Zentralinstitut für Engineering, Elektronik und
Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Evaluierung von Methoden zur C++-Integration in Python innerhalb des SELMA-Projekts

Eliane Julie Bernard

Jül-4447

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Zentralinstitut für Engineering, Elektronik und Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

Evaluierung von Methoden zur C++-Integration in Python innerhalb des SELMA-Projekts

Eliane Julie Bernard

Berichte des Forschungszentrums Jülich
Jül-4447 · ISSN 0944-2952
Zentralinstitut für Engineering, Elektronik und
Analytik (ZEA) · Systeme der Elektronik (ZEA-2)

DE-A96 (Bachelor, FH Aachen, 2024)

Vollständig frei verfügbar über das Publikations-
portal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess

Forschungszentrum Jülich GmbH · 52425 Jülich
Zentralbibliothek, Verlag
Tel.: 02461 61-5220 · Fax: 02461 61-6103
zb-publikation@fz-juelich.de
www.fz-juelich.de/zb

Erklärung

Hiermit versichere ich, dass ich die Bachelorarbeit mit dem Thema „Evaluierung von Methoden zur C^{++}-Integration in Python innerhalb des SELMA-Projekts“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Bachelorarbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Ort, Datum	Unterschrift

Kurzfassung

Die Untersuchung der Bodeneigenschaften spielt eine entscheidende Rolle in der Agrarwirtschaft, da sie essenzielle Informationen über den Mineral- und Wassergehalt des Bodens liefert. Diese Informationen sind besonders wertvoll für die Optimierung von Düngemittel- und Bewässerungsstrategien und tragen dazu bei, den steigenden Nahrungsmittelbedarf effizient zu decken. Da Messverfahren wie das Entnehmen von Bodenproben kosten- und zeitaufwändig sind, werden immer häufiger nicht-invasive Messmethoden zur Bodenanalyse eingesetzt.

Diese Bachelorarbeit ist im SELMA-Projekt angesiedelt, das sich mit der Entwicklung von einem mobilen nicht-invasiven Messinstrument zur Bestimmung der elektromagnetischen Leitfähigkeit des Bodens beschäftigt.

Die durch das Messsystem Selma gesammelten Daten werden zunächst in Echtzeit erfasst und ohne eine vorherige Sortierung gespeichert. Für die weitere Verarbeitung und Analyse der Daten wird eine in C^{++} programmierte Software, der „Decoder“, verwendet. Derzeit erfolgt die Auswertung dieser Daten und das Erstellen von Grafiken und Diagrammen mittels Matlab. Zukünftig geplant ist jedoch die Umstellung der Datenverarbeitung und -analyse auf Python, da es eine weit verbreitete Sprache in der wissenschaftlichen Gemeinschaft ist.

Die zentrale Aufgabe besteht darin, Python-Bindings für die bestehende C^{++}-Software zu entwickeln. Dadurch kann Python direkten Zugriff auf die C^{++}-Datenverarbeitung erhalten, ohne dass der Decoder übersetzt werden muss, denn das wäre mit erheblichem Zeit- und Arbeitsaufwand verbunden und könnte zu einem Leistungsverlust führen. Weiterhin soll ein Vergleich verschiedener Methoden zur Erstellung von Python-Bindings, zum Beispiel Pybind11, Ctypes und CFFI erfolgen. Die Analyse dieser Bibliotheken beinhaltet Bewertungen in Bezug auf Benutzerfreundlichkeit, Speichermanagement, Dokumentation, Effizienz und Implementierungsaufwand. Ein Benchmarking dieser Bibliotheken wird ebenfalls durchgeführt. Die Arbeit schließt mit einem umfassenden Vergleich, einem Résumé der Ergebnisse und Erkenntnisse und einem Ausblick ab.

Inhaltsverzeichnis

Abbildungsverzeichnis

Code-Snippets

Abkürzungen

DLL Dynamic Linked Library

GC Garbage Collector

GIL Global Interpreter Lock

SELMA Scalable Electromagnetic Conductivity Measuring

ZEA Zentralinstitut für Engineering, Elektronik und Analytik

1 Einleitung

Diese Arbeit befasst sich mit der Erstellung und dem Vergleich von Python-Bindings für die Datenauswertungs-Software des SELMA-Projekts (siehe Unterkapitel 1.2). Sie wurde im Institutsbereich Systeme der Elektronik des Zentralinstituts für Engineering, Elektronik und Analytik angefertigt. In diesem Kapitel wird daher das Institut sowie das SELMA-Messsystem vorgestellt und anschließend die Motivation der Arbeit erläutert.

1.1 Das Zentralinstitut für Engineering, Elektronik und Analytik

Das Zentralinstitut für Engineering, Elektronik und Analytik (ZEA) am Forschungszentrum Jülich ist in die Entwicklung von technologischen Lösungen wie Geräten, Experimenten, Prozessen, Analyse-, Mess- und Regelungsverfahren, Detektoren sowie Software-Tools und Bildgebungsverfahren involviert. Diese Aktivitäten erfolgen in Kooperation mit einer Vielzahl von Partnerinstitutionen, die sowohl interne Institute des Forschungszentrums als auch andere globale wissenschaftliche Organisationen umfassen [1].

Das ZEA setzt sich dabei aus drei Bereichen zusammen:

1. Engineering und Technologie (ZEA-1)

2. Systeme der Elektronik (ZEA-2)

3. Analytik (ZEA-3)

Diese Arbeit wurde am ZEA-2 erstellt. Deshalb wird dieser Institutsbereich im Folgenden kurz vorgestellt.

Systeme der Elektronik (ZEA-2)

Das ZEA-2 forscht an innovativen komplexen elektronischen und informationstechnischen Systemlösungen. Die Systeme reichen von Sensoren und Detektoren über Signal- und Datenverarbeitung bis hin zu Informationsextraktions- und Visualisierungsverfahren. Dabei fokussiert sich das ZEA-2 unter anderem auf die Entwicklung elektronischer Systeme für Quantum Computing, Neuromorphic Computing, sowie Mess- und Detektorsysteme. Im Bereich der Messsysteme werden vor allem Systeme für tomografische Untersuchungen in den Umwelt- und Geowissenschaften entwickelt [2].

1.2 Das SELMA-Projekt

Das SELMA-Projekt ist im Forschungsbereich der Messsysteme angesiedelt. SELMA steht für **S**calable **El**ectro**ma**gnetic Conductivity Measuring und ist ein mobiles Instrument für die tomografische Analyse der Bodenstruktur.

Ein fundiertes Verständnis der Bodeneigenschaften und -qualität ist in der heutigen Landwirtschaft von großer Bedeutung. Die elektrische Leitfähigkeit des Bodens liefert dabei entscheidende Informationen über den Mineral- und Wassergehalt, die erheblichen Einfluss auf die Ernteergebnisse haben. Die Kenntnis dieser Faktoren ist essenziell für die Optimierung von Düngung, Bewässerung und Anbautechniken. Nicht-invasive Untersuchungsmethoden spielen dabei eine wichtige Rolle, da sie es ermöglichen, den Boden ohne direkte Eingriffe zu analysieren, umweltschonende Bewertungen durchzuführen und landwirtschaftliche Methoden optimal an die spezifischen Bodenverhältnisse anzupassen, um den zukünftigen Nahrungsmittelbedarf zu sichern [3].

Das IBG-3 des Instituts für Bio- und Geowissenschaften am Forschungszentrum Jülich konzentriert sich mit seinem Forschungsanliegen auf die Agrosphäre [4]. Im Fokus steht hierbei, die Auswirkungen des Klimawandels durch geeignete Anpassungs- und Managementstrategien abzuschwächen und Modelle für Vorhersagen zu entwickeln. Durch geophysikalische und hydrologische Messverfahren soll ein vertieftes Verständnis der Wechselwirkungen zwischen Boden und Grundwasser sowie der chemischen Prozesse im Boden gewonnen werden, speziell für landwirtschaftliche Zwecke. Das IBG-3 erforscht verschiedene Methoden zur Analyse der Bodenzusammensetzung und hat bereits Erfahrung mit kommerziellen Bodenuntersuchungssystemen gesammelt [5]. Um die Bodeneigenschaften im kompletten Bereich in den oberen fünf Metern mit einer ausreichenden Genauigkeit zu analysieren, wird bisher mit verschiedenen kommerziellen Multisensorsystemen gemessen. Problematisch ist dabei die gleichzei-

tige Anwendung der Systeme, da sich diese gegenseitig stören. In Kooperation mit dem ZEA-2 wurde deshalb das SELMA-Projekt initiiert, um ein genaueres und skalierbares System für die tomografische Untersuchung des Bodens zu entwickeln.

Das Hauptanliegen des Projekts ist es, die Funktionsweise bestehender Messgeräte zu erfassen und eine verbesserte und weiterentwickelte Alternative zu schaffen. Das neue Messsystem SELMA wird für den mobilen Einsatz entwickelt und zeichnet sich durch seine Skalierbarkeit und die große Anzahl an Empfängerspulen aus. Für die ersten Versuche werden 12 Sender-Empfänger-Spulenpaare mit steigendem Spulenabstand von 0,3 bis 3,6 Meter geradlinig angeordnet. So werden Messungen bis in Tiefen von 5 Metern bei Ackerflächen ermöglicht. Das Messsystem wird dabei mit einem Fahrzeug über das zu untersuchende Gebiet bewegt, siehe Abbildung 1.1.

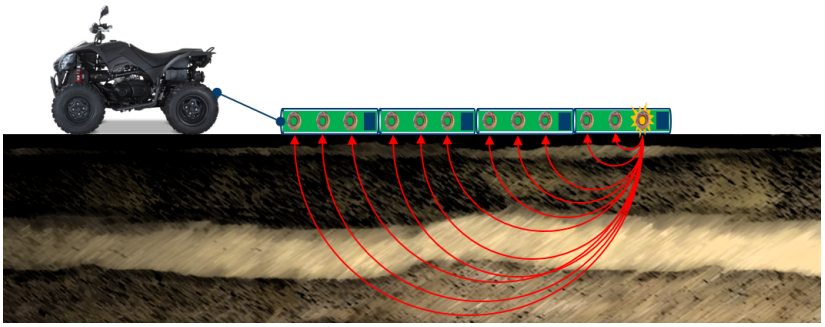

Abbildung 1.1: Veranschaulichung der Messung mit SELMA

Die Anpassungsfähigkeit des Systems ermöglicht die Nutzung auf verschiedenen Plattformen wie Traktoren oder eventuell zukünftig auch Drohnen.

1.3 Motivation

Die während der Messung gesammelten Daten werden zunächst in Echtzeit mit einer in C^{++} erstellten Firmware erfasst und in Dateien gespeichert, ohne vorherige Sortierung oder Kategorisierung. Für die weitere speicher- und zeiteffiziente Verarbeitung der Daten wird eine ebenfalls in C^{++} programmierte Software - der Decoder - verwendet. Damit werden die Daten aus den Dateien gelesen und sortiert, um die weitere Verarbeitung und Analyse vorzubereiten. Derzeit erfolgt die Auswertung und das Erstellen von Grafiken mittels der lizenzpflichtigen Software Matlab. Matlab ist eine Plattform, die speziell für numerische Berechnungen entwickelt wurde. Sie wird in wissenschaftlichen Bereichen unter anderem für Aufgaben wie Datenanalyse und Simulation eingesetzt [6].

In der Zukunft ist geplant, die Datenverarbeitung und -analyse auf die Programmiersprache Python umzustellen. Der Hauptgrund für den Übergang zu Python ist die weite Verbreitung in der wissenschaftlichen Gemeinschaft und die Reduzierung der Lizenzkosten. Python ist einfach in bestehende Projekte zu integrieren und bietet eine Vielzahl von frei zugänglichen Bibliotheken und Tools, die speziell für den Einsatz in der Datenanalyse und in anspruchsvollen numerischen Berechnungen entwickelt wurden [7, 8]. Da Python außerdem unter einer Open-Source-Lizenz steht, ist es kostenfrei nutzbar, sogar für kommerzielle Zwecke [9]. Das macht Python zu einer idealen Wahl, um Matlab zu ersetzen, ohne an Effizienz und Effektivität der Datenverarbeitung und -analyse einzubüßen [10].

Für eine direkte Nutzung des in C^{++} geschriebenen Decoders über Python ist die Entwicklung einer „Binding“-Schnittstelle notwendig. Diese Schnittstelle erlaubt es Python, auf die Funktionen des Decoders zuzugreifen, ohne diesen neu schreiben zu müssen oder den Effizienzvorteil von C^{++} vollständig zu verlieren. Das vermeidet erheblichen Zeit- und Arbeitsaufwand, da es sich bei der Software um das Produkt jahrelanger Entwicklung handelt. Das Ziel der Arbeit liegt daher nicht nur im Entwerfen dieser Python-Bindings, sondern auch in der Untersuchung und dem Vergleich verschiedener Tools. Hierbei werden die Leistungsfähigkeit und Effizienz der verschiedenen Ansätze anhand spezifischer Kriterien bewertet. Zu diesen Kriterien gehören unter anderem die Geschwindigkeit der Datenverarbeitung bzw. Laufzeit, die Speichernutzung und den Aufwand bei der Integration in bestehende Systeme. Diese Kriterien sind besonders wichtig, da die Datenverarbeitung im SELMA-Projekt große Datensätze von bis zu 125 Gigabyte pro gemessene Stunde umfasst und daher eine hohe Leistungsfähigkeit und Effizienz erfordert. Diese Untersuchungen können auch für weiterführende Projekte im ZEA-2 sehr hilfreich sein, da zur sensornahen Erfassung und Verarbeitung der Messdaten neben den Hardware-Beschreibungssprachen (wie zum Beispiel VHDL) fast ausschließlich C^{++} verwendet wird.

2 Grundlagen

Bindings sind Verbindungselemente in der Softwareentwicklung, die den Zugang zu einer Bibliothek, die in einer bestimmten Programmiersprache geschrieben wurde, in einer anderen Sprache ermöglichen. Die Entwicklung eines solchen Bindings erfolgt durch das Erstellen einer „Wrapper“-Bibliothek, die als Zwischenschicht dient. Dieser Wrapper umschließt die originale Bibliothek und bietet eine Schnittstelle, die sowohl die Syntax als auch die Funktionalitäten der Zielprogrammiersprache unterstützt.

Das Erstellen von Bindings kann eine besondere Herausforderung darstellen, da es fundamentale technische Unterschiede zwischen C^{++} und Python gibt. Diese Unterschiede reichen von den Typsystemen und Speicherverwaltungsmethoden bis hin zu den Laufzeitumgebungen, was das Design von effizienten und robusten Schnittstellen erschwert. Ein tiefgehendes Verständnis dieser Unterschiede ist daher wichtig, um Bindings zu entwickeln, die nicht nur funktionell sind, sondern auch die Performance- und Kompatibilitätserwartungen erfüllen.

2.1 Unterschiede C^{++} und Python

Im Folgenden wird detailliert erläutert, wie diese technischen Divergenzen die Entwicklung von Python-Bindings für C^{++}-Bibliotheken beeinflussen und bei der Auswahl von Strategien berücksichtigt werden müssen.

2.1.1 Datenkonvertierung

Python und C^{++} speichern ihre Datentypen intern auf unterschiedliche Weise. Unterschiede in den Sprachen zeigen sich beispielsweise in der Speicherung von Integer-Werten, was hauptsächlich auf die unterschiedlichen Typsysteme und Speicherverwaltungskonzepte der beiden Sprachen zurückzuführen ist. Python verwendet ein dynamisches Typsystem, wobei Integer wie alle anderen Datentypen als Objekte behandelt werden [11]. Jedes Integer-Objekt in Python beinhaltet nicht nur den eigentlichen Wert, sondern auch zusätzliche Informationen wie Typinformationen und

Referenzzähler, die für das automatische Speichermanagement benötigt werden [12]. Dies erhöht den Speicherbedarf für einen Integer in Python im Vergleich zu C++, führt aber zu einer größeren Flexibilität im Umgang mit Datentypen. Im Gegensatz dazu ist C++ eine statisch typisierte Sprache, bei der die Größe eines Integers zur Kompilierzeit festgelegt ist und direkt abhängig von der Plattform und dem Compiler ist (typischerweise 16, 32 oder 64 Bit) [13, 14]. C++ speichert Integer als „nackte“ Daten direkt im Speicher ohne zusätzliche Verwaltungsinformationen, was eine effizientere Speichernutzung, jedoch weniger Flexibilität im Vergleich zu Python bedeutet. Um also Daten zwischen den beiden Sprachen austauschen zu können, muss man die verschiedenen Typen umwandeln (engl. marshalling). Marshalling bezeichnet die Umwandlung von Daten in ein Format, das zur Übergabe an andere Prozesse geeignet ist, also sozusagen die Vorbereitung auf die Übertragung zwischen den beiden Sprachen. Dort müssen diese dann wieder in ihre ursprüngliche Struktur gebracht werden (engl. unmarshalling). Dieser Prozess ist wichtig, um die Datentypen richtig zu speichern und möglichst wenig Speicher zu verschwenden. Die Umwandlung von anderen typischen Datentypen wie Strings oder Complex in Python und C++ folgt ähnlichen Prinzipien wie bei den Integer-Werten, jedoch mit ihren eigenen Herausforderungen. Strings in Python sind beispielsweise Unicode-Objekte, die eine variable Byteanzahl pro Zeichen verwenden können, abhängig von den spezifischen Zeichen, die der String enthält. Dies macht Python-Strings sehr flexibel, aber auch speicherintensiver im Vergleich zu C++. Dort werden Strings üblicherweise als `std::string` oder als Arrays von Zeichen (`char arrays`) dargestellt. `std::string` ist eine Klasse, die eine dynamische Speicherverwaltung ermöglicht und mehrere Bytes pro Zeichen belegen kann. Bei der Verwendung von char arrays ist die Anzahl der Bytes pro Zeichen typischerweise ein Byte im ASCII-Format, kann jedoch auch mehrere Bytes in UTF-8 belegen. Für komplexe Datentypen wie komplexe Zahlen (`complex` in Python und `std::complex` in C++) wird das Marshalling noch komplizierter. Python speichert komplexe Zahlen als Objekte mit zwei Float-Werten, die den Real- und den Imaginärteil repräsentieren. In C++ werden diese als zwei separate Gleitkommazahlen innerhalb einer Template-Klasse gespeichert [15]. Dabei ist zu beachten, dass die Klasse selbst nicht direkt umgewandelt werden kann, sondern ihre einzelnen Komponenten separat behandelt werden müssen. Das Marshalling dieser Daten erfordert eine sorgfältige Behandlung, um sicherzustellen, dass die numerischen Werte korrekt übertragen werden und keine Präzision verloren geht.

2.1.2 Veränderliche und unveränderliche Datentypen

Beim Schreiben von Python-Bindings ist es außerdem wichtig, die Unterschiede in der Datenhandhabung beider Sprachen zu verstehen. In Python sind die primitiven Datentypen grundsätzlich immutable (unveränderlich) [16].

```
i = 10
i += 1
```

Code-Snippet 2.1: Änderung eines Integers in Python

Dieser Code in Ausschnitt 2.1 lässt es so aussehen, als ob der Wert von i geändert, also inkrementiert werden würde. Was intern jedoch tatsächlich passiert, ist, dass der ursprüngliche Integer 10 unveränderlich ist und somit unverändert bleibt [17]. Der Ausdruck `i += 1` erstellt einen neuen Integer mit dem Wert 11 und weist diesen dann der Variablen i zu. Der alte Wert 10 bleibt unangetastet und wird nicht mehr referenziert, wenn kein anderer Verweis darauf besteht. Im Gegensatz dazu würde in einer Sprache wie C++, wo Integer wie alle primitiven Datentypen mutable sind, das Inkrementieren eines Integer-Wertes wie mit `i++` den bestehenden Wert von i direkt ändern, ohne einen neuen Integer zu erzeugen. Das Konzept der Immutabilität in Python führt dazu, dass Operationen, die wie eine Modifikation eines Objektes aussehen, in Wirklichkeit oft die Zuweisung eines neuen Objektes zu einer Variablen sind [18]. Dieses Konzept hat auch Auswirkungen auf die Argument-Übergabe an Funktionen. Argumente in Python werden immer als Referenzen auf Objekte übergeben. Ob Änderungen innerhalb der Funktion den Zustand der Argumente außerhalb der Funktion beeinflussen, hängt dann davon ab, ob die Objekte veränderlich oder unveränderlich sind. Dieses Verhalten ist manchmal verwirrend, weil es sowohl Aspekte von „call by value“ als auch „call by reference“ in sich vereint, je nachdem, mit welcher Art von Objekt man arbeitet. Jede scheinbare Modifikation eines unveränderlichen Objekts wie z. B. einem Integer, Float, oder String führt zur Erstellung und Rückgabe eines neuen Objekts. Dies verhält sich ähnlich wie „call by value“, da das Originalobjekt unverändert bleibt. Bei veränderlichen Objekten wie Listen oder Dictionarys kann die Funktion Änderungen am übergebenen Objekt vornehmen, die auch außerhalb der Funktion sichtbar sind. Dies ist dem Prinzip „call by reference“ ähnlich, da die Referenz selbst nicht geändert wird, sondern das durch die Referenz referenzierte Objekt. Das kann anhand eines Beispiels verdeutlicht werden:

```
def modify(x):
    x.append(1)

list = []
modify(list)
print(list)  # Ausgabe: [1]
```

Code-Snippet 2.2: Modifizieren einer Liste in Python innerhalb einer Funktion

In diesem Beispiel 2.2 wird die Liste `list` an die Funktion `modify` übergeben, die ein Element zur Liste hinzufügt. Die Änderung ist außerhalb der Funktion sichtbar, weil die Liste in Python ein veränderliches Objekt ist.

```python
def modify(x):
    x = 10

y = 5
modify(y)
print(y)  # Ausgabe: 5
```

Code-Snippet 2.3: Modifizieren eines Integers in Python innerhalb einer Funktion

In Beispiel 2.3 wird der Wert 5, ein unveränderlicher Integer, an die Funktion übergeben. Die Zuweisung innerhalb der Funktion ändert nur den lokalen Parameter `x` und hat keinen Einfluss auf das ursprüngliche Argument `y`.

Dieses Konzept lässt sich in C++ durch Referenzen realisieren. In Beispiel 2.4 wird zunächst demonstriert, was bei der Übergabe ohne Referenz geschieht.

```cpp
#include <iostream>

void modify(int x) {
    x = 10;
}

int main() {
    int num = 5;
    modify(num);
    std::cout << "num: " << num << std::endl;  // Ausgabe: num: 5
    return 0;
}
```

Code-Snippet 2.4: Übergabe eines Arguments call-by-value in C++

In Codeausschnitt 2.4 wird das Argument `num` an die Funktion `modify` als Kopie, also „call by value“ übergeben. Die Änderungen in der Methode `modify` sind nicht global wirksam.

```cpp
#include <iostream>

void modify(int& x) {
    x = 10;
}

int main() {
    int num = 5;
    modify(num);
    std::cout << "num: " << num << std::endl;  // Ausgabe: num: 10
    return 0;
}
```

Code-Snippet 2.5: Übergabe eines Arguments call-by-reference in C++ durch Referenz

In Code-Snippet 2.5 wird `num` als Referenz übergeben. Die Änderung in der Methode hat also auch Auswirkungen auf die Variable innerhalb der `main`.

Alternativ kann auch ein Pointer auf `num` übergeben werden. Um den Wert zu erhöhen, muss dann in der Funktion dereferenziert werden, wie in folgendem Beispiel 2.6 demonstriert wird.

```cpp
#include <iostream>

void modify(int* x) {
    *x = 10;
}

int main() {
    int num = 5;
    modify(&num);
    std::cout << "num: " << num << std::endl;  // Ausgabe: num: 10
    return 0;
}
```

Code-Snippet 2.6: Übergabe eines Arguments call-by-reference in C++ durch Pointer

Möchte man also beispielsweise eine C++-Funktion aus Python heraus aufrufen, die einen übergebenen Integer-Wert erhöht, wäre es ineffizient, einen Python-Integer direkt zu übergeben. Da Python-Integers unveränderlich sind, müsste bei jeder Modifikation ein neuer Integer erstellt werden. Die effizienteste Lösung ist daher die Konvertierung des Python-Integers in einen C-Integer. Hierbei ist wieder die korrekte Umwandlung zu beachten, siehe Abschnitt 2.1.1. Zudem wäre für eine globale Änderung des Werts die Übergabe einer Referenz oder eines Zeigers und die Rückgabe des Objekts in der Funktion erforderlich. Da Python jedoch nicht direkt mit Zeigern arbeitet, gestaltet sich der Zugriff auf einen Zeiger zu einem Objekt schwierig. Einige der später vorgestellten Tools ermöglichen jedoch die Nutzung von Zeigern und speziellen C-Typen in Python, indem sie entsprechende Mechanismen und Datentypkonvertierungen anbieten, die eine effiziente Übergabe von Daten an C/C++-Funktionen erleichtern.

Ein weiterer Ansatz zur Bewältigung der Einschränkungen durch die unveränderlichen Datentypen in Python ist die Rückgabe von Tupeln aus C++-Funktionen, die mit dem Binding gebunden sind. Ein Tupel ist eine unveränderliche und geordnete Sammlung von Elementen, die es ermöglicht, mehrere Datenpunkte in einem einzigen Objekt zu gruppieren [19]. Das Konzept der Rückgabe von Tupeln basiert darauf, dass komplexe Funktionen, die möglicherweise mehrere Werte oder Objekte ändern, ihre Ergebnisse in einem einzigen Objekt zurückgeben, das mehrere Datenpunkte umfasst. Anstatt zu versuchen, unveränderliche Typen direkt zu verändern, werden die veränderten

Werte zusammen mit zusätzlichen Informationen als Tupel zurückgegeben. Das ist in folgendem Beispiel 2.7 skizziert.

```
std::tuple<int, bool> increment_and_bool (int value) {
    int new_value = value + 1;
    bool is_greater_than_ten = new_value > 10;

    // Rückgabe eines Tupels mit beiden Werten
    return std::make_tuple(new_value, is_greater_than_ten);
}
```

Code-Snippet 2.7: Umgehung der Immutabilität durch Mehrfachrückgabe mit Tupeln

2.1.3 Speichermanagement

Was das Verwalten von Speicherallokationen und -freigaben angeht, übernimmt in Python der Garbage Collector (GC) die Freigabe nicht mehr benötigter Objekte basierend auf Referenzzählung und Erkennung zyklischer Referenzen [12]. Dieser Mechanismus sorgt dafür, dass Speicherlecks vermieden werden. C++ hingegen erfordert eine manuelle Speicherverwaltung, bei der der Entwickler für die Allokation und die Freigabe von Speicher verantwortlich ist [20]. Diese Unterscheidung erfordert eine präzise Synchronisation der Lebenszyklen von Objekten, die zwischen den beiden Sprachen übergeben werden [21].

Ein zentrales Problem bei der Bindung von C++-Funktionen an Python ist vor allem die Verantwortungsfrage für die Lebensdauer und Verwaltung der zurückgegebenen Objekte [22]. Hierbei lassen sich zwei Arten von Rückgaben unterscheiden:

1. **Rückgabe per Pointer:** Python erhält einen Pointer auf ein C++-Objekt. Dies kann zu Problemen führen, da Python fälschlicherweise versuchen könnte, die Speicherverwaltung für dieses Objekt zu übernehmen. Besonders kritisch ist dies, wenn das Objekt in C++ weiterhin genutzt wird oder es sich um ein statisches Objekt handelt.

2. **Rückgabe per Wert:** Python erhält eine Kopie des Objekts. Diese Methode ist in der Regel sicher, da die Speicherverwaltung der Kopie völlig unabhängig vom Originalobjekt in C++ erfolgt.

Die größte Herausforderung stellt die Rückgabe per Pointer dar, da hier die Zuständigkeit für das Speichermanagement und die korrekte Deinitialisierung des Objekts oft unklar bleibt.

Es muss sichergestellt werden, dass Speicher, der in C++ alloziert wurde, korrekt freigegeben wird, ohne die Stabilität des Python-Programms zu beeinträchtigen. Hierbei sind oft Brückenkonstruktionen wie Wrapper nötig, die eine korrekte Integration der Speicherverwaltungsroutinen beider Sprachen ermöglichen. Objekte, die in C++ erstellt und an Python übergeben werden, müssen außerdem in einer Weise verwaltet werden, die es dem Python-GC ermöglicht, sie korrekt zu behandeln. Dies kann durch das Anpassen der Referenzzählung erfolgen, sodass das C++-Objekt die Referenzzählung des Python-Objekts bei Bedarf erhöht oder verringert. Alternativ können Smart Pointer in C++ verwendet werden, die automatisch die Lebensdauer von Objekten durch eingebaute Mechanismen wie Referenzzählung verwalten und mit dem GC-System von Python kompatibel sind [23]. Die Behandlung von zyklischen Referenzen ist besonders kritisch, wenn C++-Objekte, die Python-Objekte enthalten, wiederum von anderen Python-Objekten referenziert werden. Solche zyklischen Abhängigkeiten können dazu führen, dass der Python-GC den Speicher nicht korrekt freigibt, weil die Referenzzählung der beteiligten Objekte nie auf null fällt. Hier müssen spezielle Techniken implementiert werden, wie das Einführen von schwachen Referenzen (weak references) [24], die es ermöglichen, Zyklen zu erkennen und aufzulösen, ohne dass dies zu Speicherlecks führt.

2.1.4 Thread-Sicherheit

Beim Schreiben von Python-Bindings für C++-Systeme, die mit Multithreading arbeiten, muss zusätzlich die Thread-Sicherheit beachtet werden.

Der Global Interpreter Lock (GIL) in Python beschränkt die Ausführung von Python-Bytecode auf einen einzigen Thread gleichzeitig, wodurch die vollständige Nutzung von Multithreading-Fähigkeiten eingeschränkt wird [25]. Trotz der Einschränkungen ist Multithreading außerhalb des Python-Bytecodes möglich. Operationen wie Ein-/Ausgabe, Netzwerkkommunikation oder der Zugriff auf externe Bibliotheken können beispielsweise parallel ausgeführt werden. Um Thread-Sicherheit zu gewährleisten, müssen C++-Threads, die auf Python-Objekte zugreifen, den GIL entsprechend handhaben. Es ist jedoch wichtig zu beachten, dass der GIL nur für C++-Code freigegeben werden sollte, der keine Python-Objekte manipuliert oder auf Python-Code zugreift. Entwickler müssen sicherstellen, dass ihre C++-Funktionen threadsicher sind, bevor sie den GIL freigeben.

Zur Wahrung der Datenkonsistenz ist es ratsam, Daten in native C++-Strukturen zu kopieren, bevor sie durch mehrere Threads verarbeitet werden. Dies reduziert das Risiko von Synchronisationsfehlern und minimiert die Notwendigkeit, den GIL während der Datenverarbeitung zu halten. Bei richtiger Handhabung ermöglicht eine sorgfältige Planung und Implementierung des Python-C++-Bindings, die volle Rechenleistung

moderner Mehrkernprozessoren auszuschöpfen, während die Flexibilität und Einfachheit von Python als Schnittstelle erhalten bleibt.

2.1.5 Fehlerbehandlung

C++ und Python nutzen beide das Konzept der strukturierten Ausnahmebehandlung, wobei Ausnahmen mit `throw` bzw. `raise` geworfen und dann mit `catch` bzw. `except` gefangen werden [26]. Allerdings ist die Präzision der Fehlerbehandlung beiden Sprachen stark von der Implementierung des Programmierenden abhängig, da dieser entscheidet, wann und wie Ausnahmen behandelt werden, was eine flexible Handhabung von Fehlern ermöglicht. Es wird nicht gefordert, dass Funktionen deklarieren, welche Arten von Ausnahmen sie werfen könnten, was die Entwicklung vereinfacht, aber auch zu weniger Vorhersehbarkeit bei der Fehlerbehandlung führen kann. Unterschiede sind hier nur in der Art der Ausnahmebehandlung festzustellen. C++ verwendet typisierte Ausnahmen, während Python untypisierte Ausnahmen verwendet. Das bedeutet, dass in C++ Ausnahmen oft durch spezifische Klassen repräsentiert werden, während in Python jede beliebige Instanz als Ausnahme geworfen werden kann [27, 28]. Diese Unterschiede müssen bei der Übersetzung von Ausnahmen zwischen den beiden Sprachen berücksichtigt werden. Wenn Fehler in C++ geworfen werden, müssen sie beim Schreiben von Python-Bindings so behandelt werden, dass sie in Python als entsprechende Python-Errors erscheinen. Dies erfordert, dass die C++-Exceptions korrekt gefangen und in Python-Errors umgewandelt werden, damit sie vom Python-Code korrekt verarbeitet werden können.

3 Überblick über C++-Bindings

Da die grundlegenden Unterschiede von Python und C++, die beim Schreiben von Bindings zwischen den Sprachen beachtet werden müssen, erläutert wurden, werden im Folgenden einige Binding-Tools vorgestellt.

3.1 Ctypes

Ctypes ist ein vielseitiges Modul der Standardbibliothek von Python, das das Aufrufen von Funktionen aus dynamisch geladenen C-Bibliotheken ermöglicht. Dies erleichtert die Erstellung von Wrappern für Code in C/C++, indem es eine direkte Schnittstelle zu den entsprechenden Bibliotheken bietet. Ctypes schlägt eine Brücke zwischen der Python-Umgebung und dem maschinennahen C/C++-Code, indem es C-kompatible Datentypen anbietet [29].

Dynamisch geladene Bibliotheken, auch bekannt als DLL (`.dll`) unter Windows, Shared Objects (`.so`) unter UNIX-basierten Systemen wie Linux, und Dynamic Libraries (`.dylib`) unter macOS, werden zur Laufzeit eines Programms geladen. Im Gegensatz zu statisch geladenen Bibliotheken, die bereits beim Kompilieren des Programms eingebunden werden, wird der Code nur bei Bedarf geladen [30]. In der Praxis bedeutet das, dass Programme schneller starten und effizienter laufen, da nur die wirklich benötigten Ressourcen geladen werden.

Datenkonvertierung

In der Interaktion zwischen Python und C mittels Ctypes ist die Datenkonvertierung ein zentraler Prozess, da sie die Grundlage für den Datenaustausch zwischen den Sprachen bildet. Ctypes erleichtert das Marshalling und Unmarshalling, indem es C-kompatible Datentypen anbietet, die die direkte Interaktion mit C-Bibliotheken ermöglichen.

Beim Marshalling werden die Daten in eine Form konvertiert, die über die Sprachgrenzen hinweg übertragen werden kann. Ctypes übernimmt diese Aufgabe in Python-Bindings, indem es Python-Daten in C-Datentypen umwandelt. Zum Beispiel kann ein Python-Integer in einen C-Integer konvertiert werden:

```
import ctypes

# Python integer
p = 42

# Konvertieren in einen C-Integer
x = ctypes.c_int(p)
```

Code-Snippet 3.1: Marshalling mit Ctypes

Das Unmarshalling ist der umgekehrte Prozess, bei dem die Daten zurück in Python-Datentypen konvertiert werden, nachdem sie in C verarbeitet wurden. Diese Konvertierungen nehmen dem Entwickler Arbeit ab und erleichtern so das Schreiben von Bindings zwischen C/C++ und Python.

Veränderliche und unveränderliche Datentypen

In Abschnitt 2.1.2 wurde bereits die Unterscheidung zwischen veränderlichen (mutablen) und unveränderlichen (immutablen) Datentypen in Python sowie deren Bedeutung für die Interaktion mit C++ diskutiert. Besonders im Kontext der Python-C++-Bindings ist diese Unterscheidung relevant, da die Übergabe von Daten zwischen den beiden Sprachen sorgfältig verwaltet werden muss, um Effizienzverluste und unwirksame Modifikationen zu vermeiden.

Immutable Datentypen wie Integer und Strings in Python können nicht modifiziert werden; stattdessen wird bei scheinbaren Modifikationen eine neue Instanz generiert und zugewiesen. Diese Eigenschaft führt zu Herausforderungen, wenn solche Typen effizient an eine C++-Funktion übergeben werden sollen, die Modifikationen am übergebenen Wert vornimmt. Mit Ctypes ist es möglich, C-Datentypen in Python zu definieren, die direkt von C++-Funktionen manipuliert werden können. Zum Beispiel ermöglicht Ctypes das Erstellen von mutablen Datentypen, wie `c_int` oder `c_double`, die effizient Werte an C++-Funktionen übergeben können, welche die Daten direkt modifizieren, ohne dass eine Kopie erstellt wird. Das Konzept von Immutabilität und Mutabilität wird hierbei elegant umgangen, indem die Python-Typen in kompatible C-Datentypen konvertiert werden, die die direkte Manipulation unterstützen [29].

Ein Beispiel hierfür ist die Übergabe eines `ctypes.c_int` an eine C++-Funktion, die den Wert direkt modifiziert, anstatt eine Kopie zu verwenden. Der Vorteil dieser Methode ist die Erhaltung von Speicher- und Ausführungseffizienz, da unnötige

Objekterstellungen vermieden werden und die Funktion direkt auf den Speicher des übergebenen Wertes wirken kann. Allerdings bringt diese Methode auch einige Nachteile mit sich, zum Beispiel die erhöhte Komplexität und Fehleranfälligkeit bei der Handhabung solcher Direktmanipulationen. Da C++ eine niedrigere Ebene der Abstraktion bietet und direkter auf Hardware-Ressourcen zugreift, kann ein Fehler in der Speicherverwaltung oder in der Übergabe der Parameter zu schwerwiegenden Fehlern führen, wie z.B. Speicherlecks oder Speicherüberschreibungen, die schwer zu diagnostizieren und zu beheben sind [31].

Darüber hinaus erlaubt Ctypes die Nutzung von Pointern und Referenzen in einer Art und Weise, die mit C++ vergleichbar ist. Dies ist besonders nützlich, wenn Funktionen in C++ geschrieben sind, die Parameter per Referenz oder über Pointer erwarten. Zum Beispiel kann ein `ctypes.POINTER(c_int)` oder ein `ctypes.byref(c_int())` genutzt werden, um eine Referenz bzw. einen Pointer auf eine Variable zu übergeben. Dies ermöglicht es C++-Funktionen, den Wert direkt zu ändern, ohne eine Kopie anzufertigen.

Speichermanagement

Wie bereits in Abschnitt 2.1.3 festgehalten, obliegt es dem Entwickler, den Speicher in C/C++ eigenständig zu verwalten. Dies steht im Kontrast zu Python, wo der GC sich um die Freigabe von Speicher kümmert. Objekte, die in Python erstellt werden, einschließlich jener, über die Ctypes-Bibliothek definierten C-typischen Datentypen, werden vom GC überwacht. Die ctypes-Bibliothek ermöglicht es, dass solche Datentypen wie `c_int`, `POINTER(c_int)`, Strukturen und weitere C-kompatible Typen von der C-Schnittstelle manipuliert werden können.

Trotz der Fähigkeit, von C aus manipuliert zu werden, bleiben diese Typen jedoch unter der Aufsicht des Python-GC. Dies bedingt eine präzise Steuerung ihrer Lebenszyklen, um sicherzustellen, dass sie nicht in Python freigegeben werden, solange sie noch in der C-Umgebung aktiv genutzt werden. Das Halten der entsprechenden Python-Referenzen für die Dauer ihrer Notwendigkeit ist daher sehr wichtig, um Probleme wie hängende Zeiger (engl. dangling pointer) oder unerwartetes Verhalten zu verhindern.

In C/C++ wird Speicher häufig mit `new` alloziert, wie im Fall der Funktion `add_vector` aus Code-Snippet 3.2, die Speicher für ein Array von Double-Werten reserviert. Dieser Speicher muss später manuell mit `delete` freigegeben werden, um Speicherlecks zu vermeiden.

```
double* add_vector(double *a, double *b, int len){
    // Allokiert Speicher für den Ergebnisvektor c mit der Länge len
    double* c = new double[len];
    // für jedes Element der Eingabevektoren
    for (int i=0; i < len; ++i)
        // Addiert die entsprechenden Elemente von a und b
        c[i] = a[i] + b[i];
    // Gibt den Zeiger auf den neuen Vektor c zurück
    return c;
}
```

Code-Snippet 3.2: Speicherallokation mit `new` in C++

Auch auf der Python-Seite muss speziell darauf geachtet werden, dass der durch C++ reservierte Speicher korrekt freigegeben wird, da der GC diesen nicht automatisch erfasst. Der einzige sichere Weg diesen Speicher wieder freizugeben, ohne Speicherlecks etc. zu riskieren, ist die Freigabe auf der C++-Seite. Zwar ist der Zugriff auf Speicherfunktionen über die Bindings theoretisch möglich, aber Funktionen wie `malloc` und `free` von der C-Bibliothek in Python verhalten sich anders als die C++-spezifischen Operatoren `new` und `delete`, da `malloc` und `free` keinen Konstruktor oder Destruktor aufrufen [32]. In Bezug auf die Verwendung von Ctypes gibt es keine eingebaute Lösung für die Handhabung der Speicherfreigabe, die automatisch mit dem GC synchronisiert wäre. Die empfohlene Verfahrensweise besteht darin, die Speicherfreigabe explizit zu steuern, indem C-Funktionen zur Freigabe des Speichers aufgerufen werden, sobald die entsprechenden Python-Objekte nicht mehr benötigt werden. Entwickler müssen daher eine sorgfältige Speicherverwaltung betreiben.

Thread-Sicherheit

Die Verwendung von Ctypes ermöglicht es, den GIL von Python zu umgehen. Der GIL beschränkt die gleichzeitige Ausführung von Python-Code auf einen Thread, um Konsistenz und Datenintegrität zu gewährleisten, siehe Abschnitt 2.1.4. Der entscheidende Vorteil von Ctypes besteht darin, dass während der Ausführung einer durch Ctypes aufgerufenen C-Funktion der GIL temporär freigegeben wird [33]. Dies erlaubt es anderen Python-Threads, parallel fortzufahren. Solch eine Freigabe des GIL kann die Leistungsfähigkeit von Python-Anwendungen auf Multicore-Systemen erheblich steigern, da es echte parallele Ausführung von Threads ermöglicht.

Trotz dieser signifikanten Vorteile hinsichtlich der Leistungssteigerung durch Parallelverarbeitung ist bei der Nutzung von Ctypes besondere Vorsicht geboten. Wenn die verwendete C-Bibliothek nicht threadsicher ist, kann paralleler Zugriff durch mehrere Threads in Datenkorruption und Systemabstürzen resultieren. Entwickler müssen

daher sicherstellen, dass alle verwendeten externen Bibliotheken für den Einsatz in einem Multithreading-Kontext geeignet und entsprechend sicher konzipiert sind.

Fehlerbehandlung

Ctypes wurde für die Arbeit mit C-Bibliotheken entwickelt und berücksichtigt die Erweiterungen von C++ wie Exceptions und Objekte nicht. Daher fehlen Ctypes die Mechanismen, um C++-Exceptions zu erkennen oder zu behandeln. Wenn eine über Ctypes aufgerufene C++-Funktion eine Exception auslöst, die nicht innerhalb der C++-Umgebung abgefangen wird, führt dies in der Regel zu einem unerwarteten Absturz des aufrufenden Python-Programms. Daher sollte Ctypes vorzugsweise für C++-Bibliotheken eingesetzt werden, die ihre Ausnahmen intern handhaben. Um dennoch C++-Code und dessen Fehlermanagement in Python zu integrieren, ohne die Stabilität zu gefährden, kann eine C++-Code in externen C-Funktionen gekapselt werden, der in der Lage ist, C++-Exceptions zu fangen und in Fehlercodes umzuwandeln. Diese Fehlercodes können dann in Python überprüft und als Python-Exceptions geworfen werden, siehe Beispiel 3.10.

3.2 Pybind11

Pybind11 ist eine leistungsfähige Bibliothek für C++-Entwickler, die eine nahtlose Integration von C++-Code in Python ermöglicht. Durch seine Header-Only-Architektur vereinfacht sie den Einbindungsprozess des C++-Codes, da keine vorkompilierten Bibliotheken notwendig sind. Im Vergleich zu Ctypes, das primär für C-Bindings konzipiert ist, bietet Pybind11 spezifische Unterstützung für C++-Funktionalitäten, einschließlich Klassen, Templates und Exceptions, was den Austausch und die Manipulation von Daten zwischen den beiden Sprachen erheblich erleichtert.

Datenkonvertierung

Pybind11 erleichtert die Konvertierung von Daten zwischen Python und C++ durch automatische Typumwandlungen für die Standardtypen, aber eben auch für viele komplexere Datentypen wie Vektoren und Maps.

Zur Veranschaulichung der automatischen Typkonvertierung, wird im folgenden Beispiel 3.3 die Bindung einer C++-Funktion `add`, die zwei Integer addiert, mittels Pybind11 dargestellt. Das erstellte Python-Modul wird `example` benannt.

```cpp
#include <pybind11/pybind11.h>
#include "meine_bib.h"

PYBIND11_MODULE(example, m) {
    m.def("add", &add, "A function which adds two numbers");
}

// Code aus meine_bib.cc
int add(int a, int b) {
    return a + b;
}
```

Code-Snippet 3.3: Binden einer add-Methode mit Pybind11

Wenn Python die `add`-Funktion aus dem Beispiel 3.3 aufruft, werden normalerweise Python-Integer übergeben. Pybind11 konvertiert diese Python-Objekte automatisch in die entsprechenden C++-Typen, die die Funktion `add` erwartet. Dies geschieht durch die internen Typkonverter von Pybind11, die speziell für eingebaute Typen wie Ganzzahlen, Fließkommazahlen und viele andere vordefiniert sind. Nachdem die `add`-Funktion in C++ das Ergebnis berechnet hat, muss das Ergebnis zurück an Python übergeben werden. Pybind11 konvertiert den C++-Integer, der von `add` zurückgegeben wird, automatisch zurück in einen Python-Integer.

Veränderliche und unveränderliche Datentypen

In C++ können Klassen und Strukturen entweder durch direkte Instanzmanipulation oder durch Wertübergabe verändert werden, siehe Abschnitt 2.1.2. Pybind11 erlaubt die direkte Übertragung von Referenzen und Pointern zwischen den Sprachen, was die Manipulation von Daten innerhalb von C++ ermöglicht. Dies funktioniert gut mit Python-Objekten, die veränderlich sind, wie `list` oder `dict`. Für immutable Typen müssen andere Strategien angewandt werden. Eine gängige Lösung für das Problem der Immutabilität ist es, die modifizierten Werte als Teil des Rückgabewerts der Funktion zurückzugeben. Dies ermöglicht es, auch immutablen Python-Datentypen modifizierte Werte zuzuweisen, indem die Rückgabewerte in Python entsprechend genutzt werden.

```cpp
int increment(int value) {
    return value + 1;
}

PYBIND11_MODULE(example, m) {
    m.def("increment", &increment);
}
```

Code-Snippet 3.4: Umgehung der Immutabilität durch Rückgabe des geänderten Objekts

Pybind11 stellt außerdem eigene Typen wie `py::list`, `py::dict`, und `py::str` zur Verfügung, die eine direktere Manipulation von Python-Objekten ermöglichen. Diese Typen spiegeln die nativen Python-Typen wider und können direkt von C++ aus manipuliert werden:

```cpp
void append_item(py::list lst, int item) {
    lst.append(item);
}

PYBIND11_MODULE(example, m) {
    m.def("append_item", &append_item);
}
```

Code-Snippet 3.5: Umgehung der Immutabilität durch Pybind11-Typen

In diesem Beispiel könnte jedoch auch der interne Typkonverter von Pybind11 verwendet werden, der nahtlos zwischen einem C++-Vektor und einer Python-Liste konvertiert, siehe Abschnitt zur Datenkonvertierung 2.1.1. Dies ermöglicht es, auf benutzerdefinierte Typen zu verzichten und stattdessen Standarddatentypen beider Sprachen direkt zu nutzen.

Speichermanagement

Ein zentrales Problem bei der Bindung von C++-Funktionen an Python ist die Verantwortungsfrage für die Lebensdauer und Verwaltung der zurückgegebenen Objekte wie bereits in Abschnitt 2.1.3 beschrieben. Um diese Probleme effektiv zu adressieren, bietet Pybind11 eine Lösung in Form von Rückgabewert-Richtlinien an. Diese Richtlinien ermöglichen eine klare Definition der Verantwortlichkeiten in Bezug auf das Speichermanagement. Nachfolgend sind einige der wichtigsten Rückgabewert-Richtlinien und deren Anwendungsfälle beschrieben [34]:

Rückgabewerte-Richtlinie	Beschreibung
`take_ownership`	Python übernimmt die volle Verantwortung für das Objekt und wird es löschen, sobald es nicht mehr benötigt wird.
`copy`	Python erstellt eine Kopie des C++-Objekts. Dies ist besonders sicher, da das C++-Objekt und das Python-Objekt voneinander entkoppelt sind.
`move`	Python verschiebt die Werte des C++-Objekts in ein eigenes Objekt. Ähnlich wie bei der Richtlinie `copy` ist das Objekt damit davor geschützt, von einer der beiden Seiten zu früh gelöscht zu werden.
`reference`	Diese Richtlinie verweist Python auf das bestehende C++-Objekt, ohne dass Python Eigentumsrechte erhält. Sie ist geeignet für Objekte, deren Lebensdauer von C++ kontrolliert wird. Allerdings muss sichergestellt werden, dass das Objekt in C++ nicht gelöscht wird, solange es noch von Python referenziert wird.
`automatic_reference` und `automatic`	Diese Richtlinien lassen Pybind11 automatisch entscheiden, welche der oben genannten Strategien je nach Kontext am besten geeignet ist. Sie sind hilfreich, wenn der Entwickler sich unsicher ist, welche Richtlinie die passende ist.

Abbildung 3.1: Übersicht der Rückgabewerte-Richtlinien in Pybind11

Thread-Sicherheit

Pybind11 unterstützt Multi-Threading durch die Freigabe des GIL während der Ausführung von C++-Funktionen, ähnlich wie Ctypes. Dies ermöglicht es anderen Python-Threads, während dieser Zeit zu laufen, und verbessert so die Leistung in Multi-Core-Systemen. Um den GIL freizugeben, kann der Befehl `py::call_guard<py::gil_scoped_release>()` bei der Funktionsdefinition verwendet werden [17]. Durch die richtige Verwendung dieser Funktion kann Pybind11 echte Parallelität in Python-Anwendungen ermöglichen.

Fehlerbehandlung

Pybind11 stellt einen C++-Ausnahmehandler bereit, der C++-Exceptions abfangen und in entsprechende Python-Errors umwandeln kann [17]. So können Fehler, die im

C++-Code auftreten, korrekt an Python weitergegeben, jedoch nicht von dort geprüft werden. Für die Umwandlung verwendet Pybind11 standardmäßig eine vordefinierte Zuordnung. Zum Beispiel wird eine C++-Exception vom Typ `std::exception` in einen Python-RuntimeError umgewandelt.

Pybind11 bietet auch die Möglichkeit durch die Verwendung des `py::register_exception_translator()`, benutzerdefinierte Ausnahmeübersetzer zu registrieren. Die folgenden beiden Codeausschnitte 3.2 demonstrieren die Ausnahmeübersetzerregistrierung für eine selbst geschriebene Exception und anschließend ein Beispiel für das Auftreten der Exception in Python.

```cpp
// C++ Code
class CppExp : public std::runtime_error {
public:
    CppExp(const char* msg) : std::runtime_error(msg) {}
};

//Pybind11 Code
void init_module(py::module_ &m) {
    py::register_exception<CppExp>(m, "PyExp");
}

PYBIND11_MODULE(my_module, m) {
    m.doc() = "Beispielmodul für Exceptionuebersetzung";

    init_module(m);

    m.def("throw_cpp_exp", []() {
        throw CppExp("Cpp-Exception wurde geworfen.");
    });
}

```

```python
import my_module

try:
    my_module.throw_cpp_exp()
except my_module.PyExp as e:
    print(str(e))


```

Code-Snippet 3.6: Definieren eigener Exception-Translator

Dieser Ansatz ist besonders nützlich, wenn bestimmte C++-Exceptions in entsprechende Python-Error umgewandelt werden sollen. Diese Funktionalität ermöglicht es Entwicklern, das Exceptionhandling für ihre spezifischen Anforderungen anzupassen. Solch ein flexibler Ansatz zur Fehlerbehandlung erleichtert die Integration von C++-Komponenten in Python-Anwendungen.

3.3 CFFI

CFFI (C Foreign Function Interface) ist ein Binding-Tool, das die Interoperabilität zwischen Python und C ermöglicht. CFFI bietet zwei Modi: den API-Modus und den ABI-Modus. Der API-Modus (Application Programming Interface) erfordert Header-Dateien und bietet eine stärkere Typprüfung, während der ABI-Modus (Application Binary Interface) ohne Header-Dateien auskommt und so schneller und einfacher zu verwenden ist, da der Code zur Laufzeit kompiliert wird. Der ABI-Modus nutzt allerdings Inline-Code, was bei bereits vorkompilierten C-Bibliotheken die Neudefinition innerhalb des Bindings erfordert und zu Redundanz führt [35].

Datenkonvertierung

Im Zusammenhang mit der Datenkonvertierung zwischen Python und C++ mithilfe von CFFI ist es wichtig, die Unterschiede zwischen dem ABI-Modus und dem API-Modus zu verstehen. Der ABI-Modus erlaubt direkten Zugriff auf kompilierte C-Bibliotheken, ohne vorher die Funktionssignaturen und Datentypen in Python zu deklarieren. In diesem Modus erfolgt die Konvertierung von Datentypen wie `int`, `long` und C-Strings (`char*`) automatisch, wobei CFFI die Größenanpassung und die Konvertierung in Python-kompatible Formate übernimmt. CFFI passt die Größen der Integer-Typen entsprechend den Anforderungen von Python und C++ an, wobei automatisch die verschiedenen Integer-Größen (wie `int` und `long`) korrekt konvertiert werden.

Der API-Modus (Application Programming Interface) erfordert hingegen eine explizite Definition der C-Schnittstellen in Python, was eine genauere Kontrolle über die Datenkonvertierung ermöglicht. Dies ist insbesondere bei komplexen Datenstrukturen oder speziellen Anforderungen an die Speicherverwaltung vorteilhaft. Mittels `ffi.cdef` werden Typen präzise definiert, was eine korrekte Konvertierung sicherstellt.

Beide Modi bieten Vorteile, abhängig von den technischen Anforderungen und den Merkmalen der zu konvertierenden Daten.

Für komplexe Datentypen, wie Strukturen oder Klassen, unterstützt CFFI die Definition von entsprechenden C++-Strukturen in Python. Beispielsweise kann eine in Python definierte C-Struktur an eine C-Funktion übergeben werden, die ihre Felder modifiziert. Diese Modifikationen sind dann direkt in der Python-Umgebung sichtbar, ohne dass eine Rückkonvertierung erforderlich ist. Ein praktisches Beispiel hierfür ist eine Funktion, die eine Struktur erhält und ihre Integer-Felder modifiziert. In Python könnte das folgendermaßen implementiert werden:

```
from cffi import FFI
ffi = FFI()

# Definition der C-Struktur in Python
ffi.cdef("""
    typedef struct {
        int x;
        int y;
    } Point;

    void some_function(Point *p);
""")

# Erstellung einer Instanz dieser Struktur
p = ffi.new("Point *", {'x': 10, 'y': 20})

# Aufruf einer C-Funktion, die diese Struktur modifiziert
ffi.C.some_function(p)

# Zugriff auf die modifizierten Werte in Python
x = p.x
y = p.y
```

Code-Snippet 3.7: Bindung eines Structs mit CFFI

Veränderliche und unveränderliche Datentypen

CFFI bietet spezifische Methoden und Techniken, um die Interaktion zwischen Python und C/C++ effizient und sicher zu gestalten, besonders im Umgang mit veränderlichen und unveränderlichen Datentypen.

Für unveränderliche Datentypen wie Strings gibt es in CFFI keine vollautomatische Konvertierung, die alle notwendigen Schritte, wie Kodierung und Dekodierung, übernimmt. Das bedeutet, dass Entwickler explizit angeben müssen, wie Daten konvertiert und behandelt werden sollen.

```
ffi.cdef("void print_const_char(const char* s);")
python_string = "Hello, CFFI!"
ffi.C.print_const_char(python_string.encode('utf-8'))
```

Code-Snippet 3.8: Übergabe eines unveränderlichen Python-Strings an C mit CFFI

Hier muss der Python-String manuell in ein Byte-Array (`oder char*` in C) umgewandelt werden. Dies geschieht durch die `encode()`-Methode, die den String in UTF-8 kodiert, was in C als `const char*` verwendet werden kann. CFFI übernimmt nicht automatisch die Kodierung, da es unterschiedliche Erwartungen darüber geben kann, wie Strings behandelt werden sollen (z.B. Kodierung, Null-Terminierung).

Bei veränderlichen Datentypen wie Listen und Dictionaries verwendet CFFI hingegen Referenzen, um die Daten effizient zwischen Python und C zu übertragen. Dieser Ansatz ermöglicht es, dass Änderungen, die auf der C-Seite vorgenommen werden, direkt in Python sichtbar sind, ohne dass eine Rückkopie notwendig ist:

```
ffi.cdef("void increment_values(int* array, size_t length);")
array = ffi.new("int[]", [1, 2, 3])
ffi.C.increment_values(array, 3)
print(list(array)) #Ausgabe: [4, 5, 6]
```

Code-Snippet 3.9: Beispiel für die Übergabe eines veränderlichen Datentyps mit CFFI

In diesem Fall handhabt CFFI die Speicherallokation und gibt eine Referenz auf den allokierten Speicher zurück, die dann direkt an die C-Funktion übergeben wird. Diese Art der Datenübertragung ist relativ automatisch und erfordert keine zusätzlichen Konvertierungsschritte von Seiten des Entwicklers.

Speichermanagement

Die Unterschiede im Speichermanagement zwischen Python und C/C++ können durch verschiedenen Ansätze überbrückt werden. Eine der dazu benutzen CFFI-Funktionen ist `ffi.new()`, mit welcher Speicher für C-Datentypen auf der Python-Seite allokiert werden kann [35]. Der von `ffi.new()` allokierte Speicher wird automatisch vom Python-GC verwaltet, sodass der Entwickler sich nicht um die manuelle Freigabe kümmern muss.

Zusätzlich stellt CFFI die Methode `ffi.gc()` zur Verfügung, um benutzerdefinierte Freigabefunktionen zu erstellen [35]. Diese Methode kann verwendet werden, um Speicher, der in C allokiert wurde, automatisch freizugeben, wenn das zugehörige Python-Objekt vom GC gesammelt wird. Zum Beispiel kann man mit `ffi.gc(ptr, free)` einen Zeiger `ptr` registrieren, mit dem die Funktion `free` aufgerufen wird, wenn das Objekt nicht mehr benötigt wird [35].

Um mit Pointern und Arrays umzugehen, bietet CFFI die Methoden `ffi.from_buffer()` und `ffi.buffer()`, die es ermöglichen, Python-Objekte und C-Puffer zwischen den Sprachen zu teilen. Diese Methoden sind nützlich, um Daten zwischen Python und C auszutauschen, ohne unnötige Kopien zu erstellen.

Thread-Sicherheit

Das Schreiben von Bindings mit CFFI für Multithreading-Anwendungen erfordert besondere Beachtung in Bezug auf die Thread-Sicherheit. Genau wie bei der Verwendung von Ctypes und Pybind11, hängt die Thread-Sicherheit stark von der Art und Weise ab, wie thread-sicher die C/C++-Bibliotheken selbst implementiert sind. CFFI selbst führt keine zusätzlichen Thread-Sicherheitsmechanismen ein, sondern überlässt dies den zugrunde liegenden Bibliotheken. CFFI ermöglicht es jedoch genau wie die anderen vorgestellten Tools auch, den GIL während der Ausführung von Binding-Funktionen freizugeben, vorausgesetzt, die C/C++-Funktionen sind dafür ausgelegt. Dies erlaubt eine parallele Ausführung in C/C++, während Python-Code weiterhin durch den GIL geschützt wird.

Entwickler müssen daher sicherstellen, dass alle in C/C++ geschriebenen Funktionen, die über CFFI aufgerufen werden, selbst threadsicher sind, insbesondere wenn sie auf gemeinsam genutzte Ressourcen zugreifen. Es empfiehlt sich, die Dokumentation der verwendeten C/C++-Bibliotheken auf Hinweise zur Threadsicherheit zu prüfen und gegebenenfalls eigene Schutzmechanismen wie Mutexe oder Semaphoren einzuführen, um Datenkonsistenz und Stabilität zu gewährleisten.

Fehlerbehandlung

Da CFFI hauptsächlich für die Interaktion mit C-Bibliotheken konzipiert ist, werden C++-spezifische Features wie Exceptions nicht direkt unterstützt. Um dennoch C++-Code und dessen Fehlermanagement effektiv in Python zu integrieren, gibt es jedoch die Möglichkeit, C++-Code in externen C-Funktionen zu kapseln, die dann von CFFI genutzt werden können. Diese C-Funktionen können C++-Exceptions fangen und in Fehlercodes umwandeln, die dann in Python überprüft und als Python-Exceptions geworfen werden können [36].

```cpp
//C++ Code
int safe_function() {
    try {
        // Funktion, die möglicherweise eine Exception wirft
        risky_function();
    } catch (const std::exception& e) {
        return -1; // Fehlercode
    }
    return 0; // Erfolg
}
```

```python
# Python Code
from cffi import FFI
ffi = FFI()
```

```
4
5 result = lib.c_safe_function()
6
7 # Überprüft das Ergebnis und wirft eine Ausnahme, wenn der
     Rückgabewert nicht 0 ist
8 if result != 0:
9     raise Exception("Ein Fehler ist aufgetreten!")
```

Code-Snippet 3.10: Wrapper für Exceptionhandling

Solche Wrapper-Funktionen als Fehlerbehandlungsmechanismen sind als Workaround notwendig, um die Robustheit und Wartbarkeit der Software zu gewährleisten. Diese Methoden erfordern einen zusätzlichen Entwicklungs- und Wartungsaufwand, bieten aber die erforderliche Flexibilität und Sicherheit, wenn C++-Bibliotheken mit CFFI in Python-Projekte integriert werden sollen.

4 Integration ins SELMA-Projekt

Die Frage nach dem optimalen Binding-Tool kann nicht universell beantwortet werden. Es kommt vielmehr auf die speziellen Beschaffenheiten der Bibliothek und die Anforderungen und Ziele des Projekts an, für das ein Binding konzipiert werden soll. Daher ist es wichtig, den Aufbau und die Funktion der SELMA-Decoder-Software zu analysieren, auch um die Ergebnisse eines späteren Benchmarkings nachvollziehen zu können.

4.1 Aufbau der Funktionen des Decoders

Als Datenverarbeitungssoftware ist der Decoder darauf ausgelegt, große Mengen an Daten von verschiedenen Sensormodulen effizient zu verarbeiten. Er nutzt insgesamt 13 spezifische Datenstrukturen, die aus mehrdimensionalen Arrays bestehen und speziell für die Handhabung der benötigten Datensätze konzipiert worden sind. Diese Structs beinhalten nach dem Ausführen der Decoder-Software unterschiedliche Datenarten, wie GNSS-Daten, Phasendaten und Modulbeschreibungen. Neben den Funktionen zur Konfiguration des Decoders existiert für jedes dieser Structs eine Methode, die das entsprechende Array mit Daten füllt – beispielsweise die Methode `get_module_description()` für das Struct `module_description`. Die meisten dieser Funktionen erwarten unter anderem einen Zeiger auf die entsprechenden Datenstrukturen. Das erschwert die Nutzung durch Python, da die Sprache nativ keine Zeiger unterstützt. Zusätzlich erschweren die Datenstrukturen des Decoders das Erstellen von Bindings, weil sie spezielle Wrapper und repräsentative Objekte in der Zielsprache benötigen. Ein weiteres wichtiges Merkmal des Decoders ist die Verwendung von Statuscodes als Rückgabewerte, die sofortiges Feedback über den Erfolg oder Misserfolg einer Operation liefern. Diese simplen Rückgabewerte vereinfachen das Erstellen von Bindings, da keine komplexen Datentypen als Rückgabe erwartet werden müssen. Zudem kann die Fehlerbehandlung auf Python-Seite darauf aufbauen.

Die Aufteilung in Header-Dateien und Quellcode unterstützt zudem die Verwendung von Header-only-Bibliotheken wie Pybind11 im Kontext von Bindings, siehe Kapitel 3.2.

Im weiteren Verlauf wird die Implementierung der verschiedenen untersuchten Bindings anhand der groben Schritte, die für die Integration notwendig waren, erläutert. Alle drei Bindings nutzen `Numpy`-Arrays als Repräsentation der Datenstrukturen des Decoders. Die Verwendung von `Numpy` bietet sich an, da es das effiziente Handling großer Datensätze ermöglicht [37]. Zudem sind die Weiterverarbeitung und das Plotten der Daten bereits auf die Arbeit mit `Numpy`-Arrays abgestimmt.

4.2 Implementierung des Bindings mit Ctypes

Zunächst muss die C++-Bibliothek (selma_decoder.so) in das Python-Skript geladen werden. Dazu wird die Funktion `load_library` aus dem Modul `numpy.ctypeslib` verwendet.

```
1 import os
2 import numpy as np
3 import ctypes as ct
4
5 selmalibname = 'selma_decoder.so'
6 script_dir = os.path.abspath(os.path.dirname(__file__))
7 selmalib = np.ctypeslib.load_library(selmalibname, script_dir)
```

Code-Snippet 4.1: Laden der Shared-Library

Die so geladene Shared-Library enthält bereits alle Funktionen des Decoders. Das im Folgenden beschriebene Python-Skript dient als Wrapper, also als Schnittstelle zwischen dieser Shared-Library und Python.

Die Bibliothek verwendet verschiedene Datenstrukturen, die in Python definiert werden müssen. Dies wird durch die Erstellung von `Numpy`-Datentypen erreicht, welche die in der C-Bibliothek definierten Strukturen widerspiegeln.

```
1 dt_phasedata=np.dtype([
2     ('sinsum',    np.float64),
3     ('cossum',    np.float64),
4     ('bias',      np.float64),
5     ('qsum',      np.float64),
6     ('phase',     np.float64),
7     ('frequency', np.float64),
8     ], align=True)
```

Code-Snippet 4.2: Definition der Numpy-Datentypen

Der Code-Schnipsel 4.2 definiert den `Numpy`-Dtype für das Array mit Phasendaten, das aus 6 Fließkommazahlen besteht. Die 12 restlichen Datenstrukturen (siehe Analyse des Decoders in Abschnitt 4.1) wurden ähnlich definiert.

Für jede Funktion des Decoders, die in Python verfügbar gemacht werden soll, muss ein Funktionsprototyp erstellt werden. Das beinhaltetet die Festlegung der Argumenttypen (`argtypes`) und des Rückgabewerts (`restype`).

```
selmalib.init_decoder.argtypes = None
selmalib.init_decoder.restype = None

selmalib.set_verbose.argtypes = [ct.c_int]
selmalib.set_verbose.restype = ct.c_int

selmalib.decode_file.argtypes = [ct.c_char_p]
selmalib.decode_file.restype = ct.c_int

selmalib.get_module_description.argtypes=[
    np.ctypeslib.ndpointer(dt_module_description, flags='aligned,
   writeable, c_contiguous'), ct.c_int]
selmalib.get_module_description.restype=ct.c_int
```

Code-Snippet 4.3: Definition der Funktionsprototypen

`restype` definiert den Rückgabetyp einer Funktion. Wenn `restype` nicht angegeben wird, nimmt Ctypes standardmäßig an, dass die Funktion `int` zurückgibt. Durch die explizite Angabe von `restype` wird sichergestellt, dass der Rückgabewert der Funktion korrekt interpretiert wird. `argtypes` definiert die Datentypen der Argumente, die an die Funktion übergeben werden. Das hilft Ctypes bei der korrekten Konvertierung der Argumente. Ohne die Angabe von `argtypes` versucht Ctypes, die Argumente ohne Konvertierung zu übergeben, was zu Fehlern führen kann, wenn sie nicht der von den C^{++}-Funktionen erwarteten Form entsprechen.

Da C Arrays immer als Pointer und mit Angabe der Länge erwartet, mussten für einige Funktionen, die Arrays als Argumente erhalten, zusätzliche Wrapper-Funktionen erstellt werden, die das Berechnen der Länge übernehmen. Dieses Vorgehen ist nicht zwingend erforderlich, erhöht jedoch die Benutzerfreundlichkeit des Moduls in Python, da dort die explizite Angabe von Arraylängen unüblich ist.

```
def get_module_description(modules: np.ndarray) -> int:
    res = selmalib.get_module_description(modules, modules.size)
    return res

def get_slot_description(slots: np.ndarray) -> int:
    res = selmalib.get_slot_description(slots, slots.size)
    return res

def get_sender_description(senders: np.ndarray, slot: int) -> int:
    res = selmalib.get_sender_description(senders, senders.size,
    slot)
    return res
[...]
```

Code-Snippet 4.4: Wrapper-Funktionen für Ctypes

Die Funktion `get_module_description` aus Codeausschnitt 4.6 Zeile 1 nimmt beispielsweise ein `NumPy`-Array namens `modules` als Eingabe und ruft die C++-Funktion `get_module_description` des Decoders auf. Der Aufruf, siehe Zeile 2, enthält das Array als Pointer wie in Codeausschnitt 4.3 festgelegt und dessen Länge `modules.size`. Das Ergebnis `res` wird als Integer zurückgegeben, siehe Zeile 3.

Dieser Wrapper kann nun von anderen Python-Skripten geladen werden. Das geschieht durch Einbindung des Wrapper-Dateipfads in den Systempfad. Anschließend kann das Modul importiert werden mit `importlib`:

```
1 # Dateiname des Bindings 'selma_wrapper.py'
2 import importlib
3 selma = importlib.import_module('selma_wrapper')
```

Code-Snippet 4.5: Importieren des Selma-Wrappers

4.3 Implementierung des Bindings mit Pybind11

Zum Erstellen eines Python-Bindings mit Pybind11 muss die Bibliothek zunächst über einen Paketmanager z.B. `Pip` installiert werden, da sie nicht Teil der Standardlibrary ist.

Zur Erstellung des Moduls wurde eine neue C++-Datei angelegt, in der die benötigten Header eingebunden wurden.

```
1 #include <pybind11/pybind11.h>
2 #include <pybind11/stl.h>
3 #include <pybind11/numpy.h>
4 #include "selma_decoder_interface.hxx"
5
6 namespace py = pybind11;
```

Code-Snippet 4.6: Einbinden der Header-Dateien für Pybind11

Diese Header, einschließlich des Haupt-Headers von Pybind11, bieten grundlegende Funktionalitäten und API-Definitionen. Der STL-Header (Standard Template Library) ermöglicht die Nutzung der STL-Strukturen wie Vektoren, Maps, Sets etc. und stellt automatische Konverter bereit. Hier erfolgte auch die Integration des `pybind11/numpy.h`-Headers, der es ermöglicht, C++ Funktionen so zu gestalten, dass sie nahtlos NumPy-Arrays akzeptieren und zurückgeben können. Pybind11 arbeitet im Gegensatz zu Ctypes auf der C++-Seite, daher werden durch das Einbinden der Header-Datei `selma_decoder_interface.hxx` alle benötigten Informationen des Decoders zugänglich gemacht, die in Python verfügbar gemacht werden sollten.

Der Hauptteil der Entwicklung des Bindings bestand darin, das eigentliche Pybind11-Modul zu erstellen, das als Schnittstelle zum C++-Code fungiert. Dieses Modul, benannt als `selma_decoder_pybind11`, wurde durch den Makroaufruf `PYBIND11_MODULE` initialisiert, wie im Codeausschnitt 4.7 zu sehen. Insbesondere wurden die `PYBIND11_NUMPY_DTYPE`-Makros verwendet, um die benutzen Datenstrukturen für die Verarbeitung der Daten, die sowohl im C++-Code als auch in Python benötigt werden, auch innerhalb der Schnittstelle festzulegen:

```
PYBIND11_MODULE(selma_decoder_pybind11, m) {

    PYBIND11_NUMPY_DTYPE(phasedata, sinsum, cossum, bias, qsum,
    phase, frequency);
    [...]

    m.doc() = "Pybind11 interface to the selma_decoder library";

    m.def("init_decoder", &init_decoder, "Initialize the decoder
          logic");

    m.def("set_verbose", &set_verbose, "Set the verbosity level",
          py::arg("verbosity"));

    m.def("decode_file", &decode_file, "Decode a data file",
          py::arg("filename"));

    [...]
}
```

Code-Snippet 4.7: Erstellen des Pybind11-Moduls

Die Numpy-Datentypen sind auf der C++-Seite als Structs und auf der Python-Seite entsprechend als Numpy-Dtypes deklariert, siehe Snippet 4.2. Die Deklaration in Pybind11 ist ausschließlich für die Registrierung der Speicherlayout-Informationen der angegebenen C++-Strukturen zuständig. Diese in C++ definierten Strukturen, die mittels des `PYBIND11_NUMPY_DTYPE`-Makros registriert werden, sind in ihrer nativen Form keine NumPy-Arrays. Vielmehr handelt es sich um C++-Datenstrukturen, die gleiche Datentypen und Speicherlayouts besitzen, wodurch sie effizient in Python als NumPy-Datentypen genutzt werden können, ohne ihre Struktur zu verändern.

Zusätzlich zur Typendefinition wurden innerhalb dieses Moduls die Methoden des Decoders definiert, um die Funktionalität des C++-Codes in Python zugänglich zu machen.

Im Kontext der Anbindung von C++-Code an Python mittels Pybind11 ist es üblich, zwischen direkten Funktionsbindungen und solchen Funktionen zu unterscheiden, die spezielle Wrapper-Funktionen benötigen. Wrapper-Funktionen kommen zum

Einsatz, wenn die direkte Übergabe von C++-Funktionen an Python entweder aufgrund von Komplexität in der Funktionslogik oder wegen inkompatibler Datentypen nicht möglich ist. Die meisten Funktionen des Decoders erwarten einen Pointer und die Länge des Arrays, siehe Abschnitt 4.1. Um dem Nutzer die Angabe der Arraygröße abzunehmen und gleichzeitig fehlerhafte Eingaben zu vermeiden, kommen Wrapper-Funktionen zum Einsatz. Im Codeausschnitt 4.8 werden diese Wrapper wie im Codeausschnitt 4.7 deklariert.

```
m.def("get_module_description", &py_get_module_description,
      "Gets the module descriptions",
       py::arg("module_description_array"));

m.def("get_slot_description", &py_get_slot_description,
      "Gets the slot descriptions",
       py::arg("slot_description_array"));

m.def("get_sender_description", &py_get_sender_description,
      "Gets the sender descriptions for a slot",
       py::arg("sender_description_array"), py::arg("slot_num"));
```

Code-Snippet 4.8: Deklarieren der Wrapper-Funktionen

Diese Wrapper-Funktionen erfüllen zwei Hauptaufgaben:

1. Erstellung des Pointers: Generierung des erforderlichen Pointers auf das Array, das die zugrundeliegende C++-Funktion benötigt.

2. Berechnung der Arraylänge: Bestimmung der Länge des Arrays automatisch, was die Fehleranfälligkeit bei manueller Eingabe reduziert.

Im folgenden Codeausschnitt 4.9 ist eine solche Wrapper-Funktion für die C++-Funktion `get_module_description` dargestellt.

```
static int py_get_module_description(py::array_t<module_description,
    py::array::c_style> input_array) {

    //Anfordern des Buffers
    auto buf = input_array.request();

    //Konvertierung des Pointers
    module_description *ptr = (module_description *) buf.ptr;

    //Berechnen der Arraylänge
    int num_modules = input_array.size();

    //Aufruf der eigentlichen C++-Methode
    int result = get_module_description(ptr, num_modules);
    return result;
};
```

Code-Snippet 4.9: Wrapper-Funktion für get_module_description

Die vollständige Datei wurde dann mit diesem Befehl kompiliert:

```
g++ -std=c++20 -g -fPIC
    <Optionen für Warnungen>
    -shared -o \
    selma_decoder_pybind11.so selma_decoder_interface.o \
    selma_decoder.o decode_config.o decode_temperatures.o \
    decode_gnss.o decode_phase.o decode_magfield.o \
    decode_pressure.o decode_acceleration.o decode_time.o \
    decode_xadc.o decode_raw.o decoder_tools.o decoder_pybind11.o
```

Code-Snippet 4.10: Compiler-Aufruf für das Pybind11-Binding

Der Befehl verwendet den Linux-spezifischen C++-Compiler `g++` zur Erstellung der neuen Bibliothek. Dabei werden Compiler-Optionen wie `std=c++20` für die `C++20`-Standardspezifikation, Debugging-Optionen mit dem Parameter `-g` und verschiedene Warnungen verwendet und unterdrückt. Die Option `-fPIC` (Position Independent Code) wird verwendet, um sicherzustellen, dass der Code einer dynamischen Bibliothek an beliebiger Speicheradresse ausgeführt werden kann, wodurch mehrere Prozesse die gleiche Bibliothek im Speicher teilen könnten. Der Befehl bindet mehrere Objektdateien, die aus dem SELMA-Decoder-Quellcode generiert wurden, zu einer einzigen dynamischen Bibliothek zusammen.

Nach der Kompilierung des Bindings, kann das Modul in Python zum Systempfad hinzugefügt werden und anschließend wie ein gewöhnliches Python-Modul importiert werden:

```
import selma_decoder_pybind11 as selma
```

Code-Snippet 4.11: Importieren des Pybind11-Selma-Decoders

4.4 Implementierung des Bindings mit CFFI

Die Integration von CFFI in das SELMA-Projekt folgt einem ähnlichen Ansatz wie bei der Verwendung von Ctypes. CFFI ermöglicht die Integration von C-Code in Python sowohl auf Quellcode- als auch auf ABI-Ebene, siehe Unterkapitel 3.3. Die Implementierung im ABI-Modus bietet sich hier an, da der Decoder als bereits vorkompilierte Bibliothek vorliegt.

Zunächst wird die C-Bibliothek in das Python-Skript geladen. Dazu wird die CFFI-Bibliothek installiert und die notwendigen Typdefinitionen und Funktionen aus der C-Bibliothek beschrieben.

```
import os
import cffi

selmalibname = 'selma_decoder.so'
ffi = cffi.FFI()
script_dir = os.path.abspath(os.path.dirname(__file__))
decoder_dir = os.path.abspath(os.path.join(script_dir, "..", "
    decoder"))
selma_path = os.path.join(decoder_dir, selmalibname)

selmalib = ffi.dlopen(selma_path)
```

Code-Snippet 4.12: Laden der C-Bibliothek mit CFFI

Anschließend müssen innerhalb des FFI-Moduls alle Methoden und Structs definiert werden. Der übergebene String entspricht in etwa der Header-Datei des Decoders in Textform.

```
ffi.cdef("""
    typedef struct {
        double sinsum;
        double cossum;
        double bias;
        double qsum;
        double phase;
        double frequency;
    } dt_phasedata;
    [...]
    void init_decoder();
    int set_verbose(int level);
    int decode_file(const char* filename);
    [...]
    """)
```

Code-Snippet 4.13: Deklarieren der Funktionen in CFFI

Durch die Deklaration der C-Header-Definitionen mit `cdef()` wird sichergestellt, dass CFFI die richtige Signatur der Funktionen und Datentypen kennt, die in der C-Bibliothek definiert sind.

Der nächste Schritt ist die Implementierung der Python-Klasse `SelmaDecoder`, die als Wrapper für die Funktionen der C-Bibliothek dient. Diese Klasse ermöglicht es, die Funktionen der C-Bibliothek auf eine Python-typische Weise zu nutzen, indem sie die notwendigen Datentypen konvertiert und die Aufrufe der C-Funktionen vereinfacht. Die Methode `__getattr__` ermöglicht dynamischen Zugriff auf die Funktionen der C-Bibliothek, was die Implementierung der Wrapper-Funktionen vereinfacht. Jede dieser Wrapper-Funktionen übernimmt ein `NumPy`-Array, berechnet die Länge, wandelt es in

das entsprechende C-Array um und ruft dann die entsprechende Funktion der C-Bibliothek auf.

```
class SelmaDecoder:
    def __init__(self):
        self.lib = selmalib
        self.ffi = ffi

    def __getattr__(self, name):
        return getattr(self.lib, name)

    def get_module_description(self, modules):
        cdata_array = self.ffi.cast("dt_module_description *",
        modules.ctypes.data)
        res = self.lib.get_module_description(cdata_array,
        modules.shape[0])
        return res
    [...]
```

Code-Snippet 4.14: Wrapper-Funktion in CFFI

Der Wrapper kann dann in Python-Skripten wie folgt geladen werden:

```
from cffi_decoder import SelmaDecoder
```

Code-Snippet 4.15: Importieren des CFFI Decoders

5 Vergleich der Bibliotheken

Die Auswahl und Anwendung der geeigneten Bindings-Tools für den Decoder des SELMA-Projekts ist ein zentraler Aspekt dieser Arbeit. In diesem Kapitel werden die Erkenntnisse der Implementierung der Tools im SELMA-Projekt geschlussfolgert und es erfolgt eine detaillierte Betrachtung der Kriterien, die für die Toolauswahl entscheidend sind. Zusätzlich wird ein Benchmarking durchgeführt, um die bestmögliche Integrationsoption zu ermitteln.

5.1 Performancevergleich

Die vom Decoder verarbeiteten Datensätze erreichen häufig Größen von mehreren Hundert Gigabyte. Aufgrund der notwendigen hohen Verarbeitungseffizienz wurde für die Implementierung C^{++} gewählt. Es ist entscheidend, dass der durch die Sprache bedingte Leistungsvorteil durch den Einsatz von Bindings möglichst erhalten bleibt. Ziel ist es daher, das Binding-Tool für das SELMA-Projekt einzusetzen, das die höchstmögliche Datenverarbeitungsrate bietet.

Der Einsatz eines Python-Bindings könnte jedoch zu Overhead führen. Dieser resultiert sowohl aus den Eigenschaften des Python-Compilers als auch aus den notwendigen Datentypkonvertierungen innerhalb von Python. Ideal wäre daher die Auswahl des Tools, das den geringsten Overhead verursacht.

Das Benchmarking der Tools erfolgt nach deren Implementierung und basiert auf drei Hauptkriterien:

- **Initialisierungsaufwand:** Abhängig davon, ob das Tool einmalig mit vielen Dateien oder mehrmals mit jeweils einer Datei aufgerufen wird, ist die Initialisierungszeit – also die Zeit, die benötigt wird, um das Modul zu laden – sowie der benötigte Speicher von Bedeutung. Während das Pybind11-Binding als ein Python-Modul importiert wird, werden die anderen beiden Bindings als Skripte geladen. Es ist interessant festzustellen, ob es dabei zu signifikanten Leistungsunterschieden kommt.

- **Laufzeit:** Die Tools sollen eine möglichst hohe Datenmenge in kürzester Zeit verarbeiten können. Daher werden alle drei Bindings mit dem gleichen Datensatz ausgeführt und die resultierenden Laufzeiten mit dem originalen C++-Programm verglichen.

- **Speicherverbrauch:** Um die Plattform des Decoders flexibel zu halten, sollte der Speicherverbrauch moderat bleiben. Um den Overhead von Python zu isolieren, wird eine möglichst kleine Eingabedatei verwendet und der während der Laufzeit allokierte und deallokierte Speicher verglichen.

Initialisierungsaufwand

Die Initialisierungseffizienz der Python-Bindings wird mit dem Tool *Memory-Profiler* gemessen [38]. Der *Memory-Profiler* in Python ist ein Analysetool, das zur Laufzeit des Programms den Speicherverbrauch überwacht. Es arbeitet, indem es bei jedem Funktionsaufruf die Speichernutzung vor und nach der Ausführung der Funktion erfasst und dadurch den exakten Speicherzuwachs identifiziert. Zusätzlich bietet der Profiler die Möglichkeit, die Speichernutzung über die Zeit zu verfolgen und die Informationen zu plotten. Die ständige Aufzeichnung der Speichernutzung kann jedoch zusätzliche Rechenzeit beanspruchen und somit die gemessenen Initialisierungszeiten leicht verlängern. Hauptziel der Messung liegt aber nicht darin, die genauen Zeitwerte zu ermitteln, sondern vielmehr die relativen Unterschiede zwischen den verschiedenen Tools herauszustellen. Um nur die Initialisierung der einzelnen Bindings zu erfassen, wurde nach dem Importieren kein weiterer Code ausgeführt.

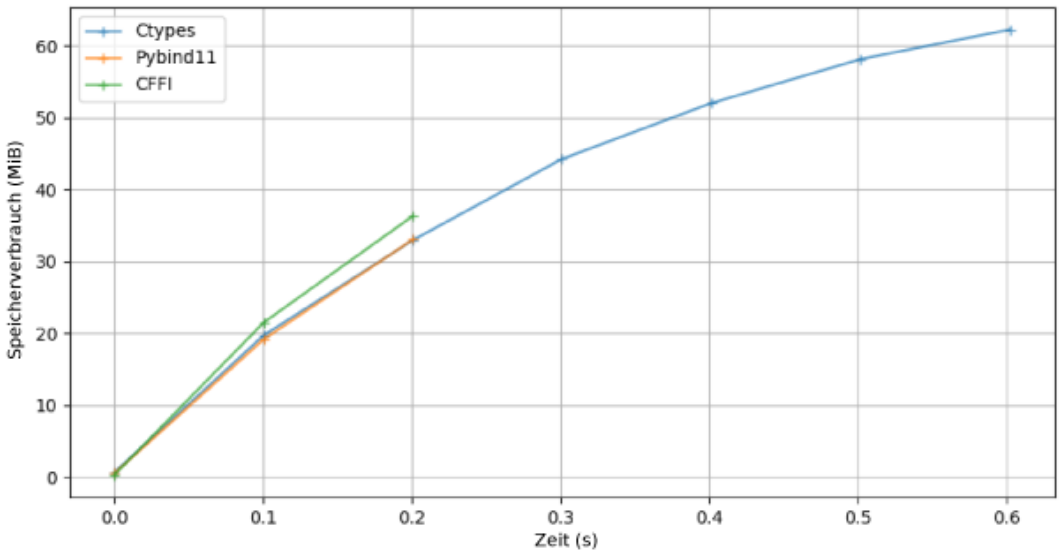

Abbildung 5.1: Initialisierung der Bindings

Die Ergebnisse zeigen, dass Ctypes mit etwas über 0,6 Sekunden sowohl am längsten zur Initialisierung benötigt, als auch mit über 60 MiB den meisten Speicher allokiert. Pybind11 und CFFI benötigen die gleiche Zeit mit etwa 0,2 Sekunden, jedoch braucht CFFI etwa 3 MiB mehr für den Initialisierungsprozess. Der Zeitvorteil von Pybind11 lässt sich durch die Vorkompilierung des Bindings erklären.

Laufzeit

Die Laufzeit wurde mit einem Datensatz von 4 Eingabedateien gemessen, die insgesamt 218,83 MiB groß sind. Außerdem wurde das originale C^{++}-Programm mit demselben Datensatz ausgeführt, um eine eventuelle durch Python verursachte Verschlechterung zu sehen. Um geringfügige Schwankungen zu minimieren, wurden zunächst 5 Messung zum Warmlaufen und anschließend jeweils 100 Messungen durchgeführt. Diese Vorbereitungen tragen dazu bei, Initialisierungs-Overheads zu reduzieren und das System in einen stabileren Zustand zu bringen, wodurch die nachfolgenden 100 Messungen die tatsächliche Leistungsfähigkeit des Codes unter optimalen Bedingungen widerspiegeln sollten.

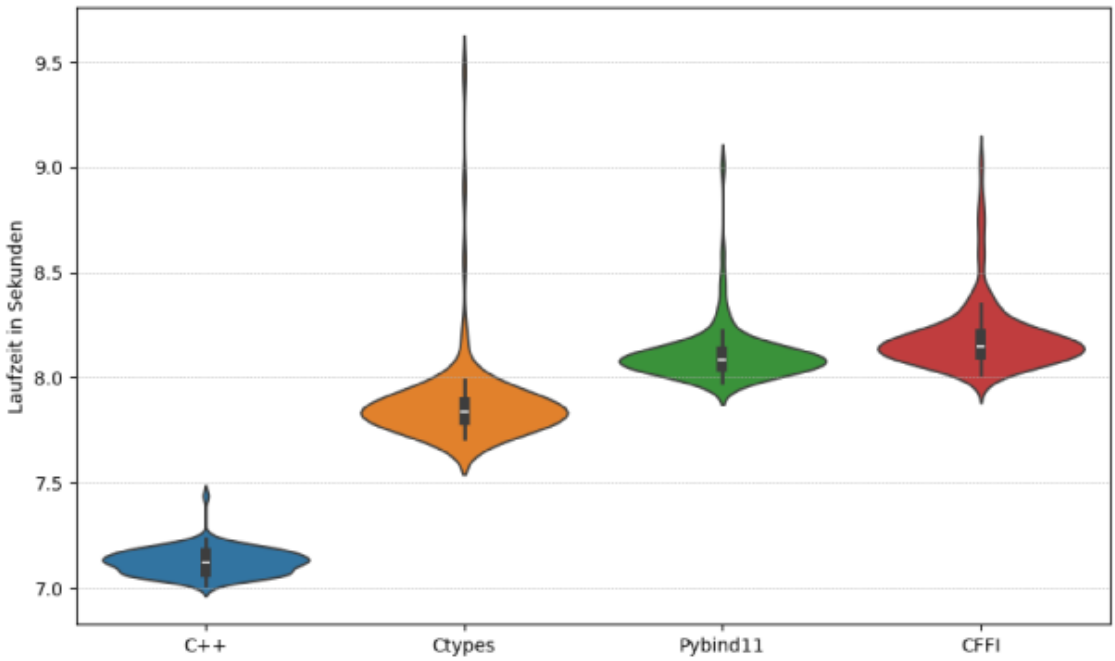

Abbildung 5.2: Verteilung der Laufzeit in einem Violinenplot

	C++	Ctypes	Pybind11	CFFI
Mittelwert	7,1255s	7,8819s	8,1130s	8,1964s
Median	7,1235s	7,8364s	8,0866s	8,1521s
Standardabweichung	0,0616s	0,2168s	0,1305s	0,1636s
Minimum	7,0110s	7,7139s	7,9768s	8,0134s
Maximum	7,4383s	9.4545s	9.0050s	9.0021s

Abbildung 5.3: Mittlere Laufzeiten der Bindings und des C++-Programms

Die Analyse der Ergebnisse, die im Violinenplot in Abbildung 5.2 grafisch dargestellt werden, zeigt, dass das native C++-Programm durchgehend die kürzesten Laufzeiten aufweist. Das bestätigt, dass C++ die höhere Ausführungseffizienz bietet.

Unter den Bindings bietet Ctypes die beste durchschnittliche Laufzeit, allerdings mit einer hohen Variabilität, was durch ein breites Spektrum in den Messwerten in Abbilung 5.2 erkennbar ist. Diese Schwankungen könnten auf Unterschiede in der System- oder der Betriebssystemkonfiguration zurückgehen. CFFI hingegen zeigt die längsten Durchschnittszeiten mit der geringsten Variabilität, während Pybind11 am durchschnittlichsten abschneidet. Diese Beobachtungen stehen im Gegensatz zu externen Benchmarkings, die aufzeigen, dass Pybind11 in manchen Situationen langsamer als andere Bindings ist, darunter auch CFFI [39].

Speicherverbrauch

Um den Speicherverbrauch der Bindings präzise zu messen, wurde statt des *Memory-Profilers* von Python das Tool *Valgrind* verwendet. *Valgrind* ist hier besser geeignet, da es neben Python auch C++ unterstützt und somit eine Analyse der Speichernutzung über beide Programmiersprachen hinweg ermöglicht. Im Gegensatz zum Python-spezifischen *Memory-Profiler*, bietet *Valgrind* auch eine tiefgreifende Untersuchung von Speicherlecks, unnötiger Speichernutzung und fehlerhaften Speicherzugriffen für eine Vielzahl von Programmiersprachen. *Valgrind* arbeitet durch die Simulation eines virtuellen Prozessors und überwacht alle Speicherzugriffe, was es ermöglicht, auch die Interpreterzeiten zu erfassen. Die verwendete Messmethode führt jedoch zu einer Verzerrung der Laufzeiten, da der Programmcode zur Laufzeit analysiert und modifiziert wird [40]. Aus diesem Grund sind in den folgenden Grafiken auch längere Laufzeiten zu sehen als die in der Tabelle 5.3 dargestellten Ergebnisse zeigen.

Das C++-Programm und die drei Bindings wurden wieder jeweils mit dem gleichen Datensatz ausgeführt. Die Zeit und der Speicher wurden dabei im direkten Vergleich zu C++ geplottet, indem die Kurven am Endpunkt aufeinander gelegt wurden. Diese

Darstellungsweise, bei der der Beginn des C^{++}-Programms als zentraler Referenzpunkt dient und die Zeit- und Speicherachsen symmetrisch von diesem Punkt aus in beide Richtungen verlaufen, ermöglicht einen klaren Vergleich der Speichernutzung.

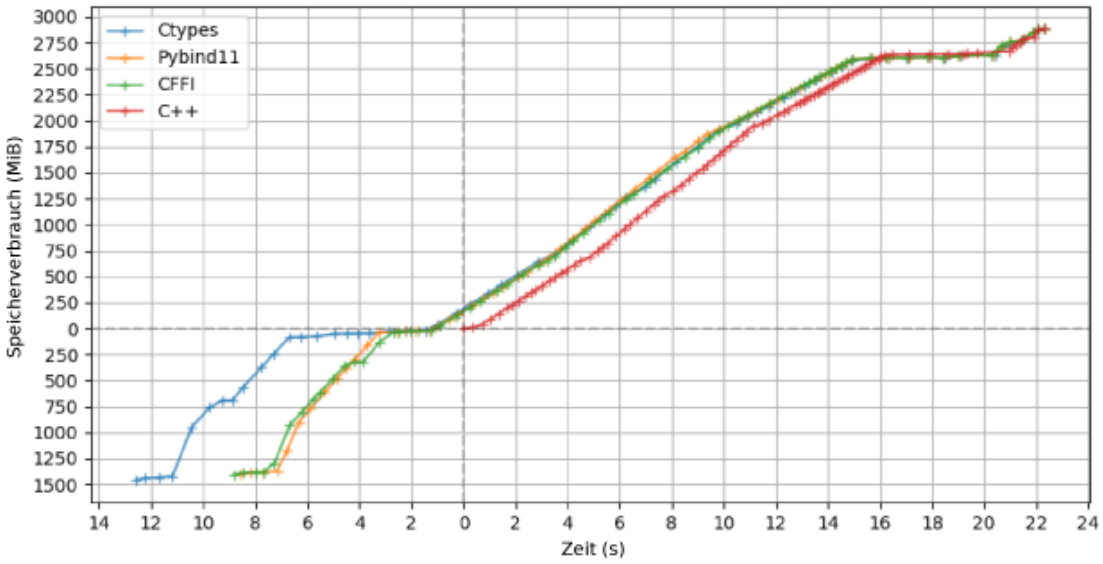

Abbildung 5.4: Speicherverbrauch der Bindings und des C^{++}-Programms

Das Ergebnis im Plot 5.4 zeigt den erheblichen Overhead, der durch die Nutzung der Python-Bindings verursacht wird und vermutlich auf den Python-Interpreter zurückzuführen ist. Im Vergleich zu der C^{++}-Implementierung benötigen die Python-Bindings bis zu 1500 MiB mehr Speicher, bie einem Gesamtverbrauch von etwa 4500 MiB. Abgesehen von diesem Overhead gibt es keine signifikanten Unterschiede im Speicherverbrauch zwischen den verschiedenen Python-Bindings und dem C^{++}-Programm.

Um den Overhead genauer zu isolieren und die Unterschiede zwischen den Bindings klarer herauszustellen, werden die Programme mit einem nur 4KB großen Datensatz aufgerufen.

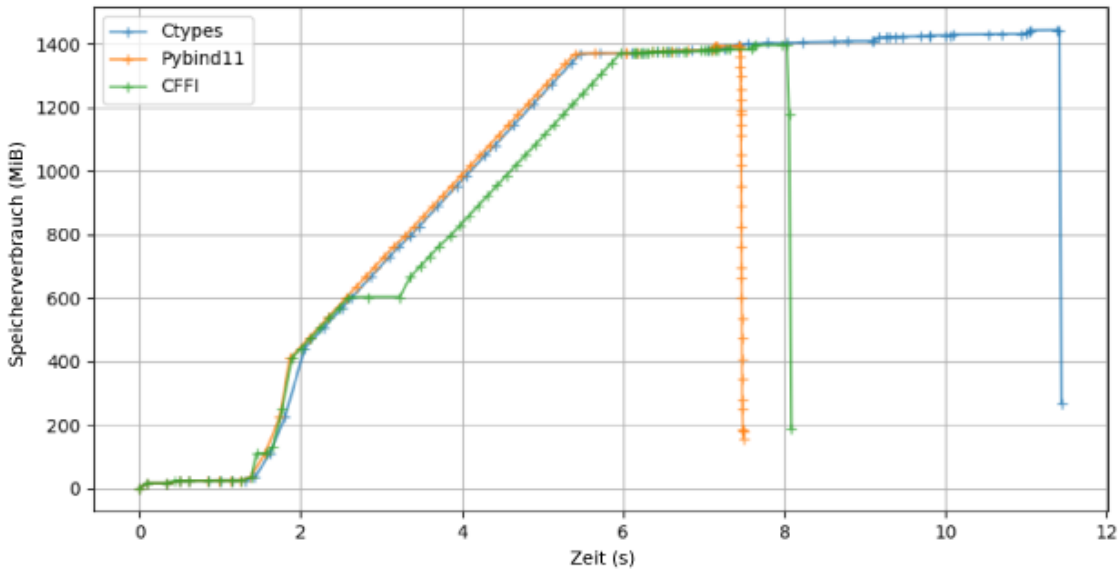

Abbildung 5.5: Speicherverbrauch der Bindings bei sehr kleinem Datensatz

Der Plot in Abbildung 5.5 veranschaulicht deutlich die Unterschiede zwischen den drei Python-Bindings, wenn das Programm selbst kaum Daten verarbeiten muss. Alle drei Bindings zeigen bis etwa 600 MiB den gleichen Speicherverbrauch. Schon vorher ist erkennbar, dass CFFI bei etwa 100 MiB eine Stufe im Verlauf hat. Ab 600 MiB stagniert der Speicher bei diesem Binding kurzzeitig und beginnt dann ein wenig später als die anderen, auf etwa 1400 MiB anzusteigen. Bei 1400 MiB beginnen alle Bindings nacheinander, den Speicher wieder freizugeben. Bei Pybind11 sinkt der Speicherverbrauch auf etwa 180 MiB, bei CFFI auf 200 MiB und bei Ctypes auf etwa 280 MiB. Besonders auffällig ist das Verhalten des Python-GC, der nach Abschluss der Operationen den Speicher wieder freigibt, was bei den Python-Bindings durch das Abfallen der Speicherkurven sichtbar wird. Im Gegensatz dazu ist das C^{++}-Programm so gestaltet, dass es den Speicher nicht explizit freigibt, sondern diese Aufgabe dem Betriebssystem überlässt, um die Laufzeit nicht in die Länge zu ziehen. Das hat zur Folge, dass die Speicherverbrauchskurve der Bindings auch am Ende der Messung nicht ganz auf null fällt. Das unterstreicht die Unfähigkeit des Python-GC, den von C^{++} allokierten Speicher zu verwalten und freizugeben, da er nur Speicher freigeben kann, der innerhalb der Python-Umgebung verwaltet wird. Die unterschiedlichen Endwerte des Speicherverbrauchs könnten auf die unterschiedlichen Ansätze der Speicherverwaltung in den Bindings zurückzuführen sein, da die Lebensverwaltung der Objekte unterschiedlich geregelt wird.

5.2 Qualitative Bewertungskriterien

Neben den quantitativen Benchmarks, die die Leistungsfähigkeit der verschiedenen Bindings messen, ist es wichtig, auch qualitative Bewertungskriterien zu berücksichtigen. Diese umfassen Faktoren wie Benutzerfreundlichkeit, Integrationsfähigkeit und die Qualität der Dokumentation, inklusive Community Support, falls vorhanden. Diese Aspekte sind entscheidend für die Gesamtbewertung der Bindings, da sie die Benutzererfahrung und die Einfachheit der Implementierung beeinflussen. In diesem Abschnitt wird daher verglichen, wie zugänglich und anwendbar die einzelnen Bibliotheken in einem realen Anwendungskontext wie dem SELMA-Decoder sind. Das geschieht anhand von drei Kriterien:

- **Entwicklerfreundlichkeit:**
  Die Entwicklerfreundlichkeit bezieht sich darauf, wie einfach und intuitiv die Entwickler mit der Software interagieren können. Dies betrifft nicht nur die Klarheit der Syntax, sondern auch den erforderlichen Aufwand für die Einarbeitung. Eine hohe Entwicklerfreundlichkeit wird durch eine umfassende Dokumentation gefördert, die klare Installationsanleitungen, Beispielszenarien, detaillierte API-Beschreibungen und Hinweise zur Fehlerbehebung bietet. Sie sollte auch spezifische Python-Beispiele und Best Practices für die Nutzung der C^{++}-Funktionalitäten enthalten. Zusätzlich kann ein eventuell vorhandener Community Support hilfreich sein. Ebenfalls dazu zählt auch wie einfach spätere Änderungen vorgenommen werden können.

- **Integration in bestehende Projekte:**
  Die Integration der Python Bindings in SELMA misst, wie gut die Bindings mit der bestehenden Decoder-Infrastruktur zusammenarbeiten. Dieses Kriterium bewertet und wie gut sich die einzelnen Tools in das Projekt einbetten ließen und wie gut es den technischen Anforderungen entspricht.

- **Benutzerfreundlichkeit:** Dieser Aspekt betrachtet, wie unkompliziert es für Nutzer ist, das Modul zu importieren und zu verwenden. Eine gute Benutzerfreundlichkeit ist auch gekennzeichnet durch ergiebige Fehlermeldungen sowie eine intuitive Bedienung des Moduls.

Ctypes ist bereits Teil der Python-Standardlibrary und deshalb leicht zugänglich für Entwickler, die bereits mit Python arbeiten. Die Dokumentation dazu findet sich direkt in den offiziellen Python-Dokumentationen und ist ausführlich gestaltet, inklusive Tutorials und praktischen Beispielen, die den Einstieg erleichtern. Obwohl

kein spezielles Forum für Ctypes existiert, ist die Technologie weit verbreitet, was sich in einer Vielzahl von Diskussionen und gelösten Problemen auf Plattformen wie *Stack Overflow* widerspiegelt. Die Syntax von Ctypes ist intuitiv. Es müssen für jede Funktion explizit Argument- und Rückgabetypen definiert werden. Diese Struktur ist leicht zu erlernen und reduziert die Einarbeitungszeit in die Bibliothek, da sich grundlegende Muster bei den meisten Anwendungen konsequent wiederholen.

Pybind11 ist ein Bindungswerkzeug, das primär auf der C^{++}-Seite der Softwareentwicklung ansetzt. Für Python-Entwickler, die nicht mit der C^{++}-Syntax vertraut sind, könnten daher längere Einarbeitungszeiten erforderlich sein als für erfahrene C^{++}-Entwickler. Die Dokumentation von Pybind11 bietet zwar eine Einführung mit Beispielen für die ersten Schritte, diese spiegeln jedoch hauptsächlich einfache Anwendungsfälle wider und sind recht knapp gehalten. Die Gliederung der Dokumentation ist übersichtlich, bietet aber nicht immer ausreichende Tiefe für komplexere Probleme. Während der Implementierung im SELMA-Projekt traten zudem gelegentlich undurchsichtige Fehlermeldungen auf, die mit den vorhandenen Dokumentationsressourcen nicht sofort aufgelöst werden konnten. Obwohl Pybind11 ein sehr leistungsfähiges Bindingtool ist, scheint es derzeit noch an umfangreichem Community-Support zu mangeln, was die Lösung spezifischer Probleme erschweren kann. Da mit Pybind11 ein Python-Modul erstellt wird, ist dieses später für den Benutzer besonders einfach einzubinden. Das Modul muss jedoch kompiliert werden, deshalb sind spätere Änderungen vergleichsweise schwieriger als bei den anderen Tools vorzunehmen.

CFFI erleichtert die Einbindung von C-Code in Python, ohne dass tiefgehende Kenntnisse der C-Syntax erforderlich sind, und bietet eine effektive Alternative zu Ctypes. Die zugehörige Dokumentation ist umfangreich, jedoch textlastig und kann stellenweise unübersichtlich sein, was das schnelle Auffinden spezifischer Informationen erschwert. Trotzdem werden auch Spezialfälle ausführlich abgedeckt. CFFI bietet sowohl einen API- als auch einen ABI-Modus, die es Entwicklern ermöglichen, verschiedene Arten von Code zu binden. Während der Implementierung sind allerdings Probleme beim Deklarieren von Wrapper-Funktionen aufgetreten, insbesondere weil ein Teil der Funktionen in CFFI gebunden und dann außerhalb als Wrapper implementiert werden musste. Dies erschwerte das Importieren der Funktionen in andere Skripte, wie es bei Pybind11 mit einem Modul möglich ist. Diese Herausforderung wurde durch die Entwicklung einer Klasse gelöst, die die `getattr`-Methode verwendet, um die restlichen Funktionen dynamisch auf einer Ebene zu binden und so eine konsistente und modulare Struktur zu ermöglichen. Neben der Dokumentation gibt es auf Plattformen wie `Libera.chat` Community-Support, wo Entwickler sich über Probleme und Anregungen austauschen können. Obwohl die Dokumentation viele Beispiele enthält, kann ihre Textlastigkeit und das gelegentliche Fehlen von klaren, präzisen Anleitungen eine Herausforderung darstellen.

5.3 Gesamtvergleich

	Ctypes	Pybind11	CFFI
Laufzeit	Gute und konsistente Laufzeit	Langsamste Laufzeit, weniger konsistent	Ähnliche Laufzeit zu Pybind11, niedrige Standardabweichung
Initialisierung	Hoher Speicherbedarf und längere Initialisierungszeit	Weniger Speicherverbrauch, schnelle Initialisierung	Weniger Speicherverbrauch, schnelle Initialisierung
Speicherverbrauch	Speicherverbrauch identisch, aber höher als bei C++		
Entwicklerfreundlichkeit	Entwicklung in Python; intuitive Syntax, umfangreiche Dokumentation, Änderungen einfach durchführbar	Entwicklung in C++; intuitive Syntax, knappe Dokumentation, Änderungen wegen Kompilierung komplizierter	Entwicklung in Python; gute Dokumentation, aber textlastig und gelegentlich unübersichtlich, Änderungen einfach durchführbar
Integration	Leichte Integration, Teil der Python-Standardbibliothek	Undurchsichtige Fehlermeldungen	Herausforderungen bei der Implementierung von Wrapper-Funktionen auf einer Ebene mit den in CFFI gebundenen Funktionen
Benutzerfreundlichkeit	Programmieren im gleichen Skript wie der Entwickler	Leicht zu importieren, genau wie bei einem nativen Python-Modul	Nutzung einer zusätzlichen Klasse oder Programmieren im gleichen Skript wie der Entwickler

Abbildung 5.6: Gesamtvergleich der Binding-Tools im SELMA-Projekt

Die Ergebnisse des Benchmarkings zeigen, dass Ctypes sich durch eine gute und konsistente Laufzeit auszeichnet, im Gegensatz zu Pybind11, welches die langsamste und am wenigsten konsistente Laufzeit aufweist. CFFI befindet sich in puncto Laufzeit ähnlich wie Pybind11, jedoch mit einer konsistenteren Standardabweichung, also stabilerer Laufzeit.

In Bezug auf die Initialisierung zeigen Pybind11 und CFFI Vorteile mit ihrem geringeren Speicherverbrauch und der schnellen Initialisierung, während Ctypes durch höheren Speicherbedarf und längere Initialisierungszeiten auffällt. Alle drei Tools haben jedoch einen höheren Gesamtspeicherverbrauch im Vergleich zur C++-Implementierung.

Ctypes bietet eine besonders intuitive Syntax und umfangreiche Dokumentation, was es besonders entwicklerfreundlich macht. Pybind11 und CFFI bieten ebenfalls eine intuitive Syntax; allerdings ist die Dokumentation bei Pybind11 knapper, und Änderungen sind aufgrund der Notwendigkeit der Kompilierung komplexer. CFFI bietet gute, wenn auch textlastige und teilweise unübersichtliche Dokumentation, ermöglicht aber einfache Änderungen.

Bei der Integration in bestehende Systeme zeigt Ctypes Vorteile durch seine Verfügbarkeit in der Python-Standardbibliothek. Trotz der undurchsichtigen Fehlermeldungen bei Pybind11, erweist sich die Implementierung nach einer initialen Einarbeitungsphase als intuitiv, und das Erstellen von Wrapper-Funktionen gestaltet sich einfach. CFFI bietet währenddessen besondere Herausforderungen bei der Implementierung von Wrapper-Funktionen. Was die Benutzerfreundlichkeit angeht, schneidet Pybind11 am besten ab, da es leicht importiert werden kann und sich wie ein natives Python-Modul verhält. Ctypes und CFFI hingegen erfordern mehr Schritte zum Importieren in ein anderes Skript, was ihre Benutzung potenziell komplizierter macht.

5.4 Empfehlung für das SELMA-Projekt

Die Ergebnisse des Benchmarkings zeigen, dass die Laufzeiten und der Speicherverbrauch der Bindings keine für das SELMA-Projekt signifikanten Abweichungen aufweisen, weshalb die Entscheidung über das am besten geeignete Tool hauptsächlich auf Basis der qualitativen Bewertungskriterien getroffen werden sollte. Obwohl es in der Performance eher mittelmäßig abschneidet, fällt die Entscheidung zugunsten von Pybind11 vor allem aufgrund seiner Entwickler- und Benutzerfreundlichkeit sowie seiner hervorragenden Integrationsfähigkeit. Pybind11 zeichnet sich durch effiziente Handhabung von C++-Features im Kontext von Python aus, unterstützt nahtlos die Konvertierung mit `Numpy`-Arrays und erleichtert das Schreiben von Wrapper-Funktionen. Es lässt sich als natives Python-Modul importieren, was die Trennung zwischen Entwicklercode und Benutzercode vereinfacht und so die Benutzerfreundlichkeit weiter erhöht. Auf Basis dieser Kriterien wird für das SELMA-Projekt die Verwendung von Pybind11 empfohlen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden verschiedene Methoden zur Integration des C^{++}-Codes der SELMA-Decoder-Software in Python untersucht und verglichen. Nach einer detaillierten Analyse der Unterschiede zwischen Python und C^{++}, die für eine erfolgreiche Integration überbrückt werden müssen, wurden die Tools Ctypes, Pybind11 und CFFI ausführlich vorgestellt. Jedes dieser Tools wurde implementiert und die Ergebnisse anhand spezifischer Kriterien evaluiert, um ihre Eignung für das Selma-Projekt zu bestimmen. Die Analyse der Benchmark-Ergebnisse hat ergeben, dass keine signifikanten Unterschiede zwischen den Bindings bestehen, die für das SELMA-Projekt von Bedeutung sind. Daher erfolgt die Auswahl des geeigneten Tools vorrangig auf Basis der qualitativen Bewertungskriterien wie Benutzerfreundlichkeit, Entwicklerfreundlichkeit und Integrationsfähigkeit. Als bevorzugtes Tool für SELMA wird auf Basis dieser Kriterien Pybind11 empfohlen, da die Integration für C^{++}-Entwickler einfach ist und durch die Möglichkeit, echte Python-Module zu erstellen, eine klare Trennung zwischen Entwickler- und Anwendercode gehalten werden kann, was die Benutzerfreundlichkeit erhöht.

Obwohl diese Arbeit eine detaillierte Analyse der drei verschiedenen Binding-Tools Ctypes, Pybind11 und CFFI bietet, bedeutet dies nicht, dass nicht potenziell überlegene Alternativen für den Einsatz im SELMA-Projekt existieren könnten. Da vor allem die Kriterien wie Speichereffizienz noch weiter verbessert werden könnten, könnte es sich lohnen, in Zukunft weitere Tools zur Einbindung von C^{++}-Code in Python zu untersuchen und zu bewerten, um vielleicht eine noch effizientere oder benutzerfreundlichere Lösung zu finden.

Darüber hinaus wäre eine weiterführende Untersuchung des in der Grafik 5.5 dargestellten Speicheroverheads sinnvoll. Es wäre von Interesse, zu bestätigen, ob dieser Overhead tatsächlich durch den Python-Interpreter verursacht wird, oder ob zusätzlich andere Faktoren eine Rolle spielen. Weitere Informationen darüber könnten zur Optimierung der Speichernutzung bei der Verwendung der Binding-Tools beitragen.

Quellenverzeichnis

[1] *Zentralinstitut für Engineering, Elektronik und Analytik (ZEA)*. URL: https://www.fz-juelich.de/de/zea (besucht am 20. 08. 2024).

[2] *Systeme der Elektronik (ZEA-2)*. URL: https://www.fz-juelich.de/de/zea/zea-2 (besucht am 20. 08. 2024).

[3] Cosimo Brogi. „Geophysics-based soil mapping for improved modelling of spatial variability in crop growth and yield“. In: (2019). URL: http://elib.uni-stuttgart.de/bitstream/11682/10763/1/C_Brogi_PhD_Thesis_2019final.pdf.

[4] *Agrosphäre (IBG-3)*. URL: https://www.fz-juelich.de/de/ibg/ibg-3 (besucht am 20. 08. 2024).

[5] *Internes Dokument des ZEA-2*.

[6] *MATLAB & Simulink - MathWorks Deutschland*. URL: https://de.mathworks.com/help/ (besucht am 20. 08. 2024).

[7] *Unlocking the Power of Numerical Methods with Python*. URL: https://www.linkedin.com/pulse/unlocking-power-numerical-methods-python-introduction-joel-tovar--aew6e (besucht am 20. 08. 2024).

[8] Great Learning Team. *Top 30 Python Libraries To Know*. März 2024. URL: https://www.mygreatlearning.com/blog/open-source-python-libraries/ (besucht am 20. 08. 2024).

[9] *History and License*. Python documentation. URL: https://docs.python.org/3/license.html (besucht am 28. 06. 2024).

[10] Luz María Alonso-Valerdi und Francisco Sepúlveda. „Implementation of a Motor Imagery based BCI System using Python Programming Language“. In: *Proceedings of the 2nd International Conference on Physiological Computing Systems* (2015). DOI: 10.5220/0005211500350043. URL: https://doi.org/10.5220/0005211500350043.

[11] *Dynamische Typisierung*. Page Version ID: 227335229. 2022. URL: https://de.wikipedia.org/w/index.php?title=Dynamische_Typisierung (besucht am 20. 08. 2024).

[12] Karishma Shukla. *How Python uses Garbage Collection for Efficient Memory Management*. 2023. URL: https://dev.to/karishmashukla/how-python-uses-garbage-collection-for-efficient-memory-management-270h (besucht am 20. 08. 2024).

[13] *Python vs. C++*. 2023. URL: https://www.ionos.de/digitalguide/websites/web-entwicklung/python-vs-c/ (besucht am 20. 08. 2024).

[14] *types - cppreference.com*. URL: https://en.cppreference.com/book/intro/types.

[15] TylerMSFT. *Complex - Cpp Standard Library*. 2023. URL: https://learn.microsoft.com/de-de/cpp/standard-library/complex?view=msvc-170 (besucht am 20. 08. 2024).

[16] James Cook. *Python Bindings*. Medium. 9. Okt. 2019. URL: https://medium.com/@benjamesian/python-bindings-c946e6573ca7 (besucht am 27. 06. 2024).

[17] Wenzel Jakob, Jason Rhinelander und Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. 2017. (Besucht am 20. 08. 2024).

[18] *Typen und Operatoren*. URL: https://www.grimm-jaud.de/index.php/python-tutorial/42-typen-und-operatoren (besucht am 20. 08. 2024).

[19] *Python Tuples*. URL: https://www.w3schools.com/Python/python_tuples.asp (besucht am 20. 08. 2024).

[20] *All You Need to Know About C++ Memory Management | Simplilearn*. URL: https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-memory-management (besucht am 20. 08. 2024).

[21] *Python Bindings: Calling C or C++ From Python – Real Python*. URL: https://realpython.com/python-bindings-overview/ (besucht am 25. 06. 2024).

[22] Saksham Sharma. *Writing Python Bindings for C++ Libraries: Easy-to-use Performance*. 2023. URL: https://www.youtube.com/watch?v=rB7c69Z5Kus (besucht am 20. 08. 2024).

[23] *smart pointers - cppreference.com*. URL: https://en.cppreference.com/book/intro/smart_pointers (besucht am 20. 08. 2024).

[24] Joel Coburn u. a. „NV-Heaps“. In: *ACM SIGPLAN Notices* 46.3 (2011). DOI: 10.1145/1961296.1950380. URL: https://doi.org/10.1145/1961296.1950380 (besucht am 27. 06. 2024).

[25] Todd A. Anderson und Tim Mattson. „Multithreaded parallel Python through OpenMP support in Numba“. In: *Proceedings of the Python in Science Conference* (2021). DOI: 10.25080/majora-1b6fd038-012. URL: https://doi.org/10.25080/majora-1b6fd038-012 (besucht am 28. 06. 2024).

[26] TylerMSFT. *Structured Exception Handling (C/C++)*. 2024. URL: https://learn.microsoft.com/de-de/cpp/cpp/structured-exception-handling-c-cpp?view=msvc-170 (besucht am 20. 08. 2024).

[27] *Built-in Exceptions*. Python documentation. URL: https://docs.python.org/3/library/exceptions.html (besucht am 28. 06. 2024).

[28] *Exception Handling*. Python documentation. URL: https://docs.python.org/3/c-api/exceptions.html (besucht am 28. 06. 2024).

[29] *ctypes — A foreign function library for Python*. Python documentation. URL: https://docs.python.org/3/library/ctypes.html (besucht am 26. 06. 2024).

[30] *Dynamic Link Library*. In: *Wikipedia*. Page Version ID: 243496535. 27. März 2024. URL: https://de.wikipedia.org/w/index.php?title=Dynamic_Link_Library&oldid=243496535 (besucht am 26. 06. 2024).

[31] D. Bhavana, M. B. Veena und Santosh Kumar Sahu. „An Automated Approach of Detection of Memory Leaks for Remote Server Controllers“. In: *EMITTER International Journal of Engineering Technology* (2020). DOI: 10.24003/emitter.v8i2.550. URL: https://doi.org/10.24003/emitter.v8i2.550 (besucht am 20. 08. 2024).

[32] *Answer to "What is the difference between new/delete and malloc/free?* 2008. URL: https://stackoverflow.com/a/240308 (besucht am 20. 08. 2024).

[33] Christopher Swenson. *Bypassing the Python GIL with ctypes — Christopher Swenson*. URL: http://caswenson.com/2009_06_13_bypassing_the_python_gil_with_ctypes.html (besucht am 28. 06. 2024).

[34] URL: https://pybind11.readthedocs.io/en/stable/advanced/functions.html (besucht am 20. 08. 2024).

[35] URL: https://cffi.readthedocs.io/en/latest/ref.html#conversions (besucht am 20. 08. 2024).

[36] Armin Rigo. *Answer to “How can I gracefully handle exceptions/crashes when calling a CFFI function?”* Okt. 2023. URL: https://stackoverflow.com/a/77330224.

[37] *NumPy*. URL: https://numpy.org/ (besucht am 20. 08. 2024).

[38] Python Memory Profiler. URL: https://github.com/pythonprofilers/memory_profiler.

[39] URL: https://yanto.fi/2022/09/benchmark-of-python-c-bindings/ (besucht am 20. 08. 2024).

[40] *Valgrind*. URL: https://valgrind.org/docs/manual/ms-manual.html (besucht am 20. 08. 2024).

Jül-4447 • Oktober 2024
ISSN 0944-2952

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum