Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Návrh a implementace open source knihovny pro kompresi molekulárních trajektorií

Plzeň 2023

Vít Kovarovič

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Vít KOVAROVIČ**
Osobní číslo:	**A19B0099P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Návrh a implementace open source knihovny pro kompresi molekulárních trajektorií**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Prozkoumejte možnosti vývoje multiplatformních knihoven v jazyce C++ a jejich integrace do projektů psaných v jazyce Python.
2. Prostudujte kompresní metodu Predictive compression of molecular dynamics trajectories, kterou bude knihovna implementovat, a její existující prototypové řešení.
3. Navrhněte strukturu a rozhraní knihovny i s ohledem na nejnovější možnosti jazyka C++.
4. Implementujte knihovnu v jazyce C++ s důrazem na přehlednost kódu a demonstrujte její možnost integrace do jednoduchého programu v jazyce Python.
5. Knihovnu důkladně zdokumentujte.
6. Ověřte funkcionalitu knihovny a to včetně porovnání kompresních výsledků s těmi dosaženými původní prototypovou implementací metody.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Ing. Jan Dvořák**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **3. října 2022**
Termín odevzdání bakalářské práce: **4. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2022

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 4. května 2023

Vít Kovarovič

Abstract

The goal of this thesis is to design and implement a cross-platform, open-source library in C++ that implements the *Predictive Molecule Compression* method for compression of molecular trajectory data and to demonstrate its integration in a Python project.

The first part analyzes options for cross-platform software development and lists technologies that enable the integration of libraries written with C++ in projects written in other languages, especially Python. Furthermore, it describes the principles of the PMC method from the perspective of its implementation.

The second part describes the design of the module structure and interface of the library and details its implementation and build system.

Finally, the compression results, execution time, and memory footprint of the library are experimentally measured and compared with the results of the previously existing prototype.

Abstrakt

Cílem této práce je navrhnout a implementovat v jazyce C++ multiplatformní, open-source knihovnu, která implementuje metodu *Predictive Molecule Compression* pro kompresi molekulárních trajektorií, a demonstrovat její integraci v projektu napsaném v jazyce Python.

První část práce analyzuje možnosti vývoje multiplatformního software a technologie, které umožňují využití knihoven napsaných v C++ v jiných jazycích, především v jazyku Python. Dále popisuje fungování kompresní metody PMC pro potřeby její implementace.

Druhá část popisuje návrh struktury a rozhraní knihovny a detaily její implementace a procesu sestavení.

V závěru jsou pomocí experimentu srovnány kompresní, časové a paměťové výsledky implementované knihovny a již existujícího prototypu.

Obsah

1 Úvod **1**

2 Možnosti vývoje knihovny **2**
2.1 Multiplatformní software 2
2.2 Integrace 4
2.2.1 Dynamická knihovna 4
2.2.2 Volání dynamické knihovny 6
2.2.3 Technologie pro tvorbu bindingu 7
2.3 Licence 11
2.3.1 Open-source software 11
2.3.2 Free software 11

3 Molekulární trajektorie **13**
3.1 Popis dat 13
3.2 Popis implementované metody 14
3.2.1 Kanonická molekula 14
3.2.2 Kódování molekuly 17

4 Limitace prototypového řešení **18**

5 Návrh knihovny **19**
5.1 Zvolené technologie 19
5.2 Struktura a rozhraní 20

6 Popis implementace **22**
6.1 Programová dokumentace 22
6.2 Systém sestavování 26
6.3 Testování kódu 27
6.4 Referenční dokumentace kódu 28

7 Validace kompresních výsledků **29**

8 Závěr **32**

Literatura **33**

A Kompletní výsledky měření **39**

1 Úvod

Cílem této práce je navrhnout a implementovat knihovnu s veřejně dostupným kódem, která implementuje metodu komprese molekulárních trajektorií publikovanou v roce 2020 pracovníky Katedry informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni v časopise *Journal of Molecular Graphics and Modelling* [23].

Molekulární trajektorie jsou výsledkem simulace molekulární dynamiky a představují pozice jednotlivých atomů v čase. Množství těchto dat pro reálnou simulaci může být značné, dosahujíc objemu až v řádu desítek gigabajtů. Z tohoto důvodu je nadmíru užitečné zredukovat velikost dat pomocí komprese. Způsoby, jak existující kompresní metody snižují velikost dat, zahrnují například redukci přesnosti souřadnic atomů, kódování rozdílů mezi atomy namísto absolutních pozic nebo aproximaci trajektorií vrcholů pomocí polynomů. Kompresní metoda *Predictive Molecule Compression*, kterou má tato práce za cíl implementovat, přináší inovaci v podobě toho, že využívá znalost vazeb mezi atomy v původních datech k tomu, aby separovala nejvýznamnější pohyby v molekule od náhodných, lokálních pohybů.

Tato práce se může opřít o již existující prototyp implementace této kompresní metody [46], avšak ta není vhodná pro veřejné využití v praxi, protože její zdrojový kód je nepřehledný a málo zdokumentovaný a protože neposkytuje hotové rozhraní, které by umožnilo snadnou integraci do jiných programů psaných v různých jazycích. Hlavním požadavkem na novou implementaci je tedy napravit tyto nedostatky a poskytnout knihovnu, která klade důraz na přehlednost kódu a důkladnou dokumentaci a která poskytuje uživatelům možnosti, jak ji integrovat do jiných programů, zejména psaných v jazycích C a Python.

Nejprve bude provedena analýza možností vývoje multiplatformních knihoven v jazyce C++ a analýza problematiky molekulárních trajektorií ve vztahu k fungování kompresní metody, kterou má tato práce implementovat. Poté následuje návrh struktury a rozhraní knihovny a popis její implementace. Nakonec na základě experimentů proběhne srovnání nově implementované knihovny a předchozí implementace.

2 Možnosti vývoje knihovny

Softwarová knihovna je souhrn hotových funkcí, tříd, datových typů atd., který může vývojář připojit ke svému programu, aby získal nějakou funkcionalitu, bez toho, aby sám napsal její implementaci [48]. Kromě toho knihovny umožňují znovupoužití kódu, kdy stejnou knihovnu můžeme použít v mnoha programech, nebo poskytují cestu, jak dosáhnout větší rychlosti, pokud knihovnu napsanou v kompilovaném jazyce, jako je například C nebo C++, použijeme v programu napsaném v jazyce interpretovaném, jako je například Python.

2.1 Multiplatformní software

Termín *platforma* může v kontextu software znamenat procesorovou architekturu, na které daný software běží, operační systém, pod kterým daný software běží, nebo konkrétní dvojici procesorové architektury a operačního systému [14]. Multiplatformní software (též přenositelný) je takový software, který je připraven tak, aby mohl být zkompilován ze stejného zdrojového kódu a následně spuštěn na alespoň dvou různých platformách.

Zkompilovat software na různých procesorových architekturách je především otázka nástrojů, tj. zda existuje překladač daného jazyka pro danou procesorovou architekturu, což je externí faktor, který nemůžeme ovlivnit změnou zdrojového kódu našeho programu. Na co si však musíme dát pozor, jsou různé standardy překladačů na různých architekturách. Například, překladač GCC jazyka C na platformě x86 definuje datový typ `char` jako 8bitové číslo se znaménkem, ale na platformě ARM je `char` definován jako 8bitové číslo bez znaménka [1]. To znamená, že pokud na platformě ARM zkompilujeme program, který pro svojí funkčnost předpokládá, že `char` je číslo se znaménkem, bude se program chovat jinak, než jsme očekávali. Následující je jednoduchý příklad programu, který se bude chovat jinak než očekáváme. Na platformě x86 se vypíše pozdrav ale na platformě ARM nikoli, protože proměnná `c` podteče a její hodnota bude kladná.

```
char c = -1;
if (c < 0)
{
    printf("Hello! \n");
}
```

Možným řešením je program zkompilovat s nastavením `-fsigned-char` překladače GCC, které zaručí chování, jaké očekáváme, ale lepší řešení je vyhnout se zvláštnostem jednotlivých architektur a kompilátorů a raději použít standardně definované datové typy jako například ty z hlavičkového souboru `cstdint`.

Zda je software přenositelný na různé operační systémy, závisí výhradně na zdrojovém kódu daného software. Základním předpokladem pro to, aby byl software přenositelný, je vyhnout se přímému používání prostředků, které jsou specifické pro jednotlivé operační systémy [37] (například POSIX API na straně systémů založených na Unixu a Win32 API na straně Microsoft Windows) a místo nich použít prostředky standardní knihovny jazyka, popřípadě prostředky nějaké externí multiplatformní knihovny. Typickým příkladem je vytvoření souboru pomocí funkce `fopen` standardní knihovny jazyka C namísto POSIXového volání `open` nebo funkce `CreateFile` z Win32 API.

Mohou nastat situace, kdy nemůžeme, nebo nechceme, nahradit nativní prostředky daného operačního systému za multiplatformní alternativy. V takové situaci můžeme v jazycích C a C++ využít symboly preprocesoru, které jsou definované pouze, pokud probíhá překlad na dané platformě. Následující je příklad toho, jak můžeme program rozdělit na dvě větve, z nichž jedna se provede pouze na operačních systémech Microsoft Windows a druhá na všech ostatních systémech:

```
#ifdef _WIN32
    // Systémy Windows
#else
    // Ostatní systémy
#endif
```

Velké množství větví v kódu se může stát nepřehledné [37]. Proto, pokud je tak možné učinit, je vhodné výše uvedené větvení zaobalit do funkce, která bude reprezentovat multiplatformní abstrakci dané funkcionality a tu používat napříč celým kódem.

Typickou vlastností multiplatformního software je podpora více překladačů. Některé překladače jsou typicky, byť ne výhradně, spjaty s některými operačními systémy (Clang na systémech společnosti Apple, GCC na systémech GNU/Linux, MSVC na systémech Windows, atd.) a jejich podpora umožňuje využití sady nástrojů, která je rozšířená na konkrétním systému. Aby software podporoval více překladačů, musí se vyhnout využívání rozšíření, funkcí, maker a dalších vlastností, které nejsou součástí standardu jazyka, ale jsou exkluzivní pro jednotlivé překladače [37]. Zároveň každý překladač vydává trochu jiná varování a generuje trochu jiný kód, což dává

příležitosti nalézt při vývoji problémy a chyby, na které bychom jinak nemuseli narazit.

V neposlední řadě musí multiplatformní software vyřešit automatizaci sestavení. Tuto roli běžně plní nástroje založené na *make* jako například *gmake* [10], *nmake* [12], *fmake* [9] a další, popřípadě alternativy jako například *Ninja* [13] nebo projektové soubory Visual Studia [11]. Všechny tyto nástroje ale cílí na svoje konkrétní platformy a jejich vstupní *Makefile* se od sebe liší, takže nejsou navzájem kompatibilní. Řešením je použít vysokoúrovňový generátor sestavení jako například *CMake* [7], *SCons* [15] nebo *meson* [43]. Tyto generátory definují konfigurační soubory, kterými lze popsat projekt a způsob jeho sestavení. Následně podle této konfigurace vygenerují na míru právě takový *Makefile* (nebo ekvivalent), o který si uživatel vstupním argumentem řekne.

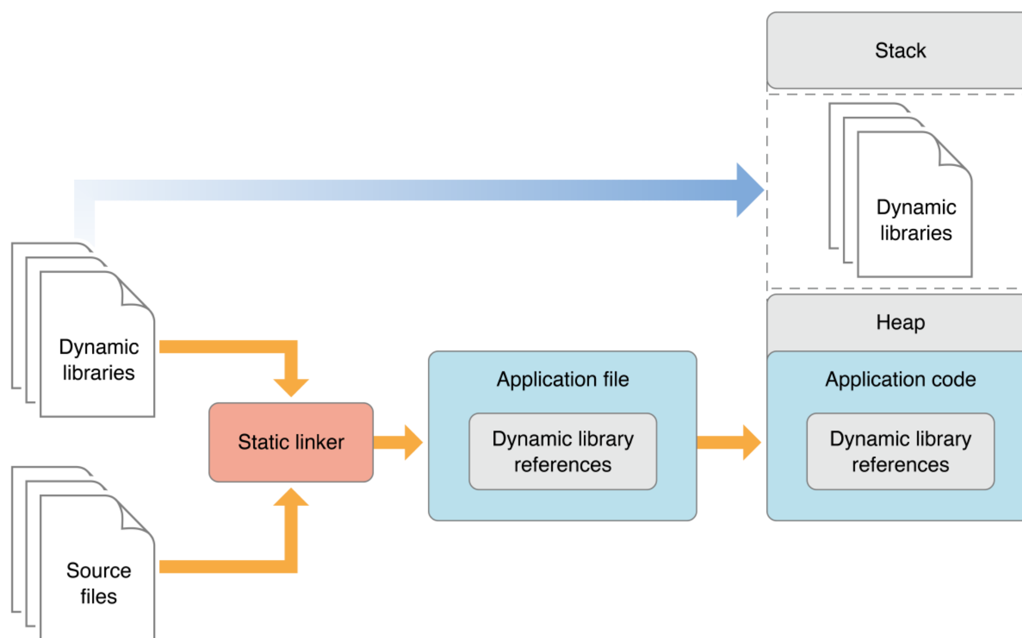
2.2 Integrace

Knihovny můžeme dělit na *statické* a *dynamické* podle toho, jakým způsobem se připojují (linkují) k programu [32]. Statické knihovny se k programu linkují v čase kompilace a stávají se neoddělitelnou součástí spustitelného binárního souboru. Naproti tomu dynamické knihovny po slinkování zůstávají oddělené a zavádí se do paměti za běhu aplikace. Díky této podstatě je možné dynamickou knihovnu použít jako modulární rozšíření do již existujících aplikací. Integrace knihovny do jiných projektů je jedním z bodů zadání této práce, budeme se tedy dále v textu věnovat jen dynamickým knihovnám.

2.2.1 Dynamická knihovna

Dynamická knihovna je samostatný soubor, který obsahuje kód a data, které může ve stejnou chvíli využít jeden či více programů. Využití dynamických knihoven snižuje velikost spustitelného souboru a snižuje nároky na operační paměť, viz obr. 2.1. Do paměti se zavádí buď při spuštění programu, nebo dokonce až za jeho běhu teprve v momentě, kdy program potřebuje využít funkcionalitu knihovny [3].

Implementace dynamických knihoven se na různých operačních systémech mírně liší. Na systémech založených na Unixu jsou dynamické knihovny známé jako *sdílené knihovny* (anglicky *shared library*, též *shared object*) a mají příponu *.so*. Kromě této přípony také musí jejich název povinně začínat předponou *lib*, například *libmylibrary.so*. Dobrovolné je verzování



Obrázek 2.1: Diagram aplikace, která využívá dynamické knihovny, v paměti (zdroj: [3])

knihovny, přičemž číslo verze knihovny patří na konec, například *libmylibrary.so.1.1*. Verzování knihoven umožňuje linkovat program vůči konkrétní verzi knihovny, díky čemuž nepřestane fungovat, i když hlavní instalace knihovny obdrží aktualizaci [62]. Soubor *.so* obsahuje jak samotný kód, tak symboly všech funkcí, které poskytuje ostatním programům, aby linker mohl knihovnu připojit k programu, který ji využívá.

Zkompilovat existující software jako sdílenou knihovnu na Unixu je přímočaré – v překladači GCC k tomu stačí přepínače `-shared`, který kód zkompiluje jako dynamickou knihovnu a `-fPIC`, který zajistí, že vygenerovaný strojový kód nebude závislý na tom, aby se nacházel na konkrétních adresách v paměti (což je nutná podmínka pro kompilaci dynamické knihovny) [2]. Konkrétně například:

```
gcc -fPIC -shared mylib.c -o libmylib.so
```

V operačním systému Microsoft Windows mají dynamické knihovny příponu *.dll* (z anglického Dynamic Link Library [4]). Systém Windows používá mnoho různých *.dll* souborů, pomocí kterých poskytuje funkce operačního systému nebo grafické rozhraní. Příklady takových systémových dynamických knihoven jsou například *KERNEL32.dll*, *GDI32.dll* nebo *USER32.dll*. Na rozdíl od Unixových sdílených knihoven obsahují *.dll* soubory pouze samotný zkompilovaný kód knihovny. Symboly, které jsou nutné ke slinko-

vání programu ke knihovně, se nacházejí v samostatném souboru, takzvané *knihovně importu* (anglicky *import library*), která má příponu *.lib*. Každá *.dll* knihovna musí mít svůj odpovídající *.lib* soubor, jinak ji nebude možné přilinkovat [4].

Zkompilovat existující program jako dynamickou knihovnu na Windows je složitější než na Unixových systémech. Systém Windows vyžaduje, aby všechny funkce, které knihovna exportuje, byly dekorovány volacím atributem *__declspec(dllexport)* [39]. Podobně, když chce program využít funkci dynamické knihovny, vyžaduje systém Windows, aby tato funkce byla v hlavičkovém souboru dekorována volacím atributem *__declspec(dllimport)* [39]. Aby byly splněny tyto podmínky, ale zároveň bylo rozhraní knihovny definováno ve stejných hlavičkových souborech jak pro import, tak pro export, používá se většinou podobný konstrukt preprocesoru:

```
#ifdef mylib_EXPORTS
    #define MYLIB_EXPORT __declspec(dllexport)
#else
    #define MYLIB_EXPORT __declspec(dllimport)
#endif
```

kde symbol `mylib_EXPORTS` se definuje (nebo nedefinuje) v čase kompilace, například takto:

```
gcc -Dmylib_EXPORTS -c mylib.c
```

což vytvoří odpovídající objektový soubor (*mylib.o*). Z něj se posléze zkompiluje samotný *.dll* soubor a jeho odpovídající knihovna importu, například takto:

```
gcc -shared -o mylib.dll mylib.o -Wl,--out-implib,mylib.lib
```

kde přepínač `-Wl,-out-implib` specifikuje, že se má vytvořit knihovna importu.

2.2.2 Volání dynamické knihovny

Standardní dynamickou knihovnu je možné zavolat takřka z jakéhokoli programovacího jazyka. Vzhledem k zadání této práce se zaměříme na jazyky C a Python. V C stačí k použití knihovny vložit do zdrojového souboru hlavičkový soubor knihovny, aby o jejích funkcích věděl překladač, a linkeru předat cestu k binárnímu souboru knihovny. Takový příkaz může vypadat například takto:

```
gcc -o main.exe main.c -L. -lmylib
```

kde tečka za přepínačem `-L` je adresář, ve kterém se budou hledat knihovny a `mylib` je název knihovny.

V jazyce Python umožňuje použití běžné dynamické knihovny modul *ctypes* ze standardní knihovny jazyka [45]. Dynamická knihovna se načte funkcí `cdll.LoadLibrary(path)`, kde `path` je řetězec, který obsahuje cestu ke knihovně v souborovém systému uživatele. Návratovou hodnotou této funkce je objekt, který ve zbytku skriptu poskytuje přístup k metodám, které jsou ve skutečnosti voláním funkcí z načtené knihovny.

Nevýhodou používání běžných dynamických knihoven v interpretovaném jazyce jako Python je, že jakékoli informace o knihovně jsou známé až v čase běhu programu. To znamená, že statická analýza vývojového prostředí nemá způsob, jak uživateli poradit, a uživatel je odkázán pouze na dokumentaci knihovny. Proto je pro účely použití knihovny v Pythonu vhodnější ji sestavit jako nativní Python modul, který se importuje běžnými prostředky jazyka. Projektu, který obaluje existující knihovnu a sestavuje jí jako Python modul, se často říká *vázání* (anglicky *binding*)[6].

2.2.3 Technologie pro tvorbu bindingu

Python/C API

Python/C API [44] je rozhraní pro přístup k interpretu Pythonu z jazyků C a C++, které se nachází v hlavičkovém souboru `Python.h`. Jedna z možností, které toto rozhraní nabízí, je zkompilovat rozšiřující modul (anglicky *extension module* pro Python.

Základem modulu jsou funkce, které modul nabízí. Následující je příklad jednoduchého bindingu funkce:

```
PyObject* func_python(PyObject* self, PyObject* input)
{
    double input_C = PyFloat_AsDouble(input);
    double result = func_C(input_C);
    return PyFloat_FromDouble(result);
}
```

Vstupem i výstupem z funkce jsou ukazatele na `PyObject`, což je struktura reprezentující obecný objekt v Pythonu. Pro práci v C je nutné tyto struktury převést na něco jiného. Součástí API jsou funkce na převod jednoduchých datových typů jako je například `double`, ale pokud je vstupem/výstupem něco složitějšího, například instance vlastní třídy, musí si uživatel sám napsat funkci, která převod provede.

Modul musí obsahovat definici pole, které pro interpret Pythonu deklaruje všechny funkce, které modul obsahuje. Pro příklad funkce, který byl uveden v textu výše, bude takové pole vypadat například takto:

```
static PyMethodDef binding_methods[] = {
    {"func_python", (PyCFunction)func_python, METH_O, NULL },
    {NULL, NULL, 0, NULL }
};
```

První atribut deklarace je název funkce, druhý je ukazatel na funkci, která obsahuje implementaci, třetí je příznak, která vyjadřuje, jak se mají zpracovávat vstupní argumenty funkce (v příkladu je `METH_O`, což je nejjednodušší způsob exkluzivní pro funkce, které mají pouze jeden vstupní argument) a čtvrtý je řetězec obsahující dokumentační komentář. Pole je povinně zakončené ukončovací prvkem, který obsahuje samé nulové hodnoty.

Dále musí modul kromě definic a implementací všech svých funkcí obsahovat také definici sebe sama, například:

```
static PyModuleDef binding_module = {
    PyModuleDef_HEAD_INIT,
    "binding_example",
    NULL,
    0,
    binding_methods
};
```

Nakonec je nutné definovat funkci, kterou interpret Pythonu zavolá v momentě, kdy načte modul.

```
PyMODINIT_FUNC PyInit_binding_example()
{
    return PyModule_Create(&binding_module);
}
```

Název funkce za `PyInit_` musí přesně odpovídat názvu výsledného zkompilovaného modulu (*.pyd* souboru).

Python API v hlavičkovém souboru `Python.h` představuje primární přístup k rozhraní jazyku Python. Výhodami je, že udržovatelé Pythonu garantují funkčnost pro jakoukoli verzi Pythonu a že nezávisí na žádném software třetí strany kromě Pythonu samotného. Nevýhodou je, že API je psané v jazyku C a tudíž poskytuje velice nízkoúrovňový přístup k tvorbě modulů. Kromě definování samotných funkcí je také potřeba režie ke konverzi vstupních a výstupních argumentů a také inicializace modulu samotného.

Boost.Python

Boost.Python [16] je součást balíku C++ knihoven *Boost* a poskytuje prostředky k propojení C++ a Pythonu, včetně možnosti obalit existující C++ rozhraní a vytvořit Python modul. *Boost.Python* využívá makra a šablony ke generování modulu. Jediné, co musí uživatel udělat, je nadefinovat binding funkcí a typů/tříd. Následující je příklad bindingu jednoduché funkce:

```
#include <boost/python.hpp>

BOOST_PYTHON_MODULE(binding_module)
{
    boost::python::def("func_python", func_C);
}
```

Výhodou knihovny *Boost.Python* je její jednoduchý syntax a široká kompatibilita se standardy jazyka C++. Nevýhodou je její závislost na zbytku balíku *Boost*, který je velmi rozsáhlý a složitý.

pybind11

pybind11 [34] je C++ knihovna, která vytváří binding existujícího C++ kódu do jazyka Python. Následující je příklad bindingu jednoduché funkce:

```
#include <pybind11/pybind11.h>

PYBIND11_MODULE(binding_module, m)
{
    m.def("func_python", &func_C, "Example function");
}
```

Syntaxem a fungováním je podobná knihovně *Boost.Python*, ale na rozdíl od ní je zcela samostatná a závisí pouze na standardní knihovně jazyka C++ ve standardu C++ 11. Další výhodou je potenciál pro mírně rychlejší čas kompilace a menší velikost zkompilovaného modulu, než při použití *Boost.Python*, jako bylo změřeno [60] při konverzi projektu *PyRosetta* z *Boost.Python* na *pybind11*.

nanobind

nanobind [57] je nejnovější C++ knihovna pro tvorbu bindingu do jazyka Python od autora *pybind11* vydaná v březnu roku 2023. Knihovna využívá

novějších verzí jazyka C++ a Python než pybind11 (C++17 a Python 3.8). Následující je příklad bindingu jednoduché funkce:

```
#include <nanobind/nanobind.h>

NB_MODULE(binding_module, m)
{
    m.def("func_python", &func_C, "Example function");
}
```

Syntaxí je takřka totožná s knihovnou *pybind11* (až na kosmetické změny), ale zásadní rozdíl je v kompatibilitě. Zatímco *pybind11* cílí na podporu veškerého existujícího C++ kódu, *nanobind* se zaměřuje na podporu jen určité podmnožiny C++, díky čemuž může být rychlejší. Jako příklad autor uvádí podporu vícenásobné dědičnosti, která byla velkým zdrojem komplexity pro *pybind11* a která v *nanobindu* už není podporovaná [59]. Podle autora se mohou moduly pomocí *nanobindu* kompilovat až 4x rychleji a mohou být až 5x menší [58].

NumpyEigen

NumpyEigen [63] je C++ knihovna, která vytváří binding pro funkce, které využívají objekty z knihoven pro vědecké výpočty jako jsou NumPy [28] nebo SciPy [56]. Tyto objekty jsou v rámci bindingu převedeny na ekvivalenty v Eigenu, což je knihovna pro lineární algebru v C++ [27].

NumpyEigen využívá pro svojí implementaci *pybind11*, ale obsahuje svůj vlastní systém pro konverzi objektů, pro který využívá vyšší standard jazyka C++ než samotný *pybind11* (standard C++ 14). Pro uživatele má *NumpyEigen* také úplně jiný syntax než *pybind11*. Následující je příklad bindingu jednoduché funkce:

```
#include <npe.h>

npe_function(create_tuple)
npe_arg(in1, dense_float, dense_double)
npe_arg(in2, npe_matches(in1))
npe_begin_code()
{
    return std::make_tuple(npe::move(in1), npe::move(in2));
}
npe_end_code()
```

2.3 Licence

Pokud chceme nějaký software vydat veřejně, musíme řešit problematiku licencí. Podle evropského práva [8] jsou počítačové programy jako duševní výtvory chráněny autorskými právy stejně jako literární a umělecká díla. Existence těchto autorských práv znamená, že autor má podle nich právo regulovat licenčním ustanovením šíření, rozmnožování nebo pozměňování svého programu. Typický komerční software je *proprietární*, což znamená, že je poskytován s licencí, která uživateli umožňuje jej používat, ale zakazuje jej modifikovat a dále šířit [22]. Z takových licencí pak vyplývá, že k proprietárnímu software není volně dostupný zdrojový kód, čímž se stává obchodním tajemstvím. Příslušné zákony stanovují, že pokud k software není poskytnuta žádná licence, jsou všechna práva vyhrazena. To znamená, že pokud chce autor umožnit ostatním, aby mohli legálně jeho program využívat nebo dokonce upravovat, musí jim k tomu poskytnout odpovídající licenci.

2.3.1 Open-source software

Open-source software definuje organizace Open Source Initiative a lze jej chápat jako software poskytovaný s takovou licencí, která umožňuje program volně šířit a modifikovat a tyto modifikace dále šířit, a ke kterému je k tomuto účelu volně dostupný zdrojový kód [52].

Mezi nejpopulárnějšími [53] open-source licencemi jsou tzv. permisivní licence. Tyto licence poskytují práva šířit a modifikovat daný software bez jakýchkoli omezení nebo nároků, včetně možnosti jej použít jako součást proprietárního software. Jedinou podmínkou, které tyto licence držiteli kladou, je povinnost distribuovat kopii licenčního ujednání v původním znění. Příklady permisivních licencí jsou MIT licence [40], ISC licence [33] nebo BSD licence [19]. Existují i přísnější licence, které navíc ukládají povinnost jasně označit ty části původního software, které byly modifikovány. Příklady takových licencí jsou Apache licence [17] nebo zlib licence [38].

2.3.2 Free software

Blízký kategorii open-source softwaru je *free software* (česky svobodný software), jehož definici spravuje organizace Free Software Foundation. Svobodný software je definován jako software, jehož uživatelé mají tyto 4 základní svobody [61]:

1. Svoboda spustit program za jakýmkoliv účelem.

2. Svoboda studovat, jak program pracuje a přizpůsobit ho svým potřebám. Nutným předpokladem této svobody je přístup ke zdrojovému kódu.

3. Svoboda redistribuovat kopie.

4. Svoboda distribuovat kopie upravených verzí programu ostatním.

Ačkoli jsou definice open-source software a svobodného software podobné, existují open-source licence, které podle FSF nesplňují podmínky pro svobodný software [55].

Protože hlavním cílem FSF je ochrana a propagace svobodného software, dala vzniknout principu tzv. *copyleft* licencí [50]. Copyleft licence ukládají povinnost zveřejnit pod stejnou copyleft licencí (včetně zdrojového kódu) jakékoli modifikace původního software, jakožto i jakýkoli jiný projekt, který by jako svojí součást využíval software, který je pod copyleft licencí. Tyto podmínky mají za následek, že jakýkoli program, který legálně využívá software pod copyleft licencí, se nutně sám stane svobodným softwarem, a znemožňují, aby software pod copyleft licencí byl využit jako součást nějakého proprietárního software (což permisivní open-source licence dovolují). Příklady copyleft licencí jsou GNU General Public License [24] a Mozilla Public License [41].

3 Molekulární trajektorie

Molekulární dynamika je výpočetní metoda, která simuluje chování modelu atomů [20]. Výsledkem této simulace jsou molekulární trajektorie, které popisují pohyby jednotlivých atomů v diskrétních časových snímcích. Získané trajektorie mohou zodpovědět některé otázky o vlastnostech modelovaného systému lépe než fyzické experimenty a jak roste výkon výpočetní techniky, roste i počet aplikací pro tyto simulace [35]. Příkladem aplikace molekulární dynamiky je simulace a studium složených proteinů [47].

3.1 Popis dat

Uchovávat vědecká data, jako jsou například molekulární trajektorie, jako prostý ASCII text je prostorově velmi neefektivní. Příčiny této neefektivity zahrnují například množství bílých znaků v textu nebo skutečnost, že typické číslo v plovoucí desetinné čárce je jako ASCII text reprezentováno na více bajtech, než by bylo v binární podobě. Proto se běžně používají binární formáty jako například NetCDF [42] nebo HDF5 [29]. Data v těchto formátech jsou ale i přesto často příliš velká, takže dalším krokem je komprese dat.

Existuje množství již používaných kompresních metod a odpovídajících souborových formátů pro uložení molekulárních trajektorií. Příkladem již existujícího rozšířeného formátu je TNG, který je součástí softwaru *GROMACS* [54] a obsahuje kombinaci souřadnic, rychlostí, sil a/nebo energií [26]. Základní princip komprese v TNG formátu je poznatek, že rozdíly v čase a prostoru mezi souřadnicemi nebo rychlostmi atomů jsou většinou menší než jejich skutečné hodnoty [49].

Kromě samotných trajektorií je také často užitečné uchovávat informace o vazbách mezi atomy v molekule. K tomu slouží například textový formát PDB [5]. Tento formát původně vznikl v roce 1976 v archivu Protein Data Bank jako lidsky čitelný formát pro děrné štítky [18], ale byl v průběhu let aktualizován až do současné verze 3.3. Protože děrné štítky měly omezenou velikost, skládá se PDB záznam z mnoha menších částí. Jedna z nich je *CONECT* – část pro popis vazeb mezi atomy. Jeden *CONECT* záznam se skládá z indexu atomu, jehož vazby popisujeme, následovaným indexy atomů, ke kterým se váže.

Pro účely návrhu knihovny nebudeme předpokládat na vstupu žádný konkrétní souborový formát. Budeme pouze očekávávat informace, které

potřebuje kompresní metoda, kterou knihovna implementuje – tedy reprezentaci molekuly jako neorientovaný graf, kde každý vrchol $\mathbf{v}_i^f$ reprezentuje pozici i-tého atomu $i \in \{1, ..., V\}$ v trojrozměrném prostoru ve snímku $f \in \{1, ..., F\}$ a každá hrana e_{ij} reprezentuje neměnnou vazbu mezi atomy i a j ve všech snímcích. Předpokládáme, že spočívá na uživateli, jak tyto informace ze svého oblíbeného formátu poskytne.

3.2 Popis implementované metody

Klíčovým aspektem kompresní metody *Predictive Molecule Compression* je, že využívá existence vazeb mezi atomy v molekule. Atomy se mezi snímky neustále drobně chvějí a otáčejí, ale tyto pohyby jsou pouze lokálního charakteru. Celkový tvar molekuly zůstává v lokálním kontextu jednoho atomu a jeho sousedních atomů obecně neměnný. Této neměnnosti se dá využít tím, že vykonstruujeme obraz molekuly, který bude zachycovat její tvar napříč všemi snímky, a ten použijeme jako referenci při kódování jednotlivých snímků. Pro tento obraz molekuly autoři metody zavedli termín *kanonická molekula* [23].

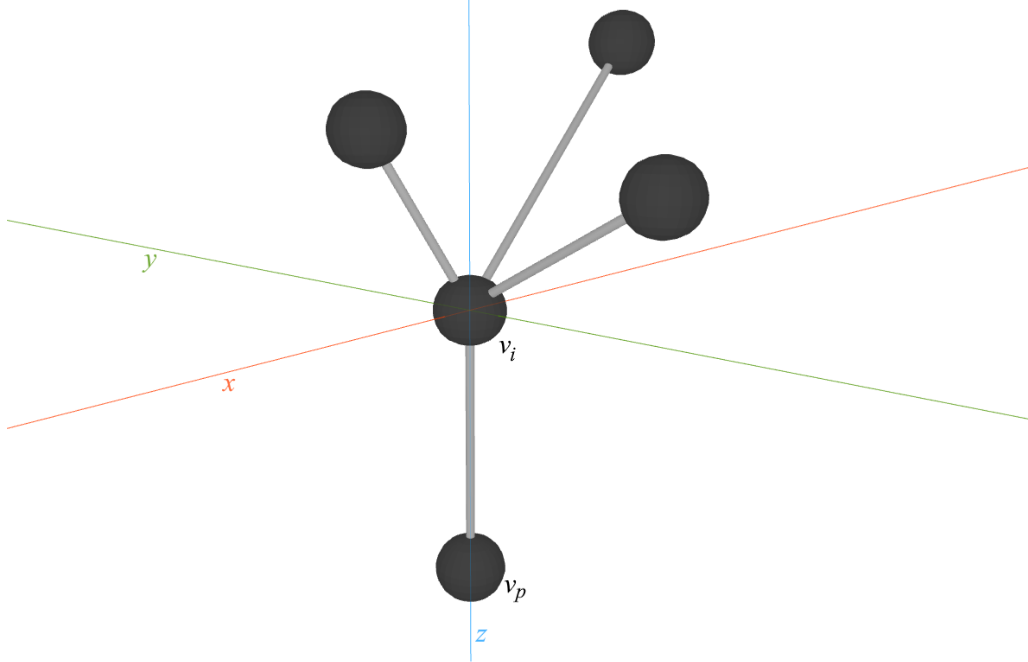
3.2.1 Kanonická molekula

Protože pro konstrukci kanonické molekuly je důležité zachytit tvar lokálního sousedství atomů, zatímco konkrétní absolutní pozice atomů není tak důležitá, můžeme kanonickou molekulu vytvářet postupně tak, že začneme jedním libovolným atomem a jeho sousedními atomy. Tyto atomy se stanou kořenem kanonické molekuly a jejich souřadnice budou stejné jako souřadnice ve vstupní molekule v libovolně zvoleném snímku.

Pozice dalších atomů v kanonické molekule se vypočítají průchodem grafu celé molekuly pomocí algoritmu prohledávání do hloubky (anglicky *Depth-First Search* [30], zkratka DFS). Pro každý zpracovávaný vrchol se nejprve v každém snímku natočí lokální sousedství tak, aby vektor $\mathbf{v}_p - \mathbf{v}_i$, kde $\mathbf{v}_i$ je pozice současně zkoumaného atomu a $\mathbf{v}_p$ je pozice posledního zkoumaného atomu, tj. jeho předchůdce, ležel na ose z a byl orientován směrem na její zápornou poloosu. Této rotace se dosáhne aplikováním transformace reprezentované maticí

$$M_{\mathbf{v}} = \begin{bmatrix} \frac{-y}{h} & \frac{x}{h} & 0 \\ \frac{zx}{h} & \frac{zy}{h} & -(\frac{x^2}{h} + \frac{y^2}{h}) \\ -x & -y & -z \end{bmatrix}, \tag{3.1}$$

kde $\mathbf{v} = (x, y, z)$ je jednotkový vektor ve směru již zmíněného vektoru $\mathbf{v}_p - \mathbf{v}_i$ a $h = \sqrt{x^2 + y^2}$, na rozdílové vektory mezi právě zpracovávaným vrcholem a každým jeho sousedním vrcholem. Výsledek rotace je ilustrován na obrázku 3.1.



Obrázek 3.1: Vizualizace zarovnání kanonického sousedství na osu Z

Poté, co jsou lokální sousedství ve všech snímcích zarovnaná ve stejné ose, můžeme spočítat podobu kanonického sousedství, tj. takovou pozici těchto atomů, která nejblíž odpovídá pozicím původních atomů napříč všemi snímky. Autoři metody toto formulují jako problém minimalizace následující energie [23]

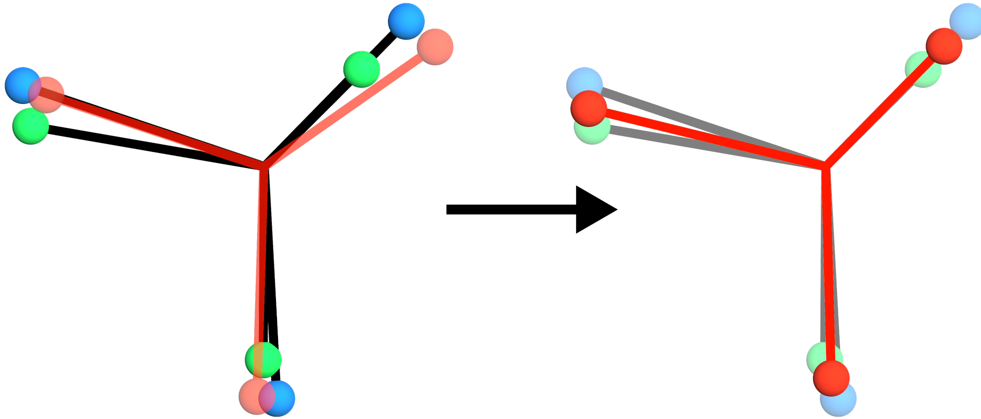
$$E = \sum_f \sum_j (\hat{v}_j - R_{\alpha f} \bar{v}_j^f)^2, \tag{3.2}$$

kde $\hat{v}_j$ je výsledná pozice sousedního atomu, $\bar{v}_j^f$ je průměr pozic původních atomů napříč všemi snímky a $R_{\alpha f}$ je tato matice pro rotační úhel α_f

$$R_{\alpha f} = \begin{bmatrix} \cos \alpha_f & -\sin \alpha_f & 0 \\ \sin \alpha_f & \cos \alpha_f & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.3}$$

a nalezli pro tento problém iterativní řešení. Začneme-li s nějakým počátečním odhadem (například pozicí sousedství v 1. snímku) a z něj vytvoříme 2D variantu matice 3.3, pomocí které přetransformujeme průměry všech sousedních atomů napříč všemi snímky, získáme řešení, které je bližší řešení rovnice

3.2, které minimalizuje energii. Na obrázku 3.2 můžeme vidět, jak se odhad kanonického sousedství (zobrazen červenou barvou) zpřesní po jedné iteraci. Podle autorů metody stačí 5 iterací, abychom získali řešení, které je velmi blízké optimálnímu řešení [23]. Po skončení všech iterací máme vhodné ka-



Obrázek 3.2: Vizualizace jedné iterace výpočtu kanonického sousedství (zdroj: [23])

nonické sousedství, ale pozice jeho atomů jsou stále natočené k ose Z. Proto musíme aplikovat opačnou transformaci k té, kterou popisuje matice 3.1, abychom získali skutečné pozice atomů v původním prostoru.

Protože informace o kanonické molekule jsou využívány při kompresi, musí být známé i při dekompresi. Proto kanonická molekula musí být součástí souboru, který je výstupem kompresní metody. Molekula je uložena diferenciálně, což znamená, že kromě počátečního vrcholu jsou všechny další vrcholy uloženy jako vzdálenosti od předchozího vrcholu, ne absolutní souřadnice. Zároveň jsou všechny hodnoty kvantizovány. Kvantizace je proces transformace spojitých hodnot nebo reálných čísel na diskrétní hodnoty. Konkrétně metoda provádí kvantizaci hodnoty x pomocí následující rovnice

$$\hat{x} = \left\lfloor \frac{x}{q} \right\rceil, \tag{3.4}$$

kde q je kvantizační konstanta, která je vstupním parametrem komprese, a $\lfloor \cdot \rceil$ je operace zaokrouhlení nad výsledkem kvantizace $\hat{x}$, čímž vzniknou zmíněné diskrétní hodnoty. Zároveň to ale znamená, že kvantizace je nevratný proces, při kterém vznikají ztráty v přesnosti dat. Těmito dvěma kroky

(ukládání vzdáleností a kvantizace) snížíme množství různých číselných hodnot v datech, čímž můžeme zvýšit efektivitu kódování dat metodami, které minimalizují entropii, jako je například Huffmanovo kódování [31].

3.2.2 Kódování molekuly

Metoda prediktivní komprese využívá kanonické molekuly, aby předpověděla pozice atomů ve všech snímcích. Tato předpověděná pozice se pak porovná se skutečnou pozicí a do výstupu se zakóduje pouze korekce předpovědi, což je rozdíl mezi těmito dvěma pozicemi. Stejně jako informace o kanonické molekule, i všechny předpovědi jsou kvantizovány rovnicí 3.4.

Molekulu zakódujeme průchodem jejího grafu pomocí algoritmu DFS. Pro každý vrchol v každém snímku se jeho lokální sousedství a lokální sousedství jeho předchůdce zarovnají na osu z pomocí transformace, kterou reprezentuje rovnice 3.1. Stejně se zarovnají i ekvivalentní sousedství v kanonické molekule. Poté se vypočítá transformace ve tvaru popsaném maticí 3.3, která natočí lokální sousedství předchůdce v kanonické molekule tak, aby odpovídalo sousedství v kódované molekule v daném snímku. Tato transformace se aplikuje na kanonické sousedství (lokální sousedství právě zkoumaného vrcholu v kanonické molekule), čímž vznikne předpověď pozice sousedství právě zkoumaného vrcholu v aktuálním snímku. Úhel, o který tato transformace natočí kanonické sousedství, se kvantizuje a uloží spolu s korekcí předpovědi. Při dekompresi dat pak stačí vypočítat z uloženého úhlu použitou transformaci, pomocí ní transformovat kanonické sousedství a nakonec přičíst korekci, abychom získali původní pozici vrcholu.

Speciálním případem je vstupní vrchol a jeho sousedství. Vstupní vrchol nemá předchůdce, takže není možné vypočítat předpověď způsobem, který byl popsán v předchozím textu. Místo toho se vstupní vrchol v daném snímku zakóduje jako jeho vzdálenost od téhož vrcholu v kanonické molekule a jeho sousedi se zakódují jako vzdálenosti od vstupního vrcholu ve stejném snímku.

4 Limitace prototypového řešení

Existující prototypové řešení [46] je nástroj v příkazové řádce napsaný v jazyku C#. Nástroj je postavený na verzi .NET Core 2.1, která je dnes již mimo aktivní podporu [21]. Program je ovládán pomocí argumentů při spuštění a neposkytuje programové rozhraní, kterým by šel ovládat, pokud by byl použit jako knihovna.

V části, která se zabývá průchodem grafu reprezentující molekulu, využívá delegáty, prostředek jazyka C#, kteří předávají různé funkce do stejné implementace algoritmu průchodu grafem podle toho, v jakém kontextu se algoritmus zavolá. Tento návrh sice umožňuje využít stejnou implementaci průchodu grafem na více místech kódu, ale zároveň jej činí značně nepřehledným, protože různé funkce mohou různě měnit stav algoritmu za jeho běhu.

Prototyp vykazuje značnou míru paměťové náročnosti. Tato je alespoň částečně důsledek samotné implementované metody, která vyžaduje možnost náhodného přístupu ke kterémukoli atomu v molekule (což znamená, že molekula musí být celá načtená v paměti). Od implementace metody v jazyce s manuální správou paměti (jako je C++) je možné si slibovat efektivnější správu paměti než od *garbage collectoru*, který je v jazyce C#.

Zdrojový kód je málo komentovaný a nemá žádnou externí dokumentaci. Repozitář prototypu nabízí pouze projektové soubory Visual Studia pro kompilaci na systému Windows, takže není pohodlně multiplatformní. Celkově prototyp není připraven na to, aby byl jinými lidmi používán, udržován a rozvíjen.

5 Návrh knihovny

5.1 Zvolené technologie

V této sekci budou uvedeny technologie, které byly na základě analýzy provedené v kapitole 2 zvoleny pro implementaci knihovny.

Technologie, kterou jsem zvolil pro multiplatformní sestavování knihovny, je generátor *CMake*, protože je široce rozšířen a podporován, včetně integrací v populárních IDE (vývojářských prostředích) jako jsou Visual Studio nebo CLion. Jeho vysoká rozšířenost je výhodou i pro účely potenciálního využití knihoven třetích stran. Mnoho knihoven totiž implementuje svá vlastní rozšíření pro *CMake*, která umožňují v souboru `CMakeLists.txt` volat speciální příkazy, které dokážou například knihovnu najít, nastavit nebo pomocí ní sestavit nějaký modul.

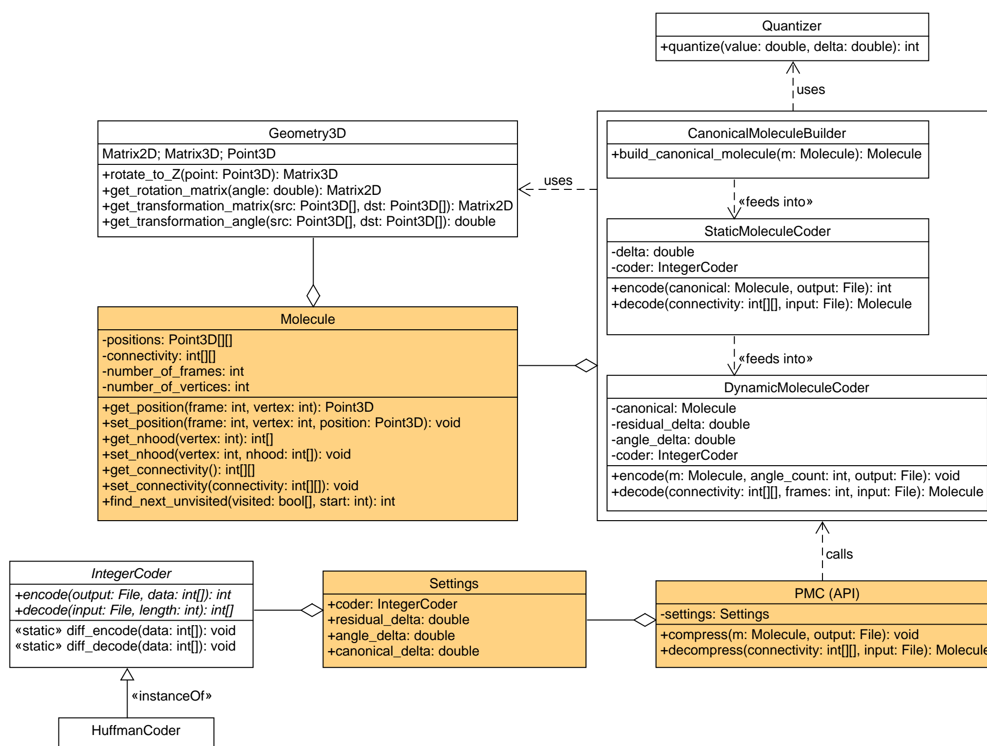
Pro účel sestavení modulu do jazyku Python jsem původně zvolil knihovnu *pybind11*, protože se jevila ze všech alternativ nejmodernější a nejrychlejší a protože je široce rozšířená, využívá moderního standardu jazyka C++ (alespoň C++11 či výše) a nepotřebuje žádné další závislosti. Navíc (přesně jak jsem zmínil výše) obsahuje rozšíření pro *CMake*, které umožňuje vyvolat kompilaci modulu velmi jednoduše pomocí jedné řádkou kódu (konkrétně příkazem `pybind11_add_module`).

Avšak, když jsem se, nedlouho poté co vyšla, dozvěděl o knihovně *nanobind*, rozhodl jsem *pybind11* nahradit právě *nanobindem*, protože slibuje ještě větší rychlost a menší moduly. Cenou za tuto rychlost je absence podpory některých funkcionalit jazyka C++ jako například vícenásobná dědičnost. Toto ovšem není pro tento projekt žádný problém, protože návrh knihovny žádnou vícenásobnou dědičnost neobsahuje (viz diagram 5.1). Navíc má stejně jako *pybind11* rozšíření pro *CMake*, díky kterému se jednoduše instaluje a používá (konkrétně se jedná o příkaz `nanobind_add_module`).

V neposlední řadě bylo třeba zvolit licenci, pod kterou bude knihovna distribuována. Protože motivací pro vydání této knihovny jako open-source software je umožnit co nejvíce lidem, aby ji používali, bylo zřejmé, že v úvahu připadají hlavně permisivní licence. Po konzultaci s vedoucím práce jsem se rozhodl pro licenci Apache verze 2.0, která je permisivní ale zároveň psaná jednoznačným jazykem, který pokrývá všechny možné případy (včetně například problematiky patentů, o které se jiné licence jako například MIT licence vůbec nezmiňují).

5.2 Struktura a rozhraní

Při návrhu knihovny se musíme zaměřit na dvě části. První je struktura modulů, ze kterých se knihovna bude skládat, a druhá je rozhraní, které knihovna bude navenek poskytovat uživateli. Tyto dvě části se navzájem ovlivňují, protože funkční rozhraní nutně vyžaduje, aby moduly, nad kterými pracuje, byly správně zapouzdřené, aby uživatel nemusel znát detaily implementace.



Obrázek 5.1: UML diagram struktury modulů projektu

Návrh diagramu struktury modulů knihovny je zobrazen na obrázku 5.1. Knihovna se skládá z devíti hlavních modulů, tři moduly, které jsou barevně zvýrazněny, reprezentují rozhraní knihovny a ty bude nutné obalit pro vytvoření bindingu.

Nejvyšší místo má modul `PMC`, který reprezentuje vstupní bod a poskytuje hlaní funkce, které uživatel bude chtít volat – `compress`, která zakóduje molekulu do souboru, a `decompress`, která dekomprimuje molekulu ze souboru. Druhou součástí rozhraní je třída `Settings`, která reprezentuje nastavení komprese, které se předá funkci `compress`. Samostatná třída pro nastavení umožní v budoucnu snadno přidat další možnosti nastavení kom-

prese bez toho, aby bylo nutné změnit hlavičku funkce `compress`.

Třetí součástí rozhraní je třída `Molecule`, která drží data o molekule. Instance této třídy vlastně reprezentují abstrakci nad vstupními a výstupním daty kompresní metody. Je důležité, aby uživatel mohl sám sestrojit instanci této třídy a naplnit ji vlastními daty. Díky tomuto přístupu není implementace knihovny závislá na konkrétních existujících formátech molekulárních dat - každý formát může být použit, pokud z jeho dat uživatel dokáže sestrojit instanci třídy `Molecule`. K tomuto účelu třída poskytuje množství metod, které umožňují různými způsoby definovat její strukturu.

Ostatní moduly jsou detaily implementace a budou uživateli skryté. Modul `Geometry3D` definuje všechny struktury pro reprezentaci bodu (atomů) v trojrozměrném prostoru a matematické operace nad nimi. Trojice modulů `CanonicalMoleculeBuilder` a `Static` a `DynamicMoleculeCoder` reprezentují jednotlivé kroky kompresní metody, jak byly popsány v sekci 3.2. Všechny tyto moduly pak využívají modul `Quantizer`, který poskytuje funkci pro kvantizaci reálných hodnot v desetinných číslech na diskrétní hodnoty v celých číslech.

Posledním modulem je `IntegerCoder`, což je abstraktní třída, která definuje rozhraní pro kodér pole celých čísel. Odkaz na konkrétní instanci této třídy je součástí instance `Settings`. Motivací za tímto návrhem je myšlenka, že by mělo být pro uživatele možné snadno změnit, jakou implementaci této funkcionality by chtěl použít. Implementace knihovny, která je výsledkem této práce, bude nabízet implementaci Huffmanova kódování. Příslušný modul je proto zakreslen v grafu, ale podrobnosti jeho struktury již ne, protože nejsou nutnou součástí knihovny. V budoucnu bude možné knihovnu rozšířit o implementace jiných způsobů kódování celých čísel.

6 Popis implementace

6.1 Programová dokumentace

Huffmanovo kódování

Modul `HuffmanCoder.cpp` implementuje Huffmanovo kódování pro účely redukce velikosti pole integerů a jeho zápisu do souboru. Nejprve získá ze vstupních dat četnost všech kódovaných symbolů a pro každý unikátní symbol vloží instanci třídy `TreeNode` do prioritní fronty. Pomocí této fronty se vytvoří kódovací strom – průchodem tímto stromem obdržíme zakódovanou formu libovolného symbolu. Pro každý symbol začneme v kořeni stromu – pokud je symbol v levém podstromu, přidáme k jeho zakódované formě bit s hodnotou 1, pokud v pravém podstromu, přidáme bit s hodnotou 0.

Do výstupního souboru je kromě samotných zakódovaných dat potřeba nejprve zapsat také nějaké informace o nich, aby je bylo možné zpětně dekódovat. Jsou to: počet bitů nutný k zapsání každého symbolu (podle rozsahu dat) a nejmenší symbol v datech. Tyto dvě informace dohromady umožňují během zápisu kódovacího stromu zapsat symboly na minimální množství bitů. Samotný kódovací strom je zakódován v pre-order pořadí, kde uzel s potomky je zanesen jako bit s hodnotou 0 a list stromu je zanesen jako bit s hodnotou 1. Pokud byl zapsán list, následuje symbol, který tento list nese, zapsán jako

$$zapsany\ symbol = symbol - minimum.$$

Tímto mapováním se zajistí, že všechny symboly jsou zapisovány jako nezáporná čísla, což umožňuje ušetřit znamínkový bit.

Nakonec se vypočte, kolik bajtů budou zakódovaná vstupní data dlouhá, podle vzorce

$$delka = \frac{\sum cetnost * delka\ kodu}{8}.$$

a tato informace je také zapsána do výstupního souboru. Po ní už následují samotná zakódovaná data.

Dekódování dat probíhá opačným postupem – tedy zpracováním vstupního souboru je zkonstruován kódovací strom, pomocí kterého jsou dekódovány uložená data. Prototypová implementace před samotným dekódováním ještě předvypočítala pro každý vrchol kódovacího stromu všech 256 (2^8) možných přechodů (vstupních bajtů) a které symboly těmito přechody dekóduje. Tyto informace uložila do tabulek a pak při dekódování v těchto tabulkách

vyhledávala. Porovnáním rychlosti výpočtu jsem ale dospěl k závěru, že při implementaci v jazyce C++ je naivní řešení (čili prosté procházení stromu a dekódování symbolů za pochodu) rychlejší než tento přístup s krokem před-výpočtu.

Prioritní fronta

Modul `HuffmanCoder.cpp` využívá vlastní implementaci prioritní fronty z modulu `PriorityQueue.cpp`. Tato prioritní fronta implementuje haldu pomocí pole a celkově napodobuje implementaci prioritní fronty, která byla využitá v prototypovém řešení. Důvodem k použití vlastní prioritní fronty byla potřeba zajistit, aby se objekty z prioritní fronty za všech okolností vracely ve stejném pořadí jako v prototypovém řešení, aby vzniklé stromy (a tudíž i Huffmanovo kódování) byly vzájemně kompatibilní. V případě využití standardní implementace prioritní fronty (jako například `std::priority_queue` nešlo toto zajistit ve všech případech (zejména v případech několikanásobné remízy při porovnávání dvou objektů).

Vstup/výstup jednotlivých bitů

Během průběhu Huffmanova kódování je potřeba pracovat se vstupními a výstupními daty po jednotlivých bitech. Za tímto účelem modul `BitIO.cpp` implementuje třídy `BitWriter` a `BitReader`, které umožňují číst/zapisovat jednotlivé bity nebo číst/zapisovat celá čísla s přesností na libovolný počet bitů a se správným řazením bitů. Obě třídy využívají kontejner `std::bitset` ze standardní knihovny jazyka C++, který umožňuje plně přenositelnou manipulaci s bity. `BitWriter` navíc implementuje bufferování svého výstupu, aby se snížilo množství volání nad výstupním proudem.

Trojrozměrná geometrie

Modul `Geometry3D.cpp` definuje všechny objekty, které knihovna používá pro reprezentaci trojrozměrného prostoru, a operace nad nimi (pomocí funkcí nebo přetěžování operátorů). Jedná se o dvojrozměrnou matici `Matrix2D`, trojrozměrnou matici `Martix3D` a bod (popř. vektor) v trojrozměrném prostoru `Point3D`. Důležité jsou funkce pro realizaci rotací, které jsou nutné pro implementaci kompresní metody. Funkce `rotate_to_Z` vypočítá trojrozměrnou matici, která natočí bod tak, aby ležel na ose Z (viz sekce 3.2.1). Funkce `get_transformation_matrix` vypočítá rotační matici, která natočí sadu bodů tak, aby byly zarovnány s druhou sadou bodů, což umožňuje určit odhad transformace mezi kanonickou a vstupní molekulou.

Molekula

Třída `Molecule` v modulu `Molecule.cpp` představuje systém atomů, který knihovna komprimuje. Molekula je reprezentována jako neorientovaný graf pozic vrcholů v daném počtu snímků. Pozice vrcholů jsou uchovány ve dvojrozměrném vektoru `positions` (myšlen standardní kontejner `std::vector`), kde první rozměr je daný snímek molekuly a druhý rozměr je index vrcholu. Na výsledné pozici je instance objektu `Point3D` reprezentující konkrétní pozici vrcholu. Seznam sousednosti grafu je uchován ve dvojrozměrném vektoru `connectivity`.

Konstruktor molekuly pouze připraví její vnitřní kontejnery – naplnění molekuly obsahem je úkolem uživatelského kódu, díky čemuž knihovna není závislá na konkrétním jednom formátu (viz sekce 3.1). Instance molekuly poskytuje metody, které umožňují kontejnery naplnit různými způsoby a v různém pořadí pozic. Jediným požadavkem je znát počet snímků a počet vrcholů v okamžiku volání konstruktoru molekuly.

Tvorba kanonické molekuly

Modul `CanonicalMoleculeBuilder.cpp` zajišťuje tvorbu kanonické molekuly pro vstupní molekulu. Konkrétně se implementace nachází ve vnitřní třídě, ke které není přímý přístup z ostatního kódu, ale modul poskytuje veřejnou funkci `build_canonical_molecule`, která vytvoří instanci třídy a zavolá její funkcionalitu. Tento návrh umožňuje využít vlastnosti třídy (například atributy instance) a zároveň zajišťuje, že v uživatelském kódu nezůstanou neuklizené instance třídy.

Modul implementuje algoritmus tvorby kanonické molekuly tak, jak je popsán v sekci 3.2.1, až na některé drobné rozdíly. Zaprvé, protože po sobě následující snímky jsou si velmi podobné, nesestrojujeme kanonickou molekulu ze všech dostupných snímku ale pouze z některých (výchozí vzorkování je každý 50. snímek). Druhým rozdílem je speciální případ, kdy v lokálním sousedství zbývá zpracovat jen jediný vrchol. V takovém případě můžeme nahradit výše popsaný maticový výpočet jednodušší kvadratickou rovnicí, která nalezne vrchol kanonické molekuly takový, aby ležel na přímce, která prochází posledními dvěma předchůdci v kanonické molekule. Tyto modifikace existují už v prototypovém řešení, ale ve vzorkování se vyskytoval kritický bug, který způsobil pád programu při pokusu zpracování molekuly, která má méně snímků, než je velikost vzorkovací frekvence.

Kódování molekuly

Modul `StaticMoleculeCoder.cpp` implementuje kompresi kanonické molekuly pomocí kvantizace a diferenciální reprezentace. Kanonickou molekulu je nutné uložit, protože informace o ní jsou nutnou podmínkou k zakódování vstupní molekuly. Modul `DynamicMoleculeCoder.cpp` implementuje kódování a kompresi vstupní molekuly tak, jak je popsáno v sekci 3.2.2. Implementace obou modulů je velice podobná té v prototypovém řešení, aby byly výsledné komprimované soubory navzájem kompatibilní, což je důležité pro férové srovnání výsledků obou implementací (viz kapitola 7).

Oba dva moduly na rozdíl od prototypové implementace nevyužívají mechanismus delegátů a implementují své vlastní metody pro průchod grafem molekuly. Díky tomu je kód pro jednotlivé metody průchodu přehlednější a ušetří se na funkčních voláních. Cenou je větší velikost výsledného binárního souboru z důvodu duplicity malé části kódu. Za zmínku také stojí oprava chyby, která se nacházela v prototypové implementaci. Kvůli použití delegátů byl na dvou různých místech duplikován seznam již navštívených vrcholů. V jedné větvi kódu ale jeden z těchto seznamů nebyl po zpracování vrcholu změněn, což mělo za následek, že v případě, kdy v grafu byla smyčka, se jeden vrchol zpracoval dvakrát.

Formát PMC a nastavení

Modul *PMC* obsahuje funkce, které zahajují proces komprese nebo dekomprese dat. Za tím účelem vytváří výstupní soubor nebo čte vstupní soubor a tím definuje formát `.pmc`. Formát `.pmc` začíná hlavičkou, která obsahuje informace o nastavení komprese, a na začátku hlavičky je tzv. *magic*, což je několik bajtů (v tomto případě čtyři) na úplném začátku souboru, které musí přesně odpovídat očekávané sekvenci, aby byl soubor identifikován jako validní `.pmc` soubor. Pokud *magic* chybí, očekává knihovna data ve formátu, který produkovala prototypová implementace, která žádnou hlavičku nedefinovala (tzv. *legacy* režim). V budoucnu by tato možnost mohla být odstraněna, aby knihovna striktně vyžadovala standardní hlavičku. Zároveň, pokud jsou parametry komprese nesmyslné (například kvantizační konstanty rovné nule), metoda neproběhne.

Nastavení je předáno uživatelem pomocí instance třídy `Settings`. Součástí této třídy je i výčtový typ `CoderSettings`, který reprezentuje zvolenou metodu kódování celých čísel. Hodnota tohoto výčtového typu je předána tovární funkci, která vrátí instanci potomka abstraktní třídy `IntegerCoder`, která odpovídá zvolené metodě.

Binding do jazyku Python

Binding do jazyku Python je definován v modulu `PythonBinding.cpp` a využívá funkcí a maker z knihovny *nanobind*. Šablonový systém *nanobindu* se ve většině případů automaticky postará o konverzi dat a objektů mezi Pythonem a C++. Výjimkou jsou metody `Molecule::set_position`, která jako argument bere instanci třídy `Point3D`, a `Molecule::get_position`, která instanci `Point3D` vrací. Místo abych definoval binding pro tuto třídu a tím umožnil její používání v Pythonu, rozhodl jsem se implementovat manuální konverzi instance `Point3D` na uspořádanou n-tici v Pythonu a naopak. Uspořádaná n-tice (anglicky tuple) je vestavěný prostředek jazyka Python a přirozenější způsob jak v Pythonu reprezentovat bod v trojrozměrném prostoru, než vlastní třída.

6.2 Systém sestavování

V souboru `CMakeLists.txt` je popsán celý proces sestavení – čili stáhnutí knihoven třetích stran, na kterých projekt závisí, kompilace samotné knihovny PMC, kompilace bindingu pro jazyk Python, kompilace jednotkových a dalších testů a vytvoření automaticky generované dokumentace.

Projekt má tři závislosti, které *CMake* není schopen automaticky nainstalovat. První je interpret jazyka Python ve verzi alespoň 3.8. Druhá je knihovna *nanobind*, kterou lze nainstalovat například pomocí instalátoru *pip* příkazem:

```
python -m pip install nanobind
```

Třetí závislost je nástroj *Doxygen*, který je použit k automatickému generování dokumentace (viz 6.4). Tento krok je dobrovolný a pokud není *Doxygen* nainstalován, bude přeskočen bez toho, aby narušil zbytek kompilace.

Sestavení lze konfigurovat z příkazové řádky v kořenovém adresáři projektu příkazem:

```
cmake <options> . -B <build_path>
```

kde `<options>` reprezentuje alespoň jednu z možností konfigurace částí projektu, které se mají sestavit, každá uvozena přepínačem `-D`. Konkrétně možnost `BUILD_BINDING` povolí sestavení bindingu, `BUILD_TESTS` povolí sestavení testů a `BUILD_DOCS` povolí automatickou generaci dokumentace. Za každou z možností musí ještě následovat informace, zda se možnost povoluje (`=ON`) či zakazuje (`=OFF`). Ve výchozím nastavení je vše povoleno. Symbol `<build_path>` reprezentuje cestu, do které bude projekt sestaven. Běžně

používaná cesta je složka `build`. Použitý příkaz tedy může vypadat například takto:

```
cmake -D BUILD_TESTS=OFF -D BUILD_DOCS=OFF . -B build
```

Po dokončení konfigurace sestavení je potřeba ve složce, která byla v předchozím kroku určena k sestavování, zahájit samotný proces sestavení. Konkrétní postup se bude lišit od zvoleného nástroje. Například v systémech GNU/Linux je výchozím nástrojem *GNU Make*, čili sestavení stačí spustit následujícím příkazem:

```
make
```

Modul pro jazyk Python, který byl vytvořen bindingem, se nachází ve složce `binding/pmc_py`. Ve výchozím stavu není modul nijak nainstalovaný a jde proto importovat pouze ve skriptu, který je spouštěn ve stejném adresáři. Pro instalaci modulu je ve složce `binding` připraven skript `setup.py`, který lze spustit následujícím příkazem:

```
python setup.py install --user
```

Po instalaci je možné voláním `import pmc_py` importovat modul v jakémkoli Python skriptu. Příkladem skriptu, který využívá modul, je `pmc.py` ve složce `python_example`. Pro demonstraci následuje jeho úryvek:

```
molecule = pmc_py.Molecule(num_frames, num_vertices)
// naplnění molekuly daty
settings = pmc_py.Settings(
    pmc_py.CoderSettings.HUFFMAN_CODER,
    residual_delta, angle_delta, canonical_delta
)
pmc_py.compress(molecule, output_file, settings)
```

6.3 Testování kódu

Kromě provedených experimentů pro ověření funkčnosti knihovny, které budou popsány v kapitole 7, byl kód testován také jednotkovými testy napsanými za pomoci C++ frameworku GoogleTest. Každá veřejně dostupná funkce v každém modulu je pokryta alespoň jedním testem. Vnitřní část implementace (`private` metody) nemá samostatné testy, protože to porušuje zásadu testování černé skříňky (black-box testing principle) [25].

Sadu testů lze v sestavovacím adresáři spustit pomocí nástroje *CMake*, konkrétně jeho součásti *CTest*, pomocí příkazu:

```
ctest
```

6.4 Referenční dokumentace kódu

Součástí sestavení je i automaticky vygenerovaná referenční dokumentace projektu ve formátu `.html`. Tato dokumentace je generována pomocí nástroje *Doxygen*. Standardní verze dokumentace obsahuje pouze informace o třídách a funkcích, které jsou veřejné (dostupné navenek), což by byla verze cílená pro programátora-uživatele. Doxygen ale umožňuje i nastavení, kdy generovaná dokumentace navíc dokumentuje i třídy a funkce, které jsou použité ve vnitřní implementaci knihovny, což by byla verze cílená pro budoucí udržovatele projektu.

7 Validace kompresních výsledků

Aby mohla nová implementace plně nahradit prototypovou implementaci, musíme zaručit, že poskytuje relevantní kompresní výsledky. Aby bylo srovnání férové, byl na novou implementaci kladen požadavek, aby udržela stejný formát souboru a byla tudíž zpětně kompatibilní při kompresi i dekompresi.

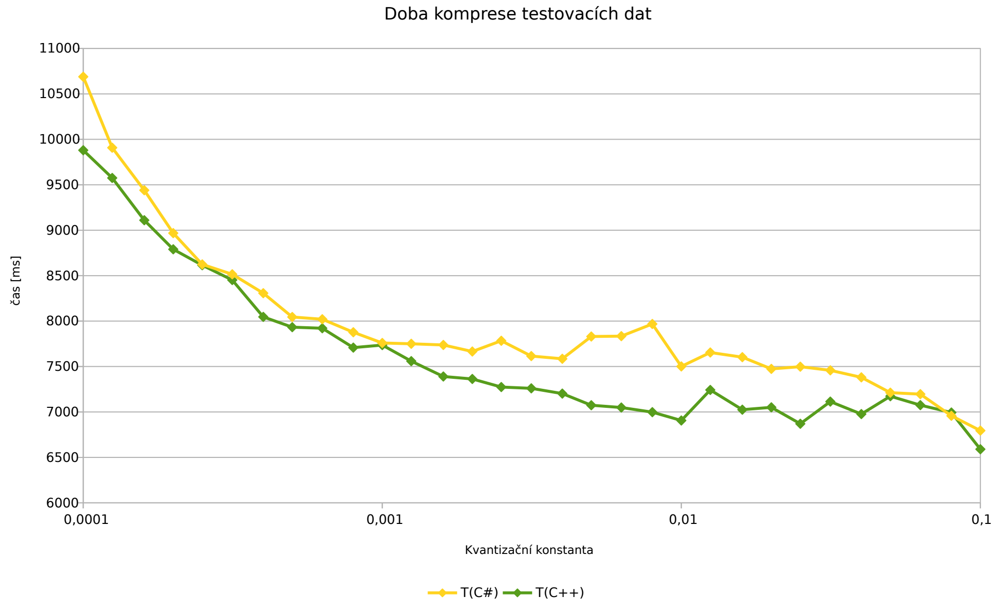
Jako testovací data mi byly poskytnuty trajektorie několika molekul. Z důvodu vysoké paměťové náročnosti ostatních molekul jsem rozhodl použít data o molekule p53, což je lidský protein, který se váže k poškozeným částem DNA a blokuje tvorbu zmutovaných buněk [36]. Data tvoří 3008 vrcholů (atomů), které byly nasnímány v 11501 snímcích, o celkové velikosti 830 MB. Pro tato data proběhly dvě měření – první pomocí prototypové implementace v jazyce C# běžící ve frameworku .NET ve verzi 7.0 na systému Windows, druhé pomocí nové implementace v jazyce C++ zkompilované překladačem GCC verze 11.1.0 běžící na systému Ubuntu 20.04. Oba programy byly spuštěny na stejném stroji s 64 bitovým procesorem Intel Core i5-8300H a 16 GB RAM.

Každé měření znamenalo provést kompresi vstupní molekuly 31×, pokaždé s jinými vstupními parametry. Konkrétně se měnil parametr q_r, což je kvantizační konstanta pro kvantizaci vrcholů vstupní molekuly. Kvantizační konstanta q_a pro kvantizaci rotačních úhlů a kvantizační konstanta q_c pro kvantizaci vrcholů v kanonické molekule zůstávaly ve všech opakováních konstantní na hodnotách 0,1 a 0,001, což jsou výchozí hodnoty uvedené v původním článku.

Testované veličiny byly tři. První veličina b_pc je počet bitů na souřadnici ve výsledném souboru, což je číslo, které vyjadřuje efektivitu komprese. Druhá veličina ϵ_{max} je maximální chyba mezi vrcholy původní molekuly a zakódované molekuly, což vyjadřuje ztrátu, ke které během komprese došlo. Poslední testovanou veličinou je doba běhu programu T, konkrétně doba samotné komprese.

Výsledky obou měření jsou uvedeny v tabulce A. Veličiny *bpc* a ϵ_{max} vyšly v obou měřeních naprosto totožné (a proto jsou v tabulce uvedeny jenom v jednom sloupečku). Tento výsledek znamená, že se povedlo, aby chování nové implementace odpovídalo chování implementace prototypové a to až na úroveň práce s čísly s plovoucí desetinnou čárkou a zaokrouhlování čísel (zejména chování při zaokrouhlování záporných čísel bylo něco, co jsem musel

ošetřit). Tudíž, zkomprimované soubory, které obě implementace produkují, jsou naprosto totožné.



Obrázek 7.1: Doba komprese testovacích dat

Rozdíly najdeme v čase T. Je nutné podotknout, že čas běhu programu bude vždy proměnlivý, protože závisí i na vnějších faktorech jako je například aktuální vytížení systému. Přesto můžeme z naměřených data usoudit, že nová implementace běží v průměru o 4,5% rychleji než prototypová implementace.

Takový rozdíl v rychlosti obou programů byl pod moje očekávání. Profilováním kódu jsem došel k závěru, že limitujícím faktorem programu je vstup a výstup dat pomocí proudu `std::fstream` ze standardní knihovny jazyka C++. Do budoucna můžeme doufat ve zlepšení ve standardech C++23 a dále, kdy manipulaci s proudy začne nahrazovat `std::print` a další nové nástroje.

Větší úspěch si zapsala nová implementace v oblasti paměťové náročnosti. Prototypová implementace napsaná v jazyce C# využívala mechanismus automatické správy paměti (garbage collector). Během komprese testovacích dat spotřebovala původní implementace průměrně 3,6 GB paměti, s maximem spotřeby až 4,8 GB. Naproti tomu nová implementace provádí manuální správu paměti a to (v souladu s pokyny pro moderní C++) pomocí prostředků standardní knihovny jako jsou chytré ukazatele (`std::unique_ptr` a `std::shared_ptr`) a kontejnery, které implementují techniku RAII [51] (Resource Allocation Is Initialization). Nová implementace během komprese

testovacích dat spotřebovala průměrně 1,4 GB paměti, s maximem 2 GB, což je zhruba o 60 % méně spotřeby paměti za dobu běhu programu.

Z výsledků experimentu je vidět, že práce splnila zadání implementovat kompresní metodu a dosáhla výrazného zlepšení v oblasti využití paměti. Rychlost komprese dat dosáhla jen malého zlepšení a její další zrychlení by mohlo by být předmětem dalšího vývoje.

8 Závěr

V rámci této práce byla navržena a implementována knihovna v jazyce C++, pomocí které je možné zkomprimovat a dekomprimovat molekulární trajektorie za použití metody *Predictive Molecular Compression*, kterou dříve navrhli pracovníci Katedry informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity v Plzni. Knihovna byla koncipována jako veřejně dostupný, open-source software a byla proto implementována s důrazem na přehledný, dlouhodobě udržitelný a rozšiřitelný kód. Důležitým aspektem také byl požadavek, aby byla knihovna multiplatformní, a proto využívá pouze technologie, které umožňují, aby ji bylo možné sestavit a spustit na více platformách. Příklady takových využitých technologií jsou například systém sestavování *CMake* nebo knihovna pro psaní testů *GoogleTest*.

Jako demonstrace možností integrace napsané knihovny byl implementován proces (za pomoci nástroje *nanobind*, který stejnou knihovnu sestaví jako dynamický modul, který je možné naimportovat ve skriptu napsaném v jazyce Python a v něm definovat vstupní data a nastavení parametrů komprese a zavolat funkci pro kompresi či dekompresi molekulárních trajektorií.

Knihovna byla úspěšně otestována na reálných datech a výsledky komprese byly porovnány s existující prototypovou implementací. Výsledky experimentů ukázaly, že knihovna dosáhla totožné komprese jako předchozí implementace, přičemž dosáhla mírně vyšší rychlosti komprese a výrazně nižších požadavků na množství paměti.

Do budoucna bude zajímavé sledovat, jestli se knihovna, spolu s metodou, kterou implementuje, po zveřejnění ve veřejném repozitáři nějak ujme v oblasti komprese molekulárních trajektorií a jestli se jí někdo další rozhodne vylepšit, rozšířit nebo implementovat do svého projektu. Jednou z oblastí, ve které vidím potenciál na vylepšení, je optimalizace kódu, čímž by se mohla ještě zvýšit rychlost komprese.

Literatura

[1] *ARM Cortex-A Series Programmer's Guide for ARMv7-A* [online]. Arm Limited, 2022. [cit. 2022/12/14]. unsigned char and signed char. Dostupné z: https://developer.arm.com/documentation/den0013/d/Porting/Miscellaneous-C-porting-issues/unsigned-char-and-signed-char.

[2] *Options for Code Generation Conventions* [online]. GCC Project, 2008. [cit. 2022/11/22]. GCC Documentation. Dostupné z: http://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html.

[3] *Overview of Dynamic Libraries* [online]. Apple, 2012. [cit. 2022/11/22]. Apple Developer Documentation. Dostupné z: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/OverviewOfDynamicLibraries.html.

[4] *What is a DLL* [online]. Microsoft, 2022. [cit. 2022/11/22]. Microsoft Windows Client Documentation. Dostupné z: https://learn.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library.

[5] *PDB Format Description* [online]. 2001. [cit. 2022/12/05]. Version 3.3. Dostupné z: https://www.wwpdb.org/documentation/file-format-content/format33/v3.3.html.

[6] *igl python bindings* [online]. libigl, 2022. [cit. 2022/11/28]. Dostupné z: https://libigl.github.io/libigl-python-bindings/.

[7] *CMake* [online]. Kitware, 2022. [cit. 2022/12/18]. Dostupné z: https://cmake.org/.

[8] *Directive 2009/24/EC of the European Parliament and of the Council of 23 April 2009 on the legal protection of computer programs (Codified version)* [online]. Evropský parlament, Rada Evropské unie, 2009. [cit. 2023/02/12]. Dostupné z: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32009L0024.

[9] *FreeBSD Make* [online]. Berkeley Softworks, 2022. [cit. 2022/12/18]. Dostupné z: https://svnweb.freebsd.org/base/stable/2.0.5/usr.bin/make/make.h?revision=8869&view=markup.

[10] *GNU Make* [online]. Free Software Foundation, Inc., 2022. [cit. 2022/12/18]. Dostupné z: https://www.gnu.org/software/make/.

[11] *Solution (.sln) file* [online]. Microsoft, 2022. [cit. 2022/12/18]. Dostupné z: https://learn.microsoft.com/en-us/visualstudio/extensibility/internals/solution-dot-sln-file?view=vs-2022.

[12] *NMAKE Reference* [online]. Microsoft, 2022. [cit. 2022/12/18]. Dostupné z: https://learn.microsoft.com/en-us/cpp/build/reference/nmake-reference?view=msvc-170.

[13] *Ninja* [online]. 2022. [cit. 2022/12/18]. Dostupné z: https://ninja-build.org/.

[14] *Platform Definition* [online]. The Linux Information Project, 2004. [cit. 2022/12/14]. Dostupné z: http://www.linfo.org/platform.html.

[15] *SCons: A software construction tool* [online]. SCons Foundation, 2022. [cit. 2022/12/18]. Dostupné z: https://scons.org/.

[16] Abrahams, D. – Seefeld, S. *Boost.Python* [online]. Boost, 2022. [cit. 2022/12/04]. Dostupné z: https://www.boost.org/doc/libs/1_80_0/libs/python/doc/html/index.html.

[17] *Apache License, Version 2.0* [online]. The Apache Software Foundation, 2023. [cit. 2023/02/13]. Dostupné z: https://www.apache.org/licenses/LICENSE-2.0.

[18] Berman, H. The Protein Data Bank: A historical perspective. *Acta crystallographica. Section A, Foundations of crystallography*. 02 2008, 64, s. 88–95. doi: 10.1107/S0108767307035623.

[19] *The 3-Clause BSD License* [online]. Open Source Initiative, 2023. [cit. 2023/02/13]. Licensed under a Creative Commons Attribution 4.0 International license. Dostupné z: https://opensource.org/license/bsd-3-clause/.

[20] Clarage, J. B. et al. A Sampling Problem in Molecular Dynamics Simulations of Macromolecules. *Proceedings of the National Academy of Sciences of the United States of America*. 1995, 92, 8, s. 3288–3292. ISSN 00278424. Dostupné z: http://www.jstor.org/stable/2367051.

[21] Damkewala, J. *.NET Core 2.1 will reach End of Support on August 21, 2021* [online]. Microsoft, 2021. [cit. 2022/12/18]. Dostupné z: https://devblogs.microsoft.com/dotnet/net-core-2-1-will-reach-end-of-support-on-august-21-2021/.

[22] Darmon, E. – Torre, D. Dual licensing strategy with open source competition. *Managerial and Decision Economics*. 2017, 38, 8, s. pp.

1082–1093. ISSN 01436570, 10991468. Dostupné z: https://www.jstor.org/stable/26608184.

[23] Dvořák, J. – Maňák, M. – Váša, L. Predictive compression of molecular dynamics trajectories. *Journal of Molecular Graphics and Modelling*. 2020, 96, s. 107531. ISSN 1093-3263. doi: https://doi.org/10.1016/j.jmgm.2020.107531. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1093326319306564.

[24] *GNU General Public License* [online]. Free Software Foundation, Inc., 2007. [cit. 2023/02/13]. Dostupné z: https://www.gnu.org/licenses/gpl-3.0.html.

[25] *Advanced GoogleTest Topics* [online]. Google Inc., 2008. [cit. 2023/04/01]. Testing Private Code. Dostupné z: http://google.github.io/googletest/advanced.html#testing-private-code.

[26] *GROMACS File Formats* [online]. 2018. [cit. 2022/11/07]. GROMACS Documentation. Dostupné z: https://manual.gromacs.org/documentation/2018/user-guide/file-formats.html.

[27] Guennebaud, G. – Jacob, B. – others. *Eigen v3* [online]. 2010. Dostupné z: http://eigen.tuxfamily.org.

[28] Harris, C. R. et al. Array programming with NumPy. *Nature*. September 2020, 585, 7825, s. 357–362. doi: 10.1038/s41586-020-2649-2. Dostupné z: https://doi.org/10.1038/s41586-020-2649-2.

[29] *HDF5 Support Page* [online]. The HDF Group, 2022. [cit. 2022/11/21]. HDF5. Dostupné z: https://portal.hdfgroup.org/display/HDF5/HDF5.

[30] Heap, D. *Depth-first search (DFS)* [online]. 2002. [cit. 2022/11/14]. Dostupné z: http://www.cs.toronto.edu/~heap/270F02/node36.html.

[31] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*. 1952, 40, 9, s. 1098–1101. doi: 10.1109/JRPROC.1952.273898.

[32] *Anatomy of Linux dynamic libraries* [online]. Techopedia Inc., 2008. [cit. 2022/11/14]. IBM Developer. Dostupné z: https://developer.ibm.com/tutorials/l-dynamic-libraries/.

[33] *ISC License* [online]. Internet Systems Consortium, Inc., 2023. [cit. 2023/02/13]. Dostupné z: https://www.isc.org/licenses/.

[34] JAKOB, W. – RHINELANDER, J. – MOLDOVAN, D. *pybind11 – Seamless operability between C++11 and Python* [online]. 2017. [cit. 2022/12/04]. https://github.com/pybind/pybind11.

[35] KARPLUS, M. – KURIYAN, J. – BERNE, B. J. Molecular Dynamics and Protein Function. *Proceedings of the National Academy of Sciences of the United States of America*. 2005, 102, 19, s. 6679–6685. ISSN 00278424. Dostupné z: `http://www.jstor.org/stable/3375414`.

[36] LAMBRUGHI, M. et al. DNA-binding protects p53 from interactions with cofactors involved in transcription-independent functions. *Nucleic Acids Research*. 09 2016, 44, 19, s. 9096–9109. ISSN 0305-1048. doi: 10.1093/nar/gkw770. Dostupné z: `https://doi.org/10.1093/nar/gkw770`.

[37] LOGAN, S. *Cross-platform development in C++*. Addison-Wesley Educational, November 2007. ISBN 978-0-321-24642-4.

[38] GAILLY, J. – ADLER, M. *zlib license* [online]. 2022. [cit. 2023/02/13]. Dostupné z: `http://zlib.net/zlib_license.html`.

[39] *__declspec* [online]. Microsoft, 2022. [cit. 2022/11/22]. C++ Language Reference. Dostupné z: `https://learn.microsoft.com/en-us/cpp/cpp/declspec`.

[40] *MIT License* [online]. SPDX Workgroup a Linux Foundation Project, 2018. [cit. 2023/02/13]. Dostupné z: `https://spdx.org/licenses/MIT.html`.

[41] *Mozilla Public License* [online]. Free Software Foundation, Inc., 2012. [cit. 2023/02/13]. Version 2.0. Dostupné z: `https://www.mozilla.org/en-US/MPL/2.0/`.

[42] *Network Common Data Form (NetCDF)* [online]. Unidata, 2022. [cit. 2022/11/21]. Dostupné z: `https://doi.org/10.5065/D6H70CW6`.

[43] PAKKANEN, J. *The Meson Build system* [online]. 2022. [cit. 2022/12/18]. Dostupné z: `https://mesonbuild.com/index.html`.

[44] *Python/C API Reference Manual* [online]. Python Software Foundation, 2022. [cit. 2022/11/28]. Python 3.11.0 Documentation. Dostupné z: `https://docs.python.org/3/c-api/intro.html`.

[45] *ctypes — A foreign function library for Python* [online]. Python Software Foundation, 2022. [cit. 2022/11/28]. Python 3.11.0 Documentation. Dostupné z: `https://docs.python.org/3/library/ctypes.html#loading-dynamic-link-libraries`.

[46] *Predictive Molecule Compression* [online]. Jan Dvořák, 2019. [cit. 2022/11/07]. Dostupné z: https://jandvorak-uwb.github.io/pmc/.

[47] Robustelli, P. – Piana, S. – Shaw, D. E. Developing a molecular dynamics force field for both folded and disordered protein states. *Proceedings of the National Academy of Sciences of the United States of America*. 2018, 115, 21, s. E4758–E4766. ISSN 00278424, 10916490. Dostupné z: https://www.jstor.org/stable/26509876.

[48] *Software Library* [online]. Techopedia Inc., 2016. [cit. 2022/11/14]. Techopedia Dictionary. Dostupné z: https://www.techopedia.com/definition/3828/software-library.

[49] Spångberg, D. – Larsson, D. S. D. – Van Der Spoel, D. Trajectory NG: portable, compressed, general molecular dynamics trajectories. *Journal of Molecular Modeling*. 2011, 17, 10, s. 2669–2685. doi: https://doi.org/10.1002/jcc.20291. Dostupné z: https://doi.org/10.1007/s00894-010-0948-5.

[50] Stallman, R. *Copyleft: Pragmatic Idealism* [online]. Free Software Foundation, Inc., 2021. [cit. 2023/02/13]. Dostupné z: https://www.gnu.org/philosophy/pragmatic.html.

[51] Stroustrup, B. – Sutter, H. *R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)* [online]. 2023. [cit. 2023/05/03]. C++ Core Guidelines. Dostupné z: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[52] *The Open Source Definition* [online]. Open Source Initiative, 2007. [cit. 2023/02/13]. Licensed under a Creative Commons Attribution 4.0 International license. Dostupné z: https://opensource.org/definition/.

[53] *Top Open Source Licenses* [online]. Black Duck Software, 2016. [cit. 2023/02/13]. Dostupné z: https://web.archive.org/web/20160719043600/https://www.blackducksoftware.com/top-open-source-licenses.

[54] Van Der Spoel, D. et al. GROMACS: Fast, flexible, and free. *Journal of Computational Chemistry*. 2005, 26, 16, s. 1701–1718. doi: https://doi.org/10.1002/jcc.20291. Dostupné z: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.20291.

[55] *Various Licenses and Comments about Them* [online]. Free Software Foundation, Inc., 2022. [cit. 2023/02/13]. Nonfree Software Licenses. Dostupné z: https://www.gnu.org/licenses/license-list.html#NonFreeSoftwareLicenses.

[56] Virtanen, P. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, 17, s. 261–272. doi: 10.1038/s41592-019-0686-2.

[57] Wenzel, J. *nanobind: tiny and efficient C++/Python bindings* [online]. 2022. [cit. 2023/04/26]. https://github.com/wjakob/nanobind.

[58] Wenzel, J. *Benchmark* [online]. 2022. [cit. 2023/04/26]. nanobind Documentation. Dostupné z: `https://nanobind.readthedocs.io/en/latest/benchmark.html`.

[59] Wenzel, J. *Removed Features* [online]. 2022. [cit. 2023/04/26]. nanobind Documentation. Dostupné z: `https://nanobind.readthedocs.io/en/latest/porting.html#removed-features`.

[60] Wenzel, J. *Benchmark* [online]. 2017. [cit. 2022/12/04]. pybind11 Documentation. Dostupné z: `https://pybind11.readthedocs.io/en/stable/benchmark.html`.

[61] *What is Free Software?* [online]. Free Software Foundation, Inc., 2022. [cit. 2023/02/13]. Dostupné z: `https://www.gnu.org/philosophy/free-sw.html#four-freedoms`.

[62] Wheeler, D. A. *Shared Libraries* [online]. 2003. [cit. 2022/11/22]. Program Library HOWTO. Dostupné z: `https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html`.

[63] Williams, F. *NumpyEigen - Fast zero-overhead bindings between NumPy and Eigen* [online]. 2022. [cit. 2022/11/29]. Dostupné z: `https://github.com/fwilliams/numpyeigen`.

A Kompletní výsledky měření

q_r	bpc	ϵ_{max}	$T_{C\#}$ [ms]	T_{C++} [ms]
0,0001	12,4468634	0,0000864	10689	9880
0,000125	11,9964226	0,0001081	9908	9575
0,00016	11,5333963	0,0001386	9440	9110
0,0002	11,1278828	0,0001729	8968	8790
0,00025	10,7362685	0,0002161	8624	8615
0,000315	10,3549927	0,0002724	8515	8452
0,0004	9,9774535	0,0003464	8307	8046
0,0005	9,6057262	0,0004324	8045	7933
0,00063	9,2525347	0,0005448	8020	7921
0,0008	8,8978046	0,0006928	7877	7708
0,001	8,5436076	0,0008647	7759	7736
0,00125	8,2070945	0,0010809	7750	7559
0,0016	7,8623306	0,0013856	7737	7390
0,002	7,5265968	0,0017321	7665	7363
0,0025	7,1876381	0,0021615	7783	7274
0,00315	6,8683517	0,0027263	7615	7260
0,004	6,5113283	0,0034641	7585	7202
0,005	6,1832316	0,0043202	7830	7074
0,0063	5,8596898	0,0054459	7834	7049
0,008	5,5059030	0,0069282	7968	6998
0,01	5,1811794	0,0086603	7501	6906
0,0125	4,8732589	0,0108080	7654	7242
0,016	4,5147981	0,0138564	7603	7025
0,02	4,1941114	0,0173205	7474	7051
0,025	3,8972815	0,0216280	7498	6870
0,0315	3,5916742	0,0272557	7458	7114
0,04	3,2734471	0,0346410	7381	6977
0,05	2,9929572	0,0433013	7212	7173
0,063	2,7546465	0,0545004	7196	7075
0,08	2,5025028	0,0692820	6958	6994
0,1	2,3200569	0,0866025	6795	6590

Tabulka A.1: Výsledky měření kompresních výsledků