kémiai tulajdonságokkal rendelkezik. A 6.5 ábra szemléltei a megoldandó feladat sematikus megvalósítását.



6.5. ábra. A megoldandó feladat sematikus ábrája

A feldat meghatározása után kezdtem el felépíteni az algoritmust az előző fejezetkben ismertetett benchmarkok segítségével. Ezek vizsgálata után lehetett elkezdeni a tervezést. Az Eigen egy high-level C++ könyvtár, amely header fájlokból szolgálja ki a kódot. Ez lineáris algebra, mátrix és vektor műveleteket, geometriai transzformációkat és numerikus megoldásokat, valamint az ehhez kapcsolódó algoritmusokat tartalmazó könyvtár. Nyílt forráskódú, amely lehetővé teszi, hogy számos projekt sikeresen alkalmazni tudja, így én is nagy figyelmet fordítottam arra, hogy beépítsem az algoritmusba. Ezen felül nem csak a nyílt forráskódú volta miatt alkalmazom, hanem mivel az úgynevezett „expression templates metaprogramming" technikát alkalmazza, ami azt jelenti, hogy fordítási időben kifejezés fát fordít és egyedi kódot generál, hogy kiértékelhesse ezt. Kifejezéssablonok és a lebegőpontos műveletek költségmodellje segítségével a könyvtár saját hurokfelbontást és vektorizálást hajt végre. Telepítése egyszerű, header fájlokból álló könyvtárral rendelkezik, így nincs lefordítandó bináris. Ezen felül nem szükséges konfigurálni sem, mint a telepítés egy lépése, valamint nincsenek függőségei más könyvtárakra. Az Eigen Core header számos megoldást kínál mátrix és tömb műveletek elvégzéséhez, lineáris algebra feladatok megoldásához. Az ABC-SMC algoritmus implementálásához többnyire 1D és 2D tömbök manipulációja szükséges. Ehhez dinamikus double pontosságú tömbök létrehozása ajánlott. A tömb egyszerű módot biztosít olyan együttható-alapú műveletek végrehajtására, amelyeknek esetleg nincs lineáris algebrai jelentése, például konstans hozzáadása a tömb minden együtthatójához, vagy két tömb együtthatós szorzása. Ezzel a tulajdonságával nagymértékben egyszerűsíti a kódot, mivel felesleges ciklusok nélkül lehet megoldani alapvető feladatokat. A Cholesky header tartalmazza a különleges mátrix műveletet, a Cholesky dekompozíciót, melynek segítségével a szimmetrikus mátrixok alsó trianguláris mátrixának és azok konjugáltjának meghatározására alkalmazható, ha-

sonlóan a LU felbontáshoz. Azonban a Cholesky felbontás sokkal hatékonyabb annál. [23] A véletlen szám generálás nemcsak az algoritmusnak fontos része, hanem a számítás tudományban is nagy szerepe van. Számos kulcsfontosságú terület kulcsfontosságú eleme úgymint globális optimalizációs feladatok, modellezés, szimuláció, robotika, játékok és még megszámlálhatatlan tudományos terület. A helyessége vagy a sebessége egy algoritmusnak kritikusan függ a minőségi véletlenszám generálásától és más kritériumoktól. Széles körben alkalmazott, helytakarékos és statisztikailag jó véletlenszám generátor a PCG véletlen generátor családba tartozó C++ könyvtár. Sebessége, felxibilitása, teljesítménye és hely foglalása miatt jó választás. Ezen felül a szintaxisa is egyszerű, így a kód olvashatóságán nem ront. Az algoritmusnak szüksége van bemenetekre, amelyek egy valós, mért spektrum, kezdeti tippek és spektrum modell. A valós spektrumot egy JSON fájl feldolgozásával juttatom be, bemenetként a programba. A kezdeti tippek is ebből a JSON fájlból kerülnek beolvasásra. A JSON fájl gyors beolvasásához egy „third party" könyvtárat használok. Ez a könyvtár a JSON for moder C++, ami megtalálható a GitHub-on és szabadon felhasználható. Könnyen használható, mivel egyetlen header file segítségével gyors JSON fájl feldolgozás érhető el. Sok esetben egy valós mérési eredmény nagysága akár a több MB nagyságú fájl is lehet, ezért is célszerű alkalmazni a JSON parser-t. Főbb előnyei, hogy szintaxisa intuitív és könnyen integrálható C++ projektekbe. Hatékonyan képes felolvasni JSON típusú fájlokat, így a memória felhasználása és a sebessége kiemelkedő. A bemeneti spektrum és a kezdeti tippek beolvasásán kívül szükséges egy konfigurációs fájl is, amivel egyszerűen lehet paraméterezni az algoritmust. Ezeket a paramétereket a program a futás kezdetén felolvassa, majd beállítja, hogy az algoritmus ezekkel a konfigurálható értékekkel elemezze a valós spektrumot. Az algoritmus kimenete egy JSON fájlba mentett statisztika, amely tartalmazza a végső statisztika adatokat, a populációból kiválasztott spektrum átlagát és annak szórását. Ezen felül az algoritmus elemzéséhez nélkülözhetetlen adatok, úgy mint a stage-ek száma, az futási idő. A C++ kódot Windows operációs rendszer alatt dll kiterjesztésű fájlra kell fordítani.

6.1. ABC-SMC algoritmus leírása

Ebben a fejezetben röviden ismertetem az ABC-SMC algoritmus érdemi elemeit. Az implementációt a PyMC3 [24] Python csomag inspirálta. A PyMC3 könyvtár több, a bayesian analízissel kapcsolatos módszert foglal magába [25]. Ez a csomag numpy és Theano könyvtárakat használja numerikus backend-ként és az Arviz csomagot az eredmények megjelenítésére. Az algoritmus pár kezdeti lépés elvégzése után iteratív jellegű,

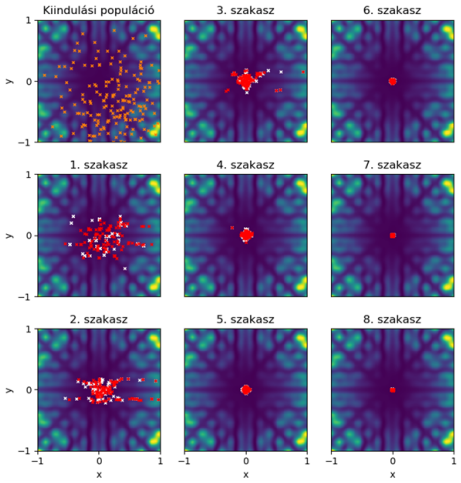
6.6. ábra. ABC-SMC algoritmus folyamatábra

ahol az egyes iterációkat szakaszoknak fogom nevezni. Az algoritmus folyamatábrája az 6.6 ábrán látható.

Az algoritmus működését a 4.3 fejezetben bemutatott minimalizálási probléma segítségével illusztrálom. Mivel az a probléma csak 2 dimenziós, a populáció fejlődése a paramétertérben jól illusztrálható. A 6.7 ábra illusztrálja az algoritmus működését nagy vonalakban. A kezdeti populációt egy durva becslés alapján generáljuk. Az illusztrációban a kezdeti becsléseket normál eloszlásokkal alapján generáljuk, kezdeti becslésünk $\widetilde{x} = 0.3 \pm 0.5; \widetilde{y} = -0.4 \pm 0.5$. A kezdeti középérték a populáció generálásához használt normál eloszlás várható értéke, míg a bizonytalanság a normál eloszlás szórása. Minden szakasz elején új populációt generálunk az előző szakasz populációjából történő súlyozott véletlen kiválasztással. Ezután a populáció minden elemére Monte-Carlo láncot indítunk, aminek hatására a populáció eloszlása a paramétertérben újra rendeződik. Minden következő szakaszban a populáció egyre jobban a minimum közelében csoportosul.

6.1.1. Kezdeti lépések

Az algoritmus elején a felhasználó megad egy előzetes valószínűség eloszlást a matematikai modell összes paraméterére. Például, ha egy illesztendő csúcs energiáját 3 ± 0.5-re becsüli (tetszőleges egységekben), akkor ennek a paraméternek a kezdeti eloszlása egy normál eloszlás lesz 3 várható értékkel és 0.5 szórással. A kezdeti populációt az előzetes



6.7. ábra. A populáció eloszlásának változása a paramétertérben az SMC algoritmus szakaszai folyamán. A fehér és piros keresztek jelölik a populációt az MCMC láncok futtatása előtt és után.

eloszlások alapján határozom meg, amelyek a paraméterek kezdeti ismeretét képviselik. Ha a modell n paraméteres, és N elemszámú mintát hozok létre $(S_1, \ldots, S_N)$, akkor összesen $N \times n$ kezdeti véletlen számot kell generálni. Az első lépésben meghatározom a mintákhoz tartozó valószínűségek logaritmusát

$$logP\left(S_i\right) = logP\left(\theta_1 = x_1, \ldots, \theta_n = x_n\right) = log\prod_{i=1}^{n} P\left(\theta_i = x_i\right) = \sum_{i=1}^{n} logP\left(\theta_i = x_i\right) \tag{6.1}$$

és a *likelihood* függvény logaritmusát, normális eloszlású véletlen hibát (zajt) feltételezve:

$$logL\left(d_i\right) = log\left(\frac{1}{\sqrt{2\pi\epsilon^2}} exp\ \left(-\frac{d_i^2}{2\epsilon^2}\right)\right) = \frac{1}{2}\left(\frac{-d_i^2}{\epsilon^2} + \frac{1}{2\pi\epsilon^2}\right) \tag{6.2}$$

ahol d_i egy előre definiált mért távolság függvény a mért spektrum és a szimulált modell között:

$$d_i\left(y, f\left(x, \theta_i\right)\right) = \frac{1}{N_s}\sum_{k=1}^{N_s}\left|y_k - f\left(x_k, \theta_i\right)\right| \tag{6.3}$$

6.1.2. Iteráció

Minden szakasz elején meghatározom a fontossági súlyokat a mintákra. A minták fontossági súlyait s-edik szakaszban a következőképpen lehet meghatározni:

$$W_i^{(s)} = \frac{\widetilde{L}^{(\beta_s - \beta_{s-1})}(d_i)}{\sum_i \widetilde{L}^{(\beta_s - \beta_{s-1})}(d_i)} \tag{6.4}$$

ahol a $\widetilde{L}(d_i) = L(d_i) - max(L(d_i))$. Miután a súly meghatározásra került, ki kell számítani az effektív minta méretet (N_{ESS}) a következő képlet alapján [26]:

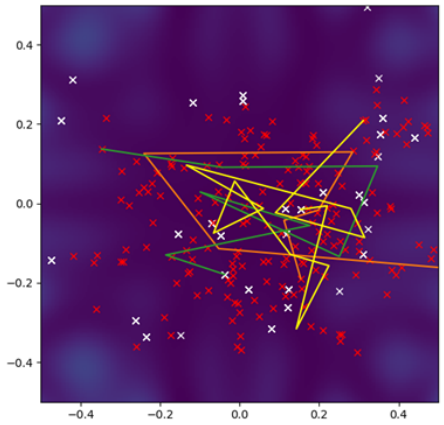
$$N_{ESS} = \left(\sum_i W_i^2 \right)^{-1} \tag{6.5}$$

Optimális teljesítmény elérése érdekében az effektív minta méretet állandó értéken kell tartani az iterációk alatt. Ezt a β_s inverz hőmérséklet állításával érhető el, így $N_{ESS} = N_\tau$ ahol a τ előre definiált paraméter (gyakran $\tau = 1/2$). A gyakorlatban β_s-t addig kell növelni, amíg N_{ESS} el nem éri a megfelelő értéket. Miután meghatároztam a $W_i^{(s)}$ fontossági súlyokat a paraméter vektorok jelenlegi generációjára ($S_i^{(s)}$), elő tudom állítani az új ($S_i^{(s+1)}$) generációt. Ehhez ($S_i^{(s)}$) generációból véletlenszerűen kiválasztok N új mintát, ahol az i-edik minta kiválasztásának valószínűsége $W_i^{(s)}$, és az ismételt kiválasztás megengedett. Ezután kiszámítom az új generáció kovariancia mátrixát, amely később felhasználható a többváltozós normális eloszlás sűrűségfüggvényének meghatározására. A következő lépés az algoritmus hangolása. Egyrészt a Metropolis lépéseinek számát az előző szakasz elfogadási aránya alapján állítható be: a magasabb elfogadási arányok esetében az adott szakasz maximális lépésszámát csökkenteni kell. Másrészt az egyes Markov-láncok átlagos lépésméretét is befolyásolni kell, úgy, hogy az elfogadási arány 0,234 közelében maradjon, amely a véletlen bolyongáson alapuló Metropolis algoritmusok optimális elfogadási rátája [27]. Ezután elindítom az N Metropolis-Hastings láncokat, amelyeket n_{steps} lépés után áll meg, ahol n_{steps} az előzőleg meghatározott lépések száma. A lépések irányait az előzőleg meghatározott, n változós normális eloszlásból generálódik. Minden lépés után kiszámítom $logP$ és $logL$ értéket minden részecskére (mintára) és ezzel meghatározható a temperált logaritmikus poszterior eloszlás:

$$logP(\theta_{t+1}) = logP(\theta_{t+1}) + \beta logL(\theta_{t+1}, y_i) \tag{6.6}$$

Generálok egy egyenletes eloszlású véletlen számot az $(0, 1)$ intervallumban, r-t. Ha $r \leq P(\theta_{t+1}) / logP(\theta_t)$, akkor elfogadható a lépés, ellenkező esetben elutasításra kerül. Az MCMC láncokat az egyes minták által a paramétertérben bejárt trajektóriák ábrázolásával illusztráltam a 6.8 ábrán. Látszik, hogy a vizsgált probléma esetében mind a három véletlenszerűen kiválasztott részecske véletlenszerű lépéseinek nagysága hasonló nagyságrendű az x és y paraméterek szórásával, ami arra utal, hogy még a viszonylag rövid

MCMC láncok is hatékonyan bejárják a rendelkezésre álló paraméterteret.



6.8. ábra. Az MCMC láncok futtatása során a populáció eloszlása újra rendeződik a paramétertérben. Az ábrán csak három véletlenszerűen kiválasztott minta trajektóriája van ábrázolva az átláthatóság kedvéért.

Végül n_{steps} lépés után (az elfogadott és az elutasított lépéseket is beleértve) a láncot megállítom és az elfogadási arány (n_{acc}/n_{steps}) elmentésre kerül az összes Markov-lánc esetében, amit a következő szakaszban fel tudok használni az algoritmus hangolására. Ezzel az aktuális szakasz végére ért és az iterációt a következő szakasszal lehet folytatni. Az iteráció addig megy, amíg $\beta < 1$. Miután az iteráció lefutott, a modell paramétereinek értékeik és a bizonytalanságaik meghatározhatók az átlagok és az N minták paraméterértékeinek szórásával.

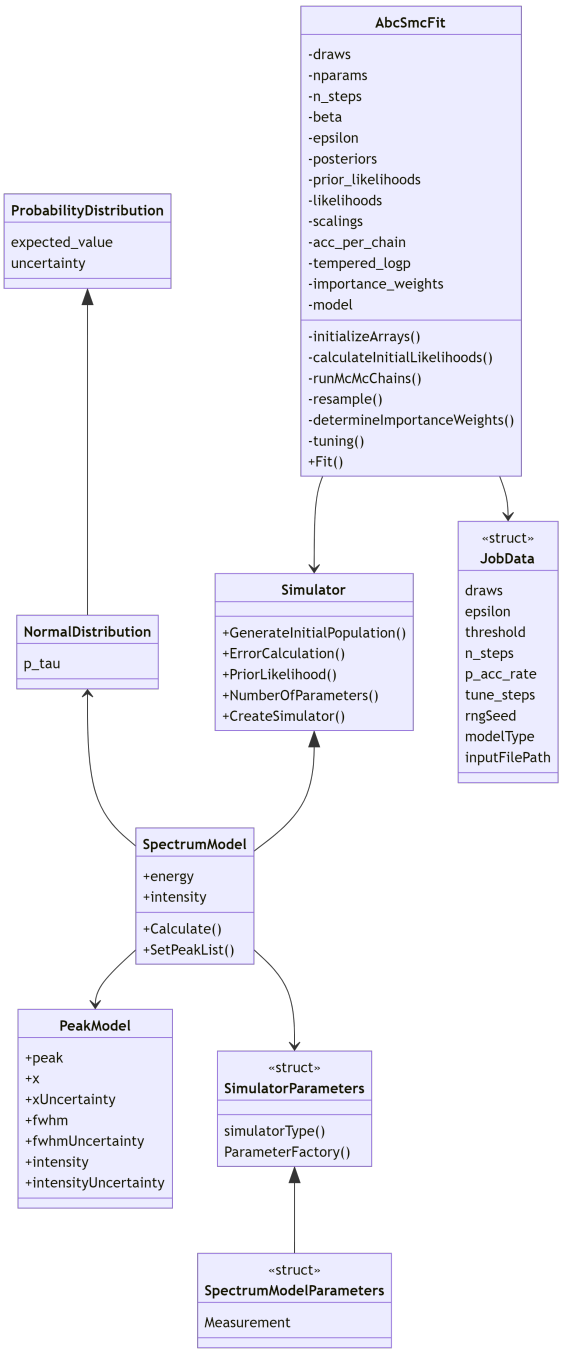
7. FEJLESZTÉS

A fejlesztés előtti nulladik lépés a fejlesztő környezet kiválasztása volt. Választásom a JetBrains által fejlesztett CLion környezetre esett, mivel kényelmes fejlesztési folyamatokat tesz lehetővé. A C++ kód fordításához számos fordító érhető el. Kézenfekvő megoldás volt a Visual Studio által használt fordítót alkalmazni. Ehhez telepíteni kellett a VisualStudio-val együtt a C++ csomagokat. Mivel a CLion környezetben fejlesztettem ezért annak a beállításaiban meg kellett adni a VisualStudio elérési útját, hogy használni tudja a környezet. A fejlesztés kezdetekor szükséges volt kialakítani egy infrastruktúrát, amely lehetővé teszi a kényelmes fejlesztést, valamint azt, hogy a kód újrafelhasználható és több platformon is alkalmazható legyen. A projektet CMake projektként hoztam létre, mivel ennek segítségével a harmadik féltől származó csomagok telepítése egyszerűbb, valamint lehetőséget biztosít arra is, hogy a forráskód fordítását szabályozni lehessen. Ez azt jelenti, hogy a fordítás elvégzése előtt alkalmazhatóak feltételek, valamint kiválasztható akár még a fordító is. Ehhez létre kellett hozni egy CMakeList fájlt, amelybe a megfelelő szintaktika mentén felépítettem a build és package struktúrát. A CMake projektekhez ajánlott mappa struktúrát is elkészítettem, amely áll egy src, doc, include, test és cmake könyvtárakból. Az src tartalmazza a forráskódot, valamint a program belépési pontját a main függvényt is. Az include mappa tartalmazza a „third party” könyvtárakat úgymint az Eigen és a PCG header fileokat. A doc könyvtárban foglalnak helyet a dokumentációs fájlok. A cmake mappa tartalmaz további CMake fájlokat. Mivel ennek az eszköznek széles utasítás palettája van, valamint egymásba ágyazható parancsai ezért szerepel a mappa a projektben. A test könyvtárba kerülnek a unit test-ek. A CMake alkalmas a teszt környezet felállításához, valamint futtatásához. A hatékony ábrák készítéséhez szükséges volt egy ábra készítő könyvtár bevezetése is. Széles körben alkalmazott python könyvtár a matplotlib. A GitHub-on megtalálható a C++ változata, amely egy becsomagolt változata a python matplotlib könyvtárnak. Így szükséges telepíteni legalább a python 3 verzióját, valamint a matplotlib csomagot is. A C++ megvalósítását ezután a CMakeList fájlba fel kell venni, hogy a futtatáshoz szükséges fájlok generálódjanak és használni lehessen a forráskódban is.

7.1. Az SMC algoritmus struktúrája

Mivel a C++ programozási nyelvvel adott a lehetőség, hogy objektum orientált programozást valósítsak meg, így ezek mentén az elvek mentén kezdtem el a struktúra kiala-

kítását. Nagyon fontos szempontnak tartottam azt, hogy a megírt kód újrafelhasználható legyen, mivel további lehetőségek vannak például arra, hogy milyen csúcs típusokat különböztethetünk meg, valamint milyen eloszlást alkalmazunk. Valamint figyelmet kapott az is, hogy a létrehozott osztályok megfeleljenek az objektum orientált programozás paradigmái között szereplő "single responsibility" elvnek is. Ez azt jelenti, hogy minden osztálynak saját felelősségi köre van. A modell készítésére egy külön osztályt vezettem be, ahogyan magának az algoritmusnak a futtatására is. Az egységbezárás elvet is tartottam azzal, hogy más osztály kívülről ne tudjon belső állapotot megváltoztatni. A leszármazást és az absztrakciót is alkalmaztam, mivel több ősosztályt hoztam létre, a kód újrafelhasználhatóság jegyében. Ezek a szülő és absztrakt osztályok tartalmaznak virtuális metódusokat, amely az objektum orientált programozás polimorfizmus megvalósítását jelenti. A fejlesztés során kialakításra került struktúra, amely segítségével az algoritmus újrafelhasználhatósága nagymértékben nőtt. A 7.9 ábra szemlélteti az osztály struktúrát.
A struktúra felépítésénél azt vettem figyelembe, hogy a könnyen lehessen különböző spektrum modelleket alkotni. Ezek a spektrum modellek tartalmazzák a bemeneti modellt, a kezdeti tippekkel és a konkrét spektrumot, amelyen az algoritmust futtatni lehet. Ezek mentén létrehoztam egy base osztályt, amelyből leszármaztam a SpektrumModell osztállyal. Ez a base class a Simulator, ahol az algoritmus spektrumra jellemző metódusai vannak. Ezek a metódusok felelősek az SMC algoritmus spektrumon végzett számításaiért. A leszármazott osztály, a SpektrumModell tartalmazza a konkrét spektrumra vonatkozó részeket, úgy mint az eneregia és az intenzitás, valamint a csúcsok kezdeti értékeit is itt kell magadni. A Simulator osztályhoz tartoznak paraméterek is, amelyeket a SimulatorParameters struct segítségével injektálok a SpectrumModell osztályba. A ParametersFactory metódusával lehet felépíteni a job fájlból kapott értékek alapján a konkrét spektrumot. Ahhoz, hogy az algoritmus számolni tudjon szükséges a SpektrumModell felépítése. Az algoritmus a számítást az AbcSmcFit osztályban van implementálva. Az osztályban számos privát változó és metódus szerepel. Ennek oka az, hogy így könnyebben lehet kezelni az osztály belső állapotának a változását. Továbbá ennek segítségével az osztályban szereplő privát metódusok paraméter listája is lecsökkent. Az AbcSmcFit osztály egyetlen publikus metódusa a Fit metódus, amely bemeneti paraméterként a job fájlból felolvasott JobData struktúrát várja. Az előzőleg felépített SpektrumModell-t a AbcSmcFit osztály a konstruktorában kapja meg, a model privát változóban tárolódik. Ezen felül különválasztottam a valószínűségi elosztálások számolására szükséges objektumokat. Létrehoztam egy ProbabilityDistribution base osztályt. Ebből származok a további eloszlások megvalósításához. Jelenleg csak a normál eloszlás kalkulációhoz van



7.9. ábra. Az SMC algoritmus osztálydiagramja

meg az implementáció, mivel az algoritmus teszteléséhez ez is elegendő. A létrehozott osztály a NormalDistribution, amely tartalmazza a konkrét számításokhoz a metódusokat. A SpektrumModell használja a NormalDistribution osztályt, ezért ahogy az 7.9 ábrán látszik, asszociál a SpektrumModell-el. A fontosabb osztályokon kívül implementálásra került több segéd struktúra is. Ezekre a rendezettség fenntartása miatt volt szükség. A

JobData struktúra tartalmazza azokat változókat, amelyekkel az algoritmust tudjuk beállítani és módosítani. A Config struktúra segítségével lehet meghatározni azt, hogy hol található a job fájl és a measurement fájl, amely a konkrét spektrumot tartalmazza.

7.2. Az SMC algoritmus fejlesztése

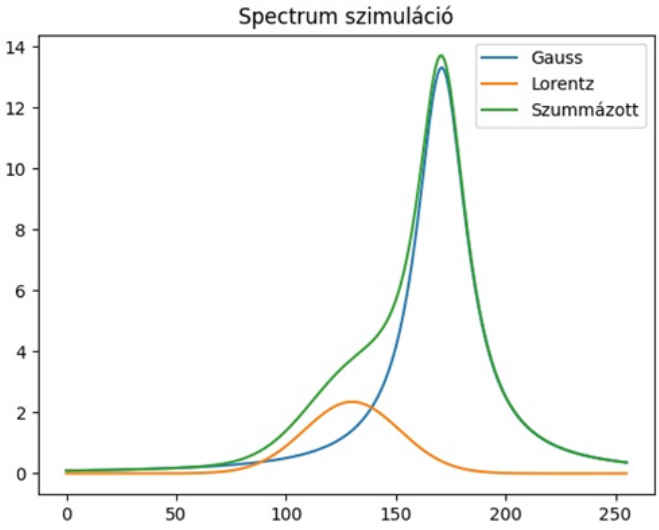
Mivel úgy éreztem, hogy ez lesz a legnagyobb kihívás, magával az SMC algoritmus C++ megvalósításával kezdtem a fejlesztést. Miután az algoritmust sikerül implementálnom, az összes hibát azonosítanom és kijavítanom, lesz értelme tovább haladnom a kiegészítő funkciók implementálásával. Az algoritmus folyamatábrájából kiindulva végzem a fejlesztést. A fejlesztés az Eigen könyvtár használata nélkül kezdődött. Azonban ez azt eredményezte, hogy a kód átláthatatlan lett, mivel nagyon sok vektor műveletet kellett elvégezni ciklus segítségével. Ezért a fejlesztés viszonylag korai szakaszában elkezdtem használni az Eigen könyvtárat (3.4 verzió). Ez jóval rövidebb, átláthatóbb forráskódot eredményezett, hiszen a for ... ciklusok legnagyobb részét ki lehetett váltani az Eigen Array osztályának használatával. A tömbök vagy vektorok kezelése egyszerűsödött, a ciklusok helyett elég matematikai operátorokat használni például arra, hogy a tömb minden eleméhez hozzáadjon egy konstanst. Az Eigen alkalmazásakor oda kell figyelni, hogy az Eigen objektumokat (Array, Matrix, Vector) mindig referenciaként kell kezelni. Amennyiben lehetséges, függvény argumentumok esetében az Eigen objektumokat érdemes konstans referenciaként átadni. Alapvetően a C++ nyelven mindig rossz ötlet az, hogy objektumot értékként adunk át, mivel ekkor egy használhatatlan másolat jön létre az átadott objektumból. Az Eigen esetében azonban az értékként való átadás egyáltalán nem támogatott, és program összeomlásához vezethet. A kiszámíthatóbb viselkedés és ezáltal tesztelhető, ellenőrizhető fejlesztés érdekében szimulált bemeneti spektrumokat használok. Ebben az esetben pontosan tudjuk, hogy az egyes paraméterek értéke mennyi, így az algoritmus sikeressége ellenőrizhető. Szimulált Gauss típusú spektrumot az alábbi kódrészlet alapján generálom.

```
Eigen::ArrayX<double> gaussian(const Eigen::ArrayX<double>& x,
double x0, double fwhm,    double intensity, int npix) {
    Eigen::ArrayX<double> spectrum(npix);
    double sigma = std::abs(fwhm) / 2.35482;
    double c0 = intensity / (sigma * 2.5066283);
    double c1 = 0.5 / (sigma * sigma);

    spectrum = c0 * exp(-c1 * (x - x0) * (x - x0));

    return spectrum;
}
```

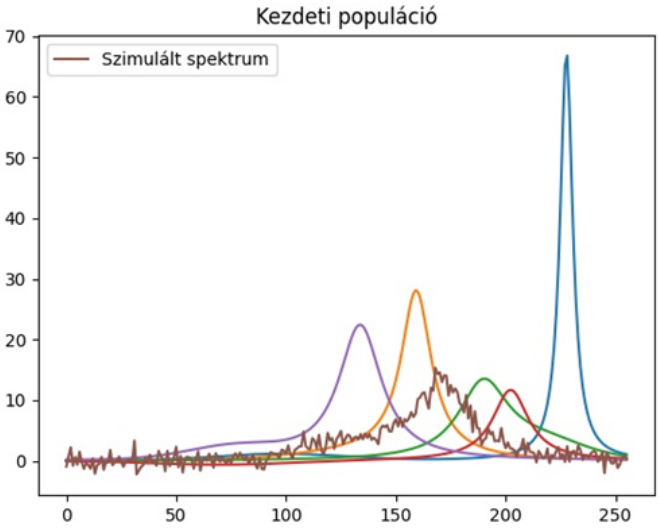
A spektrumszimuláláshoz meg kell határozni az x tengelyt, mint energia vagy ebben a megvalósításban hullámhossz és y tengelyt, mint intenzitás tengelyt. Továbbá meg kell adni a kívánt csúcs pozícióját, a félérték szélességét és intenzitását, ezen felül azt, hogy milyen típusú függvényt szeretnénk. Egyelőre két fajta csúcs függvényt implementáltam, Gauss és Lorentz görbe. Ebből a két típusból generálok egy szimulált mérést, amely a 7.10 ábrán látható. Az így kapott spektrumhoz zajt is adtam, hogy a teszt reálisabb legyen.



7.10. ábra. Szimulált spektrum

Az algoritmus pszeudo-véletlenszám generátorának seed értéke rögzíthető, vagy pedig a C++ Standard Library nem determinisztikus `std::random_device` generátorával

randomizálható. A Standard Library által biztosított pszeudo-véletlenszám generátor (`std::mt19937`) is megfelelőnek bizonyult, azonban egy modernebb Permuted Congruential Generator (PCG) nevű pszeudo-véletlenszám generátorra tértem át. Ez a generátor jelenleg a legelőnyösebb tulajdonságokkal általános felhasználású (tehát nem kriptográfiai értelemben biztonságos) pszeudo-véletlenszám generátor, jobb statisztikai tulajdonságokkal bír, mint a Mersenne-Twister típusú (`std::mt19937`), annál gyorsabb, és kevesebb memóriát foglal [28]. A szimulált függvény létrehozása után a kezdeti populáció meghatározása következik. Ehhez az előzetes eloszlásokat használom. Az előzetes eloszlásokat a felhasználó állítja be, mint kezdeti tippeket, ahol megpróbálom manuálisan meghatározni a szimulált csúcs intenzitását, félérték szélességét és pozícióját. Mivel két csúccsal szimuláltam a spektrumot ezért az utódok (posteriors) egy $N \times 6$ nagyságú vektor lesz. A vektor elemei pedig a paraméterek által megadott érték és a szórás alapján normál eloszlás segítségével kerül meghatározásra. Kezdeti populáció tartalmazza az utódokat, amelyek hasonlóan a vizsgált függvényhez és ezek közül az utódok közül lesz kiválasztva az iterációk alatt a leghasonlóbb. A kezdeti populáció mintája az eredeti függvénnyel a 7.11 ábrán látható.



7.11. ábra. Kezdeti populációból vett minták és a szimulált spektrum

A kezdeti populáció generálása után a mintához tartozó valószínűségek logaritmusát, valamint a *likelihood* függvény logaritmusát is meghatározom. A *likelihood* meghatározásához a 6.2 egyenlet alapján az alábbi kódrészletet implementáltam.

```
1       double likelihood =
2               (-(mean_abs_error * mean_abs_error) / (epsilon * epsilon) +
3               log(1.0 / (2.0 * M_PI * epsilon * epsilon))) / 2.0;
```

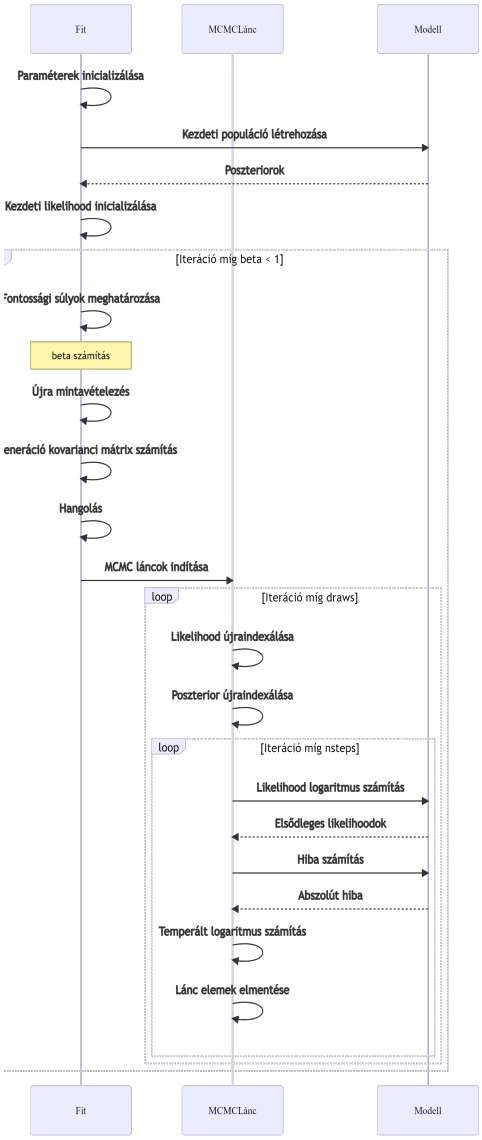
A mean_abs_error változó a kiinduló spektrum és egy lánc által meghatározott új spektrum különbésge. Ekkor a β értéke 0, így az iterációs ciklus kezdetét veszi. Minden iteráció előtt, amelyet stage néven határozok meg, kiszámításra kerülnek a fontossági súlyok. A fontossági súlyok meghatározásának szerepe van abban, hogy a *likelihood* függvény bizonyos értékei ne legyenek sokkal hangsúlyosabbak, mivel ez azt jelentené, hogy ezekben a tartományokban gyakrabban lenne mintavételezés. Ezt kompenzálják a fontossági súlyok. A Markov-lánc Monte-Carlo futtatása előtt a fontossági súlyok segítségével újra kell indexálni a *likelihood* és posterior tömböket, hogy mintavételezés egyenletesebb legyen. A súlyok és az effektív mintaméret segítségével számítom ki a következő stage β értékét is. A fejlesztés során az SMC algoritmus komplexitása nagy kihívást jelentett. Mivel az algoritmus véletlen vektorok populációjával dolgozik, kifejezetten nehéz megtalálni és azonosítani a hibákat. Mivel az én C++ implementációm a PyMC3 csomagban implementált SMC osztályon alapult, ezért a hibakeresés során az a stratégia bizonyult a leghatásosabbnak, hogy a python kód részeredményeit rendszeresen összehasonlítottam az általam fejlesztett C++ változat megfelelő részeredményeivel. Azonban ez a módszer is igen nehéznek bizonyult, mivel véletlen számok sokaságait kellett összehasonlítani a működő python implementáció és a fejlesztés alatt álló C++ algoritmus részeredményei között. Az populáció egyes elemeire származtatott értékek (paraméter koordináta, *likelihood*-érték, fontossági súly, stb.) átlagát, szórását, minimum és maximum értékeit hasonlítottam össze. Szintén informatívnak bizonyult az inverz hőmérséklet, az algoritmus hangolásánál és skálázásánál használt elfogadási ráták. A fejlesztés során számos különböző hibát kellett azonosítani és elhárítani. Egyszerű elírások matematikai formulákban, több dimenziós tömbök oszlopainak és sorainak összekeverése, rossz helyre tett zárójelek, stb. Talán a legnagyobb fejtörést a pszeudo-véletlenszám generátor nem megfelelő alkalmazása okozta. Kezdetben az egyszerűség kedvéért minden osztályban és függvényben létrehoztam a véletlenszám generátor egy-egy új példányát. Ez azt eredményezte, hogy a generált számsorok egymással korreláltak lettek, ami hibás működést eredményezett. Egy konkrét példán mutatom be ezt a problémát. A kiindulási populáció generálásakor a matematikai spektrum modellünk minden paraméterére kell N darab véletlen értéket generálni (ahol N a populáció mérete). Ha minden paraméterre új véletlenszám generátor

objektumot hoztam létre, akkor a keletkező paraméter sorozatok egymással korreláltak lettek. Ha külön-külön vizsgáltam az egyes paraméterek kiindulási populációjának eloszlását, akkor úgy tűnt, hogy jól működik a véletlen eloszlások generálása. Az algoritmus azonban ennek ellenére nem jól működött, és a hibára csak akkor jöttem rá, amikor a különböző paraméterekhez tartozó véletlenszám sorozatokat egymás mellett ábrázoltam, ahol nyilvánvalóvá vált a korreláció. A probléma megoldása az lett, hogy egyetlen egy pszeudo-véletlenszám generátort példányosítok meg, és utána ezt adom oda referenciaként az összes olyan objektumnak és függvénynek, amelyben véletlenszám generálás történik. Szintén meglepetést okozott, hogy a megvalósított SMC algoritmus már akkor is konvergenciát mutatott a globális minimumhoz, amikor még maradtak kisebb hibák a kódban. Ezért volt nagy jelentősége a bizonyítottan jól működő PyMC3 python implementációhoz hasonlítsam az általam fejlesztett C++ implementációt. Miután minden kisebb-nagyobb hibát kijavítottam, az algoritmus működése identikus az őt ihlető PyMC3 csomagban megtalálható SMC algoritmussal. Végezetül elkészítettem az algoritmus szekvencia diagramját (7.12), amely szemlélteti, hogy számos lépés és számítás szükséges a megfelelő működéshez.

Általános fejlesztési gyakorlat, hogy a gyakran használt részeket, amelyek nem tartoznak egy felelősségi kör alá, általában egyenletek számításai, mértékegység átváltások atomi műveletekkel, bővítő metódusok, külön osztályt kapnak. Ez nagyban segíti a kód újrafelhasználás alapelvét, valamint nem sérti meg azt sem, hogy egy osztálynak egy felelősségi köre maradjon. Létrehoztam egy Utils nevű osztályt, amely arra szolgál, hogy az általánosan használt metódusokat egy helyről lehessen elérni. Itt definiáltam például az újra mintavételezést is.

```
1           Eigen::ArrayX<double> resampling(
2           const Eigen::ArrayX<double>& vector,
3           const Eigen::Array<int, Eigen::Dynamic, 1>& indices)
4           {
5               Eigen::ArrayX<double> resampledResult(vector.size());
6               for (int i = 0; i < vector.size(); ++i) {
7                   resampledResult(i)= vector(indices(i));
8               }
9
10              return resampledResult;
11          }
```

Továbbá a statisztika számítást is ebben az osztályban helyeztem el. A statisztika szá-



7.12. ábra. Az AMB-SMC algoritmus szekvencia diagramja

mítást a konzolra logolom ki. Ez tartalmazza a kezdeti populáció statisztikáját, valamint nyomon követi az adott staget az iterációban lévő jelenlegi bétával, a lépések számával, ami azt mondja meg, hogy hány elfogadott lépés van az adott iterációban és a gyorsulási rátát.

```
        Starting population statistics:
                0.2867(0.4932)  -0.3938(0.5055)

        Stage:   0    Beta: 0.000000    Steps: 25    Acc. rate: 1.000000
        Stage:   1    Beta: 0.000465    Steps: 25    Acc. rate: 0.468480
        Stage:   2    Beta: 0.004813    Steps:  7    Acc. rate: 0.366857
        Stage:   3    Beta: 0.049627    Steps:  7    Acc. rate: 0.249000
        Stage:   4    Beta: 0.239617    Steps:  7    Acc. rate: 0.192143
        Stage:   5    Beta: 0.947896    Steps:  7    Acc. rate: 0.301143

        Final population statistics:
                0.0048(0.0936)  0.0068(0.0567)
```

A programnak működéséhez szüksége van mérési adatokra, a spektrum matematikai modelljének leírására, és az SMC algoritmus paramétereinek megadására. A bemenő adatok megadásának módjára számos lehetőség van. Mivel egyelőre még nem kristályosodott ki hogy az SMC algoritmus hogyan illeszkedik a nagyobb matematikai analízis keretrendszerbe, úgy döntöttem, hogy a bemeneti adatokat fájlokból olvasom be. Az általános konfigurációkat a config.json fájl tratalmazza, az elvégzendő illesztési feladatot egy szintén .json kiterjesztésű job fájl, míg a mérési adatokat egy harmadik, szintén JSON fájlban tároljuk. Mind a három fájltípushoz tartozik egy C++ *struct* objektum. A program indulásakor felolvasom a konfigurációs fájlt, amely tartalmazza a job fájl elérési útját, az eredmény fájl nevét és a loggolási szintet. A job fájl elérési útja felülírható opcionális parancssori argumentummal `algorithm.exe job=jobfolder/jobfile.json` formában, megkönnyebítve a szkriptekkel történő felhasználást. Ennek az opciónak a jelentősége a tesztek futtatásánál, illetve olyan szituációban, amikor számos mérési eredményt szeretnénk kiértékelni. A job fájl tartalmazza az algoritmus állítható paramétereit. Ebben a fájlban lehet meghatározni, hogy milyen modellt illesszen az algoritmus, a futtatni kívánt Markov láncok számát és az algoritmus numerikus paramétereit (például *threshold* és *epsilon*). A measurement fájl tartalmazza a mért spektrumot, továbbá az illeszteni kívánt csúcsot vagy csúcsokat a paraméterek kezdeti tippekkel (előzetes várakozásokat reprezentáló valószínűségi eloszlások).

8. TESZTELÉS

8.1. Az algoritmus működése

A tesztelés során ellőször arról győződtem meg, hogy az algoritmus jól működik, képes a spektrum matematikai modelljének pontos becslésére. A tesztelés során egy viszonylag egyszerű, két csúcsból álló spektrumot használtam. Ez egy viszonylag egyszerű matematikai modellt eredményez, ugyanakkor az illesztés már problémás lehet a hagyományos, globális minimumkeresésre alkalmatlan determinisztikus módszerek esetén. A szintetikus spektrumot egy Gauss és egy Lorentz függvény összegeként, a következő paraméterekkel definiáltam:

$$Lorentz: \ x_0 = \ 0.67; \ FWHM \ = \ 0.11; \ intensity \ = \ 2.3 \tag{8.1}$$

$$Gauss: \ x_0 = \ 0.51; \ FWHM \ = \ 0.20; \ intensity \ = \ 0.5 \tag{8.2}$$

A tesztelés folyamán feltételeztem, hogy nem ismerjük a paraméterek valódi értékét, tehát a kezdeti becslésünk pontatlan.
A kiindulási modell, amelyet a 7.10 ábra reprezentál a 8.2 táblázatban foglalt várható értékekkel és szórással rendelkezik.

8.2. táblázat. Kiindulási modell várható értékei és szórasai

		Kiindulási érték	Szórás
Gauss	x	0.65	0.15
	FWHM	0.09	0.04
	Intenzitás	2.65	0.5
Lorentz	x	0.45	0.30
	FWHM	0.25	0.07
	Intenzitás	0.35	0.15

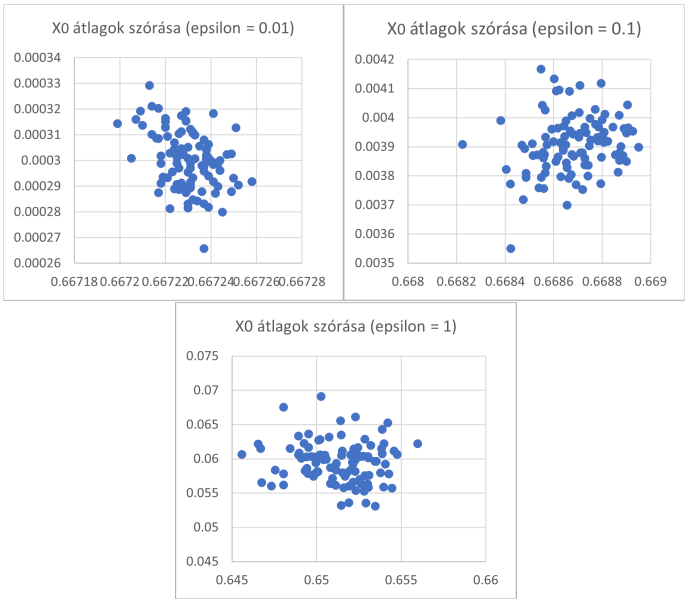
Sorozatos méréseket végeztem az algoritmus futtatásakor és elkészítettem a végleges populáció statisztikát, hogy elemezni tudjam az algoritmus működését. Az első tesztsorozatnál a mérési zaj véletlenszám generátorának seedjét rögzítettem. A tesztelés során az különböző *epsilon* paraméterekkel is kipróbáltam az algoritmust. Ez a paraméter a mérési zajjal hozható összefüggésbe. Nagyobb *epsilon* zajosabb mérésre utal, míg teljesen zajtalan mérések esetén érdemes az értékét nagyon picire állítani. Az *epsilon* paraméter

értéke nem kritikus az algoritmus konvergenciájának szempontjából, viszont meghatározza a végső populációból becsült paraméterértékek szórását. Ha nagyon kicsi *epsilon* értéket használunk az SMC algoritmus futtatása során, akkor a becsült értékek szórása kisebb lesz a reális értéknél, így alul becsüljük a zaj nagyságát. Túl nagy érték esetében viszont az iteráció túl gyorsan megáll, és a paraméterek hibáját túlbecsüljük. Az én implementációmban, az *epsilon* paraméter egy empirikus paraméter, viszont a jövőben érdemes lehet egy olyan módszert kidolgozni, amely képes *epsilon* értékét automatikusan meghatározni.
A 8.3 táblázat összesíti azokat az eredményeket, amelyek rögzített seed értékekkel és változó *epsilon* értékekkel mértem. A táblázatban látható, hogy minél kisebb az *epsilon*, az átlagok értékei annál jobban megközelítik a kívánt értéket. Azonban ehhez sokkal több lépést kell megtenni a Markov-láncokon. Valamint vegyük észre, hogy a rögzített seed adatok miatt egy iteráció alatt az értékek nem változtak. Az eredményekből az a következtetés vonható le, hogy minél kisebb az *epsilon* értéke annál pontosabban konvergál az algoritmus.

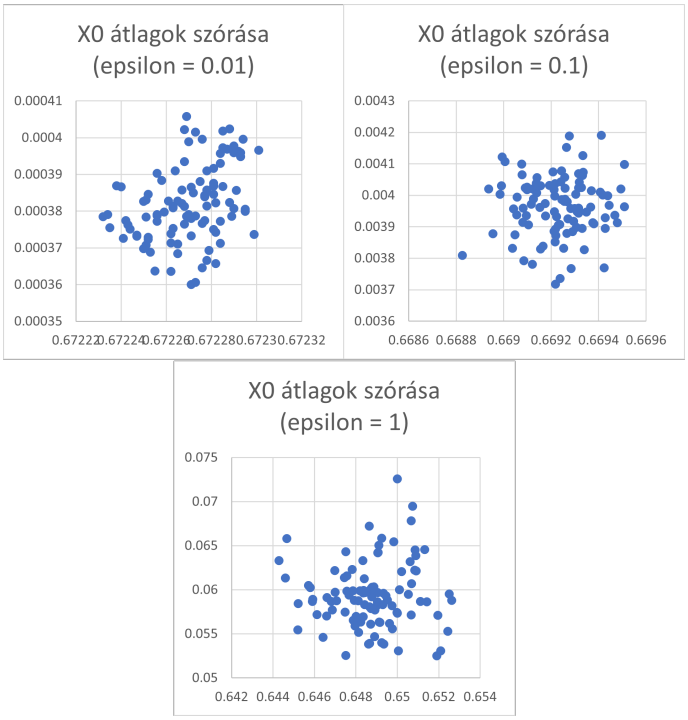
8.3. táblázat. Algoritmus statisztika fixált seedű véletlen szám generátor és különböző $epsilon$ értékekkel

		$\epsilon = 0.01$		$\epsilon = 0.1$		$\epsilon = 1.0$	
		Populációk: 20		Populációk: 9		Populációk: 2	
		Átlag	Szórás	Átlag	Szórás	Átlag	Szórás
Peak 1	x_0	0.66722	0.00030	0.66843	0.00372	0.64882	0.06011
	FWHM	0.10919	0.00103	0.11441	0.01018	0.11483	0.03744
	Intensity	2.39082	0.01829	2.35227	0.18409	2.54757	0.52555
Peak 2	x_0	0.49047	0.00184	0.50775	0.02792	0.48461	0.35310
	FWHM	0.16523	0.00399	0.21124	0.05023	0.25115	0.07943
	Intensity	0.43550	0.01217	0.46664	0.11385	0.34087	0.17382

A következő tesztesetben úgy alkalmazok véletlen szám generálást, hogy nem kerül rögzítésre a seed. Továbbá az *epsilon* értékeket hasonlóan az előző teszthez, 0.01-től 1-ig emelem. Az x_0 átlagok értékét a szórással ábrázolva 200 mintán a 8.13 ábra mutatja.
Az ábrán látszik, hogy a legkisebb szórás a legkisebb epsilon értékhez tartozik. Az eddigi tesztek úgy lettek elvégezve, hogy a szimulált mérés mindig ugyan az volt. A valósághoz közelebbi teszteredményeket kaphatóak, ha a szimulált mérési spektrum zaját minden esetben más seed értékkel generálom. Az eredmény látható a 8.14 ábrán.



8.13. ábra. X0 átlagok szórása epsilon változtatások mentén



8.14. ábra. X0 átlagok szórása epsilon változtatások mentén, rögzített seed nélkül

A *threshold* érték módosításával is hangolható az algoritmus. Ezt az értéket az effektív minta méret számításánál vesszük figyelembe. Az effektív minta méretet azonban az iteráció alatt állandó értéken kell tartani, ami az inverz hőmérséklet (β paraméter) állításával érhető el. A *threshold* paraméter értéke meghatározza, hogy milyen gyorsan változik β

értéke populációról populációra. A teszt elvégzése előtt arra számítottam, hogy a *threshold* növelésével az algoritmusnak egyre több populációra lesz szüksége hogy elérje a $\beta = 1$ értéket, és megálljon. *threshold* paramétert változtattam, amelynek az eredménye a 8.4 táblázatban látható. Jól látszik, hogy az algoritmus a várakozásoknak megfelelően működik, nagyobb effektív mintaméret (teljes mintaméret $\times threshold$) több populációt eredményez.

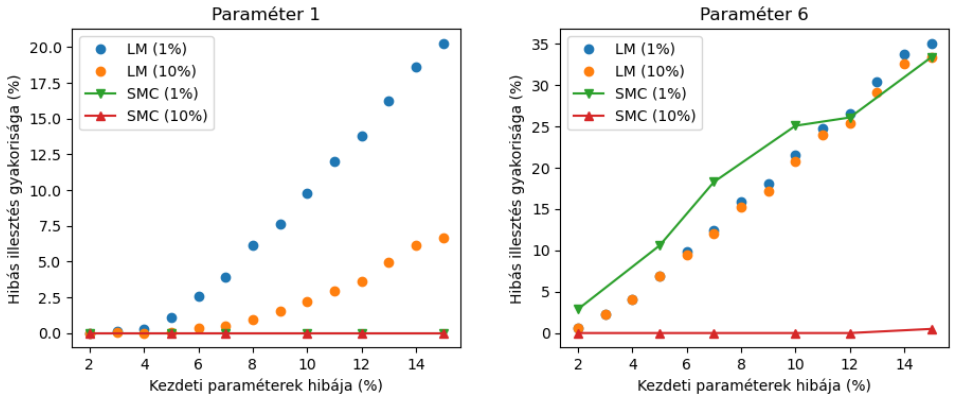
8.4. táblázat. Threshold érték algoritmusra gyakorolt hatása

Threshold	Populációk száma
0.1	9
0.2	12
0.4	18
0.5	22
0.7	33

8.2. Összehasonlítás a Levenberg-Marquardt algoritmussal

A második teszt az SMC algoritmus összehasonlítása volt az LM algoritmussal. Ehhez az előzőleg bemutatott két csúcsból felépített modellt használtam. Feltételeztem hogy a kezdeti tippünk nagyjából helyes, de a mért spektrum véletlenszerűen változik egy bizonyos mértékben. Ez egy reális szituáció valós mérések esetében, hiszen ahogy a minta tulajdonságai változnak, úgy változik a mért spektrum is. A kísérlet elvégézéséhez véletlenszerűen megváltoztattam a spektrum matematikai modelljének 6 paraméterét a kezdeti tipphez képest, úgy hogy a változás mértékét normáls eloszlással generáltam, amelynek szórása 2%, 3%, ... 15% volt. Ez a szórás megfeleltethető egy elképzelt mérési sorozat inhomogenitásának, azaz mennyire változnak a mért spektrumok mérésről mérésre. Az LM módszer esetében a véletlen generált paraméterek várható értékeiből indítottam a nemlineáris függvényillesztést (kezdeti tipp), míg az SMC módszer esetében a kiindulási populációt olyan eloszlásokkal generáltam, hogy a várható értékük megegyezzen a véletlen generált mérések paramétereinek várható értékeivel, szórásuk pedig fix 20% volt függetlenül attól, hogy mekkora volt az inhomogenitása a szimulált mérések sokaságának. Az LM illesztés esetében minden "inhomogenitás" értékhez 10000 véletlen spektrumot generáltam, és ezek illesztésének sikerességét vizsgáltam. Az SMC módszer esetében csak 1000 véletlen spektrum esetében végeztem el az illesztést a megnövekedett számí-

tási igény miatt (egy LM illesztés kb. 5 ezredmásodperc, míg az SMC illesztés kb. 5 másodpercet vesz igénybe egy modern PC-n, sorosan futtatva).
Az eredmények kiértékelése során azt tapasztaltam, hogy az első paraméter (Lorentz csúcs pozíciója) illesztése volt a legstabilabb, míg a hatodik paraméter (a Gauss csúcs intenzitása) illesztése volt a legnehezebb. Ezért a két módszer közötti különbséget ezeknek a paramétereknek a segítségével illusztrálom. Az illesztés átlagos hibája téves következtetésekhez vezethetne, mivel sok nagyon pontos illesztés egy-egy rossszul sikerült illesztéssel hasonló átlagos hibát adhat, mint ha az összes illesztés nagyjából jó, de nem teljesen precíz. Az átlagos hiba helyett ezért inkább azoknak az illesztéseknek az arányát vizsgáltam, ahol az első vagy hatodik paraméter illesztése meghaladta az 1% illetve 10%-os határértéket.



8.15. ábra. Az SMC és LM módszer összehasonlítása. Az első és hatodik paraméter illesztésének sikeressége az inhomogenitás, ha azokat az illesztéseket tekintjük sikeresnek, ahol a hiba kisebb mint 1% illetve 10%. Az inhomogenitásnak a szimulált mérések paramétereinek szórását tekintem, a kezdeti becslésünkhöz viszonyítva.

Az eredményeket a 8.15 ábrán ábrázoltam. Látható, hogy az első, robusztusabb paraméter esetében is jól elkülönül egymástól a két a módszer. Az LM módszer esetében a kb. 5%-os inhomogenitás értéknél már elkezd nőni a sikertelen illesztések aránya. Azt is megállapíthatjuk, hogy az 1% illetve 10%-os hibahatárértékhez rendelt görbék elkülönülnek egymástól, számos olyan illesztés van, ahol a paraméter illesztési hibája 1% és 10% közé esik. Ezzel szemben az SMC módszer kiválóan teljesít, minden inhomogenitás érték esetében mind az 1000 teszt illesztés jó volt.
A hatodik, nehezebben illeszthető paraméterre kapott eredmények is tanulságosak. Itt az LM illesztés esetében az 1%-os és 10%-os határértékkel definiált hibázási gyakoriság nagyon hasonló, tehát a paraméter értéke vagy pontosabb mint 1%, vagy pontatlanabb mint

10%. Az SMC módszer esetében nem ezt kaptam. Azon esetek száma, ahol az illesztés pontatlansága rosszabb mint 1% itt hasonló az LM módszerrel kapott értékekhez, viszont azon illesztések száma, ahol a paraméterre kapott becslés rosszabb mint 10% továbbra is nagyon alacsony (< 1%). Ez úgy értelmezem, hogy az algoritmus megtalálta a helyes lokális minimumot, de a sztochasztikus jellegéből adódóan a végeredmény pontatlan lett. Ennek a tesztnek a konklóziója tehát, hogy az SMC módszer sokkal robosztusabb az LM módszerhez viszonyítva, a kezdeti paraméterbecslések pontatlanságának szempontjából. Ugyanakkor az LM módszernek is vannak erényei: nagyságrendekkel kisebb számítási erőforrást igényel, és amennyiben megtalálja a globális minimumot, a végeredmény pontosabb. Ez előrevetíti, hogy a jövőben érdemes lehet kombinálni a két a módszert. Egy kezdeti SMC optimalizálás a globális minimum megtalálására, és egy gyors LM illesztés a pontos eredmény meghatározására.

9. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

Jelenleg a félvezetőipar, valamint minden olyan ágazat, ahol szükség van olyan számításokra, amelyeknek általánosságban a futasi ideje magas, lépéselőnyre tehetnek szert azok a vállalatok akik ezen az időn csökkenteni tudnak. Az SMC algoritmus rendkívül robosztusnak bizonyult a globális minimum megtalálásának szempontjából, azonban ez jelentős extra számítási kapacitást igényel. A dolgozatomban vizsgált két csúcsot tartalmazó modell illesztése kb. 2-20 másodpercet vesz igénybe (az illesztési paraméterek függvényében) párhuzamosítás nélkül. Mivel az algoritmusban indított Markov láncok egymástól függetlenek, az algoritmus triviálisan párhuzamosítható. Kifejetten nagy gyorsulás vizionálható GPU (videókártya) párhuzamosítás esetén, hiszen jól párhuzamosítható algoritmusok esetén gyakran $10 - 100\times$ gyorsulás is elérhető [29]. Eredetileg az algoritmus párhuzamosítására az OpenACC sztenderdet terveztem felhasználni, amely egy viszonylag egyszerű, direktívákon alapuló párhuzamosítási módszer, amellyel GPU-n párhuzamosítható kód is fejleszthető. Azonban fejlesztés közben jöttem rá, hogy egyelőre az OpenACC csak Linux környezetben támogatott. Az OpenACC alternatíváit az alacsonyabb szintű párhuzamosítási keretrendszerek, mint a CUDA és OpenCL jelentik. Az OpenMP API szintén egy lehetséges opció, amely az OpenACC-hez hasonlóan egyszerű kevés fejlesztői erőforrást igénybevevő párhuzamosítást tesz lehetővé. Az OpenMP is lehetővé teszi mind a CPU-n mind pedig a GPU-n történő párhuzamosítást, ugyanakkor a Windows környezetben jelenleg csak a CPU-s lehetőség adott. A fentiek tükrében a számításigényes rész (Monte-Carlo kernelek) párhuzamosítását CUDA környezetben szerettem volna elvégezni, de erre már nem jutott idő. Az algoritmus gyorsításának szempontjából a GPU párhuzamosítás jelenti a legnagyobb potenciált, de egyéb lehetőségek is megvizsgálhatók. Az algoritmus nagy része valószínűleg tökéletesen működne szimpla pontosságú lebegőpontos számokkal is (FP32), de bizonyos esetekben még az FP16-os számábrázolást is érdemes lehetne megvizsgálni. Az algoritmus optimális működésének meghatározása rengeteg tesztelést illetve benchmarkolást igényelne, amit a diplomamunkám sorám már nem volt idő elvégezni. A jövőben viszont érdemes lenne vizsgálni az algoritmus beállításainak automatikus, dinamikus változtatását (például dinamikusan változó *threshold* vagy különböző számú Markov-lánc indítása az algoritmus elején és végén). Szintén izgalmas lehet az SMC módszer kombinálása egyéb minimumkeresési módszerekkel. Az SMC algoritmus eredménye egy végső populáció, amelyet átlagolva kapjuk meg a modell paramétereinek becsült értékét, a szórás pedig az illesztés hibájáról ad információt. Ennél azonban lehetne szofisztikáltabb statisztikai elemzéseket is megvalósítani,

mint például magasabb rendű momentumok meghatározása (ferdeség és lapultság) vagy klaszter analízis elvégzésére. Ahogyan az algoritmusok futási idejének csökkentése lépéselőnyt jelent, úgy a programokhoz tartozó felhasználói felület esetleg webes elérés is komoly előnyt jelenthetne. Ezért továbbfejlesztési lehetőségként érdemes lehet megfontolni, hogy kialakítsak hozzá egy alkalmazásprogramozási felületet azaz API-t. Jelenlegi modern technológiák ezt lehetővé teszik. Gyakran alkalmazott technológia erre a Google által fejlesztett gRPC. Az RPC (Remote Procedure Call) technológia segítségével képessé tehető egy program arra, hogy különböző folyamatok között kommunikációt, adatcserét végezzen. Általában ez a technológia publish/subscribe metóduson alapul, azaz fel lehet iratkozni üzenetekre, amelyeket a feliratkozottak megkapnak ha egy objektum közzétesz a hálózaton. Ezzel a technológiával az ABC-SMC algoritmus képessé tehető arra, hogy integrálható legyen bármilyen környezetben. Például, ha szükséges lenne beilleszteni egy webes alkalmazásba, akkor ezzel könnyen megtehető. A felhasználói felület fejlesztése azért is lehetne szükségszerűen továbbfejlesztési lehetőség, mivel jelenleg az algoritmus előre elkészített fájlok beolvasásával képes egy mért spektrum elemzésére. Azonban a felhasználói felületen implementáható lenne az, hogy ezeket az adatokat dinamikusan a UI-ról beállítsuk. Valamint a bemeneti modell elkészítésére is további komoly lehetőségeket nyújtana, úgy mint a modell manipulálása a felhasználói felületen.

10. ÖSSZEGZÉS

A diplomamunkám elkészítése során a kitűzött cél a szekvenciális Monte Carlo algoritmus implementálása volt C++ nyelven. Ehhez a motivációt az adta, hogy az optikai spektroszkópián alapuló metrológiák esetében alkalmazott nemlineáris modellek illesztése hagyományos, determinisztikus módszerekkel gyakran nem kielégítő eredményt ad. Ez arra vezethető vissza, ezek a módszerek mind hajlamosak arra, hogy lokális minimumokba ragadjanak. Reményeim (illetve a szakirodalom szerint) az SMC módszer erre a problémára megoldással szolgál. Az irodalomkutatás során feltérképeztem a minimumkeresési problémák esetén alkalmazott algoritmusokat. Kezdve az legegyszerűbbektől, úgy, mint az elsőrendű gradiens módszer (Gradient Descent), a széles körben alkalmazott Levenberg-Marquardt (LM) módszerrel folytatva, amely a nemlineáris függvényillesztési problémák esetében a leggyakrabban alkalmazott módszer. Ezek az algoritmusok determinisztikus módszerek voltak. A sztochasztikus folyamatok vizsgálata során megvizsgáltam a sztochasztikus gradiens módszert, majd a heurisztikus algoritmusok vizsgálatával folytattam, mint a genetikus algoritmus és a részecske raj algoritmus. Végül megismer-

kedtem a Monte Carlo módszerekkel, illetve a Monte Carlo szimulációk esetében gyakran használt koncepciókkal. Tanulmányoztam a Markov-lánc Monte Carlo módszert, a Hamiltoni Monte Carlo módszert és végül az SMC algoritmust is. A sokféle globális optimalizációs módszer vizsgálata során meg kellett ismerkednem a valószínűségszámítás, a bayeziánus következtetés és a statisztika alapjaival.

Ezután részletesen tanulmányoztam az SMC algoritmus matematikai alapjait. Az SMC algoritmus számos, a korábban említett algoritmusokból már ismerős koncepciót használ; nagyon leegyszerűsítve a genetikus algoritmus és a Markov-lánca Monte Carlo módszer keresztezésének tekinthető. Kiderült, hogy az SMC algoritmus már implementálva van a PyMC3 python könyvtárban, amely később az általam készített C++ implementáció alapját képezte.

A program elkészítése során igyekeztem hogy a végeredmény moduláris, újra felhasználható és könnyen kibővíthető legyen. Igyekeztem a modern C++ adta lehetőségeket kiaknázni, mint például a Standard Library használata vagy a RAII megközelítés. A fejlesztés során erősen támaszkodtam az Eigen template könyvtárra, amely a numerikus műveletek kódolását nagyban megkönnyítette. Az eredmény egy jól struktúrált, fenntartható objektum-orientált szoftver lett. Jól elkülönül az optikai spektrumok modellezése és az SMC algoritmus, ez a jövőben előnyösnek bizonyulhat a szoftver bővítésének szempontjából.

Az elkészült programot teszteltem. Megbizonyosodtam, hogy jól működik, valóban képes a globális minimum megtalálására a vizsgált illesztési problémák esetében. A tesztelés során vizsgáltam, hogy az algoritmus paramétereinek változtatására hogyan változnak az eredmények. Készítettem egy összehasonlító tesztet a leggyakrabban használt nemlineáris illesztési módszerrel, a Levenberg-Marquardt (LM) algoritmussal. A teszt eredménye az lett, hogy az SMC módszer sokkal robusztusabb, és nem hajlamos lokális minimumokban megakadni az LM módszerrel szemben. Ennek az előnyös tulajdonságnak viszont meg kell fizetnünk az árát, az SMC módszer futási ideje kb. 3 nagyságrenddel nagyobb az LM módszerhez képest. Ugyan még további tesztelés szükséges, egyelőre az SMC módszer beváltotta a hozzá fűzött reményeket. Összességében a kitűzött célok nagyrészét sikerült megvalósítani. A legfontosabb fejlesztés amire már nem jutott idő a módszer párhuzamosítása volt GPU-k számára.

Hivatkozások

[1] J. Shalf, „The future of computing beyond moore's law," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, p. 20190061, Jan. 2020.

[2] P. Kopka, A. Wawrzynczak, and M. Borysiewicz, „Time dependent global optimization via bayesian inference and sequential monte carlo sampling," in *2013 Federated Conference on Computer Science and Information Systems*, pp. 363–370, 2013.

[3] D. Rudoy and P. J. Wolfe, „Monte carlo methods for multi-modal distributions," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, IEEE, Oct. 2006.

[4] *Optical Characterization of Semiconductors*. Elsevier, 1993.

[5] E. Smith and G. Dent, *Modern Raman Spectroscopy - A Practical Approach*. John Wiley & Sons, Ltd, Dec. 2004.

[6] J. Nocedal and S. J. Wright, *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering, New York: Springer, 2 ed., 2006.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipies In C: The Art of Scientific Computing*. Cambridge, England: Cambridge University Press, 1988.

[8] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods*. Springer-Verlag New York Inc., 2004.

[9] J. Nocedal and S. J. Wright, *Numerical optimization*. New York: Springer, 2006.

[10] A. Y. Zomaya, ed., *Handbook of Nature-Inspired and Innovative Computing - Integrating Classical Models with Emerging Technologies*. Springer, 2006.

[11] Wikipedia, „Gradient descent — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Gradient%20descent&oldid=1060768704`, 2021. [Online; accessed 18-December-2021].

[12] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[13] Y. Zhang, S. Wang, and G. Ji, „A comprehensive survey on particle swarm optimization algorithm and its applications," *Mathematical Problems in Engineering*, vol. 2015, pp. 1–38, 2015.

[14] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of Markov chain Monte Carlo*. CRC press, 2011.

[15] J. Lintusaari, M. U. Gutmann, R. Dutta, S. Kaski, and J. Corander, „Fundamentals and recent developments in approximate bayesian computation," 2017.

[16] A. Aaby, *Introduction to Programming Languages*. 1996.

[17] Wikipedia, „Programming language — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Programming%20language&oldid=1059923578`, 2021. [Online; accessed 18-December-2021].

[18] Wikipedia, „Strong and weak typing — Wikipedia, the free encyclopedia." `http://en.wikipedia.org/w/index.php?title=Strong%20and%20weak%20typing&oldid=1051104184`, 2021. [Online; accessed 18-December-2021].

[19] Albatross, „A Brief Description." https://www.cplusplus.com/info/description/. [Online; accessed 18-December-2021].

[20] Mirosoft, „A tour of the C# language." `https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/`, 2021. [Online; accessed 18-December-2021].

[21] kostya, „Benchmarks." `https://github.com/kostya/benchmarks`, 2021. [Online; accessed 18-December-2021].

[22] M. Fourment and M. R. Gillings, „A comparison of common programming languages used in bioinformatics," *BMC Bioinformatics*, vol. 9, Feb. 2008.

[23] G. Guennebaud, „Eigen a c++ linear algebra library." `https://downloads.tuxfamily.org/eigen/eigen_CGLibs_Giugno_Pisa_2013.pdf`, 2013. [Online; accessed 18-December-2021].

[24] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck, „Probabilistic programming in python using PyMC3," *PeerJ Computer Science*, vol. 2, p. e55, Apr. 2016.

[25] O. Martin, *Bayesian Analysis with Python*. 2018.

[26] H. Wiegand, „Kish, l.: Survey sampling. john wiley & sons, inc., new york, london 1965," *Biometrische Zeitschrift*, vol. 10, no. 1, pp. 88–89, 1968.

[27] A. Gelman, W. R. Gilks, and G. O. Roberts, „Weak convergence and optimal scaling of random walk metropolis algorithms," *The Annals of Applied Probability*, vol. 7, Feb. 1997.

[28] M. E. O'Neill, „Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation," Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, Sept. 2014.

[29] B. Suchoski, S. Stage, H. Gurung, and P. Baccam, „Gpu accelerated parallel processing for large-scale monte carlo analysis: Covid-19 parameter estimation and new case forecasting," *Frontiers in Applied Mathematics and Statistics*, vol. 8, 2022.

Ábrajegyzék

Táblázatok jegyzéke