České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra inženýrství pevných látek
Obor: Aplikace přírodních věd

Implementace algoritmu optimalizace hejnem částic do programu FOX

DIPLOMOVÁ PRÁCE

Vypracoval: Milan Kočí
Vedoucí práce: Jan Drahokoupil
Rok: 2023

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA JADERNÁ A FYZIKÁLNĚ INŽENÝRSKÁ
Katedra inženýrství pevných látek

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	**Bc. Milan Kočí**
Studijní program:	**Inženýrství pevných látek**
Akademický rok:	**2022/2023**
Název práce:	**Implementace algoritmu optimalizace hejnem částic do programu FOX**
Název práce: *(anglicky)*	**Implementation of algorithm of particle swarm optimization into programme FOX**
Jazyk práce:	**čeština**

Pokyny pro vypracování:

Cílem diplomové práce je implementovat globální optimalizační algoritmus optimalizace hejnem částic do volně přístupného programu FOX určeného na řešení krystalových struktur z práškových difrakčních dat. Funkčnost a efektivita algoritmu bude otestována na vybraných malých organických molekulách. Testovací struktury látek budou brány z krystalografických databází.

Při řešení postupujte podle následujících bodů:

I. Teoretická část

(1) Seznámení se s teoretickými základy a programovým prostředím SW FOX.
(2) Navržení postupu implemenace optimalizačního algoritmu do programu FOX.

II. Praktická část

(3) Vlastní realizace implementace algoritmu.
(4)Ověření funkčnosti a efektivity výpočty na zvoleném souboru testovacích molekul.
(5) Zpracování získaných dat a vyhodnocení dosažených výsledků a jejich diskuze.

Doporučená literatura:

[1] Favre-Nicolin V., Cerny R.: J. Appl. Cryst. **35** (2002), 734-743.
[2] Fox, Free Objects for Crystallography, http://objcryst.sourceforge.net.
[3] Clerc M.: From Theory to Practice in Particle Swarm Optimization; Handbook of Swarm Intelligence, Springer Berlin Heidelberg, 2010.
[4] Pecharsky V. K., Zavalnij P. Y.: Fundamentals of Powder Diffraction and Structual Characterization of Materials; Second Edition, Springer Science+Business Media, LLC, 2009.

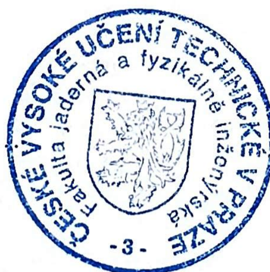
Jméno a pracoviště vedoucího práce:
Ing. Jan Drahokoupil, Ph.D., Katedra inženýrství pevných látek, Fakulta jaderná a fyzikálně inženýrská, ČVUT v Praze.

Jméno a pracoviště konzultanta:
Ing. Jan Rohlíček, Ph.D., Fyzikální ústav AV ČR v.v.i.

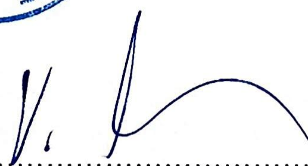
Datum zadání diplomové práce: 20. 10. 2022

Termín odevzdání diplomové práce : 3. 5. 2023

Doba platnosti zadání je dva roky od data zadání.



....
garant



....
vedoucí katedry

....
děkan

V Praze dne 20. 10. 2022

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu. Během přípravy této práce jsem využil nástroje COPILOT, ChatGPT 3.5, Google Bard, Research Rabbit a Bing AI za účelem korekce a reformulace textu, hledání autorů zabývajících se daným tématem a při programování. Po použití těchto nástrojů jsem provedl potřebné revize a úpravy obsahu a nese plnou odpovědnost za obsah této publikace.

V Praze dne ....

Milan Kočí

Poděkování

Děkuji vedoucímu práce Janu Drahokoupilovi a konzultantovi Janu Rohlíčkovi za cenné rady. Dále děkuji své rodině za podporu a trpělivost během studia.

Milan Kočí

Název práce:
Implementace algoritmu optimalizace hejnem částic do programu FOX

Vypracoval: Milan Kočí

Obor: Aplikace přírodních věd
Druh práce: Diplomová práce

Vedoucí práce: Jan Drahokoupil
FZÚ, AV ČR, v.v.i.
Konzultant: Jan Rohlíček
FZÚ, AV ČR, v.v.i.

Abstrakt: Řešení krystalových struktur látek na základě jejich práškového difrakčního záznamu je významným nástrojem v chemii, fyzice a materiálovém inženýrství. Jedním z používaných nástrojů je program FOX, do kterého byl nově implementován algoritmus optimalizace hejnem částic. Parametry algoritmu lze individuálně nastavit nebo využít přednastavených hodnot. Interakce s algoritmem může probíhat jak z grafického rozhraní, tak pomocí skriptu v programovacím jazyce Python. Dále byla optimalizace hejnem částic úspěšně aplikována na tři krystalové struktury ($PbSO_4$, paracetamol, sofosbuvir), přičemž všechny byly správně identifikovány a odpovídaly publikovaným strukturám. Porovnání s existujícím algoritmem paralelního temperování ukázalo, že PSO dosahuje lepších výsledků funkce za kratší čas, zejména pro složitější strukturu sofosbuviru. Studium parametrů PSO na struktuře paracetamolu poskytlo hlubší vhled a identifikovalo tendence, které ovlivňují konvergenci algoritmu. Nastavením těchto parametrů lze dosáhnout ještě lepších výsledků.
Klíčová slova: řešení krystalické struktury, optimalizace hejnem částic, FOX, globální optimalizace

Title:
Implementation of the particle swarm optimization algorithm into the programme FOX
Author: Milan Kočí

Abstract: Solving crystal structures of materials based on their powder diffraction pattern is an important tool in chemistry, physics and materials engineering. One of the used utilities is program FOX, into which the particle swarm optimization algorithm has been newly implemented. The parameters of the algorithm can be individually set, or pre-set values can be used. Interaction with the algorithm can take place both from the graphical interface and using a script in the Python programming language. Furthermore, the particle swarm optimization was successfully applied to three crystal structures ($PbSO_4$, paracetamol, sofosbuvir), all of which were correctly identified and corresponded to published structures. Comparison with the existing parallel tempering algorithm showed that PSO achieves better results of the function in a shorter time, especially for the more complex structure of sofosbuvir. The study of PSO parameters on the paracetamol structure provided deeper insight and identified tendencies that affect the convergence of the algorithm. Setting these parameters can achieve even better results.
Keywords: solving crystal structure, particle swarm optimization, FOX, global optimization

Obsah

Seznam obrázků ix

Seznam tabulek x

Úvod 1

1 Prášková rentgenová difrakční analýza 3
- 1.1 Difrakční záznam . . . 3
- 1.2 Srovnávání difrakčních záznamů . . . 5

2 Program Free Object for Crystallography (FOX) 7
- 2.1 Popis postupu *ab initio* řešení krystalové struktury . . . 7
- 2.2 Proces globální optimalizace ve FOXu . . . 8

3 Globální optimalizace 11
- 3.1 Představení problému . . . 11
  - 3.1.1 Metody globální optimalizace . . . 12
  - 3.1.2 No Free Lunch teorém . . . 13
- 3.2 Optimalizace hejnem částic (PSO) . . . 13
  - 3.2.1 Standardní PSO 2006 . . . 13
  - 3.2.2 Parametry algoritmu PSO . . . 15
  - 3.2.3 Výhody a nevýhody PSO . . . 18
- 3.3 Další algoritmy globální optimalizace . . . 18

4 Implementace algoritmu optimalizace hejnem částic do programu FOX 23
- 4.1 Popis kódu . . . 23
  - 4.1.1 Funkce RunParticleSwarmOptimization . . . 23
  - 4.1.2 Nastavení algoritmu PSO . . . 25
- 4.2 Interakce s algoritmem . . . 25
  - 4.2.1 Příklad použití grafického rozhraní . . . 25
  - 4.2.2 Příklad použití Python skriptu . . . 26

5 Testovací struktury 27
- 5.1 $PbSO_4$ . . . 27
- 5.2 Paracetamol . . . 31
- 5.3 Sofosbuvir . . . 34

6 Vliv změn parametrů algoritmu optimalizace hejnem částic 39
- 6.1 Konvergence algoritmu . . . 39
- 6.2 Vliv změn parametrů . . . 40

Závěr 43

Bibliografie 45

Přílohy 49

Seznam obrázků

1.1 Difrakční záznam paracetamolu z dat z [16] . . . 3

2.1 Ukázka dědičnosti v programu FOX. Tučně jsou napsané názvy tříd, v závorce vedle nich můžou být jejich parametry a pod nimi jsou ukázky některých podstatných metod. Třída `Scatterer` je virtuální. . . . 8

3.1 Ilustrace algoritmu optimalizace hejnem částic. Nová rychlost částic 1, 2 a 3 je zde rozdělena ja jednotlivé komponenty (modrá setrvačnost, zelená sociální vliv pohybu a hnědá pohyb k vlastnímu minimu). Očíslované kruhy reprezentují polohy částic, očíslované hvězdy reprezentují polohy vlastních minim a zelený pětiúhelník polohu současného globálního minima. Červená šipka označuje, kam se částice pohne v dalším kroku. . . . 14

4.1 Grafické rozhraní programu FOX s implementovaným algoritmem optimalizace hejnem částic. Uživatel si vybere algoritmus PSO a spustí optimalizaci. . . . 25

5.1 Vyřešená struktura $PbSO_4$ a její odpovídající rentgenové práškové difrakční záznamy (černě naměřený, červeně vypočítaný, zelená linka odpovídá rozdílu obou záznamů). 28

5.2 Neutronové práškové difrakční záznamy $PbSO_4$, černě naměřený, červeně napočítaný, zelená linka odpovídá rozdílu obou záznamů. . . . 29

5.3 Vývoj mediánu hodnoty aktuálního minima hodnoty LLK při řešení struktury $PbSO_4$ v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsoben první lokální optimalizací. Medián pro PSO je přibližně o 1000 vyšší než pro PT, což není příliš významný rozdíl při hodnotách LLK okolo 4000, odpovídá to přibližně desetinám procenta v R_w. . . . 30

5.4 Počet správně vyřešených struktur (LLK nižší než 5 000) při řešení struktury $PbSO_4$ v závislosti na počtu výpočtů funkce. Počet správně vyřešených struktur je u PSO nižší než u PT. . . . 30

5.5 Vyřešená struktura paracetamolu a její odpovídající naměřený a vypočtený rentgenový práškový difrakční záznam. . . . 31

5.6 Vývoj mediánu hodnoty aktuálního minima hodnoty LLK při řešení struktury paracetamolu v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsoben první lokální optimalizací. Medián pro PSO je přibližně o 1000 nižší než pro PT, což není příliš významný rozdíl při hodnotách LLK okolo 12000, odpovídá to přibližně desetinám procenta v R_w. . . . 32

5.7 Vývoj počtu správně určených struktur (LLK $<$ 15 000) při řešení struktury paracetamolu v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsobena první lokální optimalizací. PSO určilo všechny své struktury (94/100) při první lokální optimalizaci, PT při první lokální optimalizaci určilo větší procento správných struktur a navíc při druhé optimalizaci určil všechny správné struktury. . . . 33

5.8 Vývoj hodnoty optimalizované funkce částice číslo 26 v závislosti na iteraci při řešení struktury paracetamolu. Do první lokální optimalizace měla vysokou hodnotu, ale trefovala i relevantně nízké hodnoty, po lokální optimalizaci se částice pohybovala v okolí minima. . . . 34

5.9 Program FOX nalezl pro molekulu sofosbuviru v krystalové mřížce celkem 202 volných parametrů. Obrázek byl získán z [51]. . . . 35

5.10 Difrakční záznam a krystalová struktura sofosbuviru. Černá čára odpovídá naměřeným hodnotám, červená čára odpovídá vypočtenému difrakčnímu záznamu a zelená je rozdíl mezi oběma předchozími. Difrakční záznamy si odpovídají, což potvrzuje i $R_w = 7,13\%$ 36
5.11 Srovnání mediánu hodnoty aktuálního minima PSO a PT v závislosti na počtu výpočtů funkce pro strukturu sofosbuviru. Skok u 150 000 výpočtu je způsoben první lokální optimalizací. PSO nachází během prvních 150 000 výpočtů výrazně lepší hodnotu LLK než PT a cca od 50 000 výpočtu už zůstává hodnota stejná. Z této lepší polohy PSO při lokální optimalizaci přeskočí do výhodnějšího bodu než při PT a proto PSO nachází ve finále nižší hodnoty LLK než PT. Výhoda PT je, že i při dalších lokálních optimalizacích dochází k nalezení výhodnějších poloh, tj. má lepší konvergenční vlastnosti. 37
5.12 Vývoj počtu správně určených struktur (LLK<100 000) v závislosti na počtu výpočtů funkce pro strukturu sofosbuviru. Skok u 150 000 pokusů je způsoben první lokální optimalizací. PSO určilo všechny své struktury (100/100) při první lokální optimalizaci. PT jich při první lokální optimalizaci určilo jen část (22/100), zato další lokální optimalizace nacházely další správná řešení. Celkově PT nalezlo správnou strukturu v 90 pokusech ze 100. 37
5.13 Vývoj hodnoty funkce LLK pro náhodně zvolenou částici 26 pro dva různé běhy pro strukturu sofosbuviru. Pro běh 19 částice kolem 150 000 výpočtu funkce sklouzla do lokálního minima a setrvala zde až do konce jen s malými výjimkami kolem dalších lokálních optimalizací. Pro běh 21 částice do minima nesklouzla a prohledávala po celou dobu globálně. 38

6.1 Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro 50, 100 a 200 částic. 40
6.2 Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro parametr setrvačnosti 0,6; 0,721 a 0,8 41
6.3 Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro sociální parametr 0,8; 1,193 a 1,5. 42
6.4 Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro 3, 10 a 50 sousedů. 42

Seznam tabulek

5.1 Parametry algoritmu PSO použité pro řešení struktur $PbSO_4$, paracetamolu a sofosbuviru. 27
5.2 Srovnání algoritmů optimalizace hejnem částic a paralelního temperování pro řešení krystalové struktury $PbSO_4$. PT vyřešil strukturu pokaždé úspěšně, zatímco PSO dvacetkrát strukturu vůbec nevyřešil. 29
5.3 Srovnání výsledků pro aloritmy PSO a PT při řešení struktury paracetamolu. PT vyřešil strukturu vždy, zatímco PSO v 6 případech nevyřešil strukturu. PSO dosahovalo nižších hodnot LLK, ale výsledky byly méně konzistentní než u PT. 33
5.4 Srovnání výsledků pro algoritmy PSO a PT. Pro sofosbuvir dosáhl algoritmus PSO nižší nejlepší hodnoty za rychlejší čas a pro všech sto běhů. 36

Úvod

Úloha řešení krystalové struktury z práškových difrakčních dat je důležitý úkol, obzvláště pokud z daného materiálu nelze vyrobit vhodný monokrystal a použít metody monokrystalické difrakce nebo pokud je v centru zájmu právě daná polykrystalická vlastnost materiálu [1]. Obvykle lze metody na řešení krystalové struktury z prášku rozdělit na dva základní přístupy – v reciprokém prostoru a v přímém prostoru (existují i další, které se mohou vyskytovat na pomezí obou přístupů tzv. *dual-space* metody, lze přeložit jako metody duálních prostorů, patří mezi ně například metoda *charge-flipping*, lze přeložit jako metoda otáčení nábojů). Mezi metody reciprokého přístupu patří například Patersonova metoda, nevýhodou reciprokých a dual-space metod je, že pro řešení tímto způsobem je zapotřebí preciznějі naměřené intenzity difrakčního záznamu [2]. Tento problém metody přímého prostoru elegantně obcházejí. Nicméně i tyto metody mají svá omezení, a tím je složitost krystalové struktury.

Při metodách řešení krystalové struktury v přímém prostoru se porovnává naměřený difrakční záznam se záznamem vypočítaným z počítačového modelu zkoumané krystalové struktury. Do modelu jsou zahrnuty chemické informace o dané látce, jako je chemické složení, vazby mezi jednotlivými atomy v molekule apod. Z těchto informací jsou stanoveny objekty, které budou rozptylovat ionizující záření, jako jsou atomy, molekuly a mnohostěny. Každý z těchto objektů má své stupně volnosti, jako jsou polohy v mřížce, vzdálenosti, úhly a torze mezi atomy, natočení molekuly apod.

Dalším klíčovým úkolem je porovnání, zda a do jaké míry jsou naměřené a vypočtené difrakční záznamy podobné. Existuje několik kriterií, kterými lze tuto podobnost hodnotit, například R, R_w, R_B, χ^2 faktory. Tyto faktory srovnávají vzdálenosti naměřených hodnot od napočtených hodnot simulovaného záznamu. Tyto faktory lze následně využít jako objektivní funkce, jejichž extrémní hodnota v závislosti na stupních volnosti je hledána [3]. Zadefinování úlohy již umožňuje hledat její řešení.

Obvykle není ve výpočetních kapacitách prohledat celý konfigurační prostor, proto programy na řešení krystalové struktury z práškového difrakčního záznamu často používají algoritmy globální optimalizace, aby urychlili a často i umožnili řešení této úlohy. Mezi oblíbené algoritmy globální optimalizace patří simulované žíhání [4, 5, 6], paralelní temperování [5], genetický algoritmus [7], optimalizace hejnem částic [8, 2], algoritmus *big bang – big crunch* (lze přeložit *velký třesk-velký křach*) [9] a další, často vlastní – navržené výzkumným týmem, algoritmy například [1].

V rámci řešení krystalové struktury z práškových difrakčních záznamů existuje řada programů, které využívají různé algoritmy globální optimalizace. Využití simulovaného žíhání při řešení krystalové struktury v programu DASH [10, 4] spočívá v jednoduchosti užití tohoto algoritmu, kdy je možné spustit mnoho běhů z různých počátečních poloh, které rychle naleznou minimální hodnotu objektivní funkce a z nich zvolit nejvýhodněji ohodnocenou strukturu. Program FOX [5, 11, 12] má implementovány jak algoritmus simulovaného žíhání, tak paralelního temperování. Je ovšem doporučeno užití paralelního temperování, protože se tento algoritmus nezasekává v lokálních minimech tak často jako simulované žíhání. Více algoritmů globální optimalizace nabízí také program EXPO [13, 6, 9, 3], který byl původně v roce 1999 navržen tak, že řešil struktury v reciprokém prostoru, ale postupem času začal používat také metody přímého prostoru. Nejprve byl do programu zařazen algoritmus simulovaného žíhání a později algoritmus big bang – big crunch zkombinovaný se simulovaným žíháním. Kombinace těchto dvou algoritmů vedla k redukci času nutného k řešení krystalové struktury. Genetický algoritmus je využíván programem OCEANA [7], který řeší krystalové struktury bez *a priori* znalosti parametrů krystalové mřížky. Dalším programem, který zvládá řešit krystalové struktury bez znalosti parametrů krystalové mřížky, je FIDEL-GO [1]. Tento program rozšiřuje původní program FIDEL [14] o vlastní metodu globální optimalizace. V této metodě provádí velké množství lokálních optimalizací z různých úvodních struktur, lokální optimalizace je provedena pokaždé, když je splněna jedna z očekávaných podmínek. Algoritmus může připomínat evoluční algoritmus v tom, že se zde vyvíjí náhodné struktury a ty, které nesplňují

očekávané podmínky, zanikají. Pro tuto práci jsou obzvláště významné programy, které používají algoritmus optimalizace hejnem částic. Mezi ně patří programy PeckCryst [2] a GALLOP [8]. Program PeckCryst, který je rozšířením programu GEST [15], optimalizuje funkci Braggova R faktoru R_{Bg}, protože použití této funke šetří výpočetní čas a důležité strukturní parametry jsou v něm zahrnuty, je v něm použit PSO s časově proměnnou závislostí parametru setrvačnosti. Program GALLOP používá kombinaci lokální optimalizace a PSO. Na začátku optimalizace provede lokální optimalizaci setu úvodních struktur a ty následně používá jako výchozí struktury pro algoritmus PSO, k optimalizaci používá χ^2 funkci.

Tato práce se zabývá právě použitím algoritmu optimalizace hejnem částic v programu FOX. V teoretické části jsou popsány základy práškové rentgenové difrakční analýzy, program FOX a metody globální optimalizace. Praktická část se zabývá popisem a úskalími implementace algoritmu PSO do programu FOX a testováním tohoto algoritmu na vybraných strukturách $PbSO_4$, Paracetamol a Sofosbuvir. Poslední kapitola popisuje náš postup při zkoumání některých parametrů algoritmu PSO.

Kapitola 1

Prášková rentgenová difrakční analýza

Když rentgenové záření dopadne na libovolnou látku, mohou probíhat tři základní procesy – průchod záření látkou, absorpce záření látkou a difrakce záření. Obvykle dochází částečně ke všem třem procesům. Prášková rentgenová difrakční analýza se zaměřuje na difraktované záření vzniklé z polykrystalické látky.

Pomocí práškové rentgenové difrakční analýzy obvykle identifikujeme fázové složení v materiálu, ale můžeme též popisovat krystalické struktury neznámých látek, případně určovat další vlastnosti, jako jsou např. velikost částic, pnutí, apod.. Rentgenový paprsek ozáří studovaný materiál a difraktovaný paprsek vytvoří na detektoru tzv. difrakční obrazec. Pro polykrystalické látky má difrakční obrazec tvar tzv. Debyeových kroužků, což jsou soustředné kruhy s různou intenzitou dopadajícího záření. V ideálním případě jsou Debyeovy kroužky homogenní (celý kroužek má stejnou intenzitu dopadajícího záření), a proto není nutné zaznamenávat celý difrakční obrazec. Stačí pouze řez difrakčním obrazcem, nazývaný práškový rentgenový difrakční záznam (dále pouze difrakční záznam).

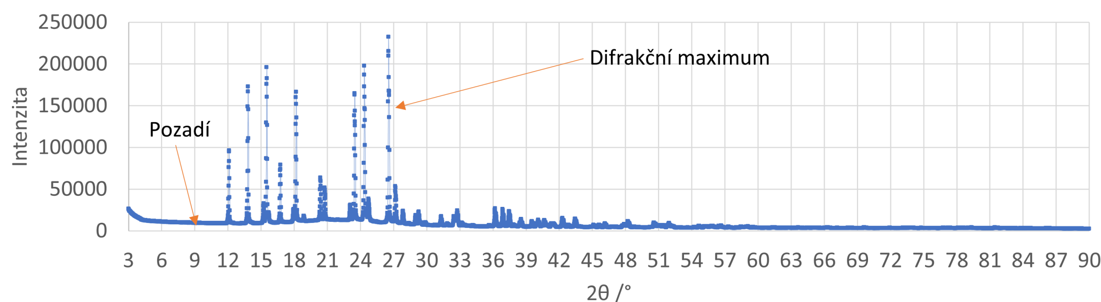
V praxi z experimentu získáváme data ve formě dvou řad čísel – nezávislá řada určuje úhel 2θ, což je úhel svíraný dopadajícím a reflektovaným paprskem, a závislá řada určuje intenzitu záření I dopadajícího na detektor. Pokud si data vyneseme do grafu (viz 1.1), získáváme typický graf známý jako práškový difrakční záznam.

1.1 Difrakční záznam

Difrakční záznam závisí na úhlu 2θ. Je charakteristický tím, že z pozadí vystupují difrakční maxima, která odpovídají difrakci na určitých krystalových rovinách krystalů prášku, jak ilustruje obrázek 1.1.

s difrakčním záznamem je možné provést tzv. indexaci (nalezení základní buňky). Každému difrakčnímu maximu jsou přiřazeny Millerovy indexy hkl, které identifikují krystalovou rovinu, ze které je záření difraktováno.

Pro každé difrakční maximum, popsané indexy hkl a úhlem 2θ, lze získat jeho integrální



Obrázek 1.1: Difrakční záznam paracetamolu z dat z [16]

intenzitu jeho intenzitu. Intenzita závisí na struktuře látky a na dalších parametrech, které souvisí se vzorkem nebo s nastavením měřícího přístroje. Závislost intenzity na úhlu 2θ lze vypočítat pomocí vztahu

$$I_{hkl}(\theta) = K \cdot p_{hkl} \cdot L(\theta) \cdot P(\theta) \cdot A(\theta) \cdot T_{hkl} \cdot E_{hkl} \cdot |F_{hkl}|^2, \tag{1.1}$$

kde K je škálovací faktor, který ve vzorci je za účelem normalizovat napočtená data vůči naměřeným; p_{hkl} je multiplicitní faktor, který určuje, kolik je symetricky ekvivalentních difrakcí; $L(\theta)$ je Lorentzův faktor, závisející na geometrii experimentu; $p(\theta)$ je polarizační faktor, který započítává částečné zpolarizování záření v průběhu difrakce; $A(\theta)$ je absorpční faktor započítávající, kolik záření bylo pohlceno při průchodu záření hmotným prostředím; T_{hkl} je texturní faktor, ve kterém je započítávána preferovaná orientace krystalků prášku; E_{hkl} je extinkční faktor, započítávající odchylku od kinematického modelu difrakce; F_{hkl} je strukturní faktor, který je určen rozložením atomů v krystalové struktuře a jejich tepelným pohybem, hkl označuje Millerovy indexy [17].

Zabývat se dopodrobna všemi parametry, které ovlivňují výslednou intenzitu, není cílem této práce. Pokud má čtenář zájem o podrobnosti o jednotlivých faktorů, doporučujeme knihu Fundamentals of Powder Diffraction and Structural Characterization of Materials [17].

Strukturní faktor F_{hkl}

Strukturní faktor ve vztahu (1.1) je jediný člen, který závisí na pozicích a druhu atomů. Zbytek členů závisí na makroskopických vlastnostech látek nebo na symetriích krystalové mřížky. Protože chceme přesně určit polohu atomů v základní buňce krystalu, je nezbytné důkladněji zkoumat strukturní faktor.

Z principů difrakce elektromagnetických vln lze pro strukturní faktor odvodit vztah

$$F_{hkl}(\vec{S}) = \sum_{j=1}^{n} g_j t_j(S) f_j(S) \exp(2\pi i(hx_j + ky_j + lz_j)), \tag{1.2}$$

kde hkl jsou Millerovy indexy; $\vec{S} = \frac{\Delta k}{2\pi}$ je vektor rozptylu; g_j je populační faktor atomu j; t_j je teplotní faktor atomu j, f_j je atomový rozptylový faktor atomu j a x_j, y_j, z_j jsou frakční polohy atomu j vůči základní buňce.

Fázový problém

Výpočet difrakčního záznamu modelu krystalu představuje relativně jednoduchý postup. Je třeba dosadit do vztahů (1.1) a (1.2) polohy atomů, informace o symetriích, konstanty vázající se k jednotlivým prvkům a experimentální detaily. Naopak, získání krystalické struktury z difrakčního záznamu je náročný úkol. V experimentu se měří pouze velikost strukturního faktoru, ne jeho fázový posun, což komplikuje zpětné stanovení struktury látky. Tato komplikace je známá jako „fázový problém“.

Existuje více různých metod, které problém fázového problému buď řeší nebo obcházejí a lze je rozdělit do třech skupin:

1. **Metody přímého prostoru**, při kterých je nejprve vytvořen model krystalu (rozestavění atomů, konformace a rozložení v buňce molekuly) a následně je optimalizována shoda napočítaného a naměřeného difrakčního záznamu.

2. **Reciproké metody**, ve kterých se hledají fáze jednotlivých strukturních faktorů, ze kterých se pak určí krystalová struktura.

3. **Dual-space metody**, při kterých se hledá rozložení elektronové hustoty v krystalu, která nejlépe odpovídá naměřenému difrakčnímu záznamu. Z elektronové hustoty se následně vypočítávají strukturní faktory a ty se porovnávají s naměřenými.

Rozšiřování difrakčních linií

Tvar difrakčního maxima lze nejlépe popsat tzv. funkcí tvaru difrakčního maxima (z angl. peak-shape function). Tato funkce závisí na třech hlavních vlivech: na geometrickém nastavení experimentu, na použitém záření a na vlastnostech vzorku [17].

Z vlastností vzorku nejvíce funkci tvaru difrakčního maxima ovlivňuje velikost krystalitů a vliv mikrodeformací, způsobují rozšiřování nebo zužování šířky funkce tvaru difrakčního maxima. Označme si šířku difrakčního maxima písmenem β, vliv velikosti krystalitů (τ) lze popsat

$$\beta = \frac{\lambda}{\tau \cdot \cos\theta}.$$

Vliv mikrodeformací (ε) lze popsat

$$\beta = k \cdot \varepsilon \tan\theta.$$

λ je vlnová délka dopadajícího záření, θ je úhel dopadu paprsku na vzorek a k je konstanta. Zmenšením velikosti krystalitů dochází k rozšiřování difrakčních linií a zvětšením vlivu mikrodeformací dojde opět k rozšíření difrakčních maxim.

Vliv textury na difrakční záznam

Textura je preferovaná orientace krystalitů ve vzorku. Může být způsobena různými faktory, například způsobem přípravy vzorku nebo způsobem jeho zpracování. Textura se projevuje v difrakčním záznamu jako posun difrakčních maxim. V případě, že je textura výrazná, může být obtížné určit strukturu látky.

1.2 Srovnávání difrakčních záznamů

Model struktury bude odpovídat reálné struktuře právě tehdy, když se budou shodovat jejich difrakční záznamy. Abychom mohli určit, zda se difrakční záznamy shodují, musíme je umět vzájemně porovnávat. Pro porovnávání difrakčních záznamů se používají různé míry shody. Klasickými mírami shody jsou:

- Profilový reziduální faktor R_p

$$R_p = \frac{\sum_{i=1}^{n} |Y_i^{obs} - Y_i^{calc}|}{\sum_{i=1}^{n} Y_i^{obs}} \cdot 100\%$$

- Vážený profilový reziduální faktor R_{wp}

$$R_{wp} = \sqrt{\frac{\sum_{i=1}^{n} w_i \left(Y_i^{obs} - Y_i^{calc}\right)^2}{\sum_{i=1}^{n} w_i \left(Y_i^{obs}\right)^2}} \cdot 100\%$$

- Braggův reziduální faktor R_B

$$R_B = \frac{\sum_{j=1}^{m} |I_j^{obs} - I_j^{calc}|}{\sum_{j=1}^{m} I_j^{obs}} \cdot 100\%$$

- Shoda modelu s pozorováním (obvykle se používá termín „chí kvadrát“) χ^2

$$\chi^2 = \frac{\sum_{i=1}^{n} w_i \left(Y_i^{obs} - Y_i^{calc}\right)^2}{n - p}$$

Proměnné ve vztazích značí: n je počet všech naměřených bodů, Y_i^{obs} je pozorovaná intenzita záření i-tého bodu difrakčního záznamu, Y_i^{calc} je vypočítaná intenzita i-tého bodu difrakčního záznamu, w_i je parametr udávající váhu i-tého bodu difrakčního záznamu, m je počet pozorovaných nezávislých difrakčních maxim, I_j^{obs} je pozorovaná integrální intenzita j-tého difrakčního maxima, I_j^{calc} je vypočítaná integrální intenzita j-tého difrakčního maxima a p je počet volných parametrů pro metodu nejmenších čtverců.

Intenzity Y_i a I_j se mezi sebou liší také v tom, že Y_i obsahuje příspěvek pozadí difrakčního záznamu, zatímco I_j jej neobsahuje. V mírách shody R_p a R_{wp} vystupuje pozadí difrakčního záznamu a v Braggově reziduálním faktoru R_B pozadí nevystupuje.

Míry shody se používají na ověření správnosti vyřešené struktury. Čím nižší jsou hodnoty měr shody, tím více jsou si podobny difrakční záznamy z experimentu a z modelu. Například při Rietveldově metodě [18] se struktura řeší minimalizací R faktorů změnami strukturních parametrů modelu krystalické látky.

Kapitola 2

Program Free Object for Crystallography (FOX)

Cílem programu FOX je *ab initio* vyřešit krystalové struktury z difrakčního záznamu (obvykle jde o data získaná z práškové difrakce). Krystalovou strukturu lze ve FOXu popsat pomocí poloh jednotlivých atomů, pomocí poloh a orientací molekul (popsaných délkami vazeb, úhly mezi třemi a čtyřmi atomy) nebo pomocí mnohostěnů anorganických prvků. Program dokáže automaticky zjistit, jestli se atom nachází na speciální pozici v krystalu a zda není sdílen mezi dvěma mnohostěny. Při hledání struktury lze využít zároveň více difrakčních záznamů (rentgenový, neutronový či elektronový). Klíčovým prvkem programu jsou algoritmy globální optimalizace, které zlepšují efektivitu řešení krystalové struktury a umožňují dostat se z nalezených lokálních minim [19].

Program řeší krystalové struktury z difrakčních záznamů pomocí metod přímého prostoru, tj. nejprve vytvoří model a následně porovnává vypočítané a naměřené difrakční záznamy.

Program FOX je implementován jako objektově orientovaný program, napsaný převážně v programovacích jazycích C++ a Python. Jednotlivé entity v programu, jako jsou atomy, difrakční záznamy atd, jsou reprezentovány třídami v jazyce C++, které jsou strukturovány do hierarchie. Příkladem této hierarchie může být situace, kde atom je potomkem rozptylového centra, a toto centrum je dále potomkem rafinovatelného objektu. Tato hierarchická struktura umožňuje jednotné zacházení se stejnými metodami pro podobné objekty a usnadňuje orientaci v kódu. Pro lepší vizualizaci dědičnosti v programu FOX lze nahlédnout do diagramu zobrazeného na obrázku 2.1.

Objektově naprogramované třídy z C++ jsou zabaleny do pythonovských objektů, pomocí nichž lze již psát skripty nutné k řešení krystalových struktur.

Kód obsahující jednotlivé objekty je dostupný z `https://github.com/diffpy/libobjcryst` a po zkompilování vytváří samostatně stojící sdílenou knihovnu ObjCryst.so, jejíž metody lze následně číst pomocí grafického prostředí nebo pomocí pythonovských zabalených objektů, které jsou dostupné z `https://github.com/diffpy/pyobjcryst`.

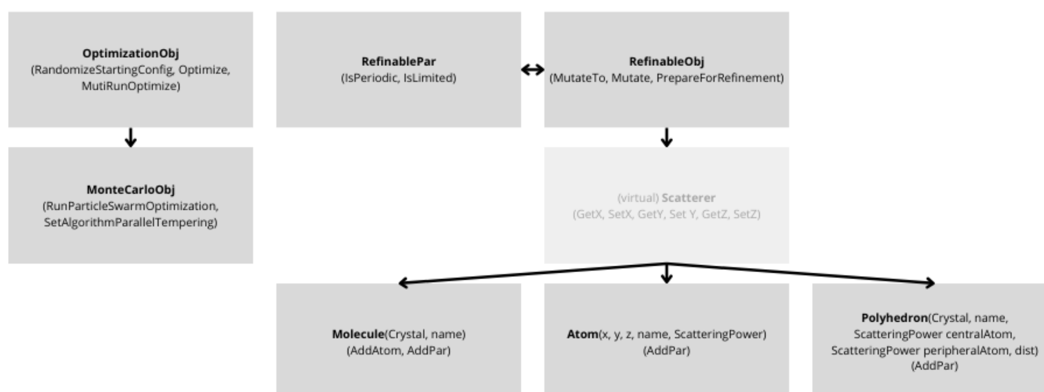
2.1 Popis postupu *ab initio* řešení krystalové struktury

Vstupními údaji programu jsou informace o objektech, se kterými se v programu bude pracovat, tj. krystal, difrakční záznam a Monte Carlo objekt (s významem metody globální optimalizace). Následně je potřeba krystal a difrakční záznam předzpracovat před tím, než bude spuštěna globální optimalizace.

Příprava difrakčního záznamu

Do objektu difrakčního záznamu je potřeba načíst experimentální difrakční data. K difrakčním záznamům je nutné doplnit informace o jejich experimentálních detailech – vlnová délka záření, polarizace záření apod. Je možné načíst více difrakčních záznamů náležících jednomu nebo více různým vzorkům.

Zpracování difrakčního záznamu dále spočívá v detekci difrakčních maxim, jejich indexaci, le Bailova fitu, určení rozměrů základní buňky a určení prostorové grupy náležící difrakčnímu



Obrázek 2.1: Ukázka dědičnosti v programu FOX. Tučně jsou napsané názvy tříd, v závorce vedle nich můžou být jejich parametry a pod nimi jsou ukázky některých podstatných metod. Třída `Scatterer` je virtuální.

záznamu z informací o extinkci některých difrakčních maxim. Získané informace lze snadno přenést na objekt krystalu.

Než je možné pokračovat s přípravou difrakčního záznamu, je nutné předem připravit objekt krystalu, jak je popsáno níže. Připravený krystal je poté spojen s difrakčním záznamem a s jeho pozadím (standardně je nastaven polynom 20. stupně). Následně jsou naměřená data porovnána s vypočítaným difrakčním záznamem a jeho pozadím. Tímto způsobem získáme připravený difrakční záznam, který je připraven pro další globální optimalizaci.

Příprava krystalu

Objekt krystalu se sestává z informací o jeho mřížce a rozložení rozptylových center, tj. atomů, molekul a mnohostěnů uvnitř něj, (všechny tyto entity jsou v programu FOX považovány za jednotlivé objekty a podle toho je s nimi i zacházeno při uživatelské interakci).

Informace o krystalové mřížce byly získány předzpracováním difrakčního záznamu a lze je snadno přiřadit krystalové mřížce. Pokud má uživatel přesnější informace o krystalové mřížce může je k objektu ještě později doplnit a pozměnit. Vyžadovanou informací krystalu jsou jeho rozptylová centra. Lze je tam importovat pomocí různých formátů souborů nebo manuálně přidávat.

2.2 Proces globální optimalizace ve FOXu

Základním úkolem globální optimalizace v programu je najít co nejnižší hodnotu objektivní funkce – funkce LogLikelihood (LLK). Tato nezáporná objektivní funkce popisuje, jak jsou si pododobné difrakční záznamy naměřené a vypočítané. Více o objektivní funkci v programu FOX je v následující podsekci.

1. Vytvoří se náhodná konfigurace tím, že se náhodně změní volné parametry.

2. Je vypočtena hodnota objektivní funkce, která je porovnána se současnou hodnotou této funkce.

3. Algoritmus globální optimalizace určí, jak se struktura bude dále vyvíjet a opakuje se krok dva.

Proces hledání započíná v přiřazením nejméně jednoho krystalu a difrakčního záznamu danému objektu globální optimalizace. Na uživateli je volba parametrů globální optimalizace – může si zvolit algoritmus, používání či nepoužívání metody nejmenších čtverců během a po proběhlém běhu, či parametry týkající se jednotlivých algoritmů.

Při spuštění hledání krystalové struktury nejprve dochází k definování stupňů volnosti struktury. Toto je plně pod kontrolou programu, jediné jak může uživatel ovlivnit stupně volnosti, které se budou optimalizovat, je při definování rozptylových center. Jednotlivé stupně volnosti jsou buď neomezené, omezené nebo periodické, podle toho je také definován prohledávaný prostor, jako

kartézský součin neomezené oblasti, omezené oblasti nebo periody. Hodnoty parametrů, odpovídajících jednotlivým stupňům volnosti, jsou náhodně pozměněny v rámci limitů nastavených pro jednotlivé typy. Toto je způsobeno přednastavenými omezeními, které cílí na větší přiblížení hodnot parametrů reálným hodnotám, jako je například skutečnost, že délky vazeb mezi atomy by měly být příliš velké, ani příliš krátké.

Průběh běhu algoritmu se liší v závislosti na volbě uživatele. Doposud si mohl uživatel zvolit mezi dvěma algoritmy globální optimalizace – simulovaným žíháním a paralelním temperováním – a s přidáním algoritmu optimalizace hejnem částic si může zvolit ze tří. Při běhu jsou aktualizovány vyobrazené modely krystalu a hodnoty objektivní funkce, aby mohl uživatel sledovat vývoj hledané struktury krystalu.

LogLikelihood – objektivní funkce

Mnoho programů používá pouze optimalizaci jednoho z faktorů míry shody, například [4, 3]. Program FOX umožňuje navíc započítávání chemické informace dodané uživatelem. Jako příklad lze uvést: pokud je v krystalu molekula, tak program předpokládá, že její vnitřní struktura je již částečně určena správně a velkou změnu vnitřních parametrů povoluje pouze s pravděpodobností $\exp(\frac{-\chi^2_{restrain}}{T})$, kde parametr T se mění dynamicky a nastavuje pravděpodobnost přijetí struktury přibližně na 70 % [19].

Objektivní funkcí v programu FOX je funkce nazývaná autory „LogLikelihood“ [11]. Tato funkce se skládá z faktoru χ^2 doplněným o speciální členy, které z tomuto faktoru zabraňují divergovat a umožňují započítávat kromě chyby měření i „informační“ chyby způsobené výpočetní nepřesností. Objektivní funkce tedy je počítaná podle vztahu

$$-\log(\text{Likelihood}) = \chi^2_{likelihood} = \sum_i \left[\frac{1}{2} \log \left(2\pi ({\sigma_i^{obs}}^2 + s^2 {\sigma_i^{calc}}^2) \right) + \frac{(Y_i^{obs} - s\langle Y \rangle_i^{calc})^2}{{\sigma_i^{obs}}^2 + s^2 {\sigma_i^{calc}}^2} \right], \quad (2.1)$$

kde i prochází všechny oblasti, ve kterých se integruje difrakční záznam, σ_i^{obs} a σ_i^{calc} jsou pozorované a vypočítané nejistoty, Y_i^{obs} je pozorovaná integrální intenzita, $\langle Y \rangle_i$ je nejpravděpodobněji vypočítaná integrální intenzita a s je škálovací faktor, který v tomto případě nelze zkrátit. První výraz odpovídá normalizaci pravděpodobnosti, která bývá obvykle zanedbávána, ale zde její význam zaručuje, že nedojde k nalezení optimálního řešení pro σ v nekonečnu.

Kapitola 3

Globální optimalizace

Globální optimalizace je důležitou úlohou v široké oblasti života. Mnoho běžných činností lze snadno převést na úlohu hledání optimálního řešení, ať už vezmeme hledání nejkratší cesty z místa na místo, snahu oslovit, co největší publikum lidí nebo najít strukturu s nejnižší energií. Ve vztahu k diplomové práci se úloha váže tím, že při řešení krystalové struktury z práškového rentgenového difrakčního záznamu je potřeba co nejpřesněji přizpůsobit simulovaný difrakční záznam naměřeným hodnotám.

Úloha globální optimalizace, jejíž matematický popis a detailní diskuse je k nalezení v sekci 3.1, představuje klíčovou součást našeho studia. Po definování této úlohy se tato kapitola věnuje efektivním způsobům jejího řešení a představuje dostupné metody. Zvláštní pozornost je věnována algoritmu optimalizace hejnem částic, kde jsou analyzovány jednotlivé parametry tohoto algoritmu a zdůrazněny jeho výhody a nevýhody. Poslední část kapitoly přináší stručný přehled dalších algoritmů, které lze využít k řešení úloh globální optimalizace.

3.1 Představení problému

Při globální optimalizaci dochází k hledání optimální hodnoty jedné funkce, viz matematický vztah (3.1), nebo více funkcí, viz podsekce 3.1. Obvykle dochází k minimalizaci hodnoty funkce, protože maximalizaci lze zadefinovat jako minimalizaci též funkce jen s opačným znaménkem. Optimalizovanou funkci zde budeme nazývat funkcí objektivní.

Nechť je $f : P \to \mathbb{R}$ funkce, kde $P \subset \mathbb{R}^n$, $n \in \mathbb{N}$ a navíc $P \subset D_f$, pak pozicí globálního minima funkce f nazýváme bod $\vec{x}^*$, pro který platí

$$\vec{x}^* = \arg\min_{\vec{x} \in P} f(\vec{x}). \tag{3.1}$$

Hodnotu funkce v bodě $\vec{x}^*$ nazýváme globálním minimem funkce f.

Nalezení optimálního řešení může být náročné, protože objektivní funkce může být složitá, tj. obsahovat velké množství jiných lokálně optimálních hodnot, které ovšem nebudou optimální globálně; může být nespojitá, či její definiční obor může mít neobvyklou topologii a další. Při hledání globálního minima úlohy řešení rentgenového práškového difrakčního záznamu budeme předpokládat, že definiční obor úlohy je spojitý a má obvyklou topologii.

Vícekriteriální optimalizace

V oblasti vícekriteriální optimalizace se zabýváme optimalizací více než jedné funkce. Pro některé z těchto funkcí může být cílem nalézt minimální hodnotu, zatímco pro jiné hodnotu maximální. Příkladem může být úloha v oblasti dopravy, kde se snažíme maximalizovat dojezd automobilu, zároveň však minimalizujeme spotřebu paliva a emise; jiným příkladem může být optimalizace energie krystalové struktury a faktoru shody jejího difrakčního záznamu s naměřenými hodnotami, kdy minimum jedné funkce nemusí odpovídat minimu funkce druhé.

Hledání minima všech funkcí, což je naše úloha, lze matematicky popsat následujícím způsobem. Pokud by bylo potřeba některou z funkcí maximalizovat, lze úlohu snadno transformovat na úlohu hledání minim všech funkcí tím, že je minimalizována negace této funkce.

Nechť jsou $f_i : P \to \mathbb{R}$ funkce, kde $i \in \{1, \ldots k\}$, $k \in \mathbb{N}$, $P \subset \mathbb{R}^n$, $n \in \mathbb{N}$ a navíc $P \subset D_{f_i}$, pak úlohou globální optimalizace je

$$\min_{\vec{x} \in P} f_i(\vec{x}) \qquad \forall i \in \{1, \ldots, k\}. \tag{3.2}$$

Řešením úlohy nazýváme vektor $\vec{x}^*$, pro který (3.2) platí.

Důležitou otázkou je, jak definovat skutečně nalezené řešení úlohy, protože optimum jedné funkce nemusí být optimem funkce jiné. Existuje více metod, například často používanou metodou je převedení úlohy na úlohu jednoho kritéria (3.1), která lze řešit například jednou z metod ze sekce 3.1.1. Převedení na úlohu optimalizace jednoho kritéria lze například

$$\vec{x}^* = \arg\min_{\vec{x} \in P} \sum_{i=1}^{k} w_i f_i(\vec{x}),$$

kde w_i jsou váhové faktory jednotlivých funkcí. Je možné sloučit více funkcí do jedné pomocí složitějších nelineárních metod. To může být někdy výhodné, protože mezi optimalizovanými funkcemi může existovat řádový rozdíl. Lineární kombinace nemusí vždy umožnit, aby se jedna funkce dostatečně projevila v konečném výsledku.

Další možností převedení je optimalizace pouze jedné funkce s podmínkami, že ostatní funkce nesmí překročit stanovené kritérium $\varepsilon_i \in \mathbb{R}^+$, tj.

$$\begin{aligned} &\vec{x}^* = \arg\min_{\vec{x} \in P} f_j(\vec{x}) \\ &f_i(\vec{x}^*) < \varepsilon_i \qquad \forall i \in \{1, \ldots k\} \setminus \{j\}. \end{aligned}$$

3.1.1 Metody globální optimalizace

Deterministické

Deterministické algoritmy lze definovat jako algoritmy, které na stejné vstupní informace vždy vydá stejné výstupní informace. Každý krok algoritmu je jasně definován a nedochází zde k neočekávaným změnám.

Výsledky těchto algoritmů jsou od samého počátku předvídatelné a lze je předem očekávat. Tato vlastnost poskytuje výhodu při hledání globálních optimálních řešení, jelikož je zajištěno dosažení globálního minima, což činí tyto algoritmy robustními. Jedním z nevýhodných aspektů těchto algoritmů je však jejich častá vysoká výpočetní náročnost.

Mezi deterministické metody globální optimalizace můžeme zařadit například vyhledávání v mřížce (z anglického *(*Grid Search)). Tato metoda funguje tak, že rozdělí prohledávaný prostor na mřížku a poté prozkoumá každý bod na této mřížce. Ačkoli je tato metoda velmi jednoduchá, je také výpočetně náročná. Pro dosažení přesného řešení je totiž nutné zvolit mřížku dostatečně jemnou.

Stochastické

Stochastické algoritmy představují určitý opak deterministických algoritmů. Obsahují prvky náhody a nepředvídatelnosti, což znamená, že nelze přesně předpovědět, jaký krok bude proveden. Toto vede k tomu, že pro stejná vstupní data můžeme získat různé výstupní informace při dvou různých bězích. Tato vlastnost také znamená, že při použití stochastického algoritmu k globální optimalizaci není možné zaručit nalezení globálního minima.

Prvek náhody umožňuje prohledávání diskrétních bodů v různých oblastech s různou pravděpodobností. V nejjednodušším případě náhodné střelby je v celém prohledávaném prostoru pravděpodobnost prohledání shodná, ale v složitějších algoritmech roste pravděpodobnost prohledávání výhodnějších pozic, čili roste efektivita prohledávání.

Stochastické algoritmy se dělí na dva typy: heuristické a metaheuristické. Heuristické algoritmy jsou algoritmy, při nichž se pokouší uhodnout optimální řešení úlohy. Toto řešení nemusí být opravdu optimální, přesné nebo to nemusí být všechna řešení. Výhodou je, že alespoň nějaké řešení bylo nalezeno.

Metaheuristické algoritmy jsou nadstavby heuristických algoritmů, pořád je při nich zahrnut prvek náhody, ale je zde vyšší pravděpodobnost dopátrat se opravdu globálního optima. Vylepšení heuristických algoritmů spočívá obzvláště v tom, že je zde zapojena jistá kombinace náhody a lokálního prohledávání.

3.1.2 No Free Lunch teorém

Tento teorém, jehož název vychází z anglického přísloví „There is no such thing as a free lunch!" (do českého jazyka lze volně přeložit jako „Zadarmo ani kuře nehrabe!"), je obecný matematický princip, který se promítá i do problematiky globální optimalizace. Jeho významem zde je, že neexistuje algoritmus globální optimalizace, který by byl obecně nejlepší na všechny druhy problémů.

Ačkoliv tvrzení No Free Lunch teorému neodpírá možnost existence tříd algoritmů, které mohou být pro konkrétní problém optimální, neznamená to, že lze obecně předpokládat, že jeden algoritmus bude úspěšný na všechny druhy problémů. Je proto důležité zkoumat každý problém s co nejvíce specifickými informacemi relevantními pro daný kontext. Podrobnější informace o aplikaci No Free Lunch teorému v oblasti optimalizace lze nalézt například v práci [20].

3.2 Optimalizace hejnem částic (PSO)

Algoritmus optimalizace hejnem částic, inspirovaný pohyby hejn ptáků či ryb, je metaheuristický algoritmus schopný řešit problém globální optimalizace. Jeho vývoj začal v roce 1995, kdy James Kennedy a Russel Eberhart vydali první článek *Particle Swarm Optimization* [21], ve kterém byl základní algoritmus navržen. Algoritmus se dále rozvíjel a byly vydány jeho standardní verze, například: SPSO 2006 [22], SPSO 2007 [23] a SPSO 2011 [24]. Další verzí PSO můžeme představit například tzv. Tribes, který se od předchozích liší v tom, že není závislý na parametrech algoritmu, ale sám v závislosti na vstupech algoritmu zvládne vyřešit daný optimalizační problém [25]. Další popsané verze jsou dostupné na stránkách *Particle Swarm Central*: `https://www.particleswarm.info/index.html`.

V této sekci je popsán jednoduchý algoritmus SPSO 2006, jedná se o základní verzi PSO, jejíž kořeny lze nalézt již v první verzi z roku 1995 a jež se stala základem všech dalších verzí, které na algoritmus navazují. Tato verze algoritmus optimalizace hejnem částic dobře charakterizuje a dává do něj dobrý vhled. V další části jsou rozepsány informace o jeho výzkumu, například závislosti na parametrech, možné úpravy algoritmu apod, vycházející z dřívějších i ze soudobých výzkumů o tomto algoritmu.

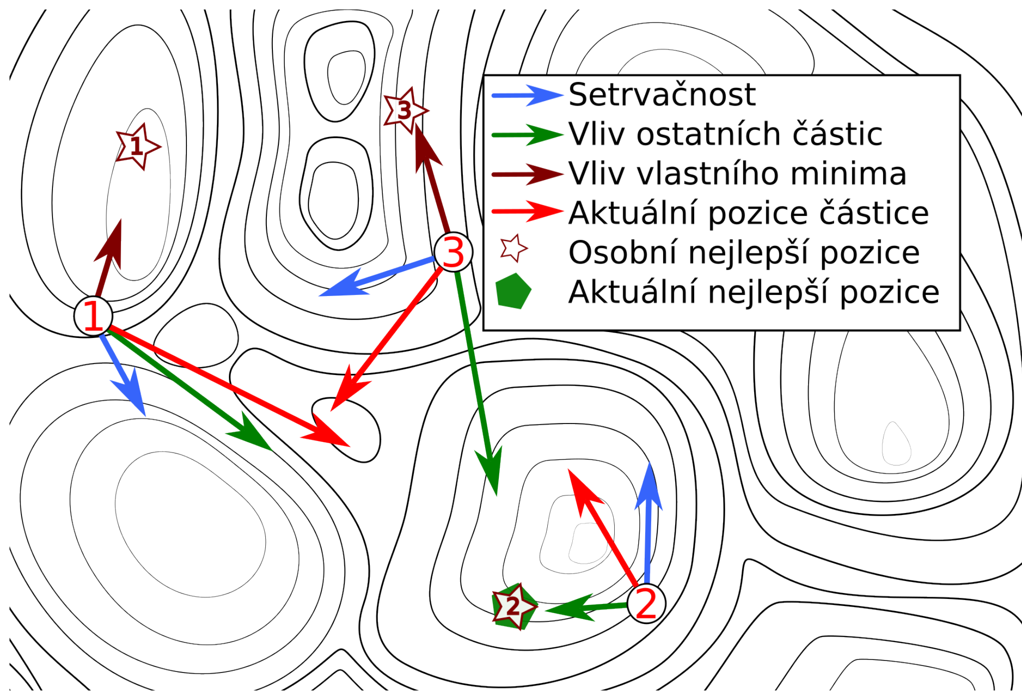
3.2.1 Standardní PSO 2006

Při optimalizaci funkce pomocí PSO je v prohledávaném prostoru rozmístěno S agentů (z historických důvodů a analogie jejich chování jsou agenti nazývány „částice" anglicky *particle*). Tyto částice mají za úkol postupně prohledávat prostor a společně se dopátrat optimální hodnoty funkce. Objektivní funkci částice optimalizují v iteračních krocích, které jsou zde analogické k časovým krokům t. O každé částici i máme uloženy následující informace:

1. Pozice v prohledávaném prostoru $\vec{x}_i(t)$
2. Hodnota objektivní funkce v dané pozici $F(\vec{x}_i)$
3. Rychlost, ze které se následně vypočítává nová pozice $\vec{v}_i(t)$
4. Pozice, ve které částice nalezla nejlepší hodnotu objektivní funkce dosavadním prohledáváním $\vec{m_i}(t)$
5. Hodnota objektivní funkce v předchozí nejlepší pozici $F(\vec{m}_i)$

Vektory $\vec{x}_i$, $\vec{v}_i$ a $\vec{m}_i$ mají stejný počet složek jako je dimenze prohledávaného prostoru.

Všech S částic dohromady tvoří „hejno" (anglicky *swarm*). Uvnitř hejna dochází ke komunikaci, při níž některé částice informují jiné částice o svých nalezených nejlepších pozicích a hodnotách. Skupina částic, která informuje konkrétní částici, se nazývá „sousedství" (z anglického *neighbourhood*). Na obrázku 3.1 si lze prohlédnout ilustraci fungování algoritmu.



Obrázek 3.1: Ilustrace algoritmu optimalizace hejnem částic. Nová rychlost částic 1, 2 a 3 je zde rozdělena ja jednotlivé komponenty (modrá setrvačnost, zelená sociální vliv pohybu a hnědá pohyb k vlastnímu minimu). Očíslované kruhy reprezentují polohy částic, očíslované hvězdy reprezentují polohy vlastních minim a zelený pětiúhelník polohu současného globálního minima. Červená šipka označuje, kam se částice pohne v dalším kroku.

Inicializace algoritmu

Před zahájením samotného iteračního algoritmu je potřeba určit informace o každé z částic. Kromě informací o jednotlivých částicích je potřeba dále přiřadit částici její sousedství, tj. seznam částic, které ji budou o sobě informovat:

1. Pozice částice $\vec{x}_i(0)$ může být vybrána buď náhodně nebo záměrně v konkrétní oblasti o které máme informaci, že je zde výhodné mít inicializovanou částici.

2. Hodnotu funkce $F(\vec{x}_i(0))$ zde lze vypočítat.

3. Rychlost částice lze zadefinovat jako $\vec{v}_i := \frac{1}{2}(\vec{x}_i - \vec{u}_i)$, kde $\vec{u}_i$ je náhodně vygenerovaná pozice v prohledávaném prostoru. Význam tohoto zlomku spočívá v tom, že pokud by částice nebyla ovlivněna dalšími vlivy, pak by skočila přesně doprostřed mezi tyto dvě vygenerované pozice.

4. Jako nejlepší nalezenou pozici částice, konkrétní funkce určíme současnou pozici částice $\vec{m}_i(0) := \vec{x}_i(0)$

5. Jako sousedství se obvykle volí náhodných K částic (doporučené je $K = 3$ [26]).

Pro celé hejno je potřeba určit současnou optimální pozici $\vec{G}(0)$ a hodnotu funkce $F(\vec{G})$ a nastavit parametry příslušící algoritmu.

6. Aktuální optimální pozice a nejlepší hodnota jsou zvoleny z pozic a hodnot všech částic, tj. jako $\vec{G}(0)$ je vybrána pozice částice $\vec{m}_i(0)$, jejíž hodnota objektivní funkce $F(\vec{m}_i)$ je nejlepší.

7. Při výpočtu nových rychlostí částic, viz 3.3, jsou jednotlivé vlivy započítávány s určitou váhou, tuto váhu představují parametry w, c_1 a c_2, jejichž význam je rozebírán v sekci 3.2.2. Hodnoty těchto parametrů je potřeba určit předem explicitně.

Iterační krok algoritmu

Pro každý iterační krok algoritmu a pro každou částici i je potřeba provést několik dílčích úkonů:

1. Určit nejlepší pozici částic ze sousedství $\vec{M}_i$.
2. Posunout každou složku d částice podle vztahů

$$v_i^d(t+1) = w \cdot v_i^d(t) + U(0,c_1) \cdot (m_i^d(t) - x_i^d(t)) + U(0,c_2) \cdot (M_i^d(t) - x_i^d(t)); \tag{3.3}$$
$$x_i^d(t+1) = x_i^d(t) + v_i^d(t+1); \tag{3.4}$$

   kde $U(0, c_j)$ označuje náhodný výběr z uniformního rozdělení intervalu $(0, c_j)$.
3. Aktualizovat hodnoty $\vec{m}_i$, případně $\vec{G}$, pokud došlo k nalezení lepší hodnoty funkce.
4. V případě, že nedošlo k vylepšení hodnoty $F(\vec{G})$, znovu zvolit nové sousedství.

Tento proces by se měl opakovat do té doby, než bude splněna některá z podmínek ukončení algoritmu. V Algoritmu 1 je uveden pseudokód algoritmu optimalizace hejnem částic.

Algorithm 1 Standardní PSO 2006

```
Initializace:
 1: for Každou částici i do
 2:     x⃗_i ← náhodná pozice
 3:     Vypočítej F(x⃗_i)
 4:     v⃗_i ← 0.5 · (x⃗_i − u⃗_i)
 5:     m⃗_i ← x_i
 6:     Vyber K sousedů částice i
 7: end for
 8: G⃗(0) ← pozice částice s nejlepší hodnotou
 9: Vypočítej F(G⃗)
Iterace:
10: repeat
11:     for Každou částici i do
12:         M⃗_i ← pozice částice s nejlepší hodnotou ze sousedství
13:         for Každou složku d do
14:             v_i^d(t+1) ← w · v_i^d(t) + U(0;c_1) · (x_i^d(t) − m_i^d(t)) + U(0;c_2) · (x_i^d(t) − M_i^d(t))
15:             x_i^d(t+1) ← x_i^d(t) + v_i^d(t+1)
16:         end for
17:         if F(x⃗_i(t+1)) < F(m⃗_i(t)) then
18:             m⃗_i(t+1) ← x⃗_i(t+1)
19:             if F(x⃗_i(t+1)) < F(G⃗(t)) then
20:                 G⃗(t+1) ← x⃗_i(t+1)
21:             else
22:                 G⃗(t+1) zůstává G⃗(t)
23:             end if
24:         else
25:             m⃗_i(t+1) zůstává m⃗_i(t)
26:         end if
27:     end for
28:     if Nedošlo ke změně pozice G⃗ then
29:         for Každou částici i do
30:             Vyber nových K sousedů částice i
31:         end for
32:     end if
33: until Kritérium ukončení
```

3.2.2 Parametry algoritmu PSO

Nyní, když byl představen základní algoritmus, je možné podívat se podrobněji na jednotlivé části algoritmu, protože samotný algoritmus nechává na uživateli velké množství vstupních parametrů, které je potřeba určit a mohou zásadně ovlivnit chování algoritmu. Seznam diskutovaných parametrů není vyčerpávající, protože existuje mnoho různých verzí PSO, které do něj přidávají další a další parametry, které se snaží pozměnit vývoj algoritmu.

Volba prohledávaného prostoru

Objektivní funkce má svůj definiční obor, proto prohledávaný prostor musí být podmnožinou definičního oboru. Navíc obvykle není potřeba prohledávat celý definiční obor funkce, protože některé části prostoru nemají logické opodstatnění být prohledávány (např. pokud by měla vzniknout chemická vazba s příliš velkou délkou).

Další specifický úkol je popsat daný prostor vhodnými souřadnicemi. Obvykle se volí souřadnice kartézské, ale lze zvolit i jiné souřadnice které popisují jiné stupně volnosti daného systému, při řešení krystalových struktur se objevují souřadnice odpovídající rotacím uvnitř molekuly a navíc souřadnice, které jsou omezené periodičností krystalu.

V literatuře [26, 27, 25] autoři uvádí, že se obvykle prohledávaný prostor volí jako hyperkvádr

$$E = \bigotimes_{i=1}^{N} [r_i^{min}; r_i^{max}], \tag{3.5}$$

kde $r_i^{min}; r_i^{max}$ jsou dolní resp. horní hranice souřadnice i. V případě řešení krystalových struktur je možné použít obdobu tohoto hyperkvádru s tím, že hranicemi mohou být rotace o plný úhel, tj. $(0°; 360°)$ nebo v případě periodických okrajů $(0; a)$, kde a je velikost mřížkového parametru.

Velikost hejna (počet částic)

Počet částic může významně ovlivnit konvergenci algoritmu i pravděpodobnost nalezení skutečného globálního optima objektivní funkce. V algoritmech SPSO 2006 a SPSO 2007 je navržené, že optimální hodnotu počtu částic S lze vypočítat pomocí jednoduchého vztahu

$$S = 10 + \lfloor \sqrt{2D} \rfloor, \tag{3.6}$$

kde D je dimenze prohledávaného prostoru a symbol $\lfloor \cdot \rfloor$ označuje dolní celou část. Bohužel takovýto odhad počtu částic je nevhodný a v [27] je ukázáno, že pro různé problémy se optimální hodnota počtu částic od hodnoty dané vztahem (3.6) může významně lišit. Dále je zde ukázáno, že optimální počet částic není dán pouze dimenzí prohledávaného prostoru, ale závisí i na konkrétní řešené úloze.

To že počet částic ovlivňuje jak pravděpodobnost nalezení globálního optima, tak konvergenci algoritmu je způsobená těmito důvody: pokud by byl počet částic příliš velký jejich pohyb bude do velké míry nahodilý a algoritmus by byl srovnatelný s algoritmem náhodné střelby v prohledávaném prostoru, proto by částice neměly tendenci zkonvergovat, navíc s velkým počtem částic roste i počet vyhodnocení objektivní funkce v každé iteraci, což může algoritmus dále zpomalovat; pokud by byl počet částic moc malý, tak částice budou mít tendenci po několika málo iteracích zkonvergovat do některého z optim, přičemž pravděpodobnost nalezení globálního optima může být velmi nízká.

Sousedství částice

Sousedství částice určuje, které částice budou informovat konkrétní částici o polohách svých minim. Z matematického hlediska vytváří topologii mezi jednotlivými částicemi. Existuje více druhů topologie, které lze využít při algoritmu PSO: jako myšlenkově nejjednoduší topologii patří ta, kde každá částice informuje každou částici – tato topologie umožňuje snadno definovat konvergenční kritérium algoritmu jednoduše tak, že k jeho konvergenci došlo poté, co všechny částice skončí v jednom bodě, nevýhodou této topologie může být, že vždy dochází k lokálnímu prohledávání prostoru pouze v jedné části prohledávaného prostoru; jiným příkladem topologie může být topologie, kdy jsou částice rozděleny do skupin – v tom případě se každá skupina chová jako vlastní hejno s topologií, kdy každá částice informuje každou.

Pro SPSO 2006 je doporučená *adaptivní náhodná topologie* [26]. Při této topologii je každá částice informovaná K jinými náhodně vybranými částicemi, to dává společně s informací o sobě dohromady až $K + 1$ částic, kterými je částice informovaná o jejich poloze a hodnotě vlastního minima. Výhodou této topologie je, že se sousedství obměňuje, a to pokaždé iteraci, kdy nedojde k vylepšení hodnoty funkce v bodě aktuální optimální pozice $\vec{G}$. Tyto změny sousedství vedou k tomu, že nedochází k uvěznění částic v lokálních optimech. V [26] je doporučeno použít hodnotu $K = 3$, při níž by mělo docházet k optimální komunikaci mezi částicemi.

Sousedství částice má velký vliv na konvergenci algoritmu a na pravděpodobnost nalezení globálního optima. Konvergence algoritmu je nastavením vhodné topologie zpomalena na optimální rychlost, tj. algoritmus zkonverguje do optima až v pozdějších fázích průběhu algoritmu. Právě toto delší prohledávání nelokálně umožňuje zvýšit pravděpodobnost nalezení globálního optima.

Parametry ovlivňující setrvačnost a sociální vliv částic

V definičním vztahu (3.3) se vyskytují parametry w, c_1 a c_2, jejich význam je po řadě: parametr, který udává vliv setrvačného pohybu částice v prostoru; parametr, který udává maximální váhu na vzdálenosti částice od předchozího nalezeného optima; a parametr, který udává maximální váhu na vzdálenosti částice od optima sousedství.

Parametr setrvačnosti pohybu částice určuje balanc mezi lokálním a globálním prohledáváním prostoru. Pokud bude hodnota parametru nízká, částice bude rychle konvergovat k pozici optimalní hodnoty funkce. Pokud bude hodnota parametru vysoká částice bude méně reagovat na své nalezené optimum a na optimum svého sousedství, takže bude prohledávat více globálně. Hodnota parametru w se v literatuře liší: v [28] se udává a je matematicky dokázána jako optimální hodnota

$$w = \frac{1}{2\ln(2)},$$

v článku [29] je udávaná typická hodnota mezi 0,8 a 1,2. Je zřejmé, že optimální hodnota bude záviset na typu problému.

Hodnoty parametrů c_1 a c_2 obvykle bývají shodné $c_1 = c_2 = c$. V závislosti na velikosti parametru c lze opět vyladit, zda bude částice prohledávat lokálně nebo globálně. V [25] je zkoumáno, kam pravděpodobně dopadne částice pro různé velikosti parametru c, směr dopadu částice je definován vektorem součtu vlastního lokálního optima s optimem částice, co se však liší je pravděpodobnost dopadu do určité vzdálenosti a rozptyl částic. Pro nízké hodnoty ($c = 1$) částice dopadala relativně blízko původní pozice a byl nízký rozptyl dopadajících pozic; pro vysoké hodnoty ($c = 2$) částice dopadala daleko od výchozí pozice a rozptyl byl výraznější. Z tohoto výzkumu plyne, že pro větší lokální prohledávání je vhodné zvolit nižší hodnotu parametru c a naopak. V [28] doporučují hodnotu

$$c = 1 + \ln 2 \approx 2,39;$$

v [26, 25] doporučují hodnotu poloviční, tj. $c \approx 1,193$. Členy definičního vztahu (3.3), které jsou násobeny uniformním rozdělením závislým na c, přináší do algoritmu schopnost hejna mezi sebou komunikovat.

Hraniční podmínky

Definice rychlosti částice umožňuje částici přesáhnout, někdy velmi výrazně, definovaný prostor, proto je potřeba následně omezit její pohyb. Existuje více možností, jak částici „vrátit“ zpět do prohledávaného prostoru: pokud je funkce definovaná i mimo prohledávaný prostor, je možné nechat částici dále vyvíjet (je pravděpodobné, že budou hodnoty funkce mimo prohledávanou oblast dostatečně nevýhodné na to, aby sociální vlivy částici přitáhly zpět do prohledávaného prostoru); pokud částice vyletí ve směru, který vykazuje periodičnost, je možné částici přesunout pomocí periodických okrajových podmínek zpět do prohledávaného prostoru; další možností je zrcadlit pohyb částice podle překročené hranice, tj. odečíst její vzdálenost od hranice od hraniční hodnoty; další možností je částici zastavit na hranici a vynulovat její rychlost.

Ideálním stavem by bylo, aby se částice k hranici prohledávaného prostoru vůbec nedostaly, ale to není obecně možné u všech řešených úloh, proto je potřeba volit vhodné hraniční podmínky nejen pro různé úlohy, ale i pro různé souřadnice jedné úlohy.

Podmínka konvergence algoritmu

Zajímavým problémem může být i zjišťování, zda algoritmus stále ještě prohledává prostor, nebo už našel globální optimum, nebo omylem zkonvergoval do lokálního optima. Pokud je známá poloha globálního optima, pak je tento úkol snadný – pouze vzdálenost nalezeného optima a globálního optima je dostatečně nízká.

Pokud poloha globálního optima není známa, je potřeba zadefinovat jinou podmínku konvergence algoritmu: nejjednodušší podmínkou může být překonat určitý počet iterací algoritmu, kdy nedochází ke zlepšení hodnoty nalezené minimální hodnoty; pokud mají částice tendenci zkonvergovat do jednoho bodu může být toto kritérium taktéž vhodné kritérium ukončení algoritmu; další možností může být, že rychlost všech částic bude nižší než stanovené kritérium; pokud bychom měli odhad, jaká by měla být optimální hodnota funkce je možné stanovit kritérium, které po překročení určitého limitu algoritmus ukončí.

Výběr vhodného kritéria ukončení může ovlivnit, zda algoritmus vůbec zkonverguje nebo zda nebude prohlášen za zkonvergovaný příliš pozdě a nebude docházet ke zbytečným dalším výpočtům.

3.2.3 Výhody a nevýhody PSO

Souhrn výhod a nevýhod algoritmu PSO vychází z článku [30]:

Výhody

- Jednoduchá implementace
- Relativně málo parametrů, které je potřeba ladit
- Možnost paralelizovat výpočty
- Robustnost
- Vysoká efektivita a pravděpodobnost nalezení minima
- Rychlá konvergence
- Nedochází zde k překryvu prohledávaných oblastí a není nutné provádět mutace optimalizovaných struktur
- Nízké výpočetní nároky
- Schopnost tvořit přesné matematické modely, které mohou řešit složité problémy

Nevýhody

- Není snadné nastavit výchozí parametry algoritmu
- Špatně se řeší příliš velký rozptyl hejna
- Příliš rychlá konvergence a uvěznění v lokálních optimech (obzvláště pro více-dimenzionální úlohy)

3.3 Další algoritmy globální optimalizace

Tato sekce si neklade za cíl představit všechny algoritmy globální optimalizace, které byly vymyšleny, ale představit několik vybraných algoritmů, které jsou dobrými reprezentanty své skupiny (deterministické, metaheuristické, stochastické) a jsou zajímavé svým návrhem. Agentem/agenty v této sekci je myšlena vyvíjející se trajektorie bodů z prohledávaného prostoru. Představa agenta, tj. osoby prohledávající prostor, je jednodušší pro vlastní vizualizaci a mnoha algoritmům umožňuje snazší vhled do nich.

Metadynamika

V roce 2002 Alessandro Laio a Michele Parrinello představili algoritmus metadynamiky [31]. V metadynamice agent prochází prohledávaným prostorem, na místech, kam již vstoupil, jsou přidávány penalizační funkce, které omezují další vstup agenta na toto místo. Agent se pohybuje ve směru největšího spádu, ale penalizační funkce umožňují to, že neprohledává pouze lokální optimum, ale podaří se mu dostat se z něj do optim dalších.

Velkou výhodou tohoto algoritmu je, že již po několika iteracích lze získat obecnou představu o tvaru objektivní funkce a udělat předběžné odhady. Dále algoritmus prohledává důkladně, pokud jsou penalizační funkce dostatečně lokální. Pomocí algoritmu metadynamiky lze získat relativně přesný tvar objektivní funkce v prohledávaném prostoru, protože postupným přidáváním penalizačních funkcí je modelována právě objektivní funkce (jen přenásobená minus jednou). Poslední výhodou je možnost paralelizace výpočtů algoritmu – pokud bude prostor prohledávat více agentů současně, dosažení minima by mělo být rychlejší. Nevýhodou algoritmu může být, že opustit oblast spadající k jednomu lokálnímu minimu může vyžadovat více výpočtů.

Jako penalizační funkce se používá obvykle funkce Gaussova (obecně vícedimenzionální), takže po prohledání prostoru má objektivní funkce přibližně tvar

$$F(\vec{s}) \approx \tau \sum_{j=0}^{N} \omega \exp\left(-\frac{1}{2}\left|\frac{\vec{s}-\vec{s}_j}{\sigma}\right|^2\right), \tag{3.7}$$

kde τ, ω a σ jsou parametry algoritmu. Nastavení těchto parametrů je zásadní pro správný běh algoritmu, neboť jinak by mohly být penalizační funkce příliš vysoké, nízké, široké nebo úzké.

Basin hopping

Algoritmus basin hopping (do českého jazyka lze přeložit jako *přeskoky mezi údolími*) byl poprvé představen v článku Davida Walese a Jonathana Doyla [32]. Algoritmus je zde představen na příkladu minimalizace energie struktury Lennardova-Jonesova clusteru.

Princip algoritmu basin hopping spočívá v relaxaci agenta do lokálního minima (například pomocí gradientního sestupu). Poté, co agent nalezne lokální minimum, je posunut do nové náhodné pozice. Podle Metropolisova kritéria je tato nová pozice buď přijata anebo není, pokud není je prováděn nový skok dokud není nová pozice přijata. V další iteraci se opakují tyto dva iterační kroky, čímž dochází k prohledávání daného prostoru.

Basin hopping se vykazuje tím, že je schopen najít globální minimum i ve vícedimenzionálních prostorech. Nevýhodou je, že může mít problémy s hlubokými údolími prohledávaného prostoru, ze kterých se obtížně přesouvá do jiné oblasti prostoru [33].

Algoritmus basin-hopping patří mezi algoritmy stochastické.

Minima hopping

Podobným algogitmem jako je basin hopping je minima hopping (do českého jazyka lze přeložit jako *přeskoky mezi minimy*). I v tomto algoritmu nejprve dochází k relaxaci do lokálního minima a následným přeskokům do minim dalších, rozdílný však je princip, jímž se mezi minimy přeskakuje. V algoritmu minima hopping se nechá systém dynamicky vyvíjet – je mu dodána „kinetická energie", pomocí níž může přeskočit do jiného minima. Dobře je tento algoritmus ilustrovatelný na příkladu dynamiky termodynamického systému, kdy částice opravdu získávají kinetickou energii. Vhodné je tento algoritmus používat v oblasti simulací fyzikálních systémů jako například molekul, krystalů, nanočástic apod, neboť tento algoritmus do značné míry simuluje skutečné pohyby částic.

Výhodou algoritmu minima hopping je, že prohledává jak lokálně, tak globálně. Oproti algoritmu basin hopping má menší tendence vracet se zpět do již nalezených minim (pokud jsou správně nastaveny jeho parametry) [34].

Simulované žíhání

Dalším metaheuristickým algoritmem je simulované žíhání. Tento algoritmus byl pojmenovaný podle procesu postupného ochlazování a zahřívání kovů, který se využívá v metalurgii k získání kýžených fyzikálních vlastností materiálu, viz článek [35].

Principem simulovaného žíhání je postupné procházení agenta prohledávaným prostorem, kdy agent přijímá všechny kroky na místa, kde nalezne výhodnější pozici, ale navíc přijímá i některé kroky, které směřují na nevýhodnější pozici. Důvodem tohoto přijímání nevýhodnějších pozic je to, že pomocí těchto kroků lze překonat nevýhodnou oblast a dostat se do okolí jiného lokálního minima. Přijímání nevýhodných kroků je obvykle řízeno parametrem teploty T, který se vyskytuje v Metropolitním kritériu:

$$p \sim \exp \frac{F_{n-1} - F_n}{kT}, \tag{3.8}$$

kde p je pravděpodobnost přijetí nevýhodného kroku, n označuje kolikátá iterace algoritmu právě probíhá a k je konstanta, pomocí níž se dá nastavovat závislost na teplotě (odpovídá fyzikálnímu významu Boltzmanovy konstanty).

Parametr teploty se postupně v průběhu algoritmu snižuje a tím klesá i pravděpodobnost přijetí nevýhodného kroku. Tento postup umožňuje nejprve prohledávat prostor globálně a v závěru lokálně. Teplota se nemusí snižovat po celou dobu běhu algoritmu, ale můžou zde nastávat zvýšení za účelem zvýšení pravděpodobnosti opuštění lokálního minima.

Kromě teploty je podstatným parametrem simulovaného žíhání také délka kroku, který agent vykoná, lze pomocí něj nastavovat lokálnost/globálnost prohledávání. S velkým krokem může agent přeskakovat z jednoho bodu prohledávaného prostoru na další a tím pádem prohledávat globálně, s malým krokem se bude agent držet v menší oblasti a bude prohledávat více lokálně.

Paralelní temperování

Algoritmus paralelního temperování funguje na podobném principu jako simulované žíhání. Opět se náhodně prochází prohledávaným prostorem a s určitou pravděpodobností se přijímají i nevýhodné kroky. Rozdílem však je, že prostor neprohledává jeden agent, ale více agentů. Tito agenti mají pevně stanovené pravděpodobnosti přijímání nevýhodných kroků, které jsou stanovené

pomocí parametru teploty, která je dosazována do Metropolitního kritéria 3.8. Toto umožňuje, že někteří z agentů (s vysokým parametrem teploty) prohledávají prostor globálně, někteří naopak lokálně. Kromě tohoto může navíc nastat změna teplot agentů i a j, tato změna nastane s pravděpodobností

$$p = \min\left\{1, \exp\left[+(F(\vec{x}_i) - F(\vec{x}_j))\left(\frac{1}{k_i T_i} - \frac{1}{k_j T_j}\right)\right]\right\}. \tag{3.9}$$

Zaměňování teplot agentům umožňuje to, že se žádný z agentů by se neměl natrvalo zaseknout v jednom lokálním optimu, ale bude z něj moci vyskočit. Navíc to umožňuje prohledat optimum lokálně hned poté, co do něj některý agent skočí při svém globálním prohledávání.

Mezi výhody algoritmu patří to, že prohledává oblast současně globálně i lokálně a že lokální prohledávání se soustředí na výhodné oblasti. Oproti algoritmu simulovaného žíhání není nutné tento algoritmus spouštět vícekrát, protože zde už vycházíme z různých startovacích pozic. Algoritmus je výhodné použít při hledání globálního optima, avšak neumožňuje efektivně zmapovat prohledávaný prostor (nelokalizuje tranzitní stavy apod).

Tabu search

Algoritmus tabu search je založen na tom, že agent procházející prohledávaným prostorem lokálně prohledává své okolí. Další krok volí tak, aby měl výhodnější pozici a navíc aby se vyhnul zakázaným oblastem (tabu). Tento proces se opakuje, dokud není proveden dostatečný počet možných kroků. Oblasti tabu umožňují agentovi dostat se z lokálního minima.

Při tomto algoritmu je dobře ošetřeno, že se nepočítá vícekrát funkce v jednom a tom samém bodě. Nevýhodou může být, že je nutné ukládat velkou část trajektorie agenta, což může vyžadovat větší paměťové nároky na výpočty.

Algoritmus byl navržen a publikován v roce 1986 v článku [36] a je považován za metaheuristiku.

Genetický algoritmus

Je to metaheuristický optimalizační algoritmus, který je inspirovaný procesem přirozeného výběru. V tomto algoritmu dochází k vývoji populace, kde členem populace je myšlen bod prohledávaného prostoru, tak aby do další generace postoupili jen ti členové s nejlepší hodnotou objektivní funkce. Z původní *rodičovské* populace je vytvořena nová rozšířená populace, tj. obsahuje celou rodičovskou populaci a nějaké členy navíc. Všichni členové rozšířené populace jsou ohodnoceni objektivní funkcí, z těchto ohodnocených členů je vybráno několik, kteří „přežijí“ do další generace, tj. stanou se novou rodičovskou populací, a zbytek členů je zapomenut. Tímto dochází k postupnému vylepšování vlastností členů populace, tj. k optimalizaci objektivní funkce.

Procesy vytváření rozšířené populace z populace rodičovské mohou být různé, například křížení či mutace. Při křížení dochází k vytvoření dvou nový členů populace z dvou členů populace rodičovské tím, že se k vytvoření nových členů použijí části informací obou členů rodičů (například předání informace o poloze v prohledávaném prostoru). U mutace dochází k náhodné změně části rodiče (například náhodný posun v prohledávaném prostoru).

Bayesovská optimalizace

Při Bayesovské optimalizaci je nejprve vypočítána objektivní funkce v několika bodech, které reprezentují prohledávaný prostor. Následně se provede odhad, jak funkce může vypadat v prohledávaném prostoru. Body, ve kterých je hodnota vypočítaná, se berou jako fixní a body, v nichž hodnota známa není, se odhadují. Poté je vypočítána nová (*akviziční*) funkce, která vyjadřuje, kam je výhodné umístit následující bod, ve kterém se bude funkce vypočítávat. Pomocí optimalizace akviziční funkce je určen bod, ve kterém je proveden další výpočet objektivní funkce. Tento proces se opakuje dokud není překročen počet iterací algoritmu. Výsledkem je mapa funkce, ze které by mělo být patrné optimum objektivní funkce.

Tento druh optimalizace je vhodný na optimalizaci a mapování funkcí, u nichž je výpočetně náročné určit jejich hodnotu, protože algoritmus odhaduje průběhy funkce i v bodech, kde nebyla funkce vypočítaná. Další výhodou je, že ze své podstaty toleruje nepřesnosti při výpočtu funkce. Omezením metody je, že je vhodná na prohledávání prostoru o maximální dimenzi 20 [37].

Algoritmus velkého třesku-velkého křachu

Algoritmus velkého třesku-velkého křachu (v anglickém originále *Big Bang-Big Crunch*) je metaheuristický algoritmus, který je inspirovaný vznikem a zánikem vesmíru a vzájemným gravitačním ovlivňováním těles.

V úvodní fázi velkého třesku jsou agenti náhodně rozmístěni v prohledávaném prostoru. V druhé fázi velkého křachu se agenti pohybují směrem k těžišti populace. Těžiště je určeno podle vztahu

$$\vec{x}_c = \frac{\sum_{i=1}^{N} \frac{1}{f^i} \vec{x}^i}{\sum_{i=1}^{N} \frac{1}{f^i}}, \tag{3.10}$$

kde N je počet agentů, f^i je hodnota objektivní funkce v bodě $\vec{x}^i$ a $\vec{x}_c$ je těžiště populace. V této fázi dochází k „gravitačnímu kolapsu“ populace, tj. agenti se přibližují k sobě a postupně dochází k velkému křachu, tj. agenti skončí v jednom bodě. V další iteraci algoritmu dochází k velkému třesku, tj. agenti jsou náhodně rozmístěni v prohledávaném prostoru a algoritmus se opakuje [38].

Další algoritmy využívající inteligenci hejna

Existují další algoritmy, které využívají inteligenci hejna, ale nejsou založeny přímo na algoritmu PSO. Obvykle byly inspirovány chováním skupiny zvířat. Mezi tyto algoritmy patří například optimalizace pomocí mravenčí kolonie [39], optimalizace hejnem světlušek [40], optimalizace hejnem koček [41] a další.

Kapitola 4

Implementace algoritmu optimalizace hejnem částic do programu FOX

Pro implementaci algoritmu byla zvolena jeho verze Standardní PSO 2006 [22] z důvodu, že je přehledná a lze zde snadno dolaďovat konkrétní parametry, aby docházelo ke kýženým vlivům na algoritmus, tj. ladit rychlost konvergence, poměr mezi globálním a lokálním prohledáváním apod.

Algoritmus byl napsán v programovacím jazyku C++ do modulu RefObjCryst programu FOX. V tomto modulu jsou napsány všechny metody, které souvisí s optimalizací krystalových struktur, mimo jiné i metody algoritmů paralelního temperování a simulovaného žíhání.

Kód je k dispozici na GitHub: knihovna libObjCryst na adrese `https://github.com/kocimil1/libobjcryst` a skriptovací rozhraní v Pythonu na adrese `https://github.com/kocimil1/pyobjcryst`.

4.1 Popis kódu

Hlavní část přidaného kódu je zapsána ve funkci `RunParticleSwarmOptimization`, která je metodou třídy `MonteCarlo`, právě v této funkci algoritmus PSO probíhá. Dále byly implementovány další podpůrné funkce a metoda `SetAlgorithmParticleSwarmOptimization`, která nastavuje parametry algoritmu.

4.1.1 Funkce RunParticleSwarmOptimization

Inicializace

Před zahájením hlavního průběhu cyklu, popsaného v sekci 3.2, je potřeba inicializovat proměnné algoritmu a uložit některé hodnoty. Jsou zadefinována pole `x`, `v`, `m` a `M`, která odpovídají polohám, rychlostem, vlastním lokálním minimům částic a společnému globálnímu minimu. Do pole `x` jsou uloženy polohy jednotlivých částic, tj. hodnoty odpovídající volným parametrům krystalové struktury, pole `v` je naplněno náhodnými hodnotami z intervalu $(-0,5; 0,5)$ a hodnoty pole `m` jsou zkopírovány z pole `x`. Následně je pro všechny částice vypočítána jejich hodnota objektivní funkce, uložen snímek jejich hodnot a vybrána částice, která má nejnižší hodnotu objektivní funkce. Tato částice je prohlášena za současnou nejlepší a pole `M` je naplněno jejími hodnotami poloh.

Po tomto procesu tedy máme připravené polohy částic, rychlosti částic, polohy vlastních minim a polohu globálního minima. Navíc je ještě připraveno prázdné pole `Neighbourhoods`, do kterého se budou ukládat indexy příslušící sousedům částic.

Hlavní cyklus

Hlavní část cyklu probíhá v iteracích, jejichž počet je limitován podílem umožněného počtu výpočtů `nbTrial` ku počtu částic `nbPart`, protože při jedné iteraci je provedeno `nbPart` výpočtů objektivní funkce.

Hlavní cyklus lze rozdělit na čtyři základní části:

1. Výběr sousedství – pokaždé když v předchozí iteraci nedojde ke zlepšení hodnoty globálního minima jsou částicím přiřazeny noví sousedé.

2. Dynamický vývoj částic – částicím jsou přiřazeny nové polohy a rychlosti podle vztahů (3.3). Výjimku z tohoto má částice, pokud má nejnižší hodnotu objektivní funkce ze svého sousedství, která neuvažuje posun k minimu svého sousedství z důvodu, aby nedocházelo k příliš velkému vychýlení jedním směrem.

3. Ohodnocení a srovnání částic – všechny částice jsou ohodnoceny objektivní funkcí a následně jsou aktualizovány jejich hodnoty a polohy lokálních minim a globálního minima.

4. Test konvergence – je otestováno, zda algoritmus již nezkonvergoval, viz Podpůrné funkce 4.1.1.

Většina činností je nezávislá na vlastnostech programu FOX a lze použít obecně – výběr sousedů, srovnání částic, výpočet hodnot nové pozice a rychlosti, test konvergence. Program FOX zajišťuje obnovování a ukládání struktur příslušících jednotlivým částicím, tj. pohyb částice vychází z předchozího snímku hodnot parametrů; výpočet hodnoty objektivní funkce a pohyb parametrů struktury.

Funkce `move` (viz sekci Padpůrné funkce 4.1.1) hraje klíčovou roli v koordinaci pohybu mezi parametry částice a parametry struktury, protože změnou hodnot v poli `x` se automaticky nezmění hodnota optimalizovaného parametru simulované struktury. Tato funkce se zaměřuje na správné sladění chování jednotlivých parametrů struktury vzhledem k hodnotám parametrů částic. Například pokud je parametr simulované struktury periodický, pak je nutné zajistit, aby hodnota parametru částice byla v základním intervalu, protože hodnota parametru simulované struktury je vypočítána pomocí funkce `Mutate` nebo `MutateTo`, které hodnotu parametru ošetřují a přiřadí hodnotu v základním intervalu. Tím by byla ztracena informace o reálné hodnotě parametru částice a tím i o reálné hodnotě parametru simulované struktury. Proto funkce `move` ošetřuje tyto situace.

Další zpracování výsledků

Po skončení hlavního cyklu jsou obnoveny hodnoty parametrů struktury částice s nejlepší hodnotou objektivní funkce a lze provést další lokální optimalizaci pomocí metody nejmenších čtverců. Dále jsou smazány všechny snímky parametrů, které byly provedeny při běhu funkce a smazány i dynamicky alokovaná pole. Vymazání polí vede k tomu, že nedochází ke ztrátám paměti a při vícenásobném spouštění funkce k zpomalování procesů.

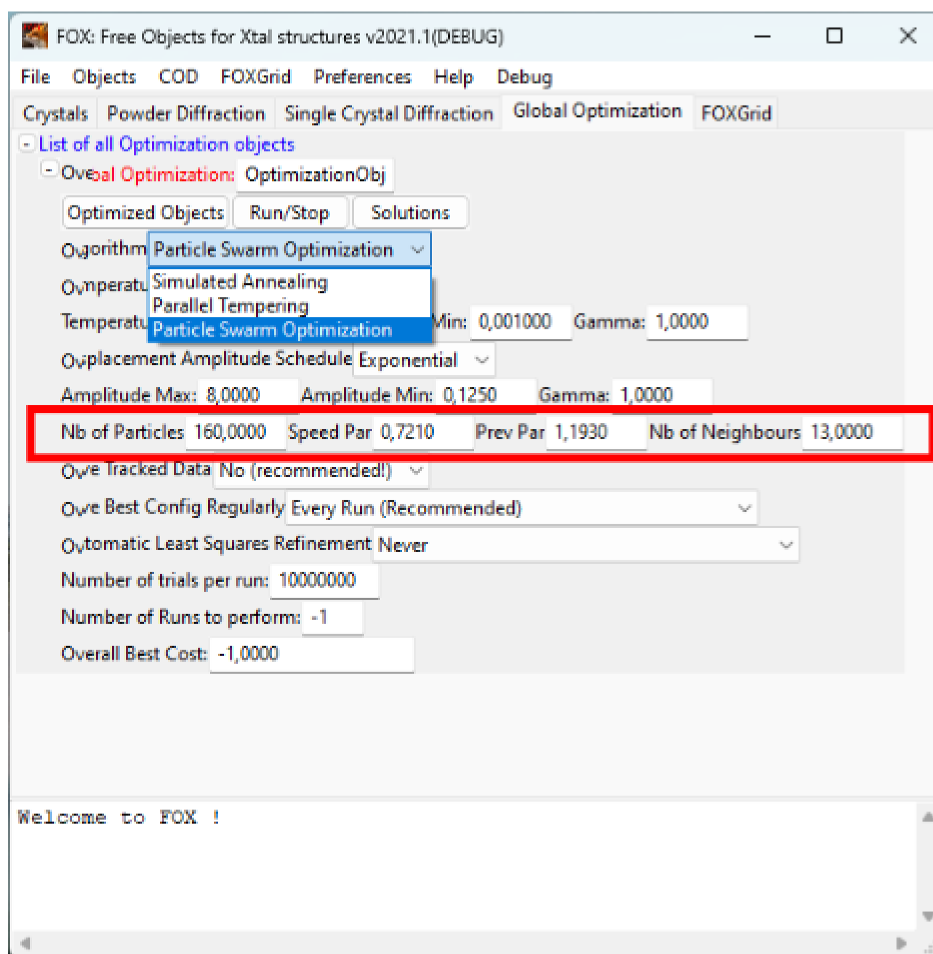
Podpůrné funkce

Mezi podpůrné funkce jsou zařazeny funkce `move`, `converged` a vlastní funkce programu FOX `Refine`, která při použití na objekt `LSQ` provádí metodu nejmenších čtverců.

Funkce `move` zařizuje komunikaci mezi programem FOX a poli poloh a rychlostí, navíc ošetřuje typy parametrů. Pohyb parametru simulovaného objektu lze v programu FOX provést pomocí funkcí `Mutate` a `MutateTo`, problém nastává v tom, že při použití těchto funkcí může docházet k ošetření hodnoty parametrů (například periodický parametr, který má hodnotu mimo základní interval, získá hodnotu v základním intervalu) a tím je pro pole polohy a rychlosti `x` a `v` ztracena informace o reálné poloze a rychlosti struktury. Proto funkce `move` ošetřuje tato vychýlení mimo základní intervaly a navíc zde probíhá upravení hodnot pozic částice. Funkce `move` navíc ošetřuje nekomutativitu kvaternionů odpovídajících rotaci molekul – přiřadí poli `x` hodnotu, která odpovídá současné hodnotě parametru kvaternionu. Tím je opět získána hodnota parametru a lze ji použít při výpočtu dalších rychlostí.

Podpůrná funkce `converged` provádí kontrolu zda algoritmus již splňuje některé z kritérií konvergence. (Kontrolu neprovádí pokaždé, jen pokud se po nastavený počet kroků nemění hodnota globálního minima.) Mezi kritéria konvergence jsme zařadili překročení limitu objektivní funkce zadaného uživatelem a situaci, kdy průměrná rychlost jedné částice je nižší než 0,01. V kódu je navíc implementován test konvergence pomocí blízkosti částic, ale není vhodné jej používat z důvodu kvadratické náročnosti testování.

Metoda nejmenších čtverců zde reprezentuje lokální optimalizační metodu a byla již implementována v programu FOX dříve. Její použití v rámci PSO ovšem umožňuje rychlejší konvergenci a nalezení výhodnějších minim, proto lze nastavit, že v průběhu algoritmu bude provedena optimalizace metodou nejmenších čtverců – deset procent částic s nejnižší hodnotou objektivní funkce



Obrázek 4.1: Grafické rozhraní programu FOX s implementovaným algoritmem optimalizace hejnem částic. Uživatel si vybere algoritmus PSO a spustí optimalizaci.

je optimalizováno a tím většinou získávají nové nejlepší vlastní polohy a hodnoty. (Lokální optimalizace u více částic částečně zabraňuje tomu, aby všechny částice konvergovaly do jednoho minima. Důvodem je, že si částice vyměňují své sousedy, a pokud jsou nalezena alespoň dvě minima, pak částice mohou přeskakovat mezi těmito minimy. Provedení lokální optimalizace u všech částic by mohlo trvat delší dobu a částice více vzdálené od lokálního minima by mohly konvergovat do minima pomalu.)

4.1.2 Nastavení algoritmu PSO

Nastavení parametrů PSO probíhá přes funkci `SetAlgorithmParticleSwarmOptimization`, jejíž parametry odpovídají parametrům PSO – počet částic, parametr setrvačnosti, sociální parametr a počet sousedů. Smyslem této funkce je oddělení nastavování algoritmu od samotného běhu algoritmu, což umožňuje bezpečnější běh algoritmu (uživatel se nemůže rozhodnout v probíhajícím algoritmu změnit jeho parametry).

4.2 Interakce s algoritmem

4.2.1 Příklad použití grafického rozhraní

Použití grafického rozhraní umožňuje uživateli snadno a přehledně provést úlohu řešení krystalové struktury. Pokud se uživatel rozhodne, že chce použít algoritmus optimalizace hejnem částic, musí pouze při nastavování objektu MonteCarlo vybrat možnost Particle Swarm Optimization a spustit optimalizaci. Navíc může nastavit parametry algoritmu, pokud vyžaduje jiné chování algoritmu. Na obrázku 4.1 je snímek interakce skrze grafické rozhraní probíhá.

4.2.2 Příklad použití Python skriptu

Prostředí Pythonu je výhodné pro spouštění komplikovanějších automatizovaných procesů při řešení struktury. Například pokud je řešena struktura a nejsou známy optimální parametry konkrétního optimalizačního algoritmu, pak lze pomocí skriptu přednastavit prohledávání prostoru parametrů algoritmu a nemuset po každém doběhnutém výpočtu přenastavovat parametry ručně. Použití algoritmu PSO na řešení krystalové struktury pomocí Python skriptu lze více způsoby – voláním optimalizačních funkcí nebo zavoláním samotné funkce `RunParticleSwarmOptimization`.

Uživateli je důrazně doporučováno používat metody `MultiRunOptimize(nb_run, nb_trial, final_cost=0, max_time=-1)` nebo `Optimize(nb_trial, final_cost=0, max_time=-1)`, protože automaticky připravují program na optimalizaci, tj. vyhledají parametry, které je potřeba optimalizovat, a ostatní zablokují; informují zbytek programu o probíhající optimalizaci; náhodně změní strukturu, aby prohledávání nezačínalo pokaždé ze stejné výchozí apod. Samy tyto funkce následně volají funkci `RunParticleSwarmOptimization`, ve které algoritmus PSO probíhá. Příklad použití metody `MultiRunOptimize` s konkrétním nastavením jeho parametrů počtu běhů `nb_run` a počtu výpočtů na jeden běh `nb_trial` je uvedeno zde:

```
1  NbRun = 100
2
3  algorithm = 2 # 0 -> Simulated Annealing; 1 -> Parallel Tempering; 2 -> Particle
       Swarm Optimization
4  mc.GetOption("Algorithm").SetChoice(algorithm)
5
6  # Automatic LSQ after 150k trials
7  mc.GetOption("Automatic Least Squares Refinement").SetChoice(2)
8
9  mc.SetAlgorithmParticleSwarmOptimization(nbParticles=100, parFormerSpeed=0.6,
       parFormerMinima=1.193)
10
11 mc.MultiRunOptimize(nb_run=NbRun, nb_trial=1e6)
```

Mimo samotné volání funkce `MultiRunOptimize` je potřeba zvolit algoritmus PSO a je možné nastavit (ne)použití metody nejmenších čtverců při optimalizaci a parametry algoritmu PSO.

Při použití funkce `RunParticleSwarmOptimization(nb_step, final_cost=0, max_time=-1, ScattTransl = 1.0, ScattConform = 1.0, ScattOrient = 1.0)` je potřeba nejprve připravit parametry sturktury k optimalizaci, proto je nejprve potřeba zavolat funkci `PrepareRefParList()`, která prohledá všechny dostupné parametry a určí, které lze optimalizovat. Dále je opět možné nastavit použití metody nejmenších čtverců v průběhu algoritmu a parametry PSO. Funkce `RunParticleSwarmOptimization` provede jeden běh algoritmu PSO na optimalizaci krystalové struktury. Parametry funkce `RunParticleSwarmOptimization` odpovídají počtu výpočtů funkce LLK, srovnávající difrakční záznamy, finální hodnotě, maximálnímu času a maximální velikosti změn parametrů translačního pohybu s molekulou, pohybu s atomem a rotačního pohybu s molekulou. Tyto parametry jsou zde umožněny, protože PSO vyžaduje jinou velikost maximálních změn parametrů než paralelní temperování nebo simulované žíhání. Příklad použití `RunParticleSwarmOptimization` je uveden zde:

```
1  mc.PrepareRefParList()
2
3  # Automatic LSQ after 150k trials
4  mc.GetOption("Automatic Least Squares Refinement").SetChoice(2)
5
6  mc.SetAlgorithmParticleSwarmOptimization(nbParticles=100, parFormerSpeed=0.6,
       parFormerMinima=1.193)
7
8  mc.RunParticleSwarmOptimization(nb_trial=1e6)
```

Kapitola 5

Testovací struktury

V této kapitole jsou popsány struktury, které byly použity pro testování algoritmu PSO v programu FOX. Konkrétně byly testovány struktury $PbSO_4$, paracetamolu a sofosbuviru. Všechny struktury byly vyřešeny pomocí programu FOX a byly použity pro porovnání dvou algoritmů – optimalizace hejnem částic a paralelní temperování. První dvě struktury byly vyřešeny v operačním systému Ubuntu 22.04 a procesor má 13th Gen Intel(R) Core(TM) i5-13600K. Struktura sofosbuviru byla vyřešena na metacentru: *Computational resources were provided by the ELIXIR-CZ project (ID:90255), part of the international ELIXIR infrastructure.*

V tabulce 5.1 jsou uvedeny parametry algoritmu PSO, které byly použity pro řešení struktur $PbSO_4$, paracetamolu a sofosbuviru. Tyto parametry byly zvoleny na základě experimentů, literatury a zkušeností autora. Pro strukturu paracetamolu byly parametry zvoleny na základě experimentů, které byly provedeny na této struktuře (viz kapitola 6). Pro všechny struktury byl počet iterací algoritmu PSO zvolen tak, aby bylo dosaženo konvergence algoritmu. Pro strukturu $PbSO_4$ bylo zvoleno 100 000 iterací, pro strukturu paracetamolu 400 000 iterací a pro strukturu sofosbuviru 1 000 000 iterací. Pro strukturu $PbSO_4$ byla provedena pouze jedna lokální optimalizace, pro strukturu paracetamolu byly provedeny tři lokální optimalizace a pro strukturu sofosbuviru bylo provedeno sedm lokálních optimalizací. Pro strukturu $PbSO_4$ byl parametr setrvačnosti zvolen na hodnotu 0,5, pro strukturu paracetamolu na hodnotu 0,6 a pro strukturu sofosbuviru na hodnotu 0,721. Sociální parametry byly pro všechny struktury zvoleny na hodnotu 1,193. Pro všechny struktury byl počet sousedů zvolen na hodnotu 3.

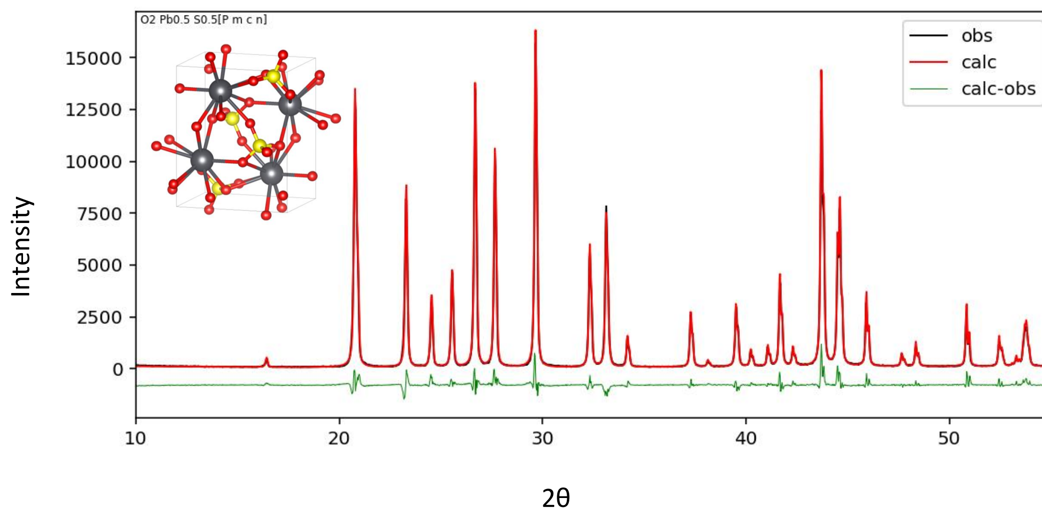
Tabulka 5.1: Parametry algoritmu PSO použité pro řešení struktur $PbSO_4$, paracetamolu a sofosbuviru.

Parametr	$PbSO_4$	Paracetamol	Sofosbuvir
Počet částic	100	100	100
Počet iterací	100 000	400 000	1 000 000
Počet lokálních optimalizací	1	3	7
Parametr setrvačnosti	0,5	0,6	0,721
Sociální parametry	1,193	1,193	1,193
Počet sousedů	3	3	3

5.1 $PbSO_4$

Síran olovnatý $PbSO_4$ je bílý krystalický prášek, který není rozpustný ve vodě. Pro člověka je toxický při vdechnutí nebo při kontaktu s kůží. Používá se při výrobě dalších chemikálií, v litografii a při výrobě olověných baterií. Síran olovnatý krystalizuje v ortorombické krystalové mřížce s prostorovou grupou $P\,n\,m\,a$ s parametry mřížky 8,516 Å; 5,399 Å a 6,989 Å[42].

Cílem řešení této struktury bylo aplikovat postup z tutoriálu k programu FOX [43], kde se pouze pozmění algoritmus, kterým se řeší krystalová struktura. Tutoriál ovšem není obvyklým řešením krystalové struktury $PbSO_4$, ale je řešením krystalové struktury z rentgenových a zároveň neutronových práškových difrakčních záznamů.



Obrázek 5.1: Vyřešená struktura $PbSO_4$ a její odpovídající rentgenové práškové difrakční záznamy (černě naměřený, červeně vypočítaný, zelená linka odpovídá rozdílu obou záznamů).

Řešení krystalové struktury

Nejprve byla do programu načtena rentgenová difrakční data a nastavena vlnová délka rentgenového záření na vlnovou délku odpovídající rentgence s měděnou anodou. Rentgenový práškový difrakční záznam si lze prohlédnout na obrázku 5.1. Na rentgenovém difrakčním záznamu bylo nalezeno 20 difrakčních maxim. Difrakční maxima byla oindexována a byla získána ortorombická krystalová mřížka s parametry 5,40 Å; 6,97 Å a 8,49 Å. Byly provedeny dva LeBailovy fity na rozsahu difrakčního záznamu omezeném maximální hodnotou $\frac{\sin\theta}{\lambda} = 0,3$. Při hledání prostorové grupy, program FOX nalezl více struktur, které měly výhodnou hodnotu míry shody difrakčních záznamů a prostorová grupa $P\,m\,c\,n$ (stejná grupa jako $P\,n\,m\,a$, jen má jiné pořadí parametrů mřížky) se nacházela mezi nimi. Při řešení neznámé struktury by bylo nutné prohledat více grup, ale protože je struktura $PbSO_4$ známa, tak jsme zvolili správnou prostorovou grupu.

Do programu byla vložena informace o prvcích, která budou působit jako rozptylová centra krystalu a byla do krystalu vložena tak, že atom olova byl vložen jako samostatný atom a atomy olova a kyslíku byly vloženy do krystalu jako čtyřstěn, který má ve svém středu síru a ve vrcholech kyslíky.

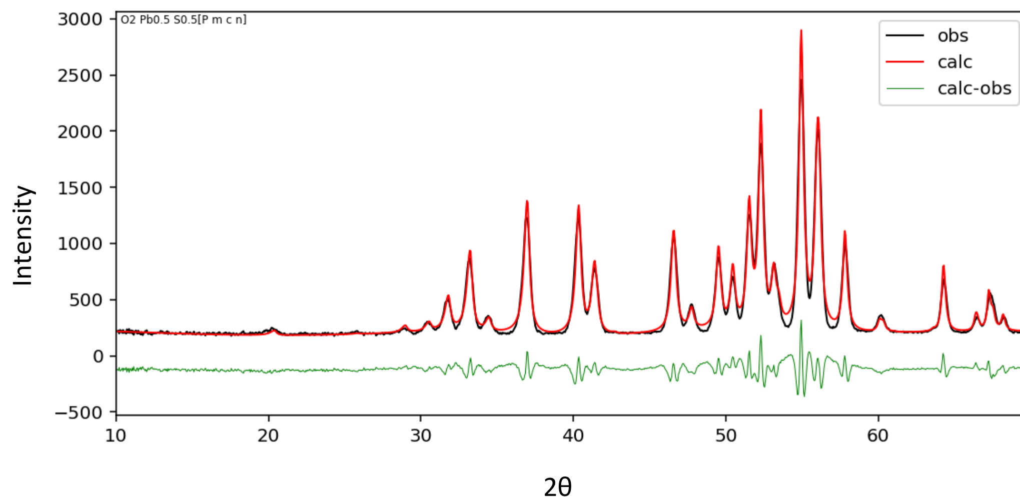
Do programu byl načten neutronový difrakční záznam a nastavena jeho vlnová délka 1,909 Å, lze si jej prohlédnout na obrázku 5.2 černě. Byl shora omezen rozsah difrakčního záznamu $\frac{\sin\theta}{\lambda} = 0,3$ a byl nastaven Cagliottiho parametr $W = 0.001$. Byl proveden fit škálovacího parametru a následně dva LeBailovy fity difrakčních záznamů.

Zkonstruovali jsme `MonteCarlo` objekt a přidali do něj krystal a oba difrakční záznamy. Byly vypnuty extrakční módy pro oba difrakční záznamy a proběhl fit škálovacích parametrů. Nastavili jsme parametry algoritmu PSO na 100 částic, parametr setrvačnosti na 0,5 a sociální parametry byly nechány na hodnotě 1,193 a počet sousedů na 3. Bylo spuštěno 100 běhů algoritmu s různými počátečními strukturami a omezením počtu kroků na 100 000. Na konci běhu byla provedena lokální optimalizace metodou nejmenších čtvrců.

Stejný postup byl proveden i pro algoritmus paralelního temperování, abychom mohli algoritmy porovnat a statisticky vyhodnotit.

Statistické zpracování výsledků a porovnání s algoritmem paralelního temperování

Optimalizaci struktury jsme provedli stokrát s cílem získat statisticky relevantní výsledky. Tyto výsledky jsou prezentovány v tabulce 5.2. Analýza tabulky naznačuje, že PSO dokázal nalézt strukturu s vyšší shodou v porovnání s algoritmem PT přibližně za stejný časový úsek, avšak v několika případech nevyřešil krystalovou strukturu $PbSO_4$ (v těchto případech měl PSO hodnotu

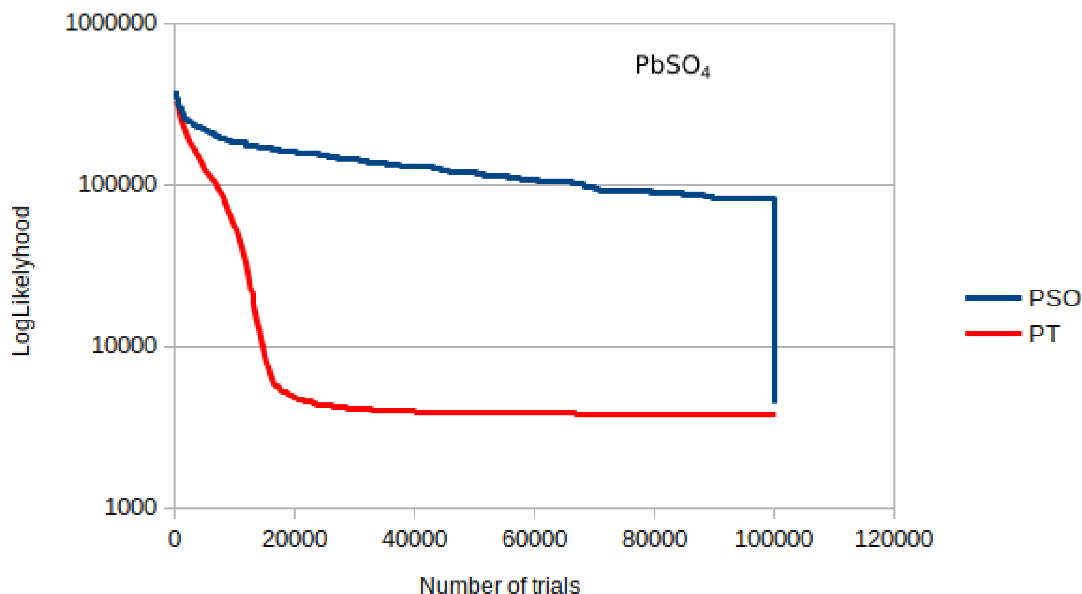


Obrázek 5.2: Neutronové práškové difrakční záznamy $PbSO_4$, černě naměřený, červeně napočítaný, zelená linka odpovídá rozdílu obou záznamů.

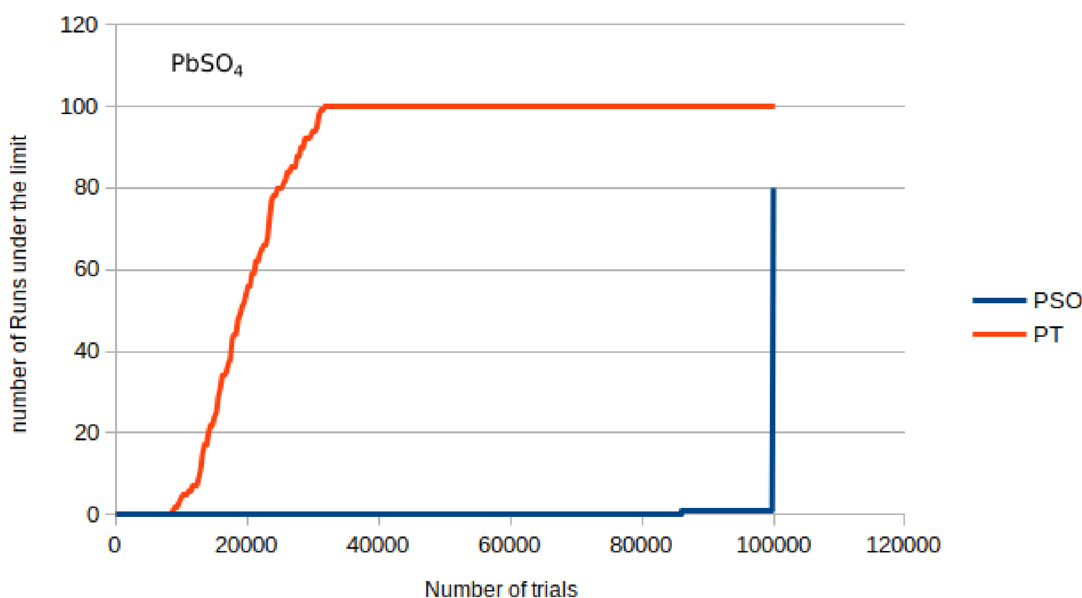
Tabulka 5.2: Srovnání algoritmů optimalizace hejnem částic a paralelního temperování pro řešení krystalové struktury $PbSO_4$. PT vyřešil strukturu pokaždé úspěšně, zatímco PSO dvacetkrát strukturu vůbec nevyřešil.

Algoritmus	Nejlepší hodnota LLK	Průměrná doba řešení	Počet správných struktur
PSO	3652,06	10,2 s	80/100
PT	3757,92	10,4 s	100/100

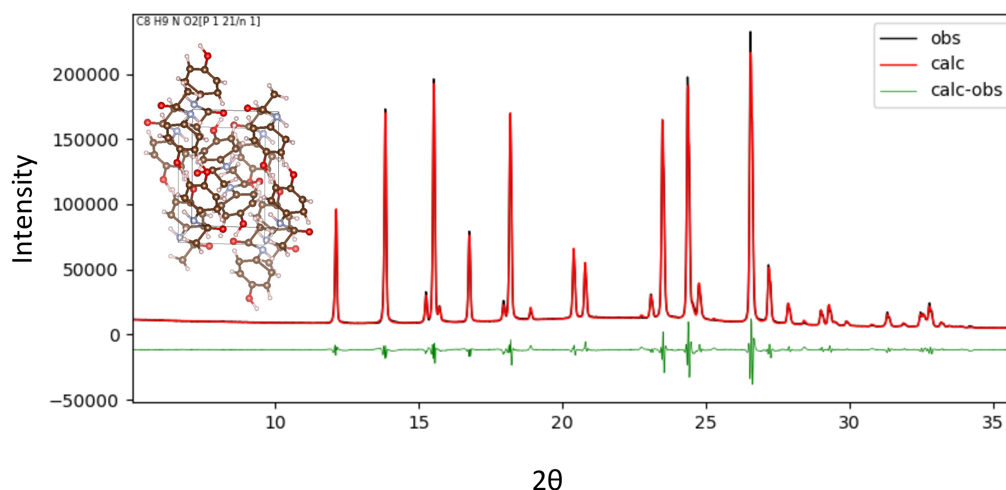
LLK řádově vyšší). Statisticky byl algoritmus paralelního temperování pro tento systém úspěšnější, neboť poskytoval konzistentnější výsledky než PSO (standardní odchylka LLK $\sigma_{PT} = 45$ ve srovnání s $\sigma_{PSO} = 5155$). Předpokládáme, že lepší výsledky pro PT mohou být způsobeny tím, že program FOX byl od počátku testován na struktuře $PbSO_4$ a tudíž by již měl všechny relevantní chyby odstraněny.



Obrázek 5.3: Vývoj mediánu hodnoty aktuálního minima hodnoty LLK při řešení struktury $PbSO_4$ v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsoben první lokální optimalizací. Medián pro PSO je přibližně o 1000 vyšší než pro PT, což není příliš významný rozdíl při hodnotách LLK okolo 4000, odpovídá to přibližně desetinám procenta v R_w.



Obrázek 5.4: Počet správně vyřešených struktur (LLK nižší než 5 000) při řešení struktury $PbSO_4$ v závislosti na počtu výpočtů funkce. Počet správně vyřešených struktur je u PSO nižší než u PT.



Obrázek 5.5: Vyřešená struktura paracetamolu a její odpovídající naměřený a vypočtený rentgenový práškový difrakční záznam.

5.2 Paracetamol

Paracetamol [*N-(4-hydroxyfenyl)acetamid*] je důležitým a široce používaným léčivem díky svým antipyretickým a analgetickým účinkům [44]. Tato organická molekula tvoří několik stabilních krystalických polymorfů, z nichž nejstabilnější fromou je monoklinická forma s prostorovou grupou *P21/n* a parametry mřížky 7,0941 Å; 9,2322 Å, 11,6196 Å a 97,821° [45]. Tato forma se také stala objektem našeho výzkumu.

Experimentální data byla získána z práce [16], k jejichž měření byl použit RTG Práškový difraktometr Empyrean od firmy PANalytical s anodou z mědi, tj. rentgenovým zářením s $K_{\alpha 1} \sim 1.5405980$ Å; napětí na rentgence bylo 45 kV a protékající proud 40 mA; práškový vzorek byl umístěn v kapiláře; bylo použito fokusující eliptické zrcadlo; rentgenový paprsek byl paralelizován pomocí Sollerových clon s maximální rozbíhavostí paprsku 0,02 rad a dopadající rentgenové záření bylo registrováno detektorem PIXcel-3D detector. Bylo provedeno opakované měření tedy 20 měření po 1,05 hodinách. Naměřená data byla nakonec sečtena a výsledný naměřený práškový difrakční záznam je zobrazený na obrázku 5.5 černě.

Řešení krystalové struktury

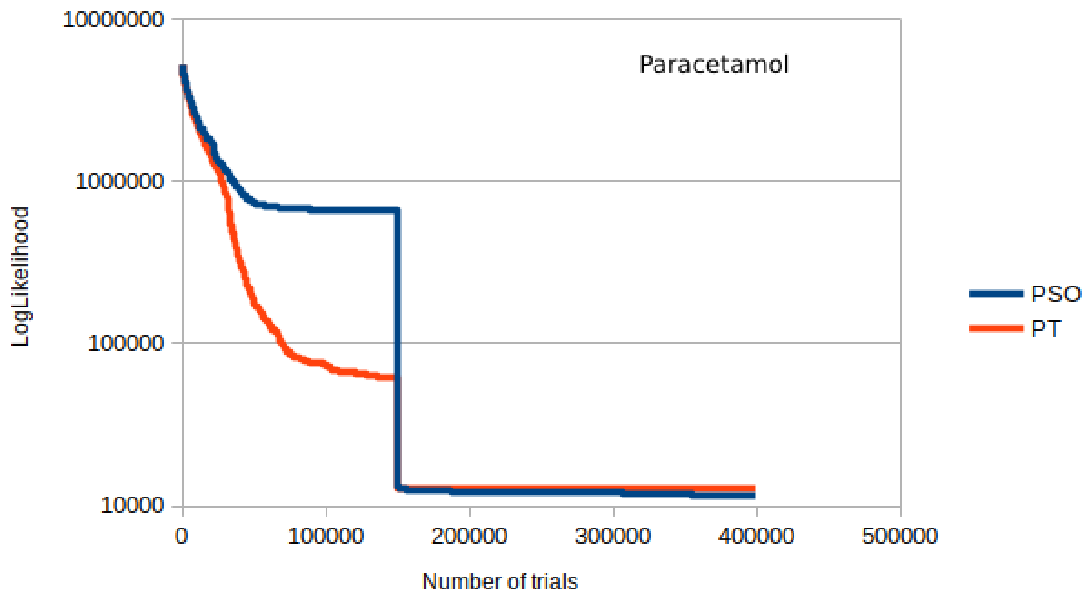
Postup řešení krystalové struktury paracetamolu se skládal z několika kroků, z nichž pro tuto práci nejvýznamější krok, tj. globální optimalizace, byl proveden až na konci.

Do programu byla načtena experimentální data a nastavena vlnová délka rentgenového záření. Difrakční záznam byl automaticky prohledán a tím byly identifikována difrakční maxima, z těchto bylo zvoleno prvních 20. Tato difrakční maxima byla oindexována pomocí programu DICVOL [46], který je součástí programu FOX. Program správně určil krystalovou mřížku struktury, tedy její parametry, a potvrdil, že je monoklinická.

Byl vytvořen krystal o určených parametrech a byly provedeny tři LeBailovy fity, aby byly nalezeny parametry vztahující se k experimentálním datům [47]. Po této rafinaci byl residuální faktor R_w přibližně kolem 5,3 % (tato hodnota se mírně lišila při různých spouštěních programu). Následně byla hledána prostorová grupa – program prohledal všechny monoklinické prostorové grupy a určil jako prostorovou grupu $P21/n$, což odpovídá očekávání.

Teprve v této fázi byla do krystalu umístěna molekula paracetamolu. Návrh molekuly pochází z geometrické optimalizace náčrtu pomocí molekulární mechaniky modulu Forcite programu MaterialStudio (BIOVIA firmy ©2018 Dassault Systèmes). Byla vypnuta možnost dynamické okupance a proveden fit škálovacího parametru.

Krok globální optimalizace se sestavoval z nastavení parametrů algoritmu – počet částic 100, parametr setrvačnost 0,6 a sociální parametry na 1,193 a nastavení, že každých 150 000 výpočtů funkce se provede lokální optimalizace pomocí metody nejmenších čtverců. Algoritmus optimalizace



Obrázek 5.6: Vývoj mediánu hodnoty aktuálního minima hodnoty LLK při řešení struktury paracetamolu v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsoben první lokální optimalizací. Medián pro PSO je přibližně o 1000 nižší než pro PT, což není příliš významný rozdíl při hodnotách LLK okolo 12000, odpovídá to přibližně desetinám procenta v R_w.

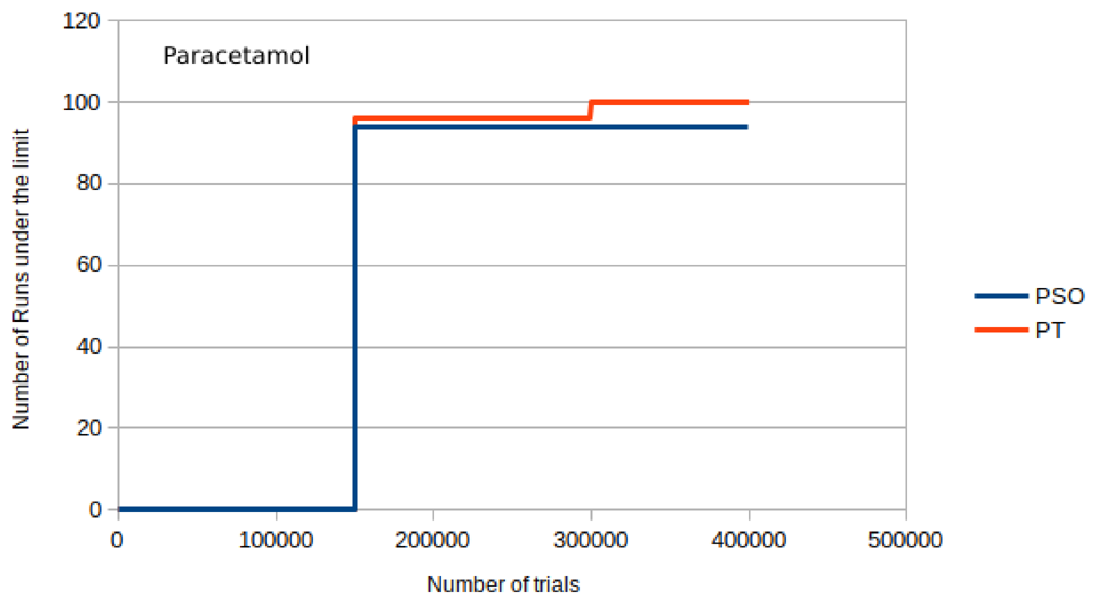
hejnem částic byl spuštěn stokrát z různých výchozích poloh s kritériem konvergence provedení 400 000 výpočtů funkce, z důvodu zisku statistického výsledku. Pro možnost porovnání algoritmů PSO a paralelního temperování jsme stejný postup provedli i s tímto algoritmem.

Statistické zpracování výsledků a porovnání s algoritmem paralelního temperování

Pro získání statistického výsledku byla provedena globální optimalizace stokrát. Statistické výsledky jsou zpracovány na obrázcích 5.6, 5.7 a 5.8 a v tabulce 5.3. Na obrázku 5.6 je vykresleno, jak klesá medián hodnoty optimalizované funkce. Oba algoritmy z počátku optimalizují postupně, avšak PT po cca 50 000 výpočtech funkce pokračuje v nacházení nižších hodnot, zatímco PSO již vykazuje jistou konvergenci hodnot LLK. Při dosažení 150 000 kroku u obou algoritmů došlo k významnému skoku k nízkým hodnotám, přičemž u PSO je skok výraznější. Algoritmy při tomto skoku obvykle dosáhly minima (u PSO 94 ze 100 běhů, u PT 96 ze 100 běhů). u PSO se hodnota mediánu následně ještě o trochu snižovala a u PT zůstávala již prakticky konstantní. PSO dosahovalo řádově o nízké jednotky tisíců nižší optimální hodnoty hodnot než PT, ale také 6 běhů nedosáhlo globálního minima, tedy nevyřešilo krystalickou strukturu. Nižší optimální hodnoty PSO než PT lze vysvětlit tím, že se mnoho částic seběhlo kolem minima a lépe prohledaly tuto oblast prostoru (u PT prohledává oblast jen několik agentů, zatímco u PSO se postupně seběhne celé hejno).

Na obrázku 5.8 je vykreslena hodnota LLK pro jednu částici (náhodně jsme zvolili částici 26) pro jeden z běhů. Je vidět, že částice do první lokální optimalizace prohledává prostor globálně, tj. má vysoké hodnoty LLK a nízké hodnoty trefuje zřídkakdy. Při první lokální optimalizaci zřejmě nebyla tato částice zvolena jako ta, která je lokálně optimalizovaná (toto dokládá větší šířka čáry v grafu okolo 150 000 výpočtu funkce), ale rychle klesla za některou z lokálně zoptimalizovaných částic. Okolo výpočtu 300 000 funkce na čas zvýšila svou hodnotu LLK, zřejmě proto, že se změnily hodnoty nejlepší částice a částice 26 se přesouvala k částici s nižší hodnotou přes oblast, kde je LLK vyšší. Důkazem této hypotézy může být i to, že po překonání této oblasti měla částice lehce nižší aktuální hodnotu.

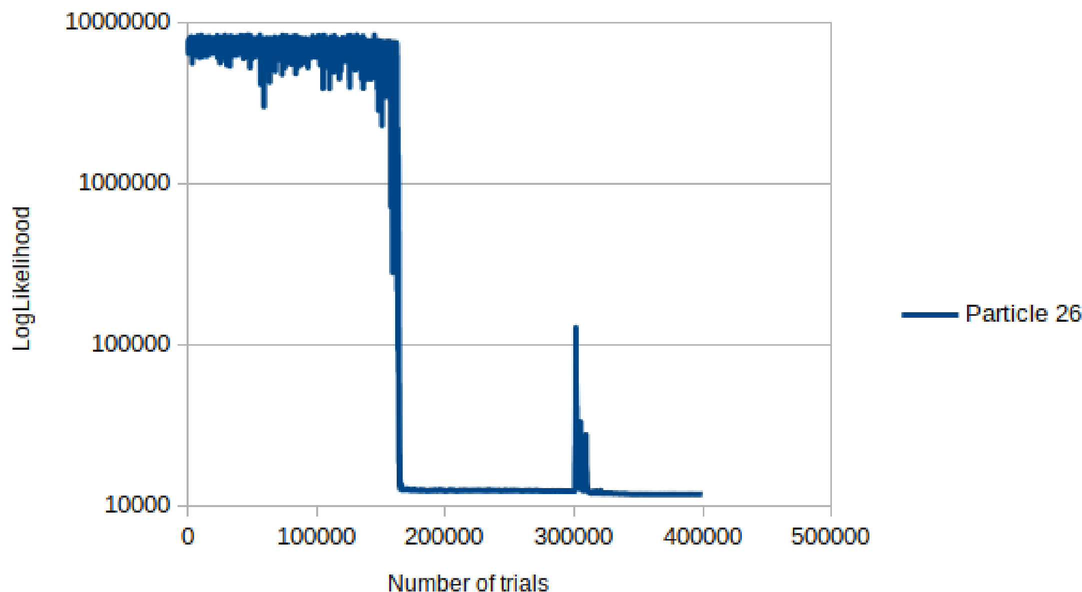
Dalším pozorováním, které lze provést při porovnávání algoritmů PSO a PT je, že PSO potřebuje méně času na provedení stejného počtu výpočtů. U paracetamolu je tento čas v průměru zanedbatelný (liší se o 3,3 s, tj. přibližně 7 %), ale u složitější látky – sofosbuviru, viz sekce 5.3, je rozdíl významnější.



Obrázek 5.7: Vývoj počtu správně určených struktur (LLK < 15 000) při řešení struktury paracetamolu v závislosti na počtu výpočtů funkce. Skok u 150 000 pokusu je způsobena první lokální optimalizací. PSO určilo všechny své struktury (94/100) při první lokální optimalizaci, PT při první lokální optimalizaci určilo větší procento správných struktur a navíc při druhé optimalizaci určil všechny správné struktury.

Tabulka 5.3: Srovnání výsledků pro aloritmy PSO a PT při řešení struktury paracetamolu. PT vyřešil strukturu vždy, zatímco PSO v 6 případech nevyřešil strukturu. PSO dosahovalo nižších hodnot LLK, ale výsledky byly méně konzistentní než u PT.

Algoritmus	Nejlepší hodnota LLK	Průměrná doba řešení	Počet správných struktur
PSO	10403,1	45,20 s	94/100
PT	12064,9	48,47 s	100/100



Obrázek 5.8: Vývoj hodnoty optimalizované funkce částice číslo 26 v závislosti na iteraci při řešení struktury paracetamolu. Do první lokální optimalizace měla vysokou hodnotu, ale trefovala i relevantně nízké hodnoty, po lokální optimalizaci se částice pohybovala v okolí minima.

5.3 Sofosbuvir

Sofosbuvir je označení pro *N-[[P(S),2'R]- 2'-Deoxy- 2'-fluoro-2-methyl-P-phenyl-5'-uridylyl]-l-alanine 1-methylethyl ester*. Strukturu této organické molekuly si lze prohlédnout na obrázku 5.9. Sofosbuvir se používá jako léčivo na hepatitidu typu C, protože molekula inhibuje NS5B polymerázu, která je zásadní při replikaci viru hepatitidy [48]. Sofosbuvir je látkou, která prokázala schopnost krystalizace, což otevírá možnost výroby tabletových forem [49].

Existují dvě různé stabilní krystalické formy sofosbuviru – ortorombická a monoklinická [50]. V této práci se zabýváme ortorombickou strukturou, protože nám byla poskytnuta data této struktury. Základní parametry struktury jsou: prostorová grupa *P 21 21 21* a parametry krystalové struktury 28,0131 Å; 17,0659 Å a 5,2519 Å.

Řešení krystalové struktury

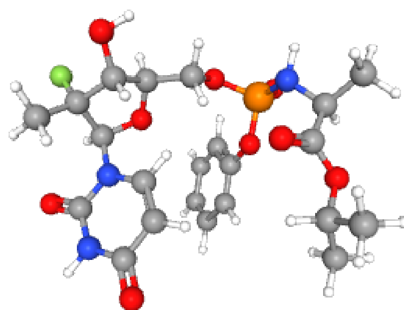
Při řešení krystalové struktury sofosbuviru byl postup analogický řešení struktury paracetamolu (sekce 5.2), tj. použití algoritmu globální optimalizace proběhne až na konci postupu.

Byla načtena experimentální data ve formátu (úhel 2θ, intenzita), nastavená vlnová délka odpovídající rentgence s měděnou anodou ($K_{\alpha 1} \sim 1{,}5405980$ Å). Naměřená data si lze prohlédnout na obrázku 5.10 černě. V difrakčním záznamu byla automaticky identifikována difrakční maxima, která byla následně oindexována programem DICVOL.

Program správně určil, že difrakční záznam patří ortorombické struktuře a určil parametry struktury jako 5,25 Å, 17,04 Å a 27,96 Å, které jsme uvažovali, že je dostatečně blízko skutečným hodnotám. Následně byly provedeny LeBaiovy rafinace, kdy se postupně uvolňovaly parametry pozadí, Cagliotiho parametr W, nulový posun, zbylé Cagliotiho parametry, pseudo-Voightův parametr, parametry asymetrie difrakčního maxima a anisotropní parametry a posunutí vzorku.

Před zahájením hledání prostorové grupy byl omezen rozsah difrakčního záznamu na $\frac{\sin\theta}{\lambda} = 0,2$. Byly prozkoumány všechny triklinické, monoklinické a ortorombické prostorové grupy a program FOX zvolil jako nejlepší prostorovou grupu *P m n 21*, avšak grupu *P 21 21 21* označil také mezi vhodné kandidáty. Pro větší obecnost řešení by bylo vhodné zkusit najít řešení pro všechny prostorové grupy s dobrou hodnotou míry shody, protože však toto nebylo předmětem této práce, nastavili jsme grupu na tu správnou a pokračovali v řešení krystalové struktury sofosbuviru.

Molekula sofosbuviru byla získána z [51] ve formátu sdf a převedena na Fenskeho-Hallovu matici pomocí on-line konvertoru [52]. Molekulu v tomto formátu již bylo možné načíst do programu FOX a vložit ji do krystalu. Byla vypnuta dynamická okupance, protože u takto komplexní



Obrázek 5.9: Program FOX nalezl pro molekulu sofosbuviru v krystalové mřížce celkem 202 volných parametrů. Obrázek byl získán z [51].

molekuly lze předpokládat, že molekuly nebudou ve speciálních polohách.

Globální optimalizace struktury probíhala na funkci mezi dvěma objekty – krystalem (tj. napočítaným difrakčním záznamem) a experimentálními daty (tj. naměřeným difrakčním záznamem). Bylo znemožněno optimalizovat parametry pozadí difrakčního záznamu z důvodu, že by mohl program FOX umožnit rafinaci těchto parametrů. Byl vypnut extrakční mód a proběhl fit škálovacího parametru. Parametry PSO byly nastaveny na 100 částic, parametr setrvačnosti byl nastaven na 0,721, sociální parametr byl nastaven na 1,193 a parametr sousedství byl nastaven na 3. Byla vybrána možnost automatické lokální optimalizace po 150 000 výpočtech pomocí metody nejmenších čtverců. Algoritmus měl k nalezení struktury k dispozici 1 000 000 výpočtů funkce LLK. Algoritmus byl spuštěn stokrát z různých výchozích pozic, pro získání statistického výsledku. Výsledek pro jeden běh si lze prohlédnout na obrázku 5.10, na kterém je patrné, že difrakční záznamy si odpovídají ($R_w = 7,13\%$).

Stejný postup byl proveden i pro algoritmus paralelního temperování, aby bylo možné tyto algoritmy vzájemně porovnat.

Statistické zpracování výsledků a porovnání s algoritmem paralelního temperování

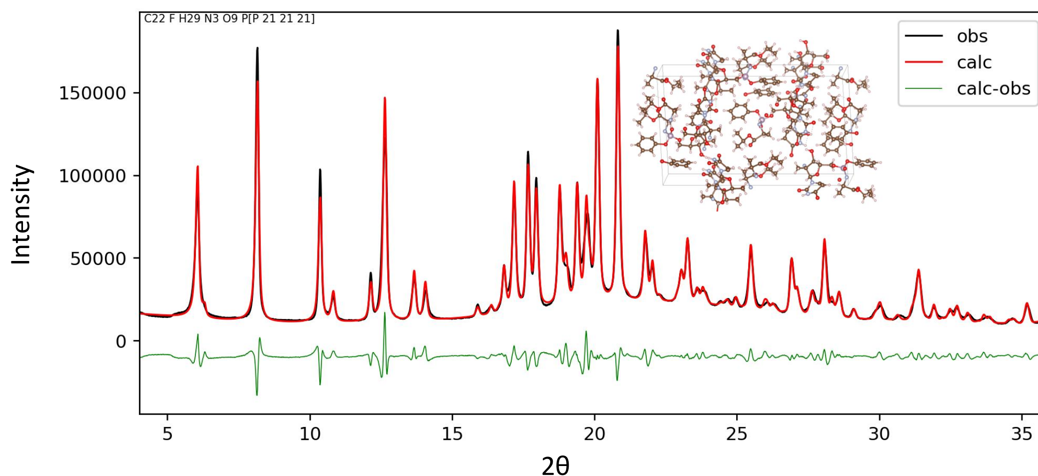
Pro získání statistických výsledků a umožnění porovnání algoritmů PSO a PT byla provedena globální optimalizace stokrát z různých výchozích poloh pro každý z algoritmů. Výsledky jsou prezentovány v obrázcích 5.11, 5.12 a 5.13, a jsou shrnuty v tabulce 5.4.

Na obrázku 5.11 jsou porovnány mediány nalezených hodnot PSO a PT v závislosti na počtu provedených výpočtů. PSO dosahuje lepších minim za nižší počet výpočtů funkce. Teoreticky by pro PSO pro sofosbuvir stačilo provést pouze 50 000 výpočtů funkce a následně provést lokální optimalizaci. To je z důvodu, že do přibližně 50 000 výpočtů dochází k prudkému poklesu hodnoty funkce, který se dále nezmění až do lokální optimalizace při kroku 150 000. Následné změny byly jen marginální v rámci následujících lokálních optimalizací.

V případě PT bylo nutné provést více lokálních optimalizací, avšak ani ty nepomohly algoritmu dosáhnout tak nízkých hodnot LLK jako PSO (rozdíl je v řádu desítek tisíc LLK, což odpovídá přibližně rozdílu $\Delta R_w = 0,5\,\%$). Výhodou algoritmu PT je, že konverguje pomaleji, a proto by měl pravděpodobně v případě složitějších struktur vyšší pravděpodobnost nalezení správné struktury.

Obrázek 5.12 ilustruje počet běhů, které dosáhly hodnoty LLK nižší než 100 000. PSO úspěšně identifikoval všechny struktury již při první lokální optimalizaci, přičemž většina běhů dosáhla LLK kolem 60 000. Naopak u paralelního temperování docházelo k postupnému nalezení struktur s nízkým LLK během jednotlivých lokálních optimalizací. Při první lokální optimalizaci dosáhlo pouze 22 z 100 běhů hodnoty LLK pod 100 000. Tento výsledek koresponduje s grafem 5.11.

Pro sofosbuvir byla také analyzována závislost hodnoty funkce na počtu výpočtů pro jednu částici, viz obrázek 5.13. Tato závislost je prezentována pro dva běhy, konkrétně běh 19 a běh 21, protože zde se chování částic liší. V běhu 19 částice dosáhla minima během první lokální optimalizace, zatímco v běhu 21 se částice během celého průběhu nedostala k minimu. Výhodou



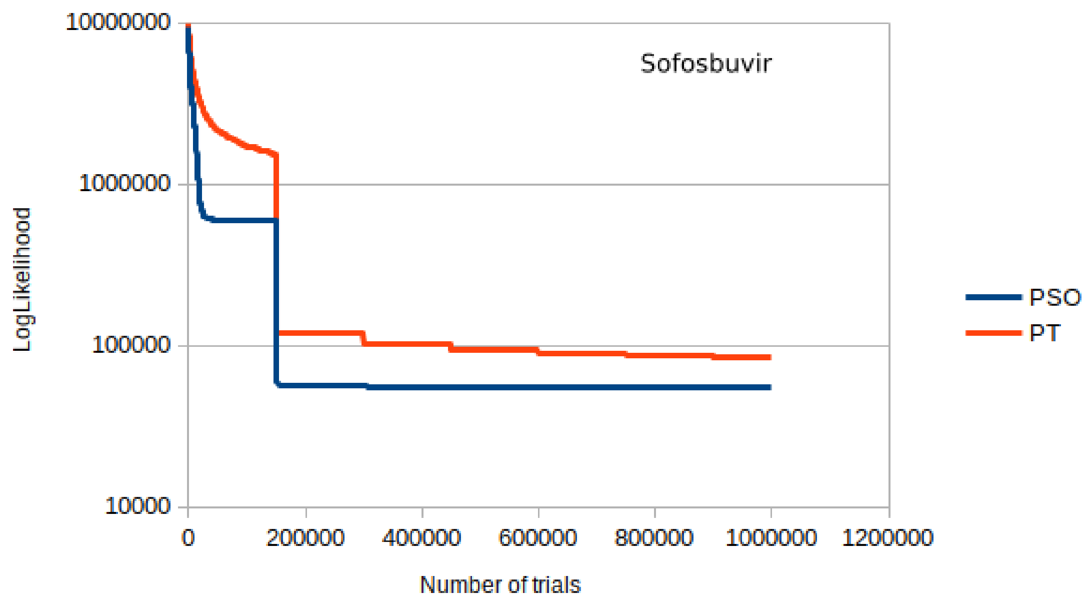
Obrázek 5.10: Difrakční záznam a krystalová struktura sofosbuviru. Černá čára odpovídá naměřeným hodnotám, červená čára odpovídá vypočtenému difrakčnímu záznamu a zelená je rozdíl mezi oběma předchozími. Difrakční záznamy si odpovídají, což potvrzuje i $R_w = 7,13\%$

Tabulka 5.4: Srovnání výsledků pro algoritmy PSO a PT. Pro sofosbuvir dosáhl algoritmus PSO nižší nejlepší hodnoty za rychlejší čas a pro všech sto běhů.

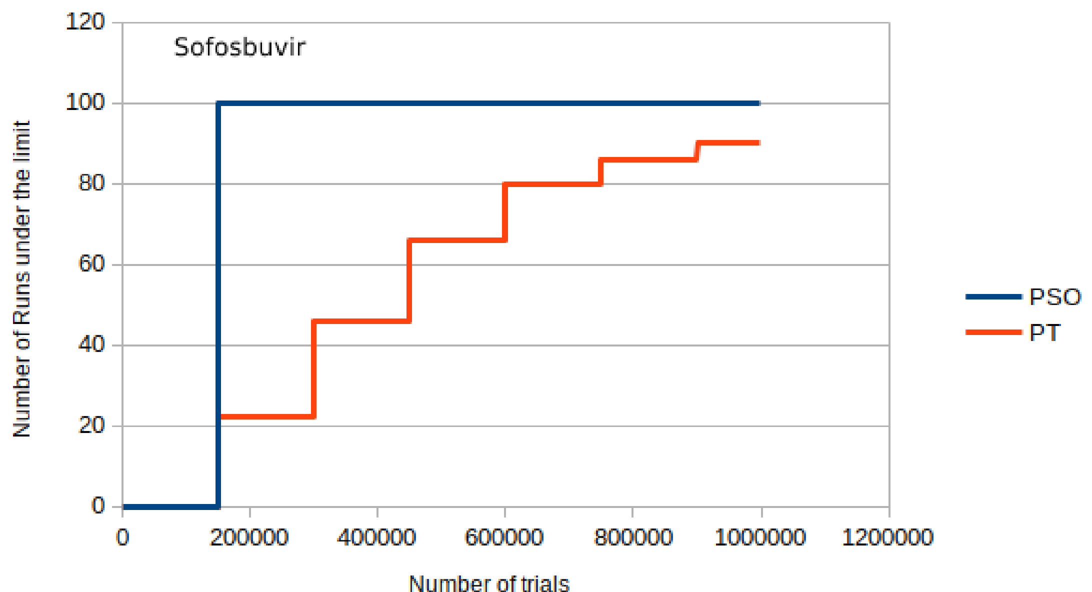
Algoritmus	Nejlepší hodnota LLK	Doba řešení	Počet správných struktur
PSO	47928,6	789,0 s	100/100
PT	60804,4	1300,7 s	90/100

nekonvergence částice v běhu 21 může být skutečnost, že pravděpodobně prozkoumávala prostor globálně po celou dobu. Naopak nevýhodou může být, že výpočty její funkce v závěru algoritmu byly již irelevantní.

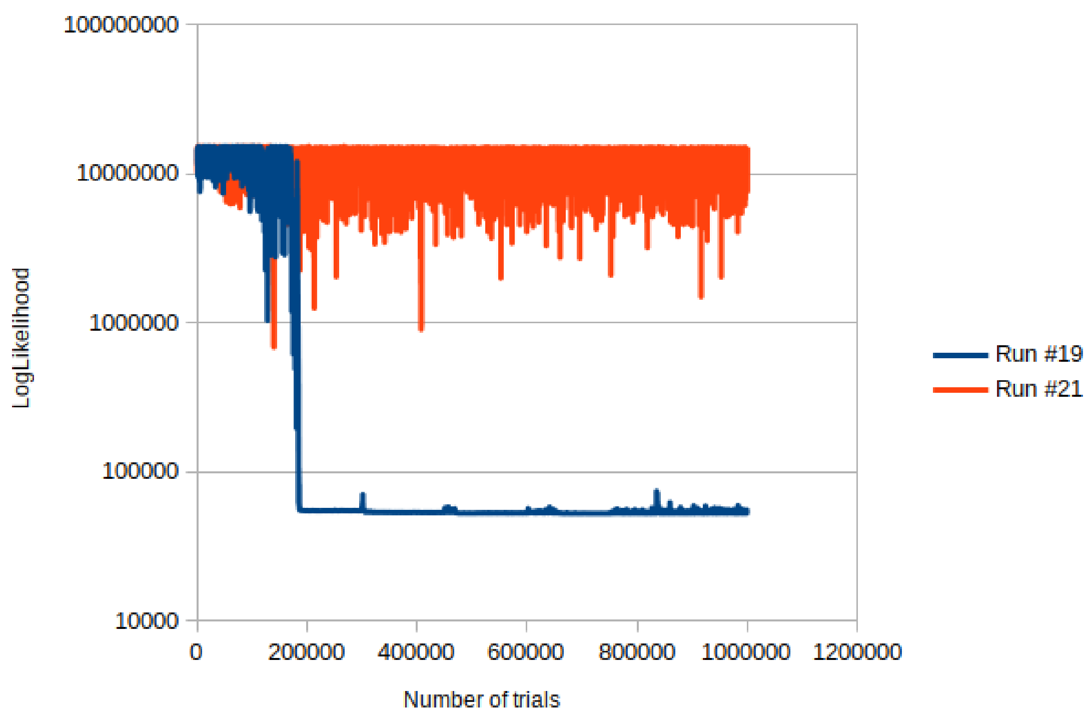
Ze srovnání PSO a PT v tabulce 5.4 plyne, že PSO našel krystalovou strukturu sofosbuviru, která má vypočítaný difrakční záznam podobnější naměřenému difrakčnímu záznamu za dobu odpovídající přibližně dvěma třetinám času, které potřeboval algoritmus paralelního temperování, a navíc se mu to podařilo ve sto případech ze sta běhů.



Obrázek 5.11: Srovnání mediánu hodnoty aktuálního minima PSO a PT v závislosti na počtu výpočtů funkce pro strukturu sofosbuviru. Skok u 150 000 výpočtu je způsoben první lokální optimalizací. PSO nachází během prvních 150 000 výpočtů výrazně lepší hodnotu LLK než PT a cca od 50 000 výpočtu už zůstává hodnota stejná. Z této lepší polohy PSO při lokální optimalizaci přeskočí do výhodnějšího bodu než při PT a proto PSO nachází ve finále nižší hodnoty LLK než PT. Výhoda PT je, že i při dalších lokálních optimalizacích dochází k nalezení výhodnějších poloh, tj. má lepší konvergenční vlastnosti.



Obrázek 5.12: Vývoj počtu správně určených struktur (LLK<100 000) v závislosti na počtu výpočtů funkce pro strukturu sofosbuviru. Skok u 150 000 pokusů je způsoben první lokální optimalizací. PSO určilo všechny své struktury (100/100) při první lokální optimalizaci. PT jich při první lokální optimalizaci určilo jen část (22/100), zato další lokální optimalizace nacházely další správná řešení. Celkově PT nalezlo správnou strukturu v 90 pokusech ze 100.



Obrázek 5.13: Vývoj hodnoty funkce LLK pro náhodně zvolenou částici 26 pro dva různé běhy pro strukturu sofosbuviru. Pro běh 19 částice kolem 150 000 výpočtu funkce sklouzla do lokálního minima a setrvala zde až do konce jen s malými výjimkami kolem dalších lokálních optimalizací. Pro běh 21 částice do minima nesklouzla a prohledávala po celou dobu globálně.

Kapitola 6

Vliv změn parametrů algoritmu optimalizace hejnem částic

Pro vhodné nastavení parametrů algoritmu PSO je nutné mít kriteria, podle kterých lze hodnotit vlastnosti parametrů. Obecně je žádoucí, aby algoritmus za co nejkratší čas získal co nejnižší hodnotu optimalizované funkce bez ohledu, na které látce je optimalizace prováděna. Proto je v první části diskutován výběr kritérií, ze kterých bude zřejmé, jak se algoritmus chová, jestli nalezl globální minimum a zda vůbec zkonvergoval. V další části je zkoumán vliv jednotlivých parametrů algoritmu na jeho konvergenci a úspěšnost hledání globálního minima.

6.1 Konvergence algoritmu

Rozpoznání dosažení skutečného globálního minima optimalizované funkce je otázkou, která byla v rámci práce zkoumána. Navíc z definice stochastického algoritmu plyne, že nemusí být vždy dosaženo skutečného globálního minima, ale pouze lokálního minima, proto konvergencí algoritmu zde budeme nazývat stav, kdy už nám jeho další vývoj nepřináší nové relevantní informace o funkci prohledávaného prostoru.

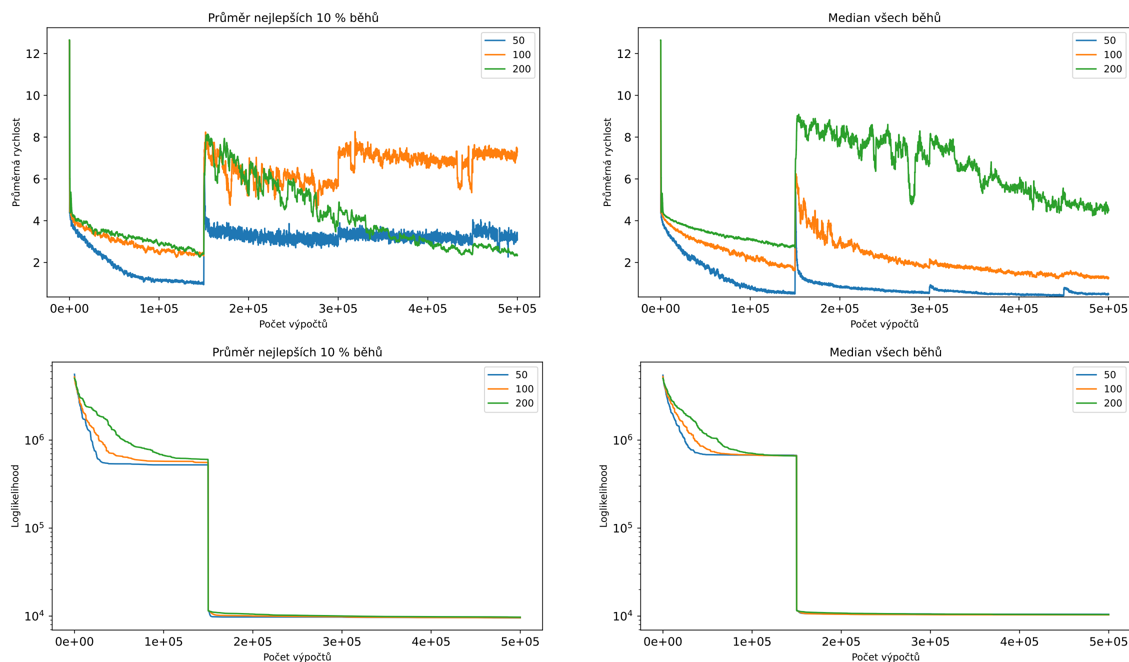
Bylo zkoumáno několik kritérií konvergence a z nich následně vybráno to, které nejlépe splňovalo nastavené požadavky: lineární závislost složitosti kritéria na počtu částic, nezávislost na systému a univerzální použití při různém nastavení parametrů algoritmu.

Mezi základní konvergenční testy PSO lze zařadit:

- Překročení limitní hodnoty – poté, co se algoritmus dosáhne pod uživatelem nastavenou hodnotu optimalizované funkce je ukončen.
- Částice zkonvergovaly do jednoho bodu – vzdálenost mezi každýma dvěma částicemi není vyšší než stanovená hodnota.
- Ustal pohyb částic – suma absolutních hodnot rychlostí částic je nižší než nastavený limit.
- Nedochází ke snižování hodnoty hledaného minima – po určitý počet iterací algoritmu nedochází k vylepšení hodnoty globálního minima.

Ani jedno z výše uvedených kriterií nesplňuje všechny nastavené požadavky kritéria konvergence algoritmu (limitní hodnota optimalizované funkce není univerzální pro všechny systémy, konvergence částic do jednoho bodu má kvadratickou závislost na počtu částic, suma absolutních hodnot rychlostí částic roste s počtem částic a počet iterací, kdy se nemění hodnota globálního minima, není nezávislý na konkrétním systému). Proto bylo navrženo kritérium konvergence, které obchází některé nevýhody těchto základních konvergenčních testů: nejprve je testován počet iterací, po které se nezměnila hodnota globálního minima, tento test má konstantní závislost na počtu částic a lze jej snadno vyhodnotit; druhým testem je ověřit, zda hodnota optimalizované funkce není nižší než hodnota nastavená uživatelem (pokud ji uživatel nenastaví, toto kritérium je ignorováno); posledním testem je kontrola, zda ustal pohyb částic – aby bylo kritérium zbaveno závislosti na počtu částic je hodnocena průměrná rychlost všech částic, zda je nižší než stanovený limit.

Zkoumáním průměrné rychlosti částic lze odhadnout, zda částice prohledávají prostor lokálně či globálně. Pokud bude průměrná rychlost částic vysoká, bude částice skákat daleko, takže bude



Obrázek 6.1: Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro 50, 100 a 200 částic.

prohledávat globálně. Pokud bude nízká, tak lze předpokládat, že se nachází poblíž minima svého sousedství a prohledává tedy lokálně.

6.2 Vliv změn parametrů

Změny chování algoritmu jsme zkoumali na struktuře paracetamolu, kde jsme omezili počet výpočtů srovnání difrakčních záznamů na 500 000. Pro každé nastavení parametrů jsme provedli 100 běhů algoritmu, za účelem získání stochastických výsledků. Jako výchozí a srovnávací chování bylo považováno nastavení parametrů podle [26], kde byl změněn počet částic na 100, ostatní parametry tedy byly: parametr setrvačnosti 0,721; sociální parametry 1,193 a počet sousedů 3. Při každém 150 000 výpočtu objektivní funkce byla provedena lokální optimalizace pomocí metody nejmeněích čtverců, která v grafech způsobuje výrazné skoky v násobcích této hodnoty.

Počet částic

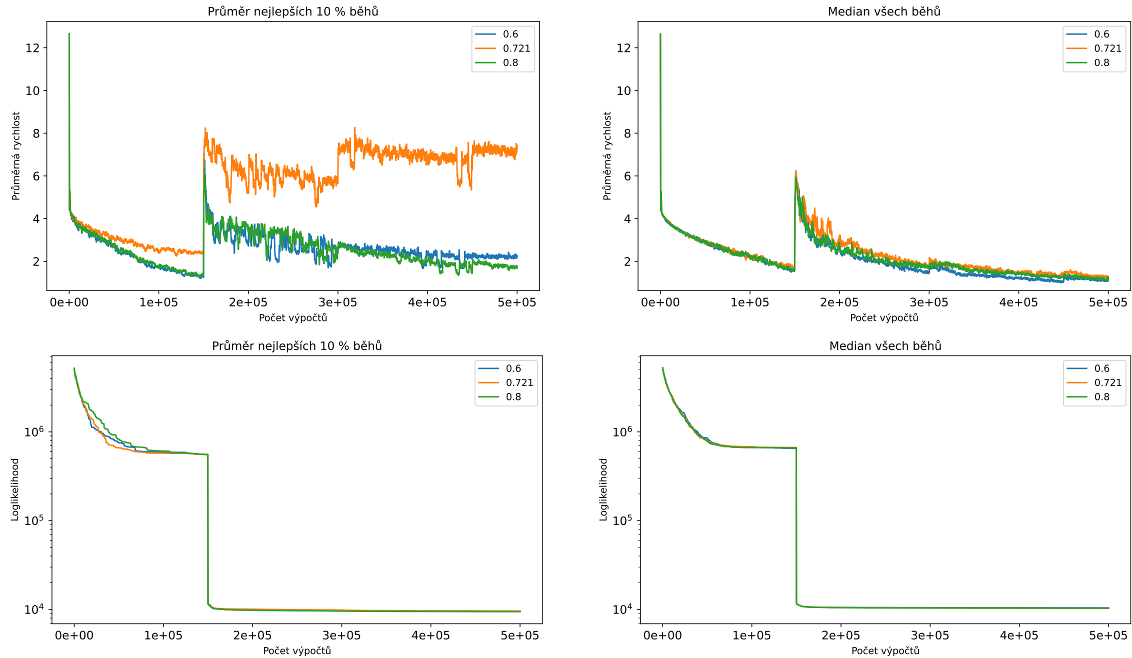
Počet částic jsme zkoumali pro hodnoty 50, 100 a 200 a výsledné statistické hodnoty jsou vyneseny v grafech 6.1. Z grafů je patrné, že pro 50 částic LLK konverguje rychleji, než pro 200 částic. To je způsobeno tím, že méně částic snáze zkonverguje do jednoho bodu (minima). Z grafu rychlostí je patrné, že medián rychlostí částic je pro 200 částic vyšší než pro 50 částic. To je způsobeno tím, že změny sousedství umožňují volbu mezi více částicemi, a tudíž umožňují získat větší rychlost výběrem vzdálenější částice. Pro 100 částic je chování algoritmu mezi těmito dvěma extrémy.

Výrazný je také útlum průměrné rychlosti pro 50 částic, který je způsoben tím, že částice zkonvergovaly do jednoho bodu a již se nepohybují. Lokální optimalizace vyvedla tyto zkonvergované částice z minima a nejúspěšnější běhy začly prohledávat více globálně, tj. částice získaly vyšší rychlost. Analogicky to platí pro 100 částic. Pro 200 částic je rychlost sice vyšší, ale globální prohledávání neumožňuje běhům nalezení lepšího minima, proto u úspěšných běhů naopak dochází k poklesu rychlosti a tendenci prohledávat více lokálně.

Parametr setrvačnosti

Parametr setrvačnosti jsme zkoumali pro hodnoty 0,6; 0,721 a 0,8 a výsledné statistické hodnoty jsou vyneseny v grafech 6.2. Grafy mediánů průměrných rychlosí a objektivní funkce LLK jsou téměř totožné, proto z nich nelze mnoho usuzovat o závislosti chování algoritmu na tomto parametru. Z grafů průměrných hodnot 10 % nejlepších běhů je patrné, že pro hodnotu 0,721 bylo

výhodnější snižovat rychlost pomaleji a lokální optimalizaci si udržovat rychlost vyšší, zatímco pro hodnoty 0,6 a 0,8 bylo výhodnější rychlost snižovat rychleji a po lokální optimalizaci ji opět snižovat. To je způsobeno tím, že pro hodnotu 0,721 se částice mohly dostat do lokálního minima a následně se z něj dostat, zatímco pro hodnoty 0,6 a 0,8 se částice do lokálního minima dostaly a následně se z něj již nedostaly.



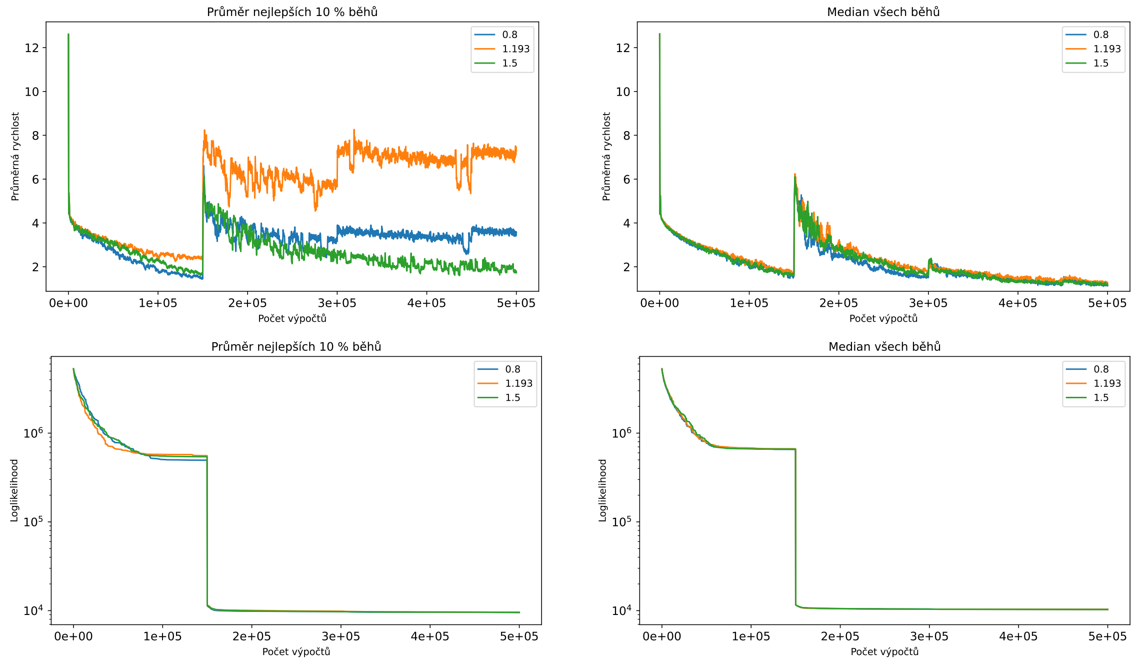
Obrázek 6.2: Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro parametr setrvačnosti 0,6; 0,721 a 0,8

Sociální parametr

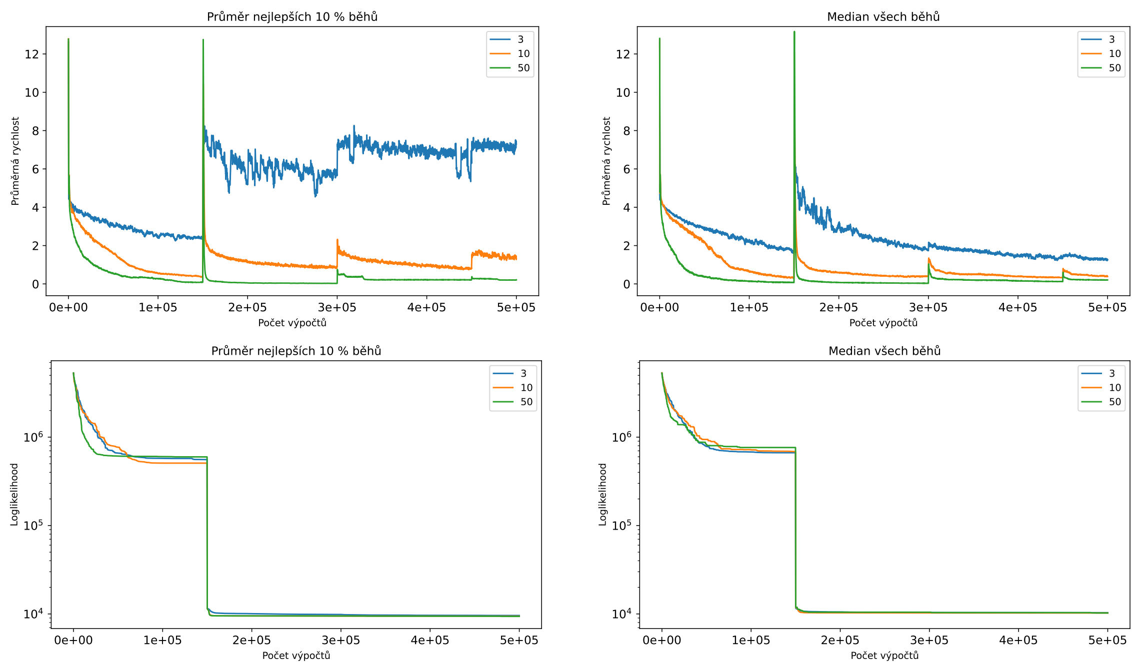
Sociální parametr jsme zkoumali pro hodnoty 0,8; 1,193 a 1,5 a výsledné statistické hodnoty jsou vyneseny v grafech 6.3. Grafy mediánů průměrných rychlosí a objektivní funkce LLK jsou téměř totožné, proto z nich nelze mnoho usuzovat o závislosti chování algoritmu na tomto parametru. Z grafů průměrných hodnot 10 % nejlepších běhů je patrné, že pro hodnotu 1,193 bylo výhodnější snižovat rychlost pomaleji a po lokální optimalizaci si udržovat rychlost vyšší. Před lokální optimalizací se rychlosti snižovaly rychleji pro hodnotu 0,8 než pro hodnotu 1,5. Po lokální optimalizaci bylo pro vyšší hodnotu sociálního parametru výhodnější opět snižovat rychlost, zatímco pro nižší rychlost bylo výhodnější držet si stálou hodnotu.

Parametr sousedství

Parametr sousedství jsme zkoumali pro hodnoty 3, 10 a 50 a výsledné statistické hodnoty jsou vyneseny v grafech 6.4. Z mediánů běhů je patrné, že pro 3 sousedy LLK konverguje nejpomaleji a nejkonzistentněji (nedochází k velkým skokům). To je způsobeno tím, že méně částic je informováno o poloze nejlepší hodnoty. Z průběhu průměru nejlepších 10 % běbů lze vyčíst, že pro 3 sousedy bylo výhodné si udržovat vyšší průměrnou rychlost, zatímco pro 10 a 50 sousedů bylo výhodné držet rychlost nižší. To je způsobeno tím, že pro 3 sousedy se částice mohly dostat do lokálního minima a následně se z něj dostat, zatímco pro 10 a 50 sousedů se částice do lokálního minima dostaly a následně se z něj již nedostaly.



Obrázek 6.3: Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro sociální parametr 0,8; 1,193 a 1,5.



Obrázek 6.4: Průměrné rychlosti částic (nahoře) a hodnoty funkce LLK (dole) pro 3, 10 a 50 sousedů.

Závěr

Byl implementován algoritmus optimalizace hejnem částic do programu FOX, jeho použití je možné jak přes grafické prostředí, tak pomocí skriptů v programovacím jazyku Python. Uživatel má možnost nastavovat si parametry algoritmu podle svého výběru nebo jsou přednastaveny na doporučované hodnoty. Použití algoritmu bylo demonstrováno na třech strukturách – $PbSO_4$, Paracetamol a Sofosbuvir, z nichž byly všechny správně vyřešeny a odpovídají publikovaným strukturám.

Při srovnání stávajícího doporučovaného algoritmu paralelního temperování s algoritmem optimalizace hejnem částic dosahuje PSO lepších hodnot funkce za kratší časový úsek hlavně pro složitější struktury jako je sofosbuviru. Celkový čas řešení byl přibližně o jednu třetinu rychlejší. Do budoucna by bylo vhodné více kontrolovat kvalitativní průběh algoritmu PSO, aby byly výsledky více konzistentní a PSO by nacházelo pouze správné struktury ne ty v lokálních minimech.

Zkoumání parametrů PSO na struktuře Paracetamolu přináší lepší vhled a představu o jednotlivých parametrech algoritmu. V práci jsou uvedeny tendence, jaké změny vedou ke zrychlení nebo zpomalení konvergence algoritmu. Do budoucna lze podle těchto tendencí nastavit chování parametrů v průběhu samotného řešení struktury a dosáhnout ještě lepších výsledků.

Lokální optimalizace v rámci PSO je klíčovým prvkem pro efektivní nalezení minim. Bez ní by bylo pro částice obtížné lokalizovat tato minima. I když jsme tento aspekt v naší práci neprozkoumali do hloubky, budoucí studie by mohly tuto oblast dále zkoumat. To by mohlo vést k rychlejšímu nalezení řešení krystalové struktury, možná i několikanásobně.

Dalšího zrychlení algoritmu by mohlo být dosaženo, pokud by se algoritmus implementoval paralelně, kdy by se jednotlivá jádra starala o vývoj každé z částic, optimální by pravděpodobně bylo algoritmus modifikovat pro GPU.

Bibliografie

1. HABERMEHL, Stefan; SCHLESINGER, Carina; SCHMIDT, Martin U. Structure determination from unindexed powder data from scratch by a global optimization approach using pattern comparison based on cross-correlation functions. *Acta Crystallographica Section B Structural Science, Crystal Engineering and Materials*. 2022, roč. 78, s. 195–213. ISSN 2052-5206. Dostupné z DOI: 10.1107/S2052520622001500.
2. FENG, Zhen Jie; DONG, Cheng; JIA, Rong Rong; DENG, Xiao Di; CAO, Shi Xun; ZHANG, Jin Cang. PeckCryst: a program for structure determination from powder diffraction data using a particle swarm optimization algorithm. *Journal of Applied Crystallography*. 2009, roč. 42, s. 1189–1193. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889809034207.
3. CUOCCI, Corrado; CORRIERO, Nicola; DELL'AERA, Marzia; FALCICCHIO, Aurelia; RIZZI, Rosanna; ALTOMARE, Angela. Direct space approach in action: Challenging structure solution of microcrystalline materials using the EXPO software. *Computational Materials Science*. 2022, roč. 210, s. 111465. ISSN 09270256. Dostupné z DOI: 10.1016/j.commatsci.2022.111465.
4. DAVID, William I. F.; SHANKLAND, Kenneth; STREEK, Jacco van de; PIDCOCK, Elna; MOTHERWELL, W. D. Samuel; COLE, Jason C. DASH: a program for crystal structure determination from powder diffraction data. *Journal of Applied Crystallography*. 2006, roč. 39, s. 910–915. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889806042117.
5. FAVRE-NICOLIN, Vincent; ČERNÝ, Radovan. FOX, 'free objects for crystallography': a modular approach to ab initio structure determination from powder diffraction. *Journal of Applied Crystallography*. 2002, roč. 35, s. 734–743. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889802015236.
6. ALTOMARE, Angela; CUOCCI, Corrado; GIACOVAZZO, Carmelo; MOLITERNI, Anna; RIZZI, Rosanna. EXPO2011: A new package for powder crystallography. *Powder Diffraction*. 2011, roč. 26, S2–S12. ISSN 0885-7156. Dostupné z DOI: 10.1154/1.3660382.
7. PADGETT, Clifford W.; ARMAN, Hadi D.; PENNINGTON, William T. Crystal Structures Elucidated from X-ray Powder Diffraction Data without Prior Indexing. *Crystal Growth and Design*. 2007, roč. 7, s. 367–372. ISSN 1528-7483. Dostupné z DOI: 10.1021/cg0605943.
8. SPILLMAN, Mark J.; SHANKLAND, Kenneth. GALLOP: accelerated molecular crystal structure determination from powder diffraction data. *CrystEngComm*. 2021, roč. 23, s. 6481–6485. ISSN 1466-8033. Dostupné z DOI: 10.1039/D1CE00978H.
9. ALTOMARE, Angela; CUOCCI, Corrado; GIACOVAZZO, Carmelo; MOLITERNI, Anna; RIZZI, Rosanna; CORRIERO, Nicola; FALCICCHIO, Aurelia. EXPO2013: a kit of tools for phasing crystal structures from powder data. *Journal of Applied Crystallography*. 2013, roč. 46, s. 1231–1235. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889813013113.
10. SPILLMAN, Mark J.; SHANKLAND, Kenneth; WILLIAMS, Adrian C.; COLE, Jason C. CDASH: a cloud-enabled program for structure solution from powder diffraction data. *Journal of Applied Crystallography*. 2015, roč. 48, s. 2033–2039. ISSN 1600-5767. Dostupné z DOI: 10.1107/S160057671502049X.
11. FAVRE-NICOLIN, Vincent; ČERNÝ, Radovan. A better FOX: using flexible modelling and maximum likelihood to improve direct-space <i>ab initio</i> structure determination from powder diffraction. *Zeitschrift für Kristallographie - Crystalline Materials*. 2004, roč. 219, s. 847–856. ISSN 2196-7105. Dostupné z DOI: 10.1524/zkri.219.12.847.55869.

12. ČERNÝ, Radovan; FAVRE-NICOLIN, Vincent; ROHLÍČEK, Jan; HUŠÁK, Michal. FOX, Current State and Possibilities. *Crystals*. 2017, roč. 7, s. 322. ISSN 2073-4352. Dostupné z DOI: 10.3390/cryst7100322.
13. ALTOMARE, Angela; BURLA, Maria Cristina; CAMALLI, Mercedes; CARROZZINI, Benedetta; CASCARANO, Giovanni Luca; GIACOVAZZO, Carmelo; GUAGLIARDI, Antonietta; MOLITERNI, Anna Grazia Giuseppina; POLIDORI, Giampiero; RIZZI, Rosanna. EXPO: a program for full powder pattern decomposition and crystal structure solution. *Journal of Applied Crystallography*. 1999, roč. 32, s. 339–340. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889898007729.
14. HABERMEHL, Stefan; MÖRSCHEL, Philipp; EISENBRANDT, Pierre; HAMMER, Sonja M.; SCHMIDT, Martin U. Structure determination from powder data without prior indexing, using a similarity measure based on cross-correlation functions. *Acta Crystallographica Section B Structural Science, Crystal Engineering and Materials*. 2014, roč. 70, s. 347–359. ISSN 2052-5206. Dostupné z DOI: 10.1107/S2052520613033994.
15. FENG, Zhen Jie; DONG, Cheng. GEST: a program for structure determination from powder diffraction data using a genetic algorithm. *Journal of Applied Crystallography*. 2007, roč. 40, s. 583–588. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889807008618.
16. KOČÍ, Milan. Predikce krystalových struktur. 2021.
17. PECHARSKY, Vitalij K.; ZAVALIJ, Peter Y. *Fundamentals of Powder Diffraction and Structural Characterization of Materials*. Second. Springer Science, 2009. ISBN 978-0-387-09578-3.
18. RIETVELD, H M. A profile refinement method for nuclear and magnetic structures. *Journal of Applied Crystallography*. 1969, roč. 2, s. 65–71. Dostupné z DOI: 10.1107/S0021889869006558.
19. FAVRE-NICOLIN, Vincent. *Fox Wiki*. [B.r.]. Dostupné také z: https://fox.vincefn.net/.
20. WOLPERT, David H.; MACREADY, William G. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*. 1997, roč. 1, s. 67–82. ISSN 1089778X. Dostupné z DOI: 10.1109/4235.585893.
21. KENNEDY, J.; EBERHART, R. Particle swarm optimization. In: IEEE, 1995, s. 1942–1948. ISBN 0-7803-2768-3. Dostupné z DOI: 10.1109/ICNN.1995.488968.
22. JAMES, Kennedy; MAURICE, Clerc. *Standard PSO version 2006*. [B.r.]. Dostupné také z: https://www.particleswarm.info/Standard_PSO_2006.c.
23. AUGER, Anne; BLACKWELL, Tim; BRATTON, Dan; CLERC, Maurice; CROUSSETTE, Sylvain; DATTASHARMA, Abhi; EBERHART, Russel; HANSEN, Nikolaus; KEKO, Hrvoje; KENNEDY, James; KROHLING, Renato; LANGDON, William; LI, Wentao; LIU, Hongbo; MIRANDA, Vladimiro; POLI, Riccardo; SERRA, Pablo; STICKEL, Manfred. *Standard PSO version 2007*. [B.r.]. Dostupné také z: https://www.particleswarm.info/standard_pso_2007.zip.
24. AUGER, Anne; BLACKWELL, Tim; BRATTON, Dan; CLERC, Maurice; CROUSSETTE, Sylvain; DATTASHARMA, Abhi; EBERHART, Russel; HANSEN, Nikolaus; HELWIG, Sabine; KEKO, Hrvoje; KENNEDY, James; KROHLING, Renato; LANGDON, William; LI, Wentao; LIU, Hongbo; MIRANDA, Vladimiro; POLI, Riccardo; SERRA, Pablo; SPEARS, William; STICKEL, Manfred; YUE, Shuai. *Standard PSO version 2011*. [B.r.]. Dostupné také z: https://www.particleswarm.info/standard_pso_2011_c.zip.
25. PÁNEK, Ondřej. Algoritmus optimalizace hejnem částic: vývoj a jeho aplikace. 2018.
26. CLERC, Maurice. From Theory to Practice in Particle Swarm Optimization. In: ed. PANIGRAHI, Bijaya Ketan; SHI, Yuhui; LIM, Meng-Hiot. Springer Berlin Heidelberg, 2011, s. 3–36. ISBN 978-3-642-17390-5. Dostupné z DOI: 10.1007/978-3-642-17390-5_1.
27. CLERC, Maurice. *Standard Particle Swarm Optimisation*. 2012.
28. CLERC, Maurice. *Stagnation Analysis in Particle Swarm Optimisation or What Happens When Nothing Happens*. 2006. Dostupné také z: https://hal.science/hal-00122031.
29. SHAARI, Gad; TEKBIYIK-ERSOY, Neyre; DAGBASI, Mustafa. The State of Art in Particle Swarm Optimization Based Unit Commitment: A Review. *Processes*. 2019, roč. 7, s. 733. ISSN 2227-9717. Dostupné z DOI: 10.3390/pr7100733.

30. GAD, Ahmed G. Particle Swarm Optimization Algorithm and Its Applications: A Systematic Review. *Archives of Computational Methods in Engineering*. 2022, roč. 29, s. 2531–2561. ISSN 1134-3060. Dostupné z DOI: 10.1007/s11831-021-09694-4.

31. LAIO, Alessandro; PARRINELLO, Michele. Escaping free-energy minima. *Proceedings of the National Academy of Sciences*. 2002, roč. 99, s. 12562–12566. ISSN 0027-8424. Dostupné z DOI: 10.1073/pnas.202427399.

32. WALES, David J.; DOYE, Jonathan P. K. Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. *The Journal of Physical Chemistry A*. 1997, roč. 101, s. 5111–5116. ISSN 1089-5639. Dostupné z DOI: 10.1021/jp970984n.

33. BAUER, Maximilian N.; PROBERT, Matt I. J.; PANOSETTI, Chiara. Systematic Comparison of Genetic Algorithm and Basin Hopping Approaches to the Global Optimization of Si(111) Surface Reconstructions. *The Journal of Physical Chemistry A*. 2022, roč. 126, s. 3043–3056. ISSN 1089-5639. Dostupné z DOI: 10.1021/acs.jpca.2c00647.

34. GOEDECKER, Stefan. Minima hopping: An efficient search method for the global minimum of the potential energy surface of complex molecular systems. *The Journal of Chemical Physics*. 2004, roč. 120, s. 9911–9917. ISSN 0021-9606. Dostupné z DOI: 10.1063/1.1724816.

35. KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. *Science*. 1983, roč. 220. ISSN 00368075. Dostupné z DOI: 10.1126/science.220.4598.671.

36. GLOVER, Fred. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*. 1986, roč. 13, s. 533–549. ISSN 03050548. Dostupné z DOI: 10.1016/0305-0548(86)90048-1.

37. FRAZIER, Peter I. A Tutorial on Bayesian Optimization. 2018.

38. EROL, Osman K.; EKSIN, Ibrahim. A new optimization method: Big Bang–Big Crunch. *Advances in Engineering Software*. 2006, roč. 37, s. 106–111. ISSN 09659978. Dostupné z DOI: 10.1016/j.advengsoft.2005.04.005.

39. DORIGO, M.; MANIEZZO, V.; COLORNI, A. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*. 1996, roč. 26, s. 29–41. ISSN 1083-4419. Dostupné z DOI: 10.1109/3477.484436.

40. KRISHNANAND, K.N.; GHOSE, Debasish. Glowworm swarm based optimization algorithm for multimodal functions with collective robotics applications. *Multiagent and Grid Systems*. 2006, roč. 2, s. 209–222. ISSN 18759076. Dostupné z DOI: 10.3233/MGS-2006-2301.

41. CHU, Shu-Chuan; TSAI, Pei-wei; PAN, Jeng-Shyang. Cat Swarm Optimization. In: Springer, 2006, s. 854–858. Dostupné z DOI: 10.1007/978-3-540-36668-3_94.

42. *Lead sulfate*. [B.r.]. Dostupné také z: https://pubchem.ncbi.nlm.nih.gov/compound/Lead-sulfate.

43. *$PbSO_4$ tutorial: indexing, spacegroup determination and structure solution*. [B.r.]. Dostupné také z: https://pyobjcryst.readthedocs.io/en/latest/examples/structure-solution-powder-pbso4.html.

44. REISS, Céleste A.; MECHELEN, Jan B. van; GOUBITZ, Kees; PESCHAR, René. Reassessment of paracetamol orthorhombic Form III and determination of a novel low-temperature monoclinic Form III-m from powder diffraction data. *Acta Crystallographica Section C Structural Chemistry*. 2018, roč. 74, s. 392–399. ISSN 2053-2296.

45. NICHOLS, Gary; FRAMPTON, Christopher S. Physicochemical Characterization of the Orthorhombic Polymorph of Paracetamol Crystallized from Solution. *Journal of Pharmaceutical Sciences*. 1998, roč. 87, s. 684–693. ISSN 00223549. Dostupné z DOI: 10.1021/js970483d.

46. BOULTIF, A.; LOUËR, D. Indexing of powder diffraction patterns for low-symmetry lattices by the successive dichotomy method. *Journal of Applied Crystallography*. 1991, roč. 24, s. 987–993. ISSN 0021-8898. Dostupné z DOI: 10.1107/S0021889891006441.

47. BAIL, A. Le; DUROY, H.; FOURQUET, J.L. Ab-initio structure determination of LiSbWO6 by X-ray powder diffraction. *Materials Research Bulletin*. 1988, roč. 23, s. 447–452. ISSN 00255408. Dostupné z DOI: 10.1016/0025-5408(88)90019-0.

48. APPLEBY, Todd C.; PERRY, Jason K.; MURAKAMI, Eisuke; BARAUSKAS, Ona; FENG, Joy; CHO, Aesop; FOX, David; WETMORE, Diana R.; MCGRATH, Mary E.; RAY, Adrian S.; SOFIA, Michael J.; SWAMINATHAN, S.; EDWARDS, Thomas E. Structural basis for RNA replication by the hepatitis C virus polymerase. *Science*. 2015, roč. 347, s. 771–775. ISSN 0036-8075. Dostupné z DOI: `10.1126/science.1259210`.

49. *Sofosbuvir (Monograph)*. 2022. Dostupné také z: `https://www.drugs.com/monograph/sofosbuvir.html`.

50. CHATZIADI, Argyro; SKOŘEPOVÁ, Eliška; ROHLÍČEK, Jan; DUŠEK, Michal; RIDVAN, Luděk; ŠOÓŠ, Miroslav. Mechanochemically Induced Polymorphic Transformations of Sofosbuvir. *Crystal Growth and Design*. 2020, roč. 20, s. 139–147. ISSN 1528-7483. Dostupné z DOI: `10.1021/acs.cgd.9b00922`.

51. *Sofosbuvir*. [B.r.]. Dostupné také z: `https://pubchem.ncbi.nlm.nih.gov/compound/Sofosbuvir`.

52. *Molecular format convertor*. [B.r.]. Dostupné také z: `https://www.webqc.org/`.

Přílohy

```
void MonteCarloObj::RunParticleSwarmOptimization(long &nbSteps, bool silent, double finalcost, double maxTime, double ScattTransl, double ScattConform, double ScattOrient)
{
    TAU_PROFILE("MonteCarloObj::RunParticleSwarmOptimization()", "void ()", TAU_DEFAULT);
    TAU_PROFILE_TIMER(timer0a, "MonteCarloObj::RunParticleSwarmOptimization() Begin 1", "", TAU_FIELD);
    TAU_PROFILE_TIMER(timer0b, "MonteCarloObj::RunParticleSwarmOptimization() Begin 2", "", TAU_FIELD);
    TAU_PROFILE_TIMER(timer1, "MonteCarloObj::RunParticleSwarmOptimization() New Config + LLK", "", TAU_FIELD);
    TAU_PROFILE_TIMER(timerN, "MonteCarloObj::RunParticleSwarmOptimization() Finish ", "", TAU_FIELD);

    // Keep a copy of the total number of steps,and decrement nbStep and number of free parameters
    int nbStep = nbSteps;
    mNbTrial = 0;
    int NbFreePar = mRefParList.GetNbParNotFixed();

    // time (in seconds) when last autoSave was made (if enabled)
    unsigned long secondsWhenAutoSave = 0;

    // Periodicity of the automatic LSQ refinements (if the option is set)
    const unsigned int autoLSQPeriod = 150000;

    if (!silent)
        cout << "Starting Particle Swarm Optimization for " << nbSteps << " trials" << endl;

    if (!silent)
        cout << "NbFreePar: " << NbFreePar << " nbPart: " << mParticles << endl;

    if (NbFreePar <= 0 || mParticles <= 0 || mNeighbourhood >= mParticles)
    {
        cout << "Cannot run Particle Swarm Optimization due to invalid parameters." << endl;
        return;
    }

    mCurrentCost = this->GetLogLikelihood();

    // Create temporary variables to store best configuration and other configurations of particles
    int bestParticle = 0;
    const int nbPart = (int)mParticles;
    CrystVector_long lastParSetIndex(mParticles);
    double *costFunctionArray = new double[nbPart];
    double *localMinimaCost = new double[nbPart];
    int convergentionNumber = 0;
    int currentIdenticalIterations = 0;
    double prevBestCost = __DBL_MAX__;
    double *x = new double[nbPart * NbFreePar];
    double *v = new double[nbPart * NbFreePar];
    double *M = new double[NbFreePar];
    double *m = new double[nbPart * NbFreePar];
    double runBestCost;

    // Communication variables
```

```
50      int nbTrialsFromLastReport = 0;
51      int nbTryReport = 3000;
52      bool needUpdateDisplay = false;
53      bool makeReport = false;
54
55      // Change maximal change of different parameter types
56      for (int i = 0; i < NbFreePar; i++)
57      {
58          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattTranslX))
59              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattTransl);
60          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattTranslY))
61              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattTransl);
62          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattTranslZ))
63              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattTransl);
64          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattConformX))
65              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattConform);
66          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattConformY))
67              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattConform);
68          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattConformZ))
69              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattConform);
70          if (mRefParList.GetParNotFixed(i).GetType()->IsDescendantFromOrSameAs(
        gpRefParTypeScattOrient))
71              mRefParList.GetParNotFixed(i).SetGlobalOptimStep(ScattOrient);
72      }
73
74      // Create a temporary parameter set to store the best configuration and initial
         configuration
75      const long paramSetAtBegining = mRefParList.CreateParamSet("MonteCarloObj:First
         parameters (PSO)");
76      const long runBestIndex = mRefParList.CreateParamSet("MonteCarloObj:Last
        parameters (PSO)");
77
78      TAU_PROFILE_STOP(timer0a);
79      TAU_PROFILE_START(timer0b);
80
81      // Save time when the optimization started
82      Chronometer chrono;
83      chrono.start();
84      float lastUpdateDisplayTime = chrono.seconds();
85      TAU_PROFILE_STOP(timer0b);
86
87      // Initialize arrays x (positions), v (velocities), M (current global minimum
        position), and m (particle's minima positions)
88
89      // Initialize the particle's positions and velocities and calculate the cost
        function
90      for (int S = 0; S < nbPart; S++)
91      {
92          mRefParList.RestoreParamSet(paramSetAtBegining);
93          for (int i = 0; i < NbFreePar; i++)
94          {
95              double r_number1 = ((double)rand() / RAND_MAX - 0.5);
96              double r_number2 = ((double)rand() / RAND_MAX - 0.5);
97              double parValue = mRefParList.GetParNotFixed(i).GetValue();
98              x[S * NbFreePar + i] = move(parValue, r_number1, i);
99              v[S * NbFreePar + i] = r_number2;
100             m[S * NbFreePar + i] = x[S * NbFreePar + i];
101         }
102
103         lastParSetIndex(S) = mRefParList.CreateParamSet();
104         mRefParList.SaveParamSet(lastParSetIndex(S));
105         costFunctionArray[S] = this->GetLogLikelihood();
106         localMinimaCost[S] = costFunctionArray[S];
107         mCurrentCost = costFunctionArray[S];
108         if (costFunctionArray[S] < costFunctionArray[bestParticle] || S == 0)
109         {
110             mRefParList.RestoreParamSet(lastParSetIndex(S));
111             for (int i = 0; i < NbFreePar; i++)
```

```
112             {
113                 M[i] = m[S * NbFreePar + i];
114             }
115             bestParticle = S;
116             mRefParList.SaveParamSet(runBestIndex);
117             runBestCost = costFunctionArray[bestParticle];
118             needUpdateDisplay = true;
119         }
120     }
121 
122     prevBestCost = runBestCost;
123     bool changeOfGlobalMinimum = false;
124 
125     // Initialize neighbourhoods
126     int K = mNeighbourhood;
127     int *neighbourhoods = new int[nbPart * K];
128     for (int i = 0; i < nbPart * K; i++)
129         neighbourhoods[i] = 0;
130 
131     // Particle Swarm Optimization iteration cycle
132     for (int iteration = 0; iteration < nbStep; iteration = iteration + nbPart)
133     {
134         int accept = 0;
135         // Select neighbours
136         if (!changeOfGlobalMinimum)
137         {
138             for (int S = 0; S < nbPart; S++)
139                 for (int k = 0; k < K; k++)
140                     neighbourhoods[S * K + k] = rand() % nbPart;
141         }
142 
143         changeOfGlobalMinimum = false;
144 
145         // Move with all particles
146         for (int S = 0; S < nbPart; S++)
147         {
148             mRefParList.RestoreParamSet(lastParSetIndex(S));
149 
150             double neighbourhoodMinimum = localMinimaCost[S];
151             int bestInHood = S;
152             for (int k = 0; k < K; k++)
153             {
154                 if (localMinimaCost[neighbourhoods[S * K + k]] <
    neighbourhoodMinimum)
155                 {
156                     neighbourhoodMinimum = localMinimaCost[neighbourhoods[S * K + k
    ]];
157                     bestInHood = neighbourhoods[S * K + k];
158                 }
159             }
160             if (S == bestInHood) // If the particle is the best in its
    neighbourhood the speed is calculated with the personal minimum only
161             {
162                 for (int i = 0; i < NbFreePar; i++)
163                 {
164                     v[S * NbFreePar + i] = mFormerSpeed * v[S * NbFreePar + i] +
    mFormerMinima * (double)rand() / RAND_MAX * (m[S * NbFreePar + i] - x[S *
    NbFreePar + i]);
165                     x[S * NbFreePar + i] = move(x[S * NbFreePar + i], v[S *
    NbFreePar + i], i);
166                 }
167             }
168             else // If the particle is not the best in its neighbourhood the speed
    is calculated with the personal and global minimum
169             {
170                 for (int i = 0; i < NbFreePar; i++)
171                 {
172                     v[S * NbFreePar + i] = mFormerSpeed * v[S * NbFreePar + i] +
    mFormerMinima * (double)rand() / RAND_MAX * (m[bestInHood * NbFreePar + i] - x[
    S * NbFreePar + i]) + mFormerMinima * (double)rand() / RAND_MAX * (m[S *
    NbFreePar + i] - x[S * NbFreePar + i]);
173                     x[S * NbFreePar + i] = move(x[S * NbFreePar + i], v[S *
    NbFreePar + i], i);
174                 }
```

```
175             }
176
177             // Calculate the cost function
178             costFunctionArray[S] = this->GetLogLikelihood();
179             mCurrentCost = costFunctionArray[S];
180             mRefParList.SaveParamSet(lastParSetIndex(S));
181         }
182
183         // Check for changes of minima
184         for (int S = 0; S < mParticles; S++)
185         {
186
187 #if 0 // Creating trajectory for a particle
188         int particle = 0;
189         if (S == particle)
190         {
191             string filename = "for_ovito/para" + to_string((int)iteration/nbPart) +
    ".cif";
192             ofstream outputFile;
193             outputFile.open(filename, ios::out);
194             Crystal &myCrystal = dynamic_cast<Crystal &>(mRefinedObjList.GetObj("
    paracetamol_crystal"));
195             myCrystal.CIFOutput(outputFile);
196             outputFile.close();
197         }
198 #endif
199             // If the cost function is lower than the personal minimum, the
    personal minimum is updated
200             if (costFunctionArray[S] < localMinimaCost[S])
201             {
202                 localMinimaCost[S] = costFunctionArray[S];
203                 for (int i = 0; i < NbFreePar; i++)
204                 {
205                     m[S * NbFreePar + i] = x[S * NbFreePar + i];
206                 }
207
208                 // If the cost function is lower than the global minimum, the
    global minimum is updated
209                 if (costFunctionArray[S] < runBestCost)
210                 {
211                     runBestCost = costFunctionArray[S];
212                     needUpdateDisplay = true;
213                     mRefParList.RestoreParamSet(lastParSetIndex(S));
214                     mRefParList.SaveParamSet(runBestIndex);
215
216                     for (int i = 0; i < NbFreePar; i++)
217                     {
218                         M[i] = m[S * NbFreePar + i];
219                     }
220                     bestParticle = S;
221                     changeOfGlobalMinimum = true;
222
223                     // If the cost function is lower than the absolute cost from
    all runs, the absolute cost is updated
224                     if (runBestCost < mBestCost)
225                     {
226                         accept = 2;
227                         mBestCost = costFunctionArray[S];
228                         mRefParList.SaveParamSet(mBestParSavedSetIndex);
229                     }
230                 }
231             }
232         }
233
234         // Autosave and update display
235         if (((mXMLAutoSave.GetChoice() == 1) && ((chrono.seconds() -
    secondsWhenAutoSave) > 86400)) || ((mXMLAutoSave.GetChoice() == 2) && ((chrono.
    seconds() - secondsWhenAutoSave) > 3600)) || ((mXMLAutoSave.GetChoice() == 3)
    && ((chrono.seconds() - secondsWhenAutoSave) > 600)) || ((mXMLAutoSave.
    GetChoice() == 4) && (accept == 2)))
236         {
237             secondsWhenAutoSave = (unsigned long)chrono.seconds();
238             string saveFileName = this->GetName();
239             time_t date = time(0);
```

```
240             char strDate[40];
241             strftime(strDate, sizeof(strDate), "%Y-%m-%d_%H-%M-%S", localtime(&date
    )); //%Y-%m-%dT%H:%M:%S%Z
242             char costAsChar[30];
243             mRefParList.RestoreParamSet(mBestParSavedSetIndex);
244             sprintf(costAsChar, "-Cost-%f", this->GetLogLikelihood());
245             saveFileName = saveFileName + (string)strDate + (string)costAsChar + (
    string) ".xml";
246             XMLCrystFileSaveGlobal(saveFileName);
247        }
248
249         // Update numbers of trials and steps
250         nbTrialsFromLastReport = nbTrialsFromLastReport + nbPart;
251         mNbTrial = mNbTrial + nbPart;
252         nbSteps = nbSteps - nbPart;
253
254         if (nbTrialsFromLastReport >= nbTryReport)
255         {
256             if (!silent)
257                 cout << "Trial :" << mNbTrial << " Best Cost=" << mBestCost << "
    Current Cost=" << mCurrentCost << endl;
258             nbTrialsFromLastReport = 0;
259 #ifdef __WX__CRYST__
260             if (0 != mpWXCrystObj)
261                 mpWXCrystObj->UpdateDisplayNbTrial();
262 #endif
263         }
264
265         // Automatic LSQ
266         if (mAutoLSQ.GetChoice() == 2)
267         {
268             if ((mNbTrial % autoLSQPeriod) < mParticles)
269             {
270                 // How many particles to LSQrefine
271                 double bestPercent = 0.1;
272                 int indexes[nbPart];
273                 for (int i = 0; i < nbPart; ++i)
274                 {
275                     indexes[i] = i;
276                 }
277                 partial_sort(indexes, indexes + int(nbPart * bestPercent), indexes
    + nbPart, [&](int a, int b)
278                              { return costFunctionArray[a] < costFunctionArray[b];
    });
279
280                 for (int i = 0; i < mRefinedObjList.GetNb(); i++)
281                     mRefinedObjList.GetObj(i).SetApproximationFlag(false);
282
283                 for (int S = 0; S < nbPart * bestPercent; S++)
284                 {
285 #ifdef __WX__CRYST__
286                     mMutexStopAfterCycle.Lock();
287                     if (mStopAfterCycle)
288                     {
289                         mMutexStopAfterCycle.Unlock();
290                         break;
291                     }
292                     mMutexStopAfterCycle.Unlock();
293 #endif
294
295                     mRefParList.RestoreParamSet(lastParSetIndex(indexes[S]));
296
297                     const REAL cost0 = this->GetLogLikelihood(); // cannot use
    currentCost(i), approximations changed...
298
299                     try
300                     {
301                         mLSQ.Refine(-30, true, true, false, 0.001);
302                     }
303                     catch (const ObjCrystException &except)
304                     {
305                     };
306                     REAL cost = this->GetLogLikelihood();
307                     if (!silent)
```

```
308                         cout << " -> " << cost << endl;
309                     if (cost < cost0)
310                     {
311                         mRefParList.SaveParamSet(lastParSetIndex(indexes[S]));
312 
313                         // Update positions and personal minima
314                         for (int i = 0; i < NbFreePar; i++)
315                         {
316                             x[indexes[S] * NbFreePar + i] = move(x[indexes[S] *
    NbFreePar + i], 0, i);
317                             m[indexes[S] * NbFreePar + i] = x[indexes[S] *
    NbFreePar + i];
318                         }
319                     }
320                 }
321                 //  Need to go back to optimization with approximations allowed (
    they are not during LSQ)
322                 for (int i = 0; i < mRefinedObjList.GetNb(); i++)
323                     mRefinedObjList.GetObj(i).SetApproximationFlag(true);
324 
325                 for (int S = 0; S < nbPart * bestPercent; S++)
326                 {
327                     // And recompute LLK - since they will be lower
328                     mRefParList.RestoreParamSet(lastParSetIndex(indexes[S]));
329                     REAL cost = this->GetLogLikelihood();
330                     if (!silent)
331                         cout << "LSQ2:" << costFunctionArray[indexes[S]] << "->" <<
     cost << endl;
332                     if (cost < costFunctionArray[indexes[S]])
333                     {
334                         mRefParList.SaveParamSet(lastParSetIndex(indexes[S]));
335                         costFunctionArray[indexes[S]] = cost;
336                         if (cost < runBestCost)
337                         {
338                             runBestCost = costFunctionArray[indexes[S]];
339                             this->TagNewBestConfig();
340                             needUpdateDisplay = true;
341                             bestParticle = indexes[S];
342                             mRefParList.SaveParamSet(runBestIndex);
343                             if (runBestCost < mBestCost)
344                             {
345                                 mBestCost = costFunctionArray[bestParticle];
346                                 mRefParList.SaveParamSet(mBestParSavedSetIndex);
347                                 if (!silent)
348                                     this->DisplayReport();
349                             }
350                         }
351                     }
352                 }
353             }
354         }
355 
356         if (true == makeReport)
357         {
358             makeReport = false;
359             if (!silent)
360             {
361                 for (int i = 0; i < nbPart; i++)
362                 {
363                     cout << "   Particle :" << lastParSetIndex(i) << ":";
364                     map<const RefinableObj *, LogLikelihoodStats>::iterator pos;
365                     for (pos = mvContextObjStats[i].begin(); pos !=
    mvContextObjStats[i].end(); ++pos)
366                     {
367                         cout << pos->first->GetName()
368                              << "(LLK="
369                              << pos->second.mLastLogLikelihood
370                              << ", w=" << mvObjWeight[pos->first].mWeight
371                              << ")  ";
372                         pos->second.mTotalLogLikelihood = 0;
373                         pos->second.mTotalLogLikelihoodDeltaSq = 0;
374                     }
375                     cout << endl;
376                 }
```

```
377                 for (int i = 0; i < nbPart; i++)
378                 {
379                     cout << "   Particle :" << lastParSetIndex(i)
380                          << " Current Cost=" << costFunctionArray[i];
381                 }
382             }
383             if (!silent)
384                 cout << "Trial :" << mNbTrial << " Best Cost=" << runBestCost << "
    ";
385             if (!silent)
386                 chrono.print();
387         }
388 
389         // Update display
390         if ((needUpdateDisplay && (lastUpdateDisplayTime < (chrono.seconds() - 1)))
     || (lastUpdateDisplayTime < (chrono.seconds() - 10)))
391         {
392             mRefParList.RestoreParamSet(runBestIndex);
393             this->UpdateDisplay();
394             needUpdateDisplay = false;
395             lastUpdateDisplayTime = chrono.seconds();
396         }
397 
398 #ifdef __WX__CRYST__
399         mMutexStopAfterCycle.Lock();
400 #endif
401         if ((mBestCost < finalcost) || mStopAfterCycle || ((maxTime > 0) && (chrono
    .seconds() > maxTime)))
402         {
403 #ifdef __WX__CRYST__
404             mMutexStopAfterCycle.Unlock();
405 #endif
406             if (!silent)
407                 cout << endl
408                      << endl
409                      << "Refinement Stopped." << endl;
410             break;
411         }
412 #ifdef __WX__CRYST__
413         mMutexStopAfterCycle.Unlock();
414 #endif
415 
416         // Test for convergence
417         mRefParList.RestoreParamSet(runBestIndex);
418         if (prevBestCost == runBestCost)
419             currentIdenticalIterations++;
420         else
421             currentIdenticalIterations = 0;
422 
423         if (convergentionNumber <= currentIdenticalIterations && converged(
    prevBestCost, x, v, currentIdenticalIterations, NbFreePar))
424         {
425             if (!silent)
426                 cout << "Algorithm has converged after " << mNbTrial << " trials."
    << endl;
427             break;
428         }
429         prevBestCost = runBestCost;
430 
431         ofstream file("miry_shody.txt", std::ios::app);
432         file << runBestCost << std::endl;
433         file.close();
434     }
435 
436     delete[] x;
437     delete[] v;
438     delete[] m;
439     delete[] M;
440     delete[] costFunctionArray;
441     delete[] localMinimaCost;
442     delete[] neighbourhoods;
443 
444     TAU_PROFILE_START(timerN);
445     if (mAutoLSQ.GetChoice() > 0)
```

```
446     { // LSQ
447         if (!silent)
448             cout << "Beginning final LSQ refinement" << endl;
449         for (int i = 0; i < mRefinedObjList.GetNb(); i++)
450             mRefinedObjList.GetObj(i).SetApproximationFlag(false);
451         mRefParList.RestoreParamSet(runBestIndex);
452         mCurrentCost = this->GetLogLikelihood();
453         try
454         {
455             mLSQ.Refine(-50, true, true, false, 0.001);
456         }
457         catch (const ObjCrystException &except)
458         {
459         };
460         if (!silent)
461             cout << "LSQ cost: " << mCurrentCost << " -> " << this->
        GetLogLikelihood() << endl;
462
463         // Need to go back to optimization with approximations allowed (they are
        not during LSQ)
464         for (int i = 0; i < mRefinedObjList.GetNb(); i++)
465             mRefinedObjList.GetObj(i).SetApproximationFlag(true);
466
467         REAL cost = this->GetLogLikelihood();
468         if (cost < mCurrentCost)
469         {
470             mCurrentCost = cost;
471             mRefParList.SaveParamSet(runBestIndex);
472             if (mCurrentCost < runBestCost)
473             {
474                 runBestCost = mCurrentCost;
475                 mRefParList.SaveParamSet(runBestIndex);
476                 if (runBestCost < mBestCost)
477                 {
478                     mBestCost = mCurrentCost;
479                     mRefParList.SaveParamSet(mBestParSavedSetIndex);
480                     if (!silent)
481                         cout << "LSQ : NEW OVERALL Best Cost=" << runBestCost <<
        endl;
482                 }
483                 else if (!silent)
484                     cout << " LSQ : NEW Run Best Cost=" << runBestCost << endl;
485             }
486         }
487         if (!silent)
488             cout << "Finished LSQ refinement" << endl;
489     }
490
491     // Restore the best parameter set and display the final report
492     mRefParList.RestoreParamSet(runBestIndex);
493     mCurrentCost = this->GetLogLikelihood();
494
495     mLastOptimTime = chrono.seconds();
496     if (!silent)
497         this->DisplayReport();
498     mCurrentCost = this->GetLogLikelihood();
499
500     if (!silent)
501         cout << "Run Best Cost:" << mCurrentCost << endl;
502     if (!silent)
503         chrono.print();
504
505     // Clear the temporary parameter sets created at the beginning
506     mRefParList.ClearParamSet(paramSetAtBegining);
507     for (int S = 0; S < nbPart; S++)
508         mRefParList.ClearParamSet(lastParSetIndex(S));
509     mRefParList.ClearParamSet(runBestIndex);
510     TAU_PROFILE_STOP(timerN);
511 }
512
513 // Sets Particle Swarm Optimization parameters
514 void MonteCarloObj::SetAlgorithmParticleSwarmOptimization(int nbParticles, float
        parFormerSpeed, float parFormerMinima)
515 {
```

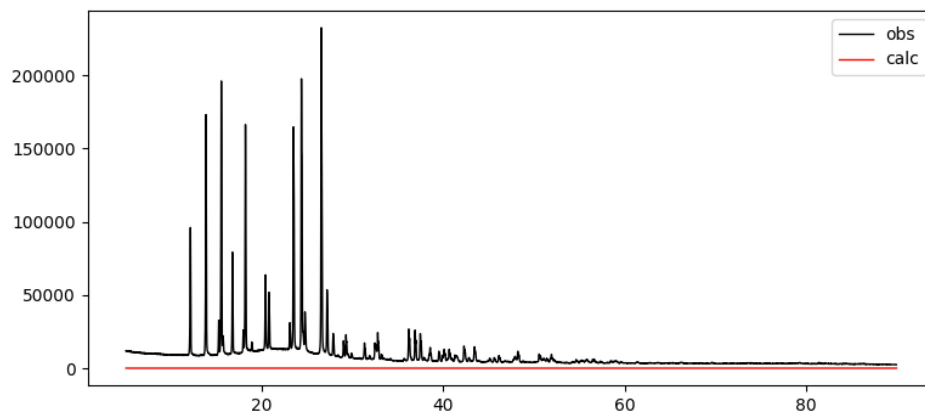
```
516     // Set optimization algorithm to Particle Swarm Optimization
517     VFN_DEBUG_MESSAGE("MonteCarloObj::SetAlgorithmParticleSwarmOptimization()", 5)
518     mGlobalOptimType.SetChoice(GLOBAL_OPTIM_PARTICLE_SWARM_OPTIMIZATION);
519     // Set PSO parameters
520     mParticles = nbParticles;
521     mFormerSpeed = parFormerSpeed;
522     mFormerMinima = parFormerMinima;
523     mNeighbourhood = 3;
524     VFN_DEBUG_MESSAGE("MonteCarloObj::SetAlgorithmParticleSwarmOptimization():End",
         3)
525 }
526 
527 // Check if the algorithm has converged
528 bool MonteCarloObj::converged(double prevBestCost, double *x, double *v, int
        sameValues, int NbFreePar)
529 {
530     int option = 1;
531     switch (option)
532     {
533     case 0:
534     {
535         double distance = 0;
536         for (int S = 0; S < int(mParticles); S++)
537         {
538             for (int T = S; T < int(mParticles); T++)
539             {
540                 for (int i = 0; i < NbFreePar; i++)
541                 {
542                     distance = distance + (x[S * NbFreePar + i] - x[T * NbFreePar +
         i]) * (x[S * NbFreePar + i] - x[T * NbFreePar + i]);
543                 }
544             }
545         }
546         if (distance > 1000)
547             return false;
548         break;
549     }
550     case 1:
551     {
552         double sumOfSpeeds = 0;
553         for (int i = 0; i < NbFreePar * int(mParticles); i++)
554             sumOfSpeeds = sumOfSpeeds + abs(v[i]);
555 
556         ofstream file("speed.txt", std::ios::app);
557         file << sumOfSpeeds << std::endl;
558         file.close();
559 
560         if (sumOfSpeeds/mParticles > 0.01)
561             return false;
562         break;
563     }
564     case 2:
565     {
566         double valueWithoutLSQ = this->GetLogLikelihood();
567         // LSQ
568         // Disable approximation flags for refined objects during LSQ refinement
569         for (int i = 0; i < mRefinedObjList.GetNb(); i++)
570             mRefinedObjList.GetObj(i).SetApproximationFlag(false);
571 
572         // Perform the LSQ refinement
573         try
574         {
575             mLSQ.Refine(-50, true, true, false, 0.001);
576         }
577         catch (const ObjCrystException &except)
578         {
579         };
580 
581         // Re-enable approximation flags for refined objects
582         for (int i = 0; i < mRefinedObjList.GetNb(); i++)
583             mRefinedObjList.GetObj(i).SetApproximationFlag(true);
584 
585         // Calculate the cost after LSQ refinement
586         double cost = this->GetLogLikelihood();
```

```
587         if ((cost - valueWithoutLSQ) * (cost - valueWithoutLSQ) > 10000000)
588             return false;
589         break;
590     }
591     case 3:
592     {
593         if (mBestCost > 1000000)
594             return false;
595         break;
596     }
597     default:
598         break;
599     }
600 
601     return true;
602 }
603 
604 // Perform particle movement and constraint handling
605 double MonteCarloObj::move(double x, double v, int i)
606 {
607     // Update the particle's position with the given velocity
608     double mutationAmplitude = mRefParList.GetParNotFixed(i).GetGlobalOptimStep();
609     x = x + v * mutationAmplitude;
610 
611     // Apply periodic or limited constraints if needed
612     if (true == mRefParList.GetParNotFixed(i).IsLimited())
613     {
614         const REAL min = mRefParList.GetParNotFixed(i).GetMin();
615         const REAL max = mRefParList.GetParNotFixed(i).GetMax();
616         if (x < min)
617         {
618             x = min;
619         }
620         else if (x > max)
621         {
622             x = max;
623         }
624     }
625     else if (true == mRefParList.GetParNotFixed(i).IsPeriodic())
626     {
627         const REAL period = mRefParList.GetParNotFixed(i).GetPeriod();
628         if (x < 0)
629         {
630             int times = (int)(x / period) - 1;
631             x = x - times * period;
632         }
633         if (x > period)
634         {
635             int times = (int)(x / period);
636             x = x - times * period;
637         }
638     }
639     mRefParList.GetParNotFixed(i).Mutate(v * mutationAmplitude);
640     x = mRefParList.GetParNotFixed(i).GetValue();
641     return x; // Return the updated particle's position
642 }
```

[1]:
```
%matplotlib widget
import os
import sys
import pyobjcryst
import numpy as np
import matplotlib.pyplot as plt
from pyobjcryst.crystal import *
from pyobjcryst.scatteringpower import ScatteringPowerAtom
from pyobjcryst.atom import Atom
from pyobjcryst.polyhedron import MakeTetrahedron
from pyobjcryst.powderpattern import *
from pyobjcryst.radiation import RadiationType
from pyobjcryst.indexing import *
from pyobjcryst.molecule import *
from pyobjcryst.globaloptim import MonteCarlo
from pyobjcryst.io import xml_cryst_file_save_global
from pyobjcryst.refinableobj import refpartype_scattdata_background
from pyobjcryst.refinableobj import refpartype_scatt_conform_dihedangle
from pyobjcryst.refinableobj import RefinablePar
```

[2]:
```
p = PowderPattern()
p.ImportPowderPattern2ThetaObs("exp_data.dat")
p.SetWavelength(1.5405980)

p.plot()
```

```
Imported powder pattern: 6474 points, 2theta=  5.002 ->  89.995, step= 0.013
```



[4]:
```
# Index
pl = p.FindPeaks(1.5, -1, 1000)
```

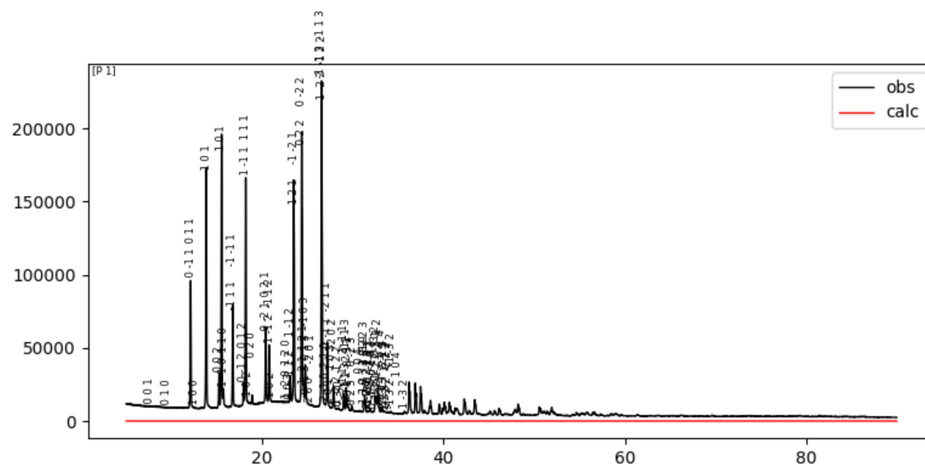
```python
if len(pl) > 20:
    pl.resize(20)  # Only keep 20 peaks
for peak in pl:
    print(peak)

ex = quick_index(pl)

print("Solutions:")
for s in ex.GetSolutions():
    print(s)
```

```
Peak dobs=0.13698+/-0.00026 iobs=5.293813e+05 (? ? ?))
Peak dobs=0.15632+/-0.00026 iobs=1.016839e+06 (? ? ?))
Peak dobs=0.17222+/-0.00026 iobs=1.373442e+05 (? ? ?))
Peak dobs=0.17533+/-0.00026 iobs=1.156594e+06 (? ? ?))
Peak dobs=0.17771+/-0.00018 iobs=3.824300e+04 (? ? ?))
Peak dobs=0.18920+/-0.00026 iobs=4.387352e+05 (? ? ?))
Peak dobs=0.20250+/-0.00022 iobs=7.900886e+04 (? ? ?))
Peak dobs=0.20523+/-0.00029 iobs=9.489007e+05 (? ? ?))
Peak dobs=0.21316+/-0.00029 iobs=3.735977e+04 (? ? ?))
Peak dobs=0.22985+/-0.00029 iobs=3.098631e+05 (? ? ?))
Peak dobs=0.23432+/-0.00029 iobs=2.377607e+05 (? ? ?))
Peak dobs=0.25984+/-0.00029 iobs=1.120684e+05 (? ? ?))
Peak dobs=0.26433+/-0.00033 iobs=9.039776e+05 (? ? ?))
Peak dobs=0.27406+/-0.00029 iobs=1.109016e+06 (? ? ?))
Peak dobs=0.27843+/-0.00029 iobs=1.440321e+05 (? ? ?))
Peak dobs=0.29827+/-0.00033 iobs=1.302337e+06 (? ? ?))
Peak dobs=0.30527+/-0.00033 iobs=2.547030e+05 (? ? ?))
Peak dobs=0.31271+/-0.00036 iobs=8.602636e+04 (? ? ?))
Peak dobs=0.32529+/-0.00036 iobs=6.029434e+04 (? ? ?))
Peak dobs=0.32831+/-0.00032 iobs=8.023384e+04 (? ? ?))
Predicting volumes from 20 peaks between d=73.003 and d= 3.046

Starting indexing using 20 peaks
      CUBIC P : V=   1724 ->  19270 A^3, max length= 80.43A
 ->   0 sols in   0.00s, best score=   0.0

 TETRAGONAL P : V=    640 ->   4617 A^3, max length= 49.96A
 ->   0 sols in   0.03s, best score=   0.0

RHOMBOEDRAL P : V=    709 ->   4845 A^3, max length= 50.76A
 ->   0 sols in   0.00s, best score=   0.0

  HEXAGONAL P : V=    874 ->   6389 A^3, max length= 55.67A
 ->   0 sols in   0.07s, best score=   0.0

ORTHOROMBIC P : V=    372 ->   2395 A^3, max length= 40.14A
 ->   0 sols in   0.08s, best score=   0.0
```

```
 MONOCLINIC P : V=    277 ->   1543 A^3, max length= 34.66A
 ->   1 sols in   0.34s, best score= 128.9

Solutions:
( 7.10  9.38 11.71  90.0  97.4  90.0 V= 773 MONOCLINIC P, 128.93045043945312)
```

[5]:

```python
uc = ex.GetSolutions()[0][0].DirectUnitCell()
c = pyobjcryst.crystal.Crystal(uc[0], uc[1], uc[2], uc[3], uc[4], uc[5], "P1")
pdiff = p.AddPowderPatternDiffraction(c)

# Plot with indexing in new figure
p.plot(diff=False,fig=None,hkl=True)
```



[6]:

```python
p.quick_fit_profile(auto_background=True,plot=False,␣
 ↪init_profile=True,verbose=True)
p.quick_fit_profile(plot=False, init_profile=False, asym=True,␣
 ↪displ_transl=True, verbose=False)

# Plot in new figure
p.plot(diff=True, fig=None, hkl=True)
print("Fit result: Rw=%6.2f%% Chi2=%10.2f  GoF=%8.2f  LLK=%10.3f" %
      (p.rw * 100, p.chi2, p.chi2/p.GetNbPointUsed(), p.llk))
```

```
No background, adding one automatically
Selected PowderPatternDiffraction:    with Crystal:
Profile fitting finished.
Remember to use SetExtractionMode(False) on the PowderPatternDiffraction object
to disable profile fitting and optimise the structure.
Fit result: Rw=  5.36% Chi2= 151141.94  GoF=   23.35  LLK= 19718.235
```

[7]:

```
p.quick_fit_profile(pdiff=None, auto_background=0, init_profile=0, plot=False,
                    zero=1, constant_width=1, width=1, eta=1, backgd=True,␣
 ↪cell=True,
                    anisotropic=1, asym=1, displ_transl=1, verbose=0)
```

[8]:

```
# Plot in new figure
p.plot(diff=True, fig=None, hkl=True)
print("Fit result: Rw=%6.2f%% Chi2=%10.2f  GoF=%8.2f  LLK=%10.3f" %
      (p.rw * 100, p.chi2, p.chi2/p.GetNbPointUsed(), p.llk))
```

```
Fit result: Rw=  5.24% Chi2= 144654.73  GoF=   22.34  LLK= 17654.561
```

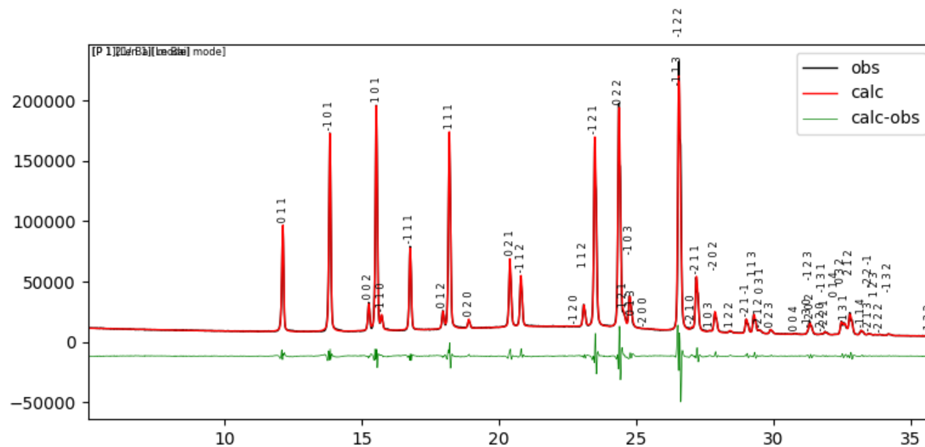
[9]:

```
p.SetMaxSinThetaOvLambda(0.2)
spgex = SpaceGroupExplorer(pdiff)

spgex.RunAll(keep_best=True, update_display=False, fitprofile_p1=False)

for sol in spgex.GetScores():
    if sol.GoF <= 2 * spgex.GetScores()[0].GoF:
        print(sol)

print("Chosen spacegroup (smallest nGoF): ", c.GetSpaceGroup())
c.GetSpaceGroup().ChangeSpaceGroup("P 1 21/n 1")

# Updated plot with optimal spacegroup
p.plot(diff=True, fig=None, hkl=True, reset=True)
for peak in pl:
    print(peak)
print("Fit result: Rw=%6.2f%% Chi2=%10.2f  GoF=%8.2f  LLK=%10.3f" %
      (p.rw * 100, p.chi2, p.chi2/p.GetNbPointUsed(), p.llk))
```

```
Beginning spacegroup exploration... 37 to go...
  (#  1) P 1           : Rwp=  6.05%  GoF=    54.39  nGoF=     8.40  (114
reflections,  0 extinct)
  (#  2) P -1          : Rwp=  6.05%  GoF=    54.39  nGoF=     8.40  (114
reflections,  0 extinct) [same extinctions as:P 1]
  (#  3) P 1 2 1       : Rwp=  6.11%  GoF=    54.62  nGoF=     5.23  ( 69
reflections,  0 extinct)
  (#  4) P 1 21 1      : Rwp=  6.09%  GoF=    54.24  nGoF=     5.11  ( 67
reflections,  2 extinct)
  (#  5) C 1 2 1       : Rwp= 44.85%  GoF=  2903.61  nGoF=   804.39  ( 37
reflections, 84 extinct)
  (#  5) A 1 2 1       : Rwp= 47.10%  GoF=  3197.32  nGoF=   820.11  ( 34
reflections, 85 extinct)
  (#  5) I 1 2 1       : Rwp= 46.72%  GoF=  3147.92  nGoF=   850.79  ( 35
reflections, 87 extinct)
  (#  6) P 1 m 1       : Rwp=  6.11%  GoF=    54.62  nGoF=     5.23  ( 69
reflections,  0 extinct) [same extinctions as:P 1 2 1]
  (#  7) P 1 c 1       : Rwp= 38.04%  GoF=  2107.87  nGoF=   786.04  ( 59
reflections, 15 extinct)
  (#  7) P 1 n 1       : Rwp=  6.13%  GoF=    54.74  nGoF=     4.71  ( 58
reflections, 17 extinct)
  (#  7) P 1 a 1       : Rwp= 38.02%  GoF=  2106.56  nGoF=   798.56  ( 60
reflections, 14 extinct)
  (#  8) C 1 m 1       : Rwp= 44.85%  GoF=  2903.61  nGoF=   804.39  ( 37
reflections, 84 extinct) [same extinctions as:C 1 2 1]
  (#  8) A 1 m 1       : Rwp= 47.10%  GoF=  3197.32  nGoF=   820.11  ( 34
reflections, 85 extinct) [same extinctions as:A 1 2 1]
  (#  8) I 1 m 1       : Rwp= 46.72%  GoF=  3147.92  nGoF=   850.79  ( 35
reflections, 87 extinct) [same extinctions as:I 1 2 1]
```

```
  (#  9) C 1 c 1       : Rwp= 44.72%  GoF=  2879.61  nGoF=   672.67  ( 31
reflections,  93 extinct)
  (#  9) A 1 n 1       : Rwp= 45.89%  GoF=  3029.24  nGoF=   673.31  ( 29
reflections,  93 extinct)
  (#  9) I 1 a 1       : Rwp= 50.49%  GoF=  3669.42  nGoF=   905.78  ( 31
reflections,  93 extinct)
  (#  9) A 1 a 1       : Rwp= 45.89%  GoF=  3029.24  nGoF=   673.31  ( 29
reflections,  93 extinct) [same extinctions as:A 1 n 1]
  (#  9) C 1 n 1       : Rwp= 44.72%  GoF=  2879.61  nGoF=   672.67  ( 31
reflections,  93 extinct) [same extinctions as:C 1 c 1]
  (#  9) I 1 c 1       : Rwp= 50.49%  GoF=  3669.42  nGoF=   905.78  ( 31
reflections,  93 extinct) [same extinctions as:I 1 a 1]
  (# 10) P 1 2/m 1     : Rwp=  6.11%  GoF=    54.62  nGoF=     5.23  ( 69
reflections,   0 extinct) [same extinctions as:P 1 2 1]
  (# 11) P 1 21/m 1    : Rwp=  6.09%  GoF=    54.24  nGoF=     5.11  ( 67
reflections,   2 extinct) [same extinctions as:P 1 21 1]
  (# 12) C 1 2/m 1     : Rwp= 44.85%  GoF=  2903.61  nGoF=   804.39  ( 37
reflections,  84 extinct) [same extinctions as:C 1 2 1]
  (# 12) A 1 2/m 1     : Rwp= 47.10%  GoF=  3197.32  nGoF=   820.11  ( 34
reflections,  85 extinct) [same extinctions as:A 1 2 1]
  (# 12) I 1 2/m 1     : Rwp= 46.72%  GoF=  3147.92  nGoF=   850.79  ( 35
reflections,  87 extinct) [same extinctions as:I 1 2 1]
  (# 13) P 1 2/c 1     : Rwp= 38.04%  GoF=  2107.87  nGoF=   786.04  ( 59
reflections,  15 extinct) [same extinctions as:P 1 c 1]
  (# 13) P 1 2/n 1     : Rwp=  6.13%  GoF=    54.74  nGoF=     4.71  ( 58
reflections,  17 extinct) [same extinctions as:P 1 n 1]
  (# 13) P 1 2/a 1     : Rwp= 38.02%  GoF=  2106.56  nGoF=   798.56  ( 60
reflections,  14 extinct) [same extinctions as:P 1 a 1]
  (# 14) P 1 21/c 1    : Rwp= 38.04%  GoF=  2105.93  nGoF=   759.35  ( 57
reflections,  17 extinct)
  (# 14) P 1 21/n 1    : Rwp=  6.11%  GoF=    54.37  nGoF=     4.57  ( 56
reflections,  19 extinct)
  (# 14) P 1 21/a 1    : Rwp= 38.02%  GoF=  2104.63  nGoF=   771.89  ( 58
reflections,  16 extinct)
  (# 15) C 1 2/c 1     : Rwp= 44.72%  GoF=  2879.61  nGoF=   672.67  ( 31
reflections,  93 extinct) [same extinctions as:C 1 c 1]
  (# 15) A 1 2/n 1     : Rwp= 45.89%  GoF=  3029.24  nGoF=   673.31  ( 29
reflections,  93 extinct) [same extinctions as:A 1 n 1]
  (# 15) I 1 2/a 1     : Rwp= 50.49%  GoF=  3669.42  nGoF=   905.78  ( 31
reflections,  93 extinct) [same extinctions as:I 1 a 1]
  (# 15) A 1 2/a 1     : Rwp= 45.89%  GoF=  3029.24  nGoF=   673.31  ( 29
reflections,  93 extinct) [same extinctions as:A 1 n 1]
  (# 15) C 1 2/n 1     : Rwp= 44.72%  GoF=  2879.61  nGoF=   672.67  ( 31
reflections,  93 extinct) [same extinctions as:C 1 c 1]
  (# 15) I 1 2/c 1     : Rwp= 50.49%  GoF=  3669.42  nGoF=   905.78  ( 31
reflections,  93 extinct) [same extinctions as:I 1 a 1]
Restoring best spacegroup: P 1 21/n 1
P 1 21/n 1    nGoF=   4.5750 GoF=  54.369 Rw= 6.11 [ 56 reflections, extinct446=
```

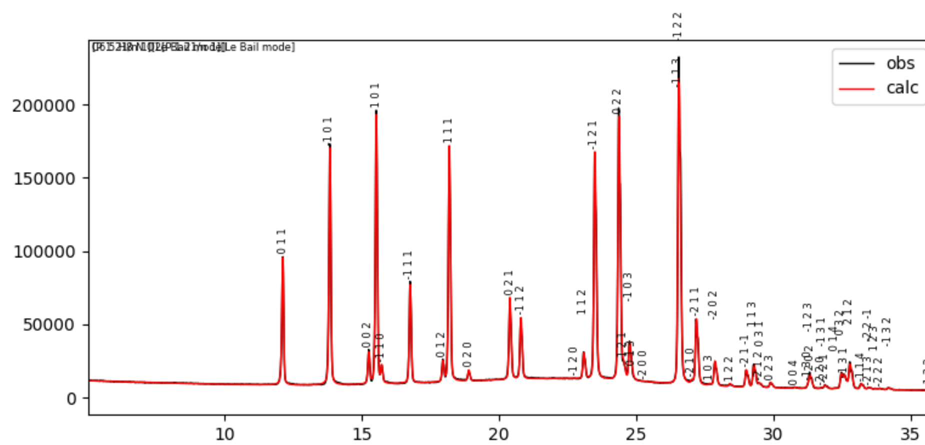
```
19]
P 1 n 1        nGoF=   4.7054 GoF=  54.737 Rw= 6.13 [ 58 reflections, extinct446=
17]
P 1 2/n 1      nGoF=   4.7054 GoF=  54.737 Rw= 6.13 [ 58 reflections, extinct446=
17]
P 1 21 1       nGoF=   5.1121 GoF=  54.244 Rw= 6.09 [ 67 reflections, extinct446=
2]
P 1 21/m 1     nGoF=   5.1121 GoF=  54.244 Rw= 6.09 [ 67 reflections, extinct446=
2]
P 1 2 1        nGoF=   5.2255 GoF=  54.616 Rw= 6.11 [ 69 reflections, extinct446=
0]
P 1 m 1        nGoF=   5.2255 GoF=  54.616 Rw= 6.11 [ 69 reflections, extinct446=
0]
P 1 2/m 1      nGoF=   5.2255 GoF=  54.616 Rw= 6.11 [ 69 reflections, extinct446=
0]
P 1            nGoF=   8.3968 GoF=  54.387 Rw= 6.05 [114 reflections, extinct446=
0]
P -1           nGoF=   8.3968 GoF=  54.387 Rw= 6.05 [114 reflections, extinct446=
0]
Chosen spacegroup (smallest nGoF):  P 1 21/n 1
Peak dobs=0.13698+/-0.00026 iobs=5.293813e+05 ( 0  1  1))
Peak dobs=0.15632+/-0.00026 iobs=1.016839e+06 ( 1  0 -1))
Peak dobs=0.17222+/-0.00026 iobs=1.373442e+05 ( 0  0  2))
Peak dobs=0.17533+/-0.00026 iobs=1.156594e+06 ( 1  0  1))
Peak dobs=0.17771+/-0.00018 iobs=3.824300e+04 ( 1  1  0))
Peak dobs=0.18920+/-0.00026 iobs=4.387352e+05 (? ? ?))
Peak dobs=0.20250+/-0.00022 iobs=7.900886e+04 ( 0  1  2))
Peak dobs=0.20523+/-0.00029 iobs=9.489007e+05 ( 1  1  1))
Peak dobs=0.21316+/-0.00029 iobs=3.735977e+04 ( 0  2  0))
Peak dobs=0.22985+/-0.00029 iobs=3.098631e+05 ( 0  2  1))
Peak dobs=0.23432+/-0.00029 iobs=2.377607e+05 ( 1  1 -2))
Peak dobs=0.25984+/-0.00029 iobs=1.120684e+05 ( 1  1  2))
Peak dobs=0.26433+/-0.00033 iobs=9.039776e+05 ( 1  2 -1))
Peak dobs=0.27406+/-0.00029 iobs=1.109016e+06 ( 0  2  2))
Peak dobs=0.27843+/-0.00029 iobs=1.440321e+05 ( 1  0 -3))
Peak dobs=0.29827+/-0.00033 iobs=1.302337e+06 ( 1  2 -2))
Peak dobs=0.30527+/-0.00033 iobs=2.547030e+05 ( 2  1 -1))
Peak dobs=0.31271+/-0.00036 iobs=8.602636e+04 ( 2  0 -2))
Peak dobs=0.32529+/-0.00036 iobs=6.029434e+04 ( 2  1  1))
Peak dobs=0.32831+/-0.00032 iobs=8.023384e+04 ( 1  1  3))
Fit result: Rw=  6.16% Chi2= 131409.49  GoF=   20.30  LLK= 14565.481
```




```
[10]: #import molekuly paracetamolu
      m = ImportFenskeHallZMatrix(c, "molecule_paracetamol.fh")
      print("Crystal Formula:", c.GetFormula())

      c.GetOption(1).SetChoice(0) # 0 = no usage of dynamical occupancy correction
```

```
Crystal Formula: C6.5 H8 N O2
```

```
[11]: p.FitScaleFactorForRw()
      p.plot(fig=None,diff=False,hkl=True)
```




```
[12]: mc = MonteCarlo()
      mc.AddRefinableObj(c)
```

```
mc.AddRefinableObj(p)

pdiff.SetExtractionMode(False)

mc.SetParIsFixed(refpartype_scattdata_background, True)
mc.SetParIsUsed("gpRefParTypeScattDataBackground", False)
```

[13]:
```
mc.SetAlgorithmParticleSwarmOptimization(100, 0.6, 1.193)
NbRun = 100
algorithm = 2
mc.GetOption("Automatic Least Squares Refinement").SetChoice(2) #each 150k trials
mc.GetOption("Algorithm").SetChoice(algorithm)

mc.MultiRunOptimize(NbRun,1e6)
```

```
Flip group with respect to: C2-C1-C6 :     -O7:H17  O7
Flip group with respect to: C1-C2-C3 :     -H13:H13
Flip group with respect to: C2-C3-C4 :     -H14:H14
Flip group with respect to: C5-C4-C3 :     -N8:C9  H18  C11  H19  H20  O10  H12
N8
Flip group with respect to: C4-C5-C6 :     -H15:H15
Flip group with respect to: C5-C6-C1 :     -H16:H16
Flip group with respect to: C9-N8-C4 :     -H12:H12
Flip group from atom N8,exchanging bonds with C9 and H12, resulting in a 180
rotation of atoms : C9  H18  C11  H19  H20  O10  H12
Flip group with respect to: C4-N8-H12 :     -C9:C9  H18  C11  H19  H20  O10
Flip group from atom C9,exchanging bonds with C11 and O10, resulting in a 180
rotation of atoms : H18  C11  H19  H20  O10
Flip group with respect to: C11-C9-N8 :     -O10:O10
Flip group with respect to: O10-C9-N8 :     -C11:H18  C11  H19  H20
Flip group with respect to: C9-C11-H18 :     -H19:H19      -H20:H20
Flip group with respect to: C9-C11-H19 :     -H18:H18      -H20:H20
Flip group with respect to: C9-C11-H20 :     -H18:H18      -H19:H19
Flip group from atom C11,exchanging bonds with H18 and H19, resulting in a 180
rotation of atoms : H18  H19
Flip group from atom C11,exchanging bonds with H18 and H20, resulting in a 180
rotation of atoms : H18  H20
Flip group from atom C11,exchanging bonds with H19 and H20, resulting in a 180
rotation of atoms : H19  H20
Found ring:C1 C2 C3 C4 C5 C6
Rings found :1, 1 unique.
List of Bond Length stretch modes
   Bond:C1-O7, Translated Atoms:  H17,O7, ; restrained to length=1.387,
sigma=0.01, delta=0.02
       Broken bonds:C1-O7,
   Bond:C2-H13, Translated Atoms:  H13, ; restrained to length=1.088,
sigma=0.01, delta=0.02
       Broken bonds:C2-H13,
```

```
   Bond:C3-H14, Translated Atoms:  H14, ; restrained to length=1.083,
sigma=0.01, delta=0.02
       Broken bonds:C3-H14,
   Bond:C4-N8, Translated Atoms:  C9,H18,C11,H19,H20,O10,H12,N8, ; restrained to
length=1.424, sigma=0.01, delta=0.02
       Broken bonds:C4-N8,
   Bond:C5-H15, Translated Atoms:  H15, ; restrained to length=1.09, sigma=0.01,
delta=0.02
       Broken bonds:C5-H15,
   Bond:C6-H16, Translated Atoms:  H16, ; restrained to length=1.091,
sigma=0.01, delta=0.02
       Broken bonds:C6-H16,
   Bond:O7-H17, Translated Atoms:  H17, ; restrained to length=0.974,
sigma=0.01, delta=0.02
       Broken bonds:O7-H17,
   Bond:N8-C9, Translated Atoms:  C9,H18,C11,H19,H20,O10, ; restrained to
length=1.39, sigma=0.01, delta=0.02
       Broken bonds:N8-C9,
   Bond:N8-H12, Translated Atoms:  H12, ; restrained to length=1.016,
sigma=0.01, delta=0.02
       Broken bonds:N8-H12,
   Bond:C9-O10, Translated Atoms:  O10, ; restrained to length=1.232,
sigma=0.01, delta=0.02
       Broken bonds:C9-O10,
   Bond:C9-C11, Translated Atoms:  H18,C11,H19,H20, ; restrained to
length=1.529, sigma=0.01, delta=0.02
       Broken bonds:C9-C11,
   Bond:C11-H18, Translated Atoms:  H18, ; restrained to length=1.099,
sigma=0.01, delta=0.02
       Broken bonds:C11-H18,
   Bond:C11-H19, Translated Atoms:  H19, ; restrained to length=1.094,
sigma=0.01, delta=0.02
       Broken bonds:C11-H19,
   Bond:C11-H20, Translated Atoms:  H20, ; restrained to length=1.099,
sigma=0.01, delta=0.02
       Broken bonds:C11-H20,
ANGLE :C2-C1-O7:H17,O7,: base rotation=0.974028 free=0
ANGLE :C6-C1-O7:H17,O7,: base rotation=0.974028 free=0
ANGLE :C1-C2-H13:H13,: base rotation=0.974028 free=0
ANGLE :C3-C2-H13:H13,: base rotation=0.974028 free=0
ANGLE :C2-C3-H14:H14,: base rotation=0.974028 free=0
ANGLE :C4-C3-H14:H14,: base rotation=0.974028 free=0
ANGLE :C5-C4-N8:C9,H18,C11,H19,H20,O10,H12,N8,: base rotation=0.795221 free=0
ANGLE :C3-C4-N8:C9,H18,C11,H19,H20,O10,H12,N8,: base rotation=0.806498 free=0
ANGLE :C4-C5-H15:H15,: base rotation=0.974028 free=0
ANGLE :C6-C5-H15:H15,: base rotation=0.974028 free=0
ANGLE :C5-C6-H16:H16,: base rotation=0.974028 free=0
ANGLE :C1-C6-H16:H16,: base rotation=0.974028 free=0
```

```
ANGLE :C1-O7-H17:H17,: base rotation=0.974028 free=0
ANGLE :C4-N8-C9:C9,H18,C11,H19,H20,O10,: base rotation=0.974028 free=0
ANGLE :H12-N8-C9:C9,H18,C11,H19,H20,O10,: base rotation=0.974028 free=0
ANGLE :C9-N8-H12:H12,: base rotation=0.974028 free=0
ANGLE :C4-N8-H12:H12,: base rotation=0.974028 free=0
ANGLE :O10-C9-C11:H18,C11,H19,H20,: base rotation=0.974028 free=0
ANGLE :C11-C9-O10:O10,: base rotation=0.974028 free=0
ANGLE :N8-C9-C11:H18,C11,H19,H20,: base rotation=0.974028 free=0
ANGLE :N8-C9-O10:O10,: base rotation=0.974028 free=0
ANGLE :C9-C11-H18:H18,: base rotation=0.974028 free=0
ANGLE :C9-C11-H19:H19,: base rotation=0.974028 free=0
ANGLE :C9-C11-H20:H20,: base rotation=0.974028 free=0
ANGLE :H19-C11-H18:H18,: base rotation=0.974028 free=0
ANGLE :H18-C11-H19:H19,: base rotation=0.974028 free=0
ANGLE :H20-C11-H18:H18,: base rotation=0.974028 free=0
ANGLE :H18-C11-H20:H20,: base rotation=0.974028 free=0
ANGLE :H20-C11-H19:H19,: base rotation=0.974028 free=0
ANGLE :H19-C11-H20:H20,: base rotation=0.974028 free=0
TORSION :C1-O7:H17,: base rotation=9.74028 freeFinished Run #0, final cost=
12274.76, nbTrial=1000000 (dt=92.9s)
Finished Run #1, final cost=    12262.79, nbTrial=1000000 (dt=96.5s)
Finished Run #2, final cost=    12885.86, nbTrial=1000000 (dt=91.7s)
Finished Run #3, final cost=    12286.24, nbTrial=1000000 (dt=88.0s)
Finished Run #4, final cost=    22426.92, nbTrial=1000000 (dt=97.9s)
Finished Run #5, final cost=    12270.93, nbTrial=1000000 (dt=95.1s)
Finished Run #6, final cost=    12280.57, nbTrial=1000000 (dt=90.8s)
Finished Run #7, final cost=    12279.84, nbTrial=1000000 (dt=93.2s)
Finished Run #8, final cost=    12307.63, nbTrial=1000000 (dt=93.7s)
Finished Run #9, final cost=    12241.67, nbTrial=1000000 (dt=94.3s)
Finished Run #10, final cost=    12264.57, nbTrial=1000000 (dt=94.3s)
Finished Run #11, final cost=    12263.58, nbTrial=1000000 (dt=93.5s)
Finished Run #12, final cost=    12315.80, nbTrial=1000000 (dt=98.7s)
Finished Run #13, final cost=   100980.35, nbTrial=1000000 (dt=117.7s)
Finished Run #14, final cost=    12251.82, nbTrial=1000000 (dt=108.0s)
Finished Run #15, final cost=    12310.40, nbTrial=1000000 (dt=115.3s)
Finished Run #16, final cost=    11736.76, nbTrial=1000000 (dt=108.2s)
Finished Run #17, final cost=    12336.40, nbTrial=1000000 (dt=117.5s)
Finished Run #18, final cost=    11775.02, nbTrial=1000000 (dt=97.9s)
Finished Run #19, final cost=    11802.47, nbTrial=1000000 (dt=99.9s)
Finished Run #20, final cost=    12266.16, nbTrial=1000000 (dt=101.4s)
Finished Run #21, final cost=    12285.45, nbTrial=1000000 (dt=93.0s)
Finished Run #22, final cost=    12282.70, nbTrial=1000000 (dt=109.0s)
Finished Run #23, final cost=    12285.24, nbTrial=1000000 (dt=127.8s)
Finished Run #24, final cost=    12286.61, nbTrial=1000000 (dt=103.9s)
Finished Run #25, final cost=    12236.44, nbTrial=1000000 (dt=129.3s)
Finished Run #26, final cost=    12304.94, nbTrial=1000000 (dt=137.0s)
Finished Run #27, final cost=    12277.15, nbTrial=1000000 (dt=114.6s)
Finished Run #28, final cost=    12178.33, nbTrial=1000000 (dt=141.2s)
```

```
Finished Run #29, final cost=    94458.54, nbTrial=1000000 (dt=144.1s)
Finished Run #30, final cost=    11889.44, nbTrial=1000000 (dt=138.3s)
Finished Run #31, final cost=    12257.72, nbTrial=1000000 (dt=131.9s)
Finished Run #32, final cost=    11974.16, nbTrial=1000000 (dt=125.8s)
Finished Run #33, final cost=    12260.75, nbTrial=1000000 (dt=158.6s)
Finished Run #34, final cost=    12262.73, nbTrial=1000000 (dt=160.3s)
Finished Run #35, final cost=    12277.05, nbTrial=1000000 (dt=140.0s)
Finished Run #36, final cost=    12333.55, nbTrial=1000000 (dt=122.6s)
Finished Run #37, final cost=    12334.75, nbTrial=1000000 (dt=129.0s)
Finished Run #38, final cost=    12168.59, nbTrial=1000000 (dt=146.0s)
Finished Run #39, final cost=    99617.29, nbTrial=1000000 (dt=174.7s)
Finished Run #40, final cost=    12253.05, nbTrial=1000000 (dt=169.4s)
Finished Run #41, final cost=    12173.91, nbTrial=1000000 (dt=151.0s)
Finished Run #42, final cost=    12288.11, nbTrial=1000000 (dt=121.2s)
Finished Run #43, final cost=    12269.27, nbTrial=1000000 (dt=127.2s)
Finished Run #44, final cost=    12290.30, nbTrial=1000000 (dt=180.8s)
Finished Run #45, final cost=    12307.08, nbTrial=1000000 (dt=164.5s)
Finished Run #46, final cost=    12279.14, nbTrial=1000000 (dt=142.4s)
Finished Run #47, final cost=   101072.54, nbTrial=1000000 (dt=200.9s)
Finished Run #48, final cost=    12249.18, nbTrial=1000000 (dt=193.9s)
Finished Run #49, final cost=    12376.98, nbTrial=1000000 (dt=189.3s)
Finished Run #50, final cost=    12266.93, nbTrial=1000000 (dt=127.5s)
Finished Run #51, final cost=    99324.51, nbTrial=1000000 (dt=152.2s)
Finished Run #52, final cost=    12268.92, nbTrial=1000000 (dt=155.1s)
Finished Run #53, final cost=    12135.75, nbTrial=1000000 (dt=143.8s)
Finished Run #54, final cost=    94651.76, nbTrial=1000000 (dt=192.8s)
Finished Run #55, final cost=    12304.91, nbTrial=1000000 (dt=192.0s)
Finished Run #56, final cost=    12215.08, nbTrial=1000000 (dt=134.3s)
Finished Run #57, final cost=    12294.29, nbTrial=1000000 (dt=169.8s)
Finished Run #58, final cost=    12268.78, nbTrial=1000000 (dt=141.4s)
Finished Run #59, final cost=    12288.41, nbTrial=1000000 (dt=138.2s)
Finished Run #60, final cost=    12218.34, nbTrial=1000000 (dt=148.6s)
Finished Run #61, final cost=    12282.70, nbTrial=1000000 (dt=143.1s)
Finished Run #62, final cost=    11866.02, nbTrial=1000000 (dt=162.0s)
Finished Run #63, final cost=    12247.09, nbTrial=1000000 (dt=155.4s)
Finished Run #64, final cost=    11608.57, nbTrial=1000000 (dt=212.1s)
Finished Run #65, final cost=    12256.34, nbTrial=1000000 (dt=231.0s)
Finished Run #66, final cost=    12238.87, nbTrial=1000000 (dt=143.9s)
Finished Run #67, final cost=    12271.29, nbTrial=1000000 (dt=214.5s)
Finished Run #68, final cost=    12335.68, nbTrial=1000000 (dt=201.8s)
Finished Run #69, final cost=    13060.53, nbTrial=1000000 (dt=148.3s)
Finished Run #70, final cost=    12273.58, nbTrial=1000000 (dt=206.3s)
Finished Run #71, final cost=    12248.43, nbTrial=1000000 (dt=220.1s)
Finished Run #72, final cost=    12318.47, nbTrial=1000000 (dt=167.3s)
Finished Run #73, final cost=    12319.74, nbTrial=1000000 (dt=160.1s)
Finished Run #74, final cost=   100169.43, nbTrial=1000000 (dt=235.2s)
Finished Run #75, final cost=    94062.20, nbTrial=1000000 (dt=212.3s)
Finished Run #76, final cost=    12107.63, nbTrial=1000000 (dt=132.1s)
```

```
Finished Run #77, final cost=    12275.34, nbTrial=1000000 (dt=151.9s)
Finished Run #78, final cost=    12261.89, nbTrial=1000000 (dt=137.6s)
Finished Run #79, final cost=    12276.55, nbTrial=1000000 (dt=210.0s)
Finished Run #80, final cost=    12196.32, nbTrial=1000000 (dt=159.8s)
Finished Run #81, final cost=    12274.78, nbTrial=1000000 (dt=162.1s)
Finished Run #82, final cost=    11794.31, nbTrial=1000000 (dt=141.4s)
Finished Run #83, final cost=    94467.11, nbTrial=1000000 (dt=212.8s)
Finished Run #84, final cost=    12664.13, nbTrial=1000000 (dt=148.0s)
Finished Run #85, final cost=    12288.39, nbTrial=1000000 (dt=166.3s)
Finished Run #86, final cost=    12299.07, nbTrial=1000000 (dt=146.9s)
Finished Run #87, final cost=    11464.15, nbTrial=1000000 (dt=161.5s)
Finished Run #88, final cost=   112315.12, nbTrial=1000000 (dt=211.7s)
Finished Run #89, final cost=    12304.39, nbTrial=1000000 (dt=152.0s)
Finished Run #90, final cost=    11788.38, nbTrial=1000000 (dt=247.2s)
Finished Run #91, final cost=    12323.68, nbTrial=1000000 (dt=238.7s)
Finished Run #92, final cost=    12249.75, nbTrial=1000000 (dt=169.7s)
Finished Run #93, final cost=    12300.14, nbTrial=1000000 (dt=142.0s)
Finished Run #94, final cost=    12430.09, nbTrial=1000000 (dt=159.7s)
Finished Run #95, final cost=    12331.02, nbTrial=1000000 (dt=160.5s)
Finished Run #96, final cost=    12294.75, nbTrial=1000000 (dt=265.5s)
Finished Run #97, final cost=    12201.06, nbTrial=1000000 (dt=227.2s)
Finished Run #98, final cost=    12272.53, nbTrial=1000000 (dt=194.8s)
Finished Run #99, final cost=    12283.33, nbTrial=1000000 (dt=148.8s)
```

[14]:
```
if (algorithm == 1):
    for i in range(NbRun):
        mc.RestoreParamSet(i, update_display=True)
        c.CIFOutput(f"results/PT/results_Paracetamol_PT_{i:02d}.cif")
elif (algorithm == 2):
    for i in range(NbRun):
        mc.RestoreParamSet(i, update_display=True)
        c.CIFOutput(f"results/PSO/results_Paracetamol_PSO_{i:02d}.cif")
```