Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Hausarbeit im Proseminar:
Python im Hochleistungsrechnen

[C/C++-Extensions]

Marcel Robohm

marcel.robohm@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr. 7159610
Fachsemester 2

Betreuerin: Kira Duwe

Inhaltsverzeichnis

1 Einleitung

Python ist eine Programmiersprache mit stetig wachsender Beliebtheit. Sie bietet durch ihre hohe Abstraktion viele Vorteile, aber auch Nachteile. Um das Beste aus mehreren Welten zu verbinden, kann man ein Native Interface nutzen. Doch inwiefern steht der Nutzen dieser Aufspaltung von Aufgaben auf zwei Programmiersprachen im Verhältnis zum Aufwand? Das wird anhand eines praktischen Beispiels überprüft.

Besonderes Augenmerk wird bei der Überprüfung des Nutzen auf die zeitliche Performance gelegt. Als zu lösendes Problem wird eine Matrixmultiplikation genutzt. Wegen der Menge an kleinen arithmetischen Rechnungen eignet sich dieses Beispiel im Kontext von Simulationen.

In dieser Hausarbeit wird sich hauptsächlich auf die Programmiersprache C++ zum Schreiben der Extension konzentriert. Ebenso wird von der Pythonimplmentation CPython ausgegangen. Alle dargestellten Funktionen werden auch von PyPy in ähnlicher Form bereitgestellt.

Anfangs wird die Motivation nochmal deutlicher nahegelegt. Danach wird auf die Besonderheiten der Programmiersprache C++ eingegangen, damit nach dem Lesen die Mechanismen und Konstrukte der C++-Extension verstanden werden können. Mit diesem Wissen wird dann schrittweise erläutert, wie man eine C++-Extension aufbauen kann. Die Erläuterung wird an dem o.g. Beispiel aufgezogen. Abschließend wird ein Performancetest vorgestellt und ein Fazit gezogen.

2 Hauptteil

2.1 Motivation

2.1.1 Performance

Ein sehr wichtiger Punkt in der Entwicklung von Software kann die Laufzeit sein. Bestimmte Anwendungen wie Grafikberechnungen oder Simulationen haben einen hohen Anspruch an die Performance. Oft gehören dazu aufwendige mathematische Berechnungen. Python benutzt einen Interpreter zur Codeausführung. Neben den Vorteilen bedeutet das aber auch, dass einige Codeoptimierungen gar nicht vorgenommen werden können. Betroffen sind alle Optimierungsversuche, die zukünftigen Code in Betracht ziehen müssen, oder allgemein eine Art von Kontextwissen bedürfen, die ein Interpreter im Gegensatz zum Compiler nicht zur Verfügung hat. Codeoptimierung funktioniert grob so, dass Informationen darüber gesammelt werden, wie sich das Programm verhalten kann und darauf basierend der Code vereinfacht wird, indem zum Beispiel Code geinlined wird oder mathematische Vereinfachungen vorgenommen werden.

Inlining beschreibt das direkte einfügen von Code an einer Stelle in der normalerweise ein Verweis auf den Code lag. So ist es möglich, den Callstack niedrig zu halten und unnötige Sprünge zu vermeiden.

Zusätzlich kommen durch Objektorientierung und Duck-Typing weitere Ungewissheiten für den Interpreter hinzu, die weitere Optimierung erschwert [Ste18].

2.1.2 Zugriff auf Maschinenebene

Neben der Performance ist der Zugriff auf die Maschinenebene ein weiterer wichtiger Aspekt der Motivation. Python ermöglicht keinen mit C vergleichbar freien Zugriff auf die Hardware wie es etwa mit Zeigern möglich wäre. Das Fehlen solcher Konstrukte wie Zeiger (2.2.3) erschwert auch eine „händische Optimierung“. Mit „händische Optimierung“ ist hier gemeint, dass die Optimierung nicht vom Compiler und automatischer Codeanalyse ausgeht, sondern durch den Programmierer selbst gemacht wird. Ein Beispiel dafür ist u.a. der Algorithmus zum Vertauschen zweier Variablen ohne Zwischenspeicher (Listing 2.1).

Listing 2.1: Tausch der Belegung zweier Variablen ([And05])

```
1 static void tausch(int& a, int& b) {
2   a = a ^ b;
3   b = a ^ b;
4   a = a ^ b;
5 }
```

Zwar gibt es bei Python schon Konstrukte, die dieses Beispiel kürzer lösen und händische Optimierung dadurch obsolet machen, doch es existieren noch nicht für alle Probleme derartige Optimierungen. Außerdem kann man auf Maschinenebene auch Programme realisieren, die darauf angewiesen sind, volle Kontrolle über die Hardware zu haben. Beispiele dafür wären Programme für eingebettete Systeme, Betriebssysteme oder Treiber [Ste18].

2.1.3 Neuimplementationen

Bekanntlich sollte man das Rad nicht bei jeder Benutzung neu erfinden. Genauso verhält es sich mit bereits geschriebener Software. Wenn eine Funktionalität existiert, die bereits in C (oder C++) realisiert ist, wäre es ein vermeidbarer Aufwand, die Funktionalität neu in Python zu implementieren. Stattdessen wäre es deutlich effizienter, wenn man auf bereits existierende Funktionalität, die in einer anderen Sprachen (hier C/C++) umgesetzt ist, zurückgreifen kann. Dieser Aspekt ist besonders dann relevant, wenn der Aufwand einer Neuimplementation groß ist [Ste18].

2.2 C++

Im Folgenden werden die besonderen Eigenschaften von C++ kurz dargestellt und so weit erklärt, wie es für das Verstehen einer C++-Extension für Python notwendig ist.

2.2.1 Überblick

C++ wird in der Regel kompiliert und gilt als eine multiparadigmatische low- bis highlevel-Sprache. Dies bezieht sich auf den Grad der Abstraktioin von der Maschinenebene. Python im Gegensatz gilt als eine stark abstrahierende Sprache. Die Lowlevelcharakteristika von C++ sind beispielsweise Zeigerarithmetik und manuelle Speicherverwaltung. „Multiparadigmatisch“ bedeutet, dass die Sprache die Programmierung in mehreren Programmierparadigmen unterstützt, wobei besonderer Fokus auf die imperative Programmierung gelegt ist [Cppa].

2.2.2 Makros

Bei Makros handelt es sich vereinfacht um funktionsähnliche Konstrukte. Sie können auch Parameter entgegennehmen und besitzen beim Aufruf eine ähnliche Syntax (Listing 2.2).

Listing 2.2: Beispielmakrobenutzung ([Cppb])

```
1 ICHBINEINMAKRO(parameter)
```

Wie man im Listing 2.2 erkennen kann, kann man einen Makroaufruf leicht mit einem Funktionsaufruf verwechseln. Unter anderem deshalb werden Makros meisten in Großbuchstaben wie im Beispiel geschrieben [Cppb]. Wenn ein Makro keine Parameter annimmt, muss man die Klammern weglassen. Im Gegensatz zu einer Funktion werden Makros nicht direkt im Maschinencode/Kompilat realisisert. Der Präprozessor verarbeitet sie und behandelt Makros wie Richtlinien zur Textkonvertierung. Dabei werden Makros eindeutig einer bestimmten Textschablone zugewiesen, in der nach Belieben die Parameter eingesetzt werden. Somit wird der Quelltext, der den Compiler erreicht, makrofrei. Wichtig anzumerken ist dabei, dass durch diese bloße Textformatierung keine Präzedenzen gelten. Das heißt, dass Operationen in einem Makro nicht gekapselt sind und von umliegendem Code an der einzufügenden Stelle gestört werden könnten. Aus diesem Grund empfiehlt es sich bei der Definition eines Makros um die gesamte Textschablone Klammern zu setzen, falls die störenden Effekte nicht in einer Form gewollt sind. Makros werden folgendermaßen definiert (Listing 2.3).

Listing 2.3: Beispielmakrodefinition ([Cppb])

```
1 #define ICHBINEINMAKRO(parameter) (parameter + 1)
2 #define PARAMETERLOS 42
3 #define VERSCHACHTELT ICHBINEINMAKRO(PARAMETERLOS)
```

Makros können sich selbst benutzen und als Konstantendefinitionen benutzt werden.

2.2.3 Zeigerarithmetik

Bei Zeigerarithmetik handelt es sich um die Möglichkeit in C++ mit Referenzen auf Speicherzellen im Arbeitsspeicher genauso zu rechnen wie mit natürlichen Zahlen. Dies kann Performancevorteile beim Iterieren über feste Datenstrukturen bieten wie im folgenden Beispiel dargestellt (Listing 2.4).

Listing 2.4: Zeigerarithmetik

```
1  int main() {
2    int array[10];
3
4    int* begin = array;
5    int* end = &array[9]; // (begin + 9)
6
7      for (int* iter = begin; iter <= end; iter++) {
8        *iter = function();
9      }
10 }
```

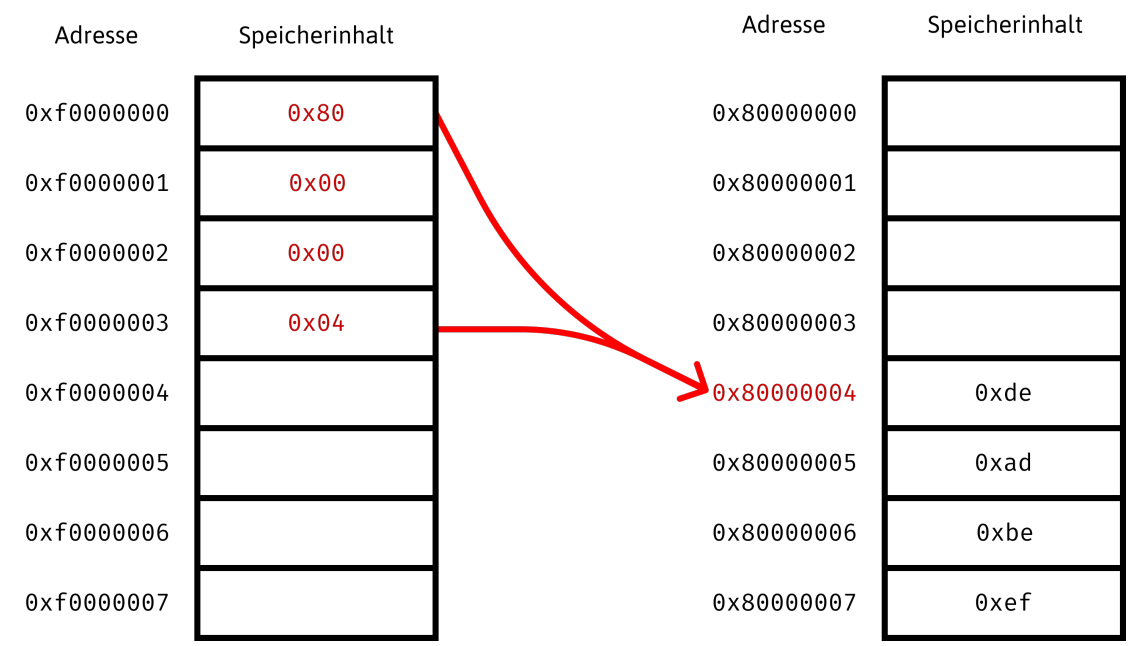

Abbildung 2.1: Zeiger

Der Stern nach einem Typnamen mach den folgenden Bezeichner zu einem Zeiger, der auf diesen Typen zeigt. Andersherum kann der Stern auch dazu genutzt werden, um auf den Inhalt zuzugreifen, auf den der Zeiger zeigt, vor dem man den Stern setzt. (Zeile 7) Hier eine schematische Darstellung eines Zeiger (2.1).

2.2.4 Speicherverwaltung

Die Speicherverwaltung in C++ funktioniert primär explizit, sofern auf dem Heap alloziiert wird. In modernem C++ gibt es Datencontainer, die über Referenzzählung explizite, manuelle Speicherverwaltung vermeiden [Ale]. Mit dem `new`-Operator kann man Speicher auf dem Heap alloziieren, den man mit dem `delete`-Operator wieder freigeben kann. In der Praxis sieht das folgendermaßen aus (Listing 2.5).

Listing 2.5: Speicherverwaltung bei C++

```
1 int main() {
2   int stack = 0;          // Allokation im Stack
3
4   int* heap = new int();  // Allokation im Heap
5   // "heap" nutzen...
6   delete heap;            // Speicherfreigabe
7 }
```

2.3 Was ist ein Native Interface?

Ein Native Interface ist eine Schnittstelle einer Programmiersprachen für andere Programmiersprachen. Ziel ist meistens eine Ausnutzung der Vorteile beider Sprachen und eine bessere Arbeitsteilung bezüglich der Sprachen. Realisiert wird es teilweise über ein Application Binary Interface (ABI) [Pyta]. Eine ABI definiert die Kommunikation zwischen 2 Programmteilen für Maschinensprache. Typische Spezifiktationen wären die Festhaltung von Registern für bestimmte Zwecke oder Binärformatentscheidungen (Big Endian / Little Endian / etc).

2.4 Native Interface in Python

Im behandelten Beispiel handelt es sich um ein Native Interface, das das Schreiben von C++-/C-Modulen für Python möglich macht. Dabei werden Vorteile der C++-Welt (2.2) genutzt. Konkret wird dieses Native Interface von Numpy genutzt, um rechenintensive Aufgaben performant in Python bzw. aus Python aufrufbar durchzuführen [Num]. Die ABI in Python ist ab Version 3 abwärtskompatibel. Um neuere Features nutzen zu können, muss man selbst die Version in der Extension definieren.

2.4.1 Verfahrensweise

Zum Testen der Vorteile gegenüber einer reinen Pythonimplementierung wird im Folgenden eine Extension aufgebaut und bezüglich Performance verglichen. Diese Extension soll eine Matrixmultiplikation durchführen.

2.4.2 Grundlegender Aufbau

Das Kernelement einer Pythonextension ist der `Python.h`-Header. Diese Datei enthält alle wichtigen, pythonspezifische Inhalte, die man braucht, um eine Python-Extension in C++ zu schreiben. Die meisten Python-Funktionen besitzen einen `Py`-Präfix. Eine einfache Funktion, die über Python aufrufbar sein soll, kann so aussehen (Listing 2.6).

Listing 2.6: Grundlegender Aufbau einer Funktion ([Vis])

```
1 #include <Python.h>
2
3 PyDoc_STRVAR(TestModule_example_doc, "Dokumentation...");
4
5 // Beispielfunktion
6 PyObject* TestModule_matrixMultiplication(PyObject* self, PyObject* ↩
     args, PyObject* kwargs) {
7   return self;
8 }
```

Mit dem Makro `PyDoc_STRVAR` kann man einen Schnittstellenkommentar für eine Funktion erstellen. Die Funktion selbst behandelt Objekte aus der Pythonwelt mit einem Zeiger auf ein Exemplar von `PyObject`. Entsprechend muss eine Funktion auch immer einen `PyObject`-Zeiger zurückgeben und 3 Parameter des selben Typs annehmen. Der erste Parameter ist eine Selbstreferenz auf das Objekt, auf dem die Funktion aufgerufen wurde, oder eine Referenz auf das Modulobjekt. Der zweite Parameter beinhaltet alle Parameter, die ohne Namen an die Funktion übergeben wurden, und der dritte Parameter enthält alle Parameter, die mit Namen an die Funktion weitergegeben wurden.

2.4.3 Speicherverwaltungssysteme

Da C++ keine direkte Speicherverwaltung anbietet, muss man den Referenzzähler des Pythoninterpreters manuell setzen. Von Referenzzählung spricht man, wenn man bei jedem Objekt auf dem Heap einen Zähler hinzufügt, der die Referenzen zählt, die auf das Objekt zeigen. Sobald eine Zählvariable auf 0 fällt, wird das dazugehörige Objekt gelöscht. Die `Python.h`-Datei stellt einen für diese manuelle Betätigung der Referenzzählung Makros zur Verfügung (Listing 2.7).

Listing 2.7: Wichtige Makros für Speicherverwaltung ([Pytb])

```
1 Py_INCREF(PyObject* o);
2 Py_XINCREF(PyObject* o);
3 Py_DECREF(PyObject* o);
4 Py_XDECREF(PyObject* o);
5 Py_CLEAR(PyObject* o);
```

Nach dem bekannten Präfix kann man schon am Namen ablesen, was sie tun. „DEC“ steht dabei für „dekrementieren“ und „INC“ steht für Inkrementieren, wobei sich beides auf den Referenzzähler des Objektes bezieht, das man als Parameter übergibt. Die Makros mit X (`Py_XINCREF` und `Py_XDECREF`) führen ihre Operation nur aus, wenn das übergebene Objekt auch noch existiert, andernfalls tun sie nichts. `Py_CLEAR` dekrementiert den Referenzzähler und setzt den eingegebenen Zeiger auf `nullptr`. Am Typen des Parameters erkennt man schon, dass sich die Operationen nur auf `PyObject`s beziehen. Das hat den Grund, dass man nur bei diesen Referenzzählung über diese Makros anwenden sollte. Innerhalb des C++-Codes wird der Speicher mit C++-internen Mitteln verwaltet. (manuell oder eigene Referenzzählung) Die Referenzzählung ist auch nur dann von Relevanz, wenn `PyObject`s innerhalb der eigenen Methode erstellt werden. Dann ist man nämlich in der Pflicht, deren Referenzzähler wieder zu dekrementieren oder sie an eine andere Methode weiter geben, die sich darum kümmert [Pytb].

2.4.4 Objektkonvertierung

Wenn man im C++-Code mit den übergebenen Parametern arbeiten möchte, muss man die `PyObjects` zu Datentypen konvertieren, mit denen man arbeiten kann. In unserem Beispiel erwarten wir abstrakt zwei Matrizen als Parameter. Matrizen werden hierbei als verschachtelte Listen realisiert. Für das Parsen der Eingabeparameter stellt die Python-API u.a. folgende Funktionen bereit (Listing 2.8).

Listing 2.8: Funktion zum Parsen der formalen Parameter ([Pytb])

```
1 int PyArg_ParseTuple(PyObject* args, const char* format, ...);
2 int PyArg_ParseTupleAndKeywords(PyObject* args, PyObject* kw, const ↩
     char* format, char* keywords[], ...);
```

Beide Funktionen aus Listing 2.8 können positionale Parameter parsen. Die untere Funktion kann zusätzlich die benannten Parameter parsen, wenn man ihr einen null-terminierten String mit den Namen der Parameter übergibt. Beide Funktionen nehmen ein format-String entgegen, der spezifiziert, zu welchen Objekten/Datentypen die Parameter geparst werden sollen. Die Ergebnisse des Parsens werden in die Referenzen kopiert, die man als Parameter anstelle der drei Punkte übergibt. Die Rückgabe kann als Wahrheitswert interpretiert werden und gibt an, ob die Operation erfolgreich war [Pytc]. Für das Matrixmultiplikationsbeispiel müssen 2 Listen aus den Parametern geparst werden. Weil Listen wiederrum original `PyObject`-Zeiger sind, sieht der Quelltext der Funktion jetzt so aus (Listing 2.9).

Listing 2.9: Funktion zum Parsen der formalen Parameter ([Pytb])

```
1 PyObject* TestModule_matrixMultiplication(PyObject* self, PyObject* ↩
     args, PyObject* kwargs) {
2   PyObject* matrixA_obj = nullptr;
3   PyObject* matrixB_obj = nullptr;
4
5   static char* keywords[] = { "matrixA", "matrixB", nullptr };
6   if (!PyArg_ParseTupleAndKeywords(args, kwargs, "OO", keywords, &↩
        matrixA_obj, &matrixB_obj)) {
7     return nullptr;
8   }
9 }
```

Weiterhin werden einem noch weitere Hilfsfunktionen und -makros geboten, um besser mit den `PyObjects` arbeiten zu können. Nützlich für die weitere Konvertierung zur Matrix sind (Listing 2.10).

Listing 2.10: Funktionen und Makros für die Arbeit mit PyObjects ([Pytb])

```
1 int PyList_Check(PyObject* obj);
2 Py_ssize_t PyList_Size(PyObject* obj);
3 PyObject* PyList_GetItem(PyObject* obj, Py_ssize_t index);
4 long PyLong_AsLong(PyObject* obj);
```

Das Makro `PyList_Check` prüft, ob der übergebene Zeiger auf eine Liste zeigt und gibt das als Wahrheitswert zurück. Die restlichen Funktionen sind selbsterklärend. Aus denen kann man sich dann eine Funktion `PyList_ToMatrix` bauen, die ein `PyObject` in eine Matrix konvertiert. Als Repräsentation für die C++-Umgebung wird ein Typalias definiert (Listing **??**).

Listing 2.11: Typalias

```
1 template<class T>
2 using Matrix = std::vector<std::vector<T>>;
```

Dabei besteht eine Matrix aus verschachtelten Vektoren mit einem gemeinsamen Typparameter `T`.

2.4.5 Exceptions

Ein wichtiges Konstrukt zum Signalisieren von Fehlern sind Exceptions. Diese können nicht einfach aus dem C++-Code geworfen werden, da der Pythoninterpreter diese nicht erkennen würde. Der Pythoninterpreter verarbeitet Exceptions mit einer globalen Variable vergleichbar mit `errno` in C, auf die regelmäßig geprüft wird. Das bedeutet für die C++-Extension, dass diese Variable mit einem adäquaten Wert gesetzt werden muss, wenn man das Erkennen einer Exception durch den Interpreter beabsichtigt. Zusätzlich muss man `nullptr` zurückgeben, damit der Interpreter weiß, dass er die globale Exceptionvariablen überprüfen muss. Dabei gibt es nicht nur eine Variable für das Anzeigen einer Exception, sondern es gibt mehrere, die Daten über die genauen Umstände des Fehler beinhalten sollen. Um diese Variablen zu setzen, bietet der `Python.h`-Header u.a. diese Funktionen an (Listing 2.12).

Listing 2.12: Funktionen zum Werfen von Exceptions([Pytb])

```
1 void PyErr_SetString(PyObject* type, const char* message);
2 void PyErr_SetObject(PyObject* type, PyObject* value);
3 void PyErr_SetNone(PyObject* type);
4 PyObject* PyErr_SetFromErrno(PyObject* type);
```

Für `PyObject* type` aus Listing 2.12 sollte man ein Exceptionobjekt einsetzen. Es gibt eine Reihe von Exceptionobjekten, die man über den `Python.h`-Header nutzen kann. Bei dem Beispiel bietet es sich an, bei einem Parametern mit falschen Typen eine Exception zu

werfen. Für diesen Zweck gibt es die `PyExc_TypeError`-Variable. Wenn die Funktion `PyList_ToMatrix` eine C++-Exception wirft, kann man sie abfangen und dann durch einen Aufruf von `PyErr_SetString(PyExc_TypeError, "...")` den Fehler dem Interpreter signalisieren. Alternativ kann man in C auch die Funktion `PyErr_SetFromErrno` nutzen, die bei einem Fehler direkt die Informationen aus der `errno`-Variable ableitet.

Das Gegenstück zum Werfen einer Exception ist das Abfangen. Dies geschieht über diese Funktkionen (Listing 2.12).

Listing 2.13: Funktionen zum Fangen von Exceptions([Pytc])

```
1 PyObject* PyErr_Occurred();
2 int PyErr_ExceptionMatches(PyObject* exc);
```

Die erste Funktion von Listing 2.12 gibt gegebenenfalls eine geworfene Exception zurück und die zweite matcht diese Exception gegen eine andere, damit man auf den genauen Exceptiontyp prüfen kann.

Der bisherige Code der Funktion für Matrixmultiplikation sieht so aus wie in Listing 2.14.

Listing 2.14: Funktionen zum Berechnen eines Matrixproduktes

```
1  PyObject* TestModule_matrixMultiplication(PyObject* self, PyObject* ↩
       args, PyObject* kwargs) {
2    PyObject* matrixA_obj = nullptr;
3    PyObject* matrixB_obj = nullptr;
4
5    static char* keywords[] = { "matrixA", "matrixB", nullptr };
6    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "OO", keywords, &↩
         matrixA_obj, &matrixB_obj)) {
7      PyErr_SetString(PyExc_TypeError, "Wrong type of parameters!");
8      return nullptr;
9    }
10
11   Matrix<long> matrixA;
12   Matrix<long> matrixB;
13
14   try {
15     matrixA = PyList_ToMatrix(matrixA_obj);
16     matrixB = PyList_ToMatrix(matrixB_obj);
17   }
18   catch (const std::exception& exception) {
19     PyErr_SetString(PyExc_TypeError, exception.what());
20     return nullptr;
21   }
```

```
22
23   Matrix<long> resultMatrix = Calculator::matrixMultiplication(↩
         matrixA, matrixB);
24
25
26   return PyList_FromMatrix(resultMatrix);
27 }
```

Die Funktion `Calculator::matrixMultiplication(matrixA, matrixB)` berechnet die Matrixmultiplikation auf einfacher Weise. Die genaue Implementation würde den Rahmen dieser Hausarbeit sprengen.

2.4.6 Abschluss des praktischen Teils

Somit wäre der Kern der Extension vollendet. Damit der Interpreter das Kompilat als aufrufbares Modul auffasst, müssen noch einige Metainformataionen hinzugefügt werden (Listing 2.4.6).

Methodenauflistung [Pytb]

```
1  // Liste aller Funktionen im Modul
2  static PyMethodDef TestModule_functions[] = {
3    { "funcName",
4        (PyCFunction)TestModule_example,
5        METH_VARARGS | METH_KEYWORDS,
6        TestModule_example_doc },
7    { nullptr, nullptr, 0, nullptr }
8    // Null-Terminator
9   };
10
11  // Modulinitialisierung
12 int exec_TestModule(PyObject* module) {
13   PyModule_AddFunctions(module, TestModule_functions);
14
15   PyModule_AddStringConstant(module, "__author__", "Marcel Robohm");
16   PyModule_AddStringConstant(module, "__version__", "1.0.0");
17   PyModule_AddIntConstant(module, "year", 2019);
18
19   return 0;
20 }
```

Durch diesen Code (Listing 2.4.6) wird dem Interpreter vermittelt, welche Funktionen das Modul hat und dessen Randdaten. Außerdem wird der Funktion das erzeugte Dokumentationsobjekt zugeordnet und es wird bestimmt, wie die Parameter übergeben werden sollen. (positional oder über Benennungen)

Der folgende Code dient der Initialisierung und Dokumentation des Moduls selbst. Außerdem werden die Slots ermittelt mit dem benötigtem Speicher, welche allerdings hier nicht näher betrachtet werden (Listing 2.4.6).

Modulaufbau [Pytb]

```
PyDoc_STRVAR(TestModule_doc, "Dokumentation des Moduls");

static PyModuleDef_Slot TestModule_slots[] = {
  { Py_mod_exec, exec_TestModule },
  { 0, nullptr }
};

static PyModuleDef TestModule_def = {
  PyModuleDef_HEAD_INIT,
  "TestModule",
  TestModule_doc,
  0,           // Groesse des benoetigten Modulspeichers
  TestModule_functions, // Methodenliste
  TestModule_slots,
  nullptr,        // Funktionen bei GC-Traversal
  nullptr,        // Destruktor mit Loeschen
  nullptr,        // Moduldestruktor
};

PyMODINIT_FUNC PyInit_TestModule() {
  return PyModuleDef_Init(&TestModule_def);

}
```

Zusätzlich wird einem hier (Listing 2.4.6) die Möglichkeit geboten, diverse Ereignisse mit Funktionen zu behandeln. Ereignisse wären beispielsweise die Zerstörung des Moduls durch den Garbage Collector von Python.

Kompiliert ist das alles von Python aus aufrufbar nach der entsprechenden `import`-Anweisung. Zum Vergleich der zeitlichen Performance ist folgend eine Implementation der Matrixmultiplikation in Python (Llisting 2.15).

Listing 2.15: Pythonimplementation der Matrixmultiplikation

```
def PyMatrixMultiplication(matrixA, matrixB):
    return [[sum(a * b for a,b in zip(X_row,Y_col))
             for Y_col in zip(*matrixB)]
                 for X_row in matrixA]
```

Zum Testen der Performance werden zwei 100-mal-100-Matrizen mit zufälligen Werten generiert und beide Implementationen aufgerufen. Die Laufzeiten der Funktionen wird gemessen. Dieser Test wird 8 mal durchgeführt. Die Angaben der Performance werden relativ angeben basierend auf der Laufzeit des Pythoncodes, der die Zahl 1 zugewiesen bekommt. Das C++-Modul wird doppelt aufgeführt, weil es Einstellmöglichkeiten bei der Optimierung durch den Compiler gibt. „`/Od`“ bedeutet keine Optimierung und „`/O2`“ bedeutet Optimierung mit Schnelligkeitsfokus.

Ausgeführtes Objekt	Relative Laufzeit	Absolute Laufzeit in Sekunden
Python	1	70,9723949
C++-Extension (`/Od`)	0,112	7,9377632
C++-Extension (`/O2`)	0,017	1,1823653

Anhand der Daten ist leicht ersichtlich, dass die C++-Extension einen deutlichen Vorteil bezüglich der Laufzeit gegenüber der puren Pythonimplementation hat.

2.5 Einschränkungen

Zu beachten ist allerdings, dass durch das Kompilieren des C++-Codes das Modul plattformabhängig wird. Das hängt damit zusammen, dass über eine ABI kommuniziert wird, welche immer architekturabhängig sind. Außerdem erhöht sich neben der Laufzeitgeschwindigkeit auch der Arbeitsaufwand in der Entwicklung deutlich. Allein die Implementation der Matrixmultiplikation brauchte mehr als 100 Zeilen C++-Code, während die Pythonimplementation ein Vierzeiler war.

3 Fazit

Anhand eines Beispiels wurde verdeutlicht, inwiefern sich die Entwicklung einer C++-Extension für Python lohnen kann. Dabei wurde der Fokus auf die zeitliche Performance gelegt. Die Laufzeit für rechenintensive Aufgaben beträgt nach der Portierung auf C++ nur noch 1,7% der Ursprungslaufzeit mit Python. Bei laufzeitkritischen Anwendungen kann es sich deutlich lohnen, bestimmte rechenintensive Aufgaben in C++ auszulagern. Das Ergebnis kann allerdings nicht verallgemeinert werden, da nicht alle Rechnungen einer Matrixmultiplikation ähneln. Trotzdem kann es einen Eindruck vermitteln, in welche Richtung der Gewinn an Performance geht. Nebenbei ist Performance nicht der einzige Grund, weshalb man eine Extension schreiben sollte 2.1.

Literaturverzeichnis

[Ale] Andrei Alexandrescu.

[And05] Sean Eron Anderson. Bit twiddling hacks. Stanford University, 1997-2005.

[Cppa] https://en.cppreference.com/w/cpp/language/main_function. Zugriff: 29.05.2019.

[Cppb] http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/c++_vorverarbeiterdirektiven_de. Zugriff: 03.08.2019.

[Num] https://github.com/numpy/numpy. Zugriff: 03.08.2019.

[Pyta] https://packaging.python.org/guides/packaging-binary-extensions. Zugriff: 29.05.2019.

[Pytb] https://docs.python.org/3/extending/extending.html. Zugriff: 29.05.2019.

[Pytc] https://cubar.nl/doc/python/c-api.pdf. Zugriff: 29.05.2019.

[Ste18] Ralph Steyer. Programmierung in Python. pages X–Y, 2018.

[Vis] https://docs.microsoft.com/de-de/visualstudio/python/working-with-c-cpp-python-in-visual-studio. Zugriff: 29.05.2019.