Technology
Arts Sciences
TH Köln

Fachhochschule
Dortmund
University of Applied Sciences and Arts

Portierung einer Java-Reaktorsimulation nach Python und Performanz-Vergleich mit verschiedenen Python-Java-Bridges

Projektarbeit im Verbundstudiengang Wirtschaftsinformatik (Master)

vorgelegt von:	Niklas Fabig
Matrikel-Nr.:	11144244 / 7206640
Adresse:	Langenfelder Str. 125
	22769 Hamburg

eingereicht bei:	Prof. Dr. Wolfgang Konen

Hamburg, 11/2021

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Hamburg, 16.11.2021

Ort, Datum

Rechtsverbindliche Unterschrift

Inhalt

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

ANOVA	Analysis of Variance
JIT	Just-in-Time
JNI	Java Native Interface
JVM	Java Virtual Machine
RL	Reinforcement Learning
VM	Virtual Machine

1 Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Portierung einer Java-Reaktorsimulation nach Python und einem Performanz-Vergleich zwischen dieser Python-Implementierung und zwei Python-Java-Bridges. Eine Java-Simulation eines Reaktors, welche von Herrn Weyers implementiert wurde, dient als Grundlage [Weyers et al., 2017, S.89-121]. Diese Java-Implementierung wird in Python übersetzt und zwei Python-Java-Bridges implementiert. Um sicherzustellen, dass die Funktionalität identisch ist, werden automatisierte Tests programmiert. Anschließend soll untersucht werden, ob die Python-Java-Bridges im Vergleich zu einer nativen Python-Implementierung einen Overhead erzeugen und wie hoch dieser Overhead ist.

In diesem Bereich gibt es bisher wenige wissenschaftliche Arbeiten. Einen vergleichbaren Performanz-Vergleich zwischen Python-Java-Bridges und nativen Implementierungen stellt das Paper von [Labanda-Jaramillo et al., 2018] dar. Das Paper ist auf Spanisch geschrieben und wenig zitiert. Teile der Ergebnisse sind verwunderlich. Beispielsweise wird dargestellt, dass die Py4J-Implementierung im arithmetischen Mittel deutlich schneller als die native Java-Implementierung ist.

Ähnliche weitere Arbeiten, wie etwa das Paper von [Redondo und Ortin, 2015] aus dem IEEE-Journal, beschäftigen sich mit verschiedenen Python-Implementierung (CPython, Jython, Cython). Dabei berücksichtigen sie allerdings keine Python-Java-Bridges.

Für die Untersuchung wird zunächst in das Thema Performanz und Benchmarks eingeführt. Es werden verschiedene statistische Verfahren zum Design, Messung und Auswertung des Experiments vorgestellt. Anschließend wird auf die verwendeten Sprachen und ihre Eigenschaften eingegangen. Im praktischen Teil der Arbeit werden zuerst die verschiedenen Implementierungen beschrieben. Anschließend wird aufgezeigt, wie durch automatisierte Tests sichergestellt wird, dass die Python-Implementierung die gleiche Funktionalität wie die Java-Implementierung aufweist. Des Weiteren wird das Design des Performanz-Experiments definiert, um anschließend die Ergebnisse darzustellen und zu analysieren. Zum Schluss werden die Ergebnisse zusammengefasst und ein Fazit erstellt.

2 Softwaretests

Das IEEE definiert im Softwarebereich ein Test wie folgt: „An activity in which a system or component is executed [...] and an evaluation is made of some aspect of the system or component.“ [IEEE, 1990]
Generell wird bei Softwaretests zwischen automatisiertem und manuellem Testen unterschieden. Performanztests, Unittests und Integrationstests stellen unterschiedliche Arten von Tests dar [Pittet, o. J.]. Im Folgenden wird auf Arten des automatisierten Testens zur Sicherstellung der Funktionalität sowie auf Performanz-Benchmarks eingegangen.

2.1 Automatisierte Tests zur Sicherstellung der Funktionalität

Bereits im Jahr 2000 beschrieb [Fowler, 2000], dass Softwaretests automatisiert sein sollten. Generell gelten automatisierte Tests im modernen Softwareengineering als Best Practices. Um die Funktionalität eines Programms sicherzustellen, werden verschiedene Arten von Tests vorgenommen. [Cohn, 2010, S. 312] hat dazu eine Test-Pyramide vorgestellt. Diese ist in Abbildung 1 dargestellt.

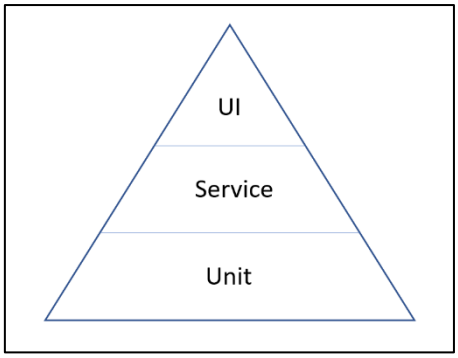

Abbildung 1: Test-Pyramide angelehnt an [Cohn, 2010, S. 312]

Die Testpyramide beschreibt drei Ebenen. Für jede dieser Ebenen können automatisierte Test programmiert werden. Die Pyramide wird oft als Grundlage im Bereich des automatisierten Testens verwendet, aber auch kritisiert. Beispielsweise schreibt [Vocke, 2018] auf Martin Fowlers Blog, dass die Pyramide zu simplifiziert sei und die Definition von Unittests und Servicetests schwierig sei. Dennoch zeige die Testpyramide, dass Tests mit unterschiedlicher Granularität implementiert werden sollten. Die Unittests besitzen den kleinsten Umfang und testen eine einzelne Methode.

Servicetests sollen laut [Cohn, 2010, S. 313] die einzelnen Komponenten oder Services ohne die UI überprüfen. Sie besitzen einen größeren Scope als die Unittests und testen mehrere Methoden zusammen.

UI-Tests überprüfen, ob das User Interface richtig funktioniert und die richtigen Funktionen aufruft. Ende-zu-Ende-Tests überprüfen die gesamte Applikation. Durch die hohe Komplexität sollte die Anzahl an Ende-zu-Ende-Tests minimiert werden [Vocke, 2018].

2.2 Software Performanz-Benchmarking

Performanz in Bezug auf Software wird von Smith und Williams folgendermaßen definiert „Performance is the degree to which a software system or component meets its objectives for timeliness.“ [Smith und Williams, 2004, S.343] Performanz von Software kann oft nicht später hinzugefügt werden, sondern muss von Anfang an bei der Softwarearchitektur berücksichtig werden [Smith und Williams, 2004, S. 344]. Um eine empirische Bewertung der Performanz von Software vorzunehmen, werden Benchmarks verwendet. Es können sowohl einzelne Teile, Klassen oder eine gesamte Applikation sowie verschiedene Implementierungen eines Programms untersucht werden [Kalibera et al., 2005, S. 853].

Vieira et al. definieren Benchmark in Bezug auf Computersysteme wie folgt: „Benchmarks are standard tools that allow evaluating and comparing different systems or components according to specific characteristics […].“ [Vieira et al., 2012, S. 283] Benchmarks sollten ein repräsentatives Anwendungsszenarios nutzen, welches an die Realität angelehnt ist, um die reale Performanz einer Anwendung bestmöglich darzustellen [Weyuker und Vokolos, 2000, S. 1449-1150].

Performanz-Benchmarks vergleichen also verschiedene Komponenten oder Systeme in Bezug auf ihre Ausführungszeiten und anderen zeitlichen Aspekten.

2.3 Auswahl eines repräsentativen Testszenarios

Die Definition eines repräsentativen Szenarios ist komplex. Viele Entscheidungen müssen beim Design des Benchmarks getroffen werden, um eine robuste Auswertung der Performanz zu ermöglichen. Es muss erörtert und definiert werden, welche Parameter überhaupt überprüft werden sollen. Performanz nimmt aus Anwendersicht eine andere Definition ein als aus Entwicklersicht. Es können beispielsweise

die Antwortzeit oder die benötigten Hardware-Ressourcen gemessen werden. In der vorliegenden Arbeit wird insbesondere die Ausführungszeit des Programms betrachtet. Des Weiteren muss entschieden werden, ob ein durchschnittlicher Workload oder ein Peak-Workload verwendet wird. [Weyuker und Vokolos, 2000, S. 1449-1150]

Vor dem Experiment muss definiert werden, ob die durchschnittliche, die beste oder die schlechteste Ausführungszeit entscheidend für den Anwendungsfall ist. Wird entweder die beste oder schlechteste Ausführungszeit genutzt, können weitere Methoden notwendig werden, da diese Zeiten häufig über bzw. unterschätzt werden. Es können beispielsweise einzelne Teile des Programms separat gemessen werden und addiert werden, um die jeweils die theoretisch beste und schlechteste Zeit zu erhalten. Detailliertere Informationen und noch genauere Möglichkeiten zur Bestimmung dieser Zeiten kann aus dem 2500 Mal zitierten Paper von [Wilhelm et al., 2008] entnommen werden.

2.3.1 Verschiedene Einflussfaktoren auf das Ergebnis

Es gibt verschiedene Faktoren, die eine Messung unbeabsichtigt beeinflussen. Pfleeger schreibt folgendes: „The aim of a good experimental design is to control for as many variables as possible, both to minimize variability among participants and to minimize the effects of irrelevant variables.“ [Pfleeger, 1995, S. 231] Es sollten möglichst viele Parameter definiert und konstant über das Experiment bleiben, um einen möglichst geringen Einfluss auf das Ergebnis zu haben. Zu diesen Faktoren gehören im IT-Bereich beispielsweise genutzte Pakete, Hardware und Betriebssystem.

Auch der CPU-Boost und das CPU-Throttling sollten berücksichtig werden. Die Regulierung des Takts funktioniert bei modernen CPU automatisch, abhängig von der CPU-Temperatur und den zurzeit benötigten Ressourcen. Für Benchmarks ist es sinnvoll diese Eigenschaften der CPU zu deaktivieren [Huybrechts et al., 2021, S. 271-274].

2.3.2 Auswertung aus statistischer Sicht

Beim statistischen Design eines Benchmarks können Fehler gemacht werden, die einen signifikanten Einfluss auf die Ergebnisse haben. Vor der Durchführung des Experiments sollten die verwendeten Verfahren definiert werden.

Der folgende Absatz basiert auf dem Paper von [Georges et al., 2007, S. 63-64]. Um zwei Gruppen miteinander zu vergleichen, können Konfidenzintervalle berechnet. Wenn sich die Intervalle der Gruppen nicht überschneiden, unterscheiden sich die Gruppen mit hoher Wahrscheinlichkeit signifikant. Alternativ können statistisch besser geeignete Arten von t-Tests eingesetzt werden.Um mehr als zwei Gruppen miteinander zu vergleichen, kann die ANOVA (Analysis of Variance) Methode und ein F-Test verwendet werden. Die ANOVA Methode und der F-Test überprüfen, ob es zwischen mindestens zwei der gemessenen Gruppen einen signifikanten Unterschied gibt. Die Methoden bestimmen nicht zwischen welchen Gruppen es einen signifikanten Unterschied gibt. Die Voraussetzungen für die ANOVA Methode sind, dass die Varianz des Messfehlers in allen Gruppen gleich ist, dass die Fehler normalverteilt sind und dass die Messungen unabhängig voneinander sind. Falls die Anzahl der Messungen in den Gruppen identisch ist, kann die Methode auch bei anderen Verteilungen eingesetzt werden. Wenn mithilfe der ANOVA Methode ein signifikanter Unterschied festgestellt wurde, kann der Tukey HSD Test angewendet werden. Der Tukey HSD Test bestimmt, zwischen welchen der Gruppen ein signifikanter Unterschied besteht. Multiple t-Tests sind zu vermeiden, da die Fehlerwahrscheinlichkeit für jeden weiteren Vergleich stark zunimmt. Die mathematischen Formeln inkl. kurzer Erläuterung sind in Anhang 1 dargestellt.

Die Auswahl der „richtigen“ statistischen Größe zur Auswertung ist im Bereich der Performanz-Vergleiche umstritten. Smith schreibt „Reducing computer performance to a single number has become one of the more controversial (and confusing) subjects in performance evaluation.“ [Smith, 1988, S. 1202] Er argumentiert, dass das arithmetische Mittel ein sinnvoller Parameter für eine Zeitmessung darstellt. Auch [Lilja, 2000, S. 40] beschreibt, dass das arithmetische Mittel für Zeitmessungen am besten geeignet ist. [Fleming und Wallace, 1986] argumentieren in Ihrem Paper, dass Benchmark-Ergebnisse mittels des geometrischen Mittelwerts zusammengefasst werden sollen.

Insgesamt stellt das arithmetische Mittel eine geeignete Kennzahl für die Ausführungszeiten dar. Allgemein sollte bei der Auswertung von Daten aber nicht nur eine Kennzahl angegeben werden. Verschiedene Streuungsparameter können beispielsweise zu einem besseren Verständnis der Daten beitragen.

Das Paper von [Kalibera et al., 2005] zeigt auf, dass bei korrekter statistischer Auswertung Benchmarks ungenau und nicht reproduzierbar sein können. Beispiele für nicht-deterministische Eigenschaften eines Softwarebenchmarks sind die Memory-Allokation und die Kompilierung des Codes. Teilweise können Ungenauigkeiten durch weitere Maßnahmen verringert oder ausgeschlossen werden. Allerdings bleiben nicht-deterministische Einflussfaktoren bestehen.

3 Verwendete Technologien

Im Folgenden sollen die verwendeten Technologien kurz vorgestellt und die wichtigsten Faktoren in Bezug auf Performanz-Vergleiche erläutert werden. Es werden die Programmiersprachen Python und Java genutzt. Laut dem TIOBE Index ist Python im November 2021 die weltweit populärste Programmiersprache. Java folgt auf Platz drei [TIOBE Software BV, 2021]. Der TIOBE Index ist umstritten. Dennoch bietet er einen Indikator für die Popularität von Programmiersprachen.

3.1 Java

Java ist eine kompilierte Sprache. Java-Code wird von einem Kompilierer in Bytecode übersetzt und läuft auf unterschiedlichen Systemen auf der Java Virtual Machine (JVM) [Ullenboom, 2021]. Java ist statisch typisiert, welches in der Theorie eine höhere Performanz gegenüber nicht-typisierten Sprachen ergibt [Tratt, 2009, S.149].

Der Just in Time (JIT) Kompilierer und die Optimierung in der virtuellen Maschine (VM) besitzen nicht-deterministische Eigenschaften. Die Gründe für diese Eigenschaften sind unteranderem das Thread-Scheduling, die Garbage Collection und weitere Java-spezifischen Effekte [Georges et al., 2007, S. 57]. Die Wissenschaft hat Strategien entwickelt, um die Java-Perfomance stabiler messen und auszuwerten.

Der folgende Abschnitt basiert auf der Arbeit von [Georges et al., 2007, S. 57]. Sie zeigen auf und empfehlen, dass je nach Programmart und Laufzeit entweder die Startup Performanz eines Java-Programms und/oder die Steady-State Performanz gemessen und ausgewertet werden sollten. Die Startup Performanz wird bei kurzen Laufzeiten von Java-Programmen verwendet, da der Overhead durch die JIT-Kompilierung und das Class Loading einen prozentual größeren Einfluss besitzt. Bei der Startup Performanz wird pro VM-Start eine Benchmark-Iteration absolviert. Dieser Prozess wird wiederholt, um später genug Messedaten für Konfidenzintervalle oder die ANOVA Methode zu erhalten. Die Messung des ersten Benchmarks sollte aus mehreren Gründen nicht berücksichtigt werden. Bei länger laufenden Java-Programmen wird die Steady-State Performanz ausgewertet, da dort der Startup Overhead nicht häufig auftritt und prozentual weniger Einfluss auf das Endergebnis besitzt. Um die Steady-State Performanz zu messen, wird wie folgt vorgegangen: Es

gibt mehrere VM-Starts und je VM-Start gibt es mehrere Benchmark-Iterationen. Für jeden VM-Start wird der Mittelwert der multiplen Benchmark-Iterationen berechnet. Der Mittelwert wird für die Ausführungen berechnet, bei denen der Variationskoeffizient unter einen bestimmten Wert fällt (z.B. 0,01 oder 0,02). Anschließend wird ein Konfidenzintervall über alle Mittelwerte aus dem vorherigen Schritt berechnet oder die ANOVA Methode angewendet.

3.2 Python

Python ist eine dynamisch-typisierte Sprache. Bei dynamisch-typisierten Sprachen werden die Typen erst während der Laufzeit überprüft und nicht während des Kompilierens [Tratt, 2009, S. 149]. Die Perfomance von dynamisch-typisierte Sprachen ist normalerweise geringer als bei statistisch-typisierte Sprachen [Tratt, 2009, S.172]. Python besitzt verschiedene Implementierungen. Cpython ist die Standard-Implementierung. Der Python-Code wir mithilfe einer virtuellen Maschine interpretiert, aber zuvor in Byte-Code kompiliert. Die PyPy-Implementierung stellt einen JIT-Kompilierer zur Verfügung, um die Performanz zur Laufzeit zu verbessern. Neben den vorgestellten Varianten gibt es noch weitere Implementierungen wie beispielsweise Jython, MicroPython und IronPython. [Reitz, 2017, S. 5-7]

[Redondo und Ortin, 2015] haben einen Vergleich zwischen den Implementierungen vorgenommen. Sie haben verschiedene Benchmarks und Szenarien berücksichtigt und sowohl die Ausführungszeit als auch den benötigten RAM gemessen. Je nach Art des Benchmarks („Microbenchmark“, „Benchmarks“, „Programs“ und „Large Applications“) und der verschiedenen Aufgaben („Common“, „Multithreading“, „I/O“ und „Metaprogramming“) performten die verschiedenen Varianten unterschiedlich. Aufgrund der Ergebnisse sei es nicht sinnvoll eine generelle Empfehlung zu geben. Des Weiteren ist zu beachten, dass nicht alle Benchmarks für jede Implementierung ausgeführt werden konnten. Das CPython konnte als einzige Implementierung alle Benchmarks ausführen.

In Vergleichen von [Crape und Eeckhout, 2020, S. 88] wird aufgezeigt, dass die PyPy-Implementierung einen Performanzvorteil gegenüber Cpython bei der Steady-State Performanz, jedoch nicht bei der Startup Performanz hat. Des Weiteren zeigen sie auf, dass eine statistisch robuste Auswertung nach den Empfehlungen aus dem Paper von [Georges et al., 2007] auch für Python sinnvoll ist.

Die Zeitmessungen in Python sollten mithilfe der Methode *time.perf_counter()* oder *time.perf_counter_ns()* durchgeführt werden, um eine genaue Performanz-Messung zu erhalten [Python Software Foundation, o. J.]. Diese Zeitmessung orientiert sich an keiner Systemzeit, die unter verschiedenen Umständen falsche Ergebnisse produziert.

3.3 Python-Java-Bridges (Py4J und Pyjnius)

Py4J ist eine Python- und Java-Bibliothek, die es ermöglicht, mit Java Objekten aus Python heraus zu interagieren. Py4J verwendet Sockets zur Kommunikation zwischen Python und Java [Dagenais, o. J.] . Die Bibliothek unterliegt der BSD Licence und wird laut github im November 2021 von 7600 Projekten verwendet [GitHub, Inc., o. J.a]. Py4J wird in PySpark, eines der meistgenutzten Big-Data Frameworks, verwendet [Apache Spark, o. J.].

Pyjnius ist eine Bibliothek, die es ermöglicht Java-Code aus einem Python-Programm zu starten. Es nutzt das Java Native Interface (JNI), um Java-Klassen und Methoden in Python verfügbar zu machen [GitHub, Inc., o. J.b]. Theoretisch erreicht Pyjnius durch das JNI eine schnellere Verarbeitungs- und Ausführungsgeschwindigkeit als Py4J.

Neben Py4J gibt es weitere Bibliotheken mit denen Java-Programme oder Klassen aus Python-Implementierungen aufgerufen werden können. Jpype stellt beispielsweise eine Möglichkeit dar.

4 Praxisteil

Im Folgenden wird beschrieben, wie die theoretischen Kenntnisse in die Praxis überführt werden. Dazu wird die Reaktor-Simulation, die Implementierung der verschieden Varianten sowie die automatisierten Tests zur Sicherstellung der Funktionalität vorgestellt. Anschließend wird das Design des Benchmarks beschrieben und die Ergebnisse des Benchmarks ausgewertet.

4.1 Vorstellung der Simulation und Ausgangslage

Die Applikation zur Simulation eines Atomreaktors stellt die Grundlage des Projektes dar. Die ursprüngliche Version wurde von [Weyers et al., 2017] in Java implementiert. Die Simulation wurde für die Forschung im Bereich Mensch-Computer-Interaktion als Case-Study entwickelt und verwendet. Die Applikation ist mittelmäßig komplex und die Ausführungszeiten im vorliegenden Szenario sind kurz.

Diese Simulation soll in einer anschließenden Masterarbeit verwendet werden, um zu überprüfen, ob RL Agenten verschiedene Szenarien der Simulation erlernen können. Da viele RL Agenten und Frameworks in Python geschrieben sind, wird eine Python-Implementierung oder eine Python-Java-Bridge für die Simulation benötigt. Aufgabe dieses Projektes ist es eine solche Lösung bereitzustellen und herauszufinden, ob eine der Varianten performanter ist. Die Performanz ist ein wichtiger Aspekt beim RL, da der Agent oft sehr viele Simulationsdurchläufe bzw. Episoden benötigt, um eine Aufgabe zu lernen.

4.2 Implementierung der verschiedenen Versionen

Um die Python-Java-Bridge umzusetzen, wurden kleine Veränderungen am Java-Code vorgenommen. Das Frontend und das Backend sollen über Python ansprechbar sein. Daher wurden noch nicht vorhandene Getter-Methoden für das Frontend implementiert. Dazu wird die Bibliothek lombok eingesetzt, welches über einfache Annotationen Getter erzeugt.

Für die Python-Java-Bridges dient die Java-Klasse *Gateway*, welche in Abbildung 2 dargestellt ist, als Einstiegspunkt.

```java
public class Gateway {
    public NPPSystemInterface getSystemInterface(){
        return new NPPSystemInterface();
    }
    public NPPAutomation getAutomation(NPPSystemInterface systemInterface){
        return new NPPAutomation(systemInterface);
    }
}
```

Abbildung 2: Klasse „Gateway“ der Java-Implementierung

Die *Gateway* Klasse besitzt zwei Methoden, die jeweils ein neues Objekt erstellen und zurückgeben. Diese beiden Objekte (*NPPSystemInterface* und *NPPAutomation*) stellen die Hauptobjekte der Simulation dar. Alle weiteren Aktionen werden auf diesen Objekten ausgeführt.

Für die Py4J-Bridge muss in der Python-Implementierung das Packet „Py4J“ installiert werden und auf der Java-Seite muss die Py4J.jar als Bibliothek hinzugefügt werden.

```java
public static void main(String[] args) {
    Gateway gateway = new Gateway();
    GatewayServer gatewayServer = new GatewayServer(gateway);
    gatewayServer.start();
    System.out.println("Gateway Server started");
}
```

Abbildung 3: Methode „main“ der Java-Implementierung

In Abbildung 3 ist die *main* Methode der Java-Implementierung dargestellt. Die *main* Methode erstellt zuerst ein Objekt der Klasse *Gateway*. Anschließend wird ein neues Objekt des *GatewayServers*, welches aus der Py4J.jar importiert wird, erzeugt. Dieser *GatewayServer* erhält im Konstruktor das Objekt der *Gateway* Klasse, welche somit direkt durch den Python-Code angesprochen werden kann. Um den Server zu starten, wird die *start* Methode auf dem *GatewayServer* ausgeführt. In Abbildung 4 ist die Python-Implementierung der Py4J-Bridge dargestellt.

```python
def main(round_number: int) -> None:
    p = subprocess.Popen(["java", "-jar", "NPP_Simu.jar"])
    while True:
        try:
            gateway = JavaGateway(java_process=p)# connect to the JVM
            backend = gateway.entry_point.getSystemInterface()
```

Abbildung 4: Python-Implementierung für die Py4J-Bridge

Die Methode startet einen Subprozess, der die jar-Datei des Java-Programms ausführt und somit den *GatewayServer* auf der Java-Seite startet. Danach erzeugt die Methode ein *JavaGateway*, welches aus dem Py4J-Paket importiert wird. Auf

diesem Objekt wird über dem *entry_point*, die Methode *getSystemInterface* aufgerufen. Der *entry_point* bildet quasy die *Gateway* Klasse aus der Java-Implementierung. Dieser Prozess ist in einer while-Schleife und einem try-Block, da der Start des Subprozesses einige Zeit benötigt und somit die ersten Versuche sich mit dem JavaGateway zu verbinden fehlschlagen. Sobald die Verbindung hergestellt wurde, können auf dem Objekt der Klasse *SystemInterface* alle weiteren Aufrufe der Java-Methoden erfolgen. Sobald der try-Block bis zum Ende durchgelaufen ist, wird über eine break-Anweisung die Schleife verlassen.

Für die Pyjnius-Bridge muss das Paket „pyjnius“ im Python-Code importiert werden. In der Java-Implementierung benötigt die Pyjnius-Bridge keine spezifischen Imports oder Implementierungen. Wie in Abbildung 5 dargestellt, fügt die *main* Methode den Pfad der jar-Datei zu der Konfiguration hinzu. Anschließend wird mit dem Pfad zur *Gateway* Klasse innerhalb der jar-Datei eine Klasse (*gateway_class*) erzeugt und ein Objekt (*gateway_object*) erstellt. Auf diesem Objekt wird die Methode *getSystemInterface* aufgerufen, welche in der Java-Klasse *Gateway* implementiert ist. Auf dem Objekt der Klasse *SystemInterface* erfolgen alle weiteren Aufrufe von den Java-Methoden, die zur Ausführung des Programms benötigt werden.

```
def main(round_number: int) -> None:
    jnius_config.add_classpath("NPP_Simu.jar")
    from jnius import autoclass  # This import must be here
    gateway_class = autoclass("org.base.Gateway")
    gateway_object = gateway_class()
    backend = gateway_object.getSystemInterface()
```

Abbildung 5: Python-Implementierung für die Pyjnius-Bridge

Zur Portierung der Java-Simulation in Python-Code hat sich der Autor an der Empfehlung von [Khandaker, 2021] orientiert. Dieser Artikel ist keine wissenschaftliche Arbeit, bietet aber einen praxisnahen Überblick. Er beschreibt Vorrausetzungen und Methoden, um eine Portierung von Code in eine andere Programmiersprache vorzunehmen.

Der Autor besitzt aufgrund von Projekten bei verschiedenen Arbeitgebern und Projekten im Rahmen des Studiums ein breites Wissen über Python und Java. Die Architektur des Codes wurde nach detaillierter Analyse und Rückfragen an den Ersteller (Herrn Weyers) vollständig nachvollzogen. Bei der Implementierung ist der Autor schrittweise vorgegangen. Zuerst hat der Autor die Einzelteile des Reaktors erstellt, dann die komplexe *time_steps* Methode und als letztes die Klasse

NPPAutomation. Dabei wurden nicht alle Design Patterns exakt übernommen. Es wurde beispielsweise eine *FullReactor* Klasse erstellt, welche alle Komponenten eines Reaktors bündelt, um leichter mit dem gesamten Reaktor zu interagieren. Des Weiteren hat der Autor einige Methoden zur besseren Übersichtlichkeit aus der Methode *time_step* extrahiert und in weitere Klassen ausgelagert. Auf die erstellten automatisierten Tests wird im nächsten Kapitel detailliert eingegangen. Die Funktionalität der Simulation wurde vollständig nachgebildet. Der gesamte Quellcode kann der digitalen Version entnommen werden.

4.3 Automatisierte Tests

Um sicherzustellen, dass die Python-Implementierung die gleiche Funktionalität wie die Java-Implementierung aufweist, hat der Autor Unit- und Servicetests implementiert. Als Testframework dient pytest. Insgesamt überprüfen die automatisierten Tests 25 Szenarien.

Die komplexeste und meistgenutzte Methode in der gesamten Applikation heißt *time_step*. Mehrere Unittests überprüfen diese Methode. Die Unittests sind nicht separat in Python und Java implementiert, da dies zu Fehlern in der Test-Programmierung führen könnte. Stattdessen sind sowohl die Tests für die Python-Implementierung als auch die Implementierung für die Java-Version in Python implementiert. Die Tests erzeugen mithilfe der Py4J-Bibliothek die relevanten Objekte auf der Java-Seite und rufen die Methoden der Java-Version aus dem Python-Code heraus auf. Die Tests vergleichen die Ergebnisse der Java-Bridge mit den Ergebnissen der ausgeführten Python-Version. Sind beide Ergebnisse identisch, ist der Test erfolgreich. So wird sichergestellt, dass die gleichen Inputs, die gleichen Outputs in beiden Implementierungen erzeugen. Ein Beispiel eines solchen Tests ist in Anhang 2 dargestellt. Um die Unittests auszuführen, müssen die jeweiligen Methoden in der Java-Implementierung *public* sein. Andernfalls könnten die Tests die Methoden nicht von außen ansprechen. Nach dem erfolgreichen Ausführen der Tests können die Methoden wieder auf *private* gesetzt werden.

Ein Testen der Gesamtlogik stellt sich aufgrund der Thread-Logik als schwierig dar. Die Implementierungen starten verschiedene Threads, um die Berechnungen durchzuführen. Wie in Kapitel 3.1 beschrieben, kann es zu minimalen nicht-deterministischen Abweichungen in Java-Projekten bei der Laufzeit kommen. In der

vorliegenden Applikation greift ein Thread auf einen anderen Thread zu und manipuliert dabei Objekte. Sobald ein Thread minimal unterschiedliche Ausführungszeiten aufweist, können unterschiedlichen Ergebnissen entstehen. In Anhang 3 wird dieses Phänomen anhand eines Beispiels detailliert erläutert. Dadurch sind automatisierte Vergleiche zwischen verschiedenen Implementierungen, die einen minimal anderen Overhead haben, nicht sinnvoll. Schon beim mehrmaligen Ausführen der Servicetests für die *NPPAutomation* Klasse entstehen Ungenauigkeiten, welche den Test fehlschlagen lassen. Durch wiederholtes Ausführen der Tests, bestehen sie erfolgreich.

Insgesamt erreichen die automatisierten Tests eine Abdeckung von 91 % der Services und 96 % der dto-Klassen bezogen auf die Python-Implementierung. Die Werte für die Testabdeckung je Klasse können aus dem Anhang 4 entnommen werden oder durch Ausführung der Tests inkl. Testabdeckung selbst bestimmt werden. Um den fehlenden Ende-zu-Ende Test zu ersetzen, hat der Autor viele manuelle Vorgänge getestet und analysiert. Dabei kam es zu kleineren Abweichungen. Grundsätzliche Fehler, die nicht auf die Thread-Logik und die minimalen Abweichungen zurückzuführen sind, konnte der Autor nicht identifizieren.

4.4 Design des Benchmarks

Im Folgenden werden die fest definierten Parameter des Experiments dargestellt.

- Python Version 3.8.10 mit CPython
- Py4J Version 0.10.9.2
- Pyjnius Version 1.4.1
- Java OpenJDK 11
- Ubuntu 20.04.3 LTS
- AMD Ryzen 5 3600, 32 GB Ram, Geforce GTX 1060 6GB
- 3 Gruppen mit je 100 Testdurchläufen

Die CPython-Variante wurde genutzt, da hier sichergestellt ist, dass alle weiteren Frameworks und Bibliotheken einwandfrei verwendet werden können. Außerdem haben [Crape und Eeckhout, 2020 aufgezeigt, dass die Startup Performanz nahezu identisch zwischen PyPy und Cpython ist. Der CPU-Boost wurde für den Benchmark deaktiviert. Das CPU-Throttling tritt bei zu hohen Temperaturen auf. Daher werden die CPU-Temperatur und die Throttling-Parameter ständig gemonitort. Die Zeit wird

mit der Methode *time.perf_counter()* gemessen, um die reale Ausführungszeit unabhängig von der Systemzeit zu erhalten. Die Python-Implementierung speichert die einzelnen Messungen in eine csv-Datei. Es soll die Startup Performanz gemessen werden, da beim RL-Training diese Zeit entscheidend ist und die Benchmarkiterationen eine kurze Ausführungszeit besitzen.

Bei der Auswertung der Daten wird das arithmetische Mittel verwendet, aber es werden auch weitere Daten, wie die Standardabweichung berechnet, um ein umfassendes Ergebnis zu erhalten. Das Signifikanzniveau α soll 0.01 betragen. Die beste und die schlechteste Zeit sind im vorliegenden Szenario nicht entscheidend. Ein späterer RL Agent benötigt viele Iteration bzw. Episoden und dabei ist es nicht entscheidend, ob eine einzelne Ausführung einen Ausreißer darstellt. Zur Auswertung wird die ANOVA Methode und die Tukey HSD Methode angewendet.

Der Autor stell die folgenden Hypothesen auf:
Nullhypothese: Bei den vorliegenden Implementierungen gibt es keine Unterschiede in der Ausführungszeit.
Alternativhypothese: Bei den vorliegenden Implementierungen ist die Ausführungszeit bei mindestens einem Implementierungspaar signifikant unterschiedlich.

Eine Iteration des Benchmarks stellt das Starten des gesamten Reaktors dar, ohne dass Teile des Reaktors kaputtgehen. Dazu werden verschiedene Parameter des Reaktors gesetzt (119 Aktionen) und die Veränderung des Zustands des Reaktors ausgegeben (1155 Aktionen). Die Simulation enthält 0,5 Sekunden „sleep“-Zeiten für jedes Öffnen und Schließen eines Ventils. Während einer „sleep“-Zeit wartet der Thread und führt keine weiteren Aktionen aus. Im vorliegenden Benchmark gibt es vier Interaktionen mit Ventilen. Die „sleep“-Zeiten sollen im Performanz-Vergleich nicht berücksichtigt werden. Deshalb werden bei der Auswertung jeweils zwei (0,5*4) Sekunden von den gemessenen Zeiten abgezogen.

4.5 Vorstellung Ergebnisse und Auswertung des Benchmarks

Die statistischen Tests zur Auswertung wurden nicht selbst in Python implementiert, da es bestehende Pakete gibt, die ausführlich getestet sind. Die Wahrscheinlichkeit, dass ein Fehler auftritt, wäre bei einer Selbstimplementierung deutlich höher als bei den verwendeten Paketen. Es wurden die Python-Pakete jupyter, matplotlib, scipy und statsmodel verwendet, um geeignete Graphen zu erstellen und die statistischen

Auswertungen vorzunehmen. Die Implementierung für die Analyse und die vollständigen Ergebnisse können der digitalen Version aus dem „analytics“ Ordner entnommen werden.

In der folgenden Auswertung wird sich auf die Ergebnisse des PCs mit dem AMD Ryzen bezogen, da ein späteres Training eines RL Agenten auf einem Server oder PC stattfinden wird. Die Ergebnisse des Raspberry Pis können dem Anhang 5 entnommen werden. Die Ausführungszeiten ohne die „sleep“-Zeiten sind beim Raspberry Pi ungefähr sieben- bis zehnmal länger.

Von allen angegebenen Zeitwerten sind bereits die zwei Sekunden „sleep“-Zeit abgezogen. In der Tabelle 1 sind die Werte für die drei Implementierungen dargestellt. Die native Python Implementierung ist, wie zu erwarten, deutlich schneller als die beiden Python-Java-Bridges. Py4J ist im arthritischen Mittel die langsamste Variante.

	Python	Py4J	Pyjnius
Beste Zeit	0.0057	0.2658	0.2330
Schlechteste Zeit	0.0111	0.3312	0.2930
Arithmetisches Mittel	0.0080	0.2836	0.2586
Standardabweichung	0.0011	0.0105	0.0108

Tabelle 1: Ergebnisse des Benchmarks in Sekunden

In Abbildung 6 und Abbildung 7 sind die Boxplots der Messungen dargestellt. Dabei geben die Y-Achsen die Ausführungszeit in Sekunden an.

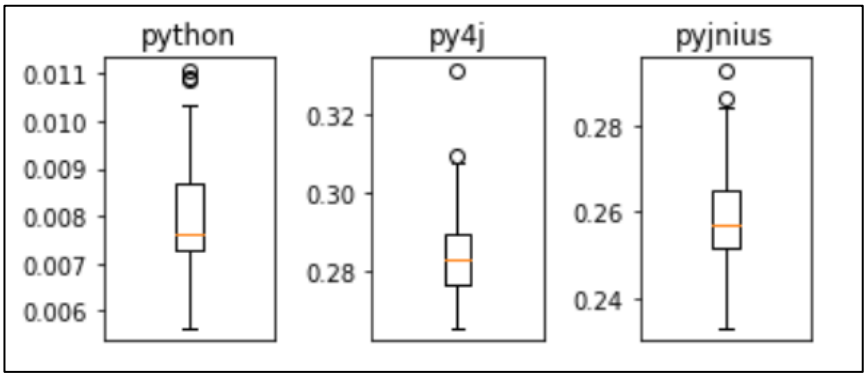

Abbildung 6: Einzelne Boxplots der Messungen (eigene Darstellung)

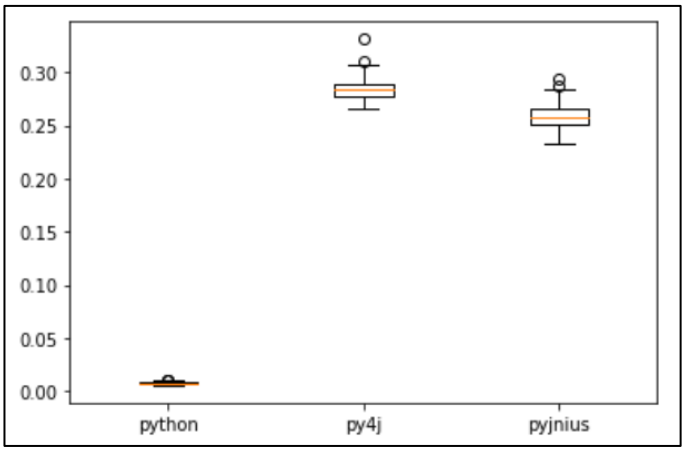

Abbildung 7: Boxplots der Messungen in einem Graph (eigene Darstellung)

Die drei Boxplots zeigen bereits, dass es vermutlich einen Unterschied zwischen der Python-Version und den Bridges gibt. Mit der ANOVA Methode und dem F-Test konnte bestätigt werden, dass es zwischen den Gruppen einen signifikanten Unterschied gibt ($F_{berechnet} = 30188\, und\, F_{Tabelle} = 4{,}678$). Damit kann die Nullhypothese abgelehnt werden. Um festzustellen zwischen welchen Gruppen es einen signifikanten Unterschied gibt, wurde der Tukey HSD Test durchgeführt. Das Ergebnis mit $\alpha = 0{,}01$ zeigt auf, dass zwischen allen drei Kombinationen ein signifikanter Unterschied besteht.

Für die vorliegende Reaktor-Simulation und das vorgestellte Szenario (Starten eines Reaktors) ist die Py4J-Bridge langsamer als die Pyjnius-Bridge. Die Pyjnius-Bridge ist signifikant langsamer als die native Python-Implementierung. Diese Ergebnisse spiegeln die Erwartungen wider. Durch die Sockets ist die Py4J-Implementierung langsamer als die Pyjnius-Bridge, die das Java Native Interface einsetzt. Bei den Ergebnissen ist zu beachten, dass es viele Interaktionen zwischen Python und Java gibt und relativ kurze Laufzeiten der nativen Java-Logik. In diesem Szenario ist die native Python-Implementierung deutlich schneller, da der größte Overhead bei den Bridges durch die Kommunikation entsteht.

5 Zusammenfassung und Fazit

In der vorliegenden Arbeit sollte die Java-Simulation in Python portiert werden und ein Performanz-Vergleich zwischen Python-Java-Bridges und der nativen Python-Implementierung vorgenommen werden. Dazu wurde die Java-Implementierung in Python übersetzt. Um sicherzustellen, dass die Funktionalität der beiden Implementierungen identisch ist, wurden insgesamt 25 Testszenarien implementiert. Die Tests überprüfen, ob die Ergebnisse der Java-Applikation und der Python-Implementierung identisch sind. Um eine statistisch robuste Auswertung vorzunehmen, wurde die ANOVA Methode und der Tukey HSD Test vorgestellt und angewendet. Es wurde ein Benchmark entwickelt, um die verschiedenen Implementierungen zu vergleichen. Die Ergebnisse des Benchmarks zeigen, dass die Python-Implementierung im vorgestellten Szenario signifikant schneller als die beiden Bridges ist. Die Pyjnius-Bridge ist signifikant schneller als die Py4J-Bridge.

In der vorliegenden Simulation gibt es viele „sleep“-Zeiten, welche die Ausführungszeit enorm erhöhen. Auch wenn die Unterschiede signifikant sind, ist die prozentuale Erhöhung durch die Verwendung von Bridges im vorliegenden Fall gering. Deshalb können Bridges im vorliegenden Szenario sinnvoll eingesetzt werden. Auch zur Überprüfung, ob die Funktionalität zwischen verschiedenen Implementierungen identisch ist, können Bridges eingesetzt werden. Diese Möglichkeit wurde mit den automatisierten Tests in der vorliegenden Arbeit vorgestellt.

Dennoch eignet sich für die anstehende Masterarbeit im Bereich des RLs die Python-Implementierung besser, da dort leichter Modifikationen vorgenommen werden können. Des Weiteren kann das gesamte Projekt dadurch in einer Sprache erstellt werden, da die benötigten Frameworks auch in Python implementiert sind. Weitere Untersuchungen könnten aufzeigen, wie zuverlässig und performant Bridges beim realen Training mit RL Agenten sind. Des Weiteren könnte beispielsweise überprüft werden, ob weniger Interaktion zwischen der Python- und Java-Implementierung zu einem geringeren Overhead der Bridges führt. In der Theorie könnte eine Python-Java-Bridge aufgrund der kompilierten Sprache und der statischen Typen schneller als eine reine Python-Implementierung sein. Um eine übertragbare Aussage über die Performanz zu erhalten könnten standardisierte Algorithmen, wie Such- und Sortierverfahren, gebenchmarkt werden.

Literaturverzeichnis

[Abdi und Williams, 2021] Abdi, H., & Williams, L. (2021) Tukey’s Honestly Signiflcant Difi-erence (HSD) Test.

[Apache Spark, o. J.] Apache Spark (o. J.) Debugging PySpark. Apache Foundation. http://spark.apache.org/docs/latest/api/python/development/debugging.html?high-light=py4j, zuletzt abgerufen am 11 November 2021.

[Cohn, 2010] Cohn, M. (2010). Succeeding with agile: Software development using Scrum (The Addison-Wesley signature series A Mike Cohn signature book). Upper Saddle River, NJ, Munich: Addison-Wesley.

[Crape und Eeckhout, 2020] Crape, A., & Eeckhout, L. (2020) A Rigorous Benchmar-king and Performance Analysis Methodology for Python Workloads. In 2020 IEEE International Symposium on Workload Characterization (IISWC), Beijing, China, 27.10.2020 - 30.10.2020, S. 83–93. IEEE. doi:10.1109/IISWC50251.2020.00017.

[Dagenais, o. J.] Dagenais, B. (o. J.) Py4J - A Bridge between Python and Java: Alter-natives to Py4J. https://www.py4j.org/about.html, zuletzt abgerufen am 11 Novem-ber 2021.

[Fleming und Wallace, 1986] Fleming, P. J., & Wallace, J. J. (1986) How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM, 29, S. 218–221. doi:10.1145/5666.5673.

[Fowler, 2000] Fowler, M. (Ed.) (2000). Refactoring: Wie Sie das Design vorhandener Soft-ware verbessern (Professionelle Softwareentwicklung). München: Addison Wesley.

[Georges et al., 2007] Georges, A., Buytaert, D., & Eeckhout, L. (2007) Statistically ri-gorous java performance evaluation. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, & G. L. Steele (Eds.), the 22nd annual ACM SIGPLAN conference, Montreal, Quebec, Canada, 21.10.2007 - 25.10.2007, S. 57–76. New York, New York, USA: ACM Press. doi:10.1145/1297027.1297033.

[GitHub, Inc., o. J.a] (o. J.) bartdag / py4j. GitHub, Inc. https://github.com/bartdag/py4j, zu-letzt abgerufen am 11 November 2021.

[GitHub, Inc., o. J.b] (o. J.) kivy / pyjnius. GitHub, Inc. https://github.com/kivy/pyjnius, zuletzt abgerufen am 11 November 2021.

[Huybrechts et al., 2021] Huybrechts, T., Mercelis, S., & Hellinckx, P. (2021) A Survey on the Software and Hardware-Based Influences on the Worst-Case Execution Ti-me. In L. Barolli, M. Takizawa, T. Yoshihisa, F. Amato, & M. Ikeda (Eds.), Advances on P2P, Parallel,

Grid, Cloud and Internet Computing, S. 271–281, Vol. 158, Lecture Notes in Networks and Systems. Cham: Springer International Publishing.

[IEEE, 1990] IEEE Standard Glossary of Software Engineering Terminology. Piscataway, NJ, USA: IEEE. doi:10.1109/IEEESTD.1990.101064.

[Kalibera et al., 2005] Kalibera, T., Bulej, L., & Tuma, P. (2005) Benchmark precision and random initial state. In SPECTS 2005, S. 853–862. SCS.

[Khandaker, 2021] Khandaker, S. (2021) Converting Code From One Programming Language to Another: Tips for Translating and Porting Code. https://medium.com/swlh/converting-code-from-one-programming-language-to-another-fe4b2ad8ed43, zuletzt abgerufen am 11 November 2021.

[Labanda-Jaramillo et al., 2018] Labanda-Jaramillo, M., Quinche, J. G., Chamba-Eras, L., Coronel-Romero, E., Granda, J.-L., & Zhingre, L. E. C. (2018) Empirical Study Between Compiled, Interpreted, and Dynamic Programming Languages Applying Stable Ordering Algorithms (Case Study: Java, Python, Jython, Jpype and Py4J). KnE Engineering, 3, S. 122. doi:10.18502/keg.v3i9.3650.

[Lilja, 2000] Lilja, D. J. (2000). Measuring computer performance: A practitioner's guide (1st ed.). Cambridge: Univ. Press.

[Pfleeger, 1995] Pfleeger, S. L. (1995) Experimental design and analysis in software engineering. Annals of Software Engineering, 1, S. 219–253. doi:10.1007/BF02249052.

[Pittet, o. J.] Pittet, S. (o. J.) The different types of software testing: Compare different types of software testing, such as unit testing, integration testing, functional testing, acceptance testing, and more! https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing, zuletzt abgerufen am 11 No-vember 2021.

[Python Software Foundation, o. J.] Python Software Foundation (o. J.) Time access and conversions: time.perf_counter. https://docs.python.org/3/library/time.html#time.perf_counter, zuletzt abgerufen am 11 November 2021.

[Redondo und Ortin, 2015] Redondo, J. M., & Ortin, F. (2015) A Comprehensive Evalu-ation of Common Python Implementations. IEEE Software, 32, S. 76–84. doi:10.1109/MS.2014.104.

[Reitz, 2017] Reitz, K. (2017). Hitchhiker's Guide für Python: Best Practices für Pro-grammierer (Animals). Heidelberg: o'Reilly.

[Smith, 1988] Smith, J. E. (1988) Characterizing computer performance with a single number. Communications of the ACM, 31, S. 1202–1206. doi:10.1145/63039.63043.

[Smith und Williams, 2004] Smith, C. U., & Williams, L. G. (2004) Software Performance Engineering. In L. Lavagno, G. Martin, & B. Selic (Eds.), UML for Real, S. 343–365. Boston: Kluwer Academic Publishers.

[TIOBE Software BV, 2021] TIOBE Software BV (2021) TIOBE Index for November 2021: November Headline: PHP about to lose its top 10 position. https://www.tiobe.com/tiobe-index/, zuletzt abgerufen am 11 November 2021.

[Tratt, 2009] Tratt, L. (2009) Chapter 5 Dynamically Typed Languages. In S. 149–184, Vol. 77, Advances in Computers. Elsevier.

[Ullenboom, 2021] Ullenboom, C. (2021). Java ist auch eine Insel: Das Standardwerk für Programmierer. Über 1.000 Seiten Java-Wissen. Mit vielen Beispielen und Übungen, aktuell zu Java 17 (16th ed.). Bonn: Rheinwerk.

[Vieira et al., 2012] Vieira, M., Madeira, H., Sachs, K., & Kounev, S. (2012) Resilience Benchmarking. In K. Wolter, A. Avritzer, M. Vieira, & A. van Moorsel (Eds.), Resi-lience Assessment and Evaluation of Computing Systems, S. 283–301. Berlin, Hei-delberg: Springer Berlin Heidelberg.

[Vocke, 2018] Vocke, H. (2018) The Practical Test Pyramid. https://martinfowler.com/articles/practical-test-pyramid.html, zuletzt abgerufen am 11 November 2021.

[Weyers et al., 2017] Weyers, B., Harrison, M. D., Bowen, J., Dix, A., & Palanque, P. (2017) Case Studies. In B. Weyers, J. Bowen, A. Dix, & P. Palanque (Eds.), The Handbook of Formal Methods in Human-Computer Interaction, S. 89–121, Human–Computer Interaction Series. Cham: Springer International Publishing.

[Weyuker und Vokolos, 2000] Weyuker, E. J., & Vokolos, F. I. (2000) Experience with performance testing of software systems: issues, an approach, and case study. IE-EE Transactions on Software Engineering, 26, S. 1147–1156. doi:10.1109/32.888628.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., et al. (2008) The worst-case execution-time problem—overview of me-thods and survey of tools. ACM Transactions on Embedded Computing Systems, 7, S. 1–53. doi:10.1145/1347375.134738

Verwendete Open Source Pakete und Bibliotheken:

jupyter: Kluyver, Thomas; Ragan-Kelley, Benjamin; Pérez, Fernando; Granger, Brian; Bussonnier, Matthias; Frederic, Jonathan et al. (2016): Jupyter Notebooks - a publishing format for reproducible computational workflows. In: Fernando Loizides und Birgit Scmidt (Hg.):

Positioning and Power in Academic Publishing: Players, Agents and Agendas. Netherlands: IOS Press, S. 87–90. Online verfügbar unter https://eprints.soton.ac.uk/403913/.

lombok: Project Lombok (o. J.). Online verfügbar unter https://projectlombok.org/.

matplotlib: Hunter, J. D. (2007): Matplotlib: A 2D graphics environment. In: Computing in Science & Engineering 9 (3), S. 90–95. DOI: 10.1109/MCSE.2007.55.

numpy: Harris, Charles R.; Millman, K. Jarrod; van der Walt, Stéfan J.; Gommers, Ralf; Virtanen, Pauli; Cournapeau, David et al. (2020): Array programming with NumPy. In: Nature 585 (7825), S. 357–362. DOI: 10.1038/s41586-020-2649-2.

py4j: Dagenais, Barthelemy (o. J.): py4j. Version 0.10.9.2. Online verfügbar unter https://github.com/bartdag/py4j.

pyjnius: pyjnius (o. J.). Version 1.4.1: Kivy. Online verfügbar unter https://github.com/kivy/pyjnius.

pytest: Krekel, Holger; Oliveira, Bruno; Pfannschmidt, Ronny; Bruynooghe, Floris; Laugher, Brianna; Bruhin, Florian (2004): pytest 6.2.5. Online verfügbar unter https://github.com/pytest-dev/pytest.

scipy: Virtanen, Pauli; Gommers, Ralf; Oliphant, Travis E.; Haberland, Matt; Reddy, Tyler; Cournapeau, David et al. (2020): SciPy 1.0: Fundamental Algorithms for Scien-tific Computing in Python. In: Nature Methods 17, S. 261–272. DOI: 10.1038/s41592-019-0686-2.

statsmodel: Seabold, Skipper; Perktold, Josef (2010): statsmodels: Econometric and statistical modeling with python. In: 9th Python in Science Conference.

Anhangsverzeichnis

1. ANOVA und Tukey HSD Berechnung

Im Folgenden werden die mathematischen Formeln für die ANOVA Methode und den Tukey HSD Test vorgestellt und erklärt. Die ANOVA Methode wird basierend auf dem Buch von [Lilja, 2000, S.71-77] vorgestellt. Gegeben sei die folgende Tabelle. Es gibt $n * k$ Messpunkte (n Messungen für jede Gruppe und k Gruppen).

Messungen	**Gruppen**						**Gesamtmittelwert**
	1	2	…	j	…	k	
1	y_{11}	y_{12}		y_{1j}		y_{1k}	
2	y_{21}	y_{22}		y_{2j}		y_{2k}	
…							
$\boldsymbol{i}$	y_{i1}	y_{i2}		y_{ij}		y_{ik}	
…							
$\boldsymbol{n}$	y_{n1}	y_{n2}		y_{nj}		y_{nk}	
Gruppenmittelwert	$\bar{\boldsymbol{y}}_{\boldsymbol{.1}}$	$\bar{\boldsymbol{y}}_{\boldsymbol{.2}}$	**…**	$\bar{\boldsymbol{y}}_{\boldsymbol{.j}}$	**..**	$\bar{\boldsymbol{y}}_{\boldsymbol{.k}}$	$\bar{\boldsymbol{y}}_{\boldsymbol{..}}$

Tabelle 1: Tabellenübersicht für die ANOVA Methode angelehnt an [Lilja, 2000, S.72]

Die Gruppenmittelwerte werden wie folgt bestimmt:

$$\bar{\boldsymbol{y}}_{\boldsymbol{.j}} = \frac{\sum_{i=1}^{n} y_{ij}}{n}$$

Der Gesamtmittelwert wird wie folgt bestimmt:

$$\bar{\boldsymbol{y}}_{\boldsymbol{..}} = \frac{\sum_{j=1}^{k} \sum_{i=1}^{n} y_{ij}}{n*\boldsymbol{k}}$$

SSA stellt die Summe der Quadrate zwischen der Differenz der verschiedenen Alternativen und dem Gesamtmittelwert dar.

$$SSA = n \sum_{j=1}^{k} (\bar{y}_{.j} - \bar{y}_{..})^2$$

SSE stellt die Variation durch zufällige Effekte innerhalb einer Alternative da und wird durch der die Summe der Quadrate zwischen der einzelnen Messung und dem Mittelwert derselben Gruppe bestimmt.

$$SSE = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \bar{y}_{.j})^2$$

SST stellt die Gesamtsumme der Quadrate dar. Es kann aufgezeigt werden, dass SSA und SSE zusammen SST ergeben.

$$SST = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \bar{y}_{..})^2 \text{ und } SST = SSA + SSE$$

Nachdem SSA und SSE bestimmt sind, wird ein F-Test durchgeführt, der aufzeigt, ob zwei Varianzwerte sich signifikant unterscheiden. Es wird die Nullhypothese aufgestellt, dass die Varianzen der Grundgesamtheiten, aus denen die Stichproben stammen gleich sind. Die Alternativhypothese ist, dass die Varianzen der Grundgesamtheiten verschieden sind.

$$s_a^2 = \frac{SSA}{k-1} \; und \; s_e^2 = \frac{SSE}{k(n-1)}$$

Für den F-Wert gilt folgende Formel mit α als Signifikanzniveau (z.B. 0.05). Der Vergleichswert wird aus einer F-Verteilungstabelle entnommen.

$$F_{berechnet} = \frac{s_a^2}{s_e^2} \text{ und } F_{Tabelle} = F_{[\alpha;(k-1),k(n-1)]}$$

Ist $F_{berechnet} > F_{Tabelle}$ dann wird die Nullhypothese abgelehnt.

Mithilfe der ANOVA Methode und dem F-Test wird nicht festgestellt zwischen welchen Gruppen es einen signifikanten Unterschied gibt. Dazu kann beispielsweise der Tukey HSD Test verwendet werden. Die Formeln können beispielsweise aus [Abdi und Williams, 2021] entnommen werden. Dieser Test wird bei mehreren Gruppen bevorzugt gegenüber dem paarweisen t-test eingesetzt, da bei einem t-test mit mehreren Gruppen die Fehlerwahrscheinlichkeit stark ansteigt. Der Tukey HSD ist wesentlich stabiler. Der HSD- Wert wird wie folgt berechnet, falls die Anzahl der Messungen n in jeder Gruppe identisch ist. q wird aus einer Tabelle entnommen.

$$HSD = q_{[\alpha;(k),(n-k)]} \sqrt{\frac{s_e^2}{n}}$$

Anschließend wird für jede Kombination der einzelnen Gruppen die Differenz der jeweiligen Mittelwerte mit dem HSD verglichen. Ist die Differenz der Mittelwerte gleich oder größer dem HSD-Wert dann besteht ein Unterschied zwischen den beiden ausgewählten Gruppen für das gewählte Signifikanzniveau.

$$HSD \leq |\bar{y}_{.j} - \bar{y}_{.j'}|$$

2. Automatisiert Test

In Abbildung 6 ist ein Test für die Methode *time_step* bzw. *timeStep* dargestellt. Im vorliegenden Fall werden Parameter überprüft, während der Reaktor überhitzt.

```
@pytest.mark.parametrize("time_steps", [2, 10])
def test_expected_reactor_status_fail(time_steps) -> None:
    """
    This method tests if the time_step methods are equal across both
    implementations. This scenario lets the reactor fail.
    """
    full_reactor = ReactorCreatorService.create_standard_full_reactor()
    background_step_service = BackgroundStepService(full_reactor)
    background_step_service.full_reactor.reactor.moderator_percent = 50
    background_step_service.time_step(time_steps)

    p = subprocess.Popen(
        [
            "java",
            "-jar",
            "NPP_Simu.jar",
        ]
    )
    sleep(1)
    gateway = JavaGateway()  # connect to the JVM
    system_interface = gateway.entry_point
    backend = system_interface.getSystemInterface()
    backend.initSimulation()
    backend.setReactorModeratorPosition(50)
    backend.timeStep(time_steps)
    assert backend.getWaterLevelReactor() ==
           int(background_step_service.full_reactor.reactor.water_level)
    assert backend.getPressureReactor() ==
           int(background_step_service.full_reactor.reactor.pressure)
    assert backend.getPowerOutlet() ==
           background_step_service.full_reactor.generator.power
    assert backend.getReactorStatus() ==
           operator.not_(
           background_step_service
           .full_reactor
           .reactor
           .overheated)
    gateway.shutdown()
    p.kill()
```

Abbildung 1: Implementierung eines automatisierten Tests für die time_stamp Methode

Um aus diesem Test auf die Java-Applikation zuzugreifen, wird ein Subprozess gestartet, welcher die jar-Datei der Java-Implementierung startet. Py4J wird genutzt, um mit diesem Prozess zu interagieren. Nach dem Starten der Py4J-Bridge wird ein Reaktor erzeugt und jeweils die *moderator_percent* und die Reactor*ModeratorPosition* auf 50 gesetzt. Anschließend werden die zu testenden Methoden *time_step* und *timeStep* ausgeführt. Nachdem beide Methoden erfolgreich durchlaufen worden sind, werden die relevanten Parameter der beiden Implemen-

tierungen miteinander verglichen. Es ist zu beachten, dass sowohl das Wasserlevel als auch der Druck des Reaktors auf der Python-Seite in einen Integer gecastet werden. In der Java-Version wird nur ein Getter bereitstellt, der dieses Umwandeln in einen Integer bereits vornimmt. Um dieselben Ergebnisse zu erhalten, werden die Ergebnisse zum Vergleichen gecastet. Alternativ gäbe es die Möglichkeit weitere Getter zu implementieren. Des Weiteren wird der Reaktor-Status verglichen. In der Java-Version gibt der Getter einfach die Negierung der *overheated* Variablen des Reaktors zurück. Auch dieser spezielle Getter wurde auf der Python-Seite nicht implementiert, da einfach auf den *overheated* Parameter zugegriffen werden kann und dieser negiert werden kann.

Wenn die Ergebnisse identisch sind, ist der Test erfolgreich und beide Varianten haben das gleiche Ergebnis für den vorliegenden Fall berechnet.
Die *pytest*-Annotation am Anfang wird genutzt, um den Test zweimal mit verschiedenen Parametern auszuführen. Im vorliegenden Fall werden die *time_step* und die *timeStep* Methode mit einmal 2 Zeitschritten und einmal 10 Zeitrschritten ausgeführt.

Um unterschiedliche Szenarien zu testen, wurden ähnliche Testfälle programmiert. Die weiteren Tests können der digitalen Version entnommen werden.

3. Probleme nicht-deterministischer Laufzeiten

In diesem Anhang wird beschrieben, wieso nicht-deterministische Java-Programme und Laufzeiten ein Problem in der vorliegenden Simulation besitzen können. Durch dieses Phänomen können keine sinnvollen automatisierten Ende-zu-Ende-Tests erstellt werden. In Abbildung 2 ist die Ausgangslage und das Problem bildlich dargestellt.

Thread A besitzt ein Objekt *X* und eine *run* Methode. Thread B besitzt die Methode *manipuliereX,* welche das Objekt *X* aus Thread A manipuliert. Der Thread A startet innerhalb seiner *run* Methode auch den Thread B und die Methode *manipuliereX*. Die Methode *manipuliereX* läuft die gesamte Zeit im Hintergrund und manipuliert so lange das Objekt *X* bis die *run* Methode aus Thread A beendet wird und damit auch der Thread B beendet wird. Hierbei können zwei Probleme auftreten. Sollte die *run* Methode durch die nicht-deterministischen Eigenschaften eine andere Ausführungszeit besitzten, kann es passieren, dass dadurch die Methode *manipuliereX* häufiger oder weniger häufig ausgeführt wird. Dieses Szenario ist durch die roten Linien dargestellt. Einmal wird die Methode *manipuliereX* dreimal aufgerufen und einmal nur zweimal. Genauso kann die Methode *manipuliereX* häufiger oder weniger häufig ausgeführt werden, da die Ausführungszeit der Methode selbst unterschiedlich lang sein kann. Somit hat die Methode *manipuliereX* nicht immer die identische Anzahl an Aufrufen. Das wiederum kann dazu führen, dass bei gleichem Input nicht garantiert werden kann, dass Objekt *X* immer identisch ist. Somit können nicht-deterministische Laufzeiten auch die Programmlogik und das Ergebnis selbst beeinflussen.

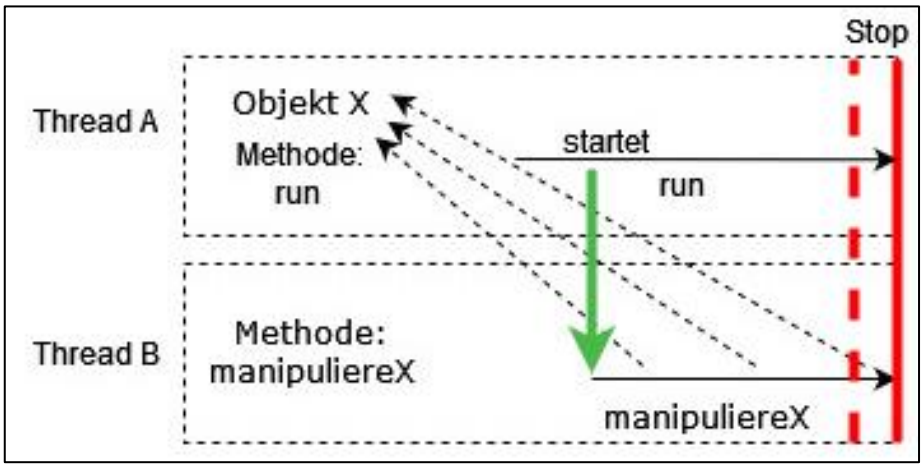

Abbildung 2: Beispielprogramm zur Erklärung von Problemen bei nicht-deterministischen Laufzeiten (eigene Darstellung)

4. Testabdeckung der Zeilen je Klasse

Die Abbildung zeigt die Testabdeckung durch die automatisierten Tests je Klasse. Die Prozentzahl bezieht sich auf die Abdeckung der einzelnen Zeilen der Python-Implementierung. Diese Zahlen können selbst ermittelt werden, indem die Tests inkl. Testabdeckung des Python-Projekts ausgeführt werden. Insgesamt konnte eine hohe Testabdeckung von über 90 % erreicht werden.

Klasse	**Zeilenabdeckung**
BackgroundStepService.py	91 %
CalculatorService.py	93 %
NPPAutomationService.py	96 %
ReactorCreatorService.py	100 %
SystemInterfaceService.py	72 %
Utils.py	100 %
Condenser.py	100 %
FullReactor.py	96 %
Generator.py	92 %
Pump.py	93 %
Reactor.py	96 %
ReactorComponent.py	90 %
SteamValve.py	100 %
Tank.py	/
Turbine.py	100 %
WaterValve.py	100 %

Tabelle 2: Testabdeckung je Zeile pro Klasse

5. Raspberry Pi Ergebnisse

Die Tests wurden auch einem Raspberry Pi 4 Model B mit 4 GB Ram und dem Raspberry Pi OS Version 10.9 inkl. Python 3.7.3 ausgeführt. Alle anderen Parameter waren identisch zum Benchmark auf dem AMD Ryzen. Die Ergebnisse des Raspbery Pis sind ca. 7-10 mal langsamer als auf dem PC (ohne „sleep“-Zeiten).

Von den dargestellten Werten sind bereits die „sleep“-Zeiten von zwei Sekunden abgezogen.

	Python	Py4J	Pyjnius
Beste Zeit	0.0250	1.9856	1.3393
Schlechteste Zeit	0.1068	2.2464	1.4542
Arithmetisches Mittel	0.0648	2.1016	1.4045
Standardabweichung	0.0116	0.0577	0.0198

Tabelle 3: Ausführungszeiten auf dem Raspberry Pi

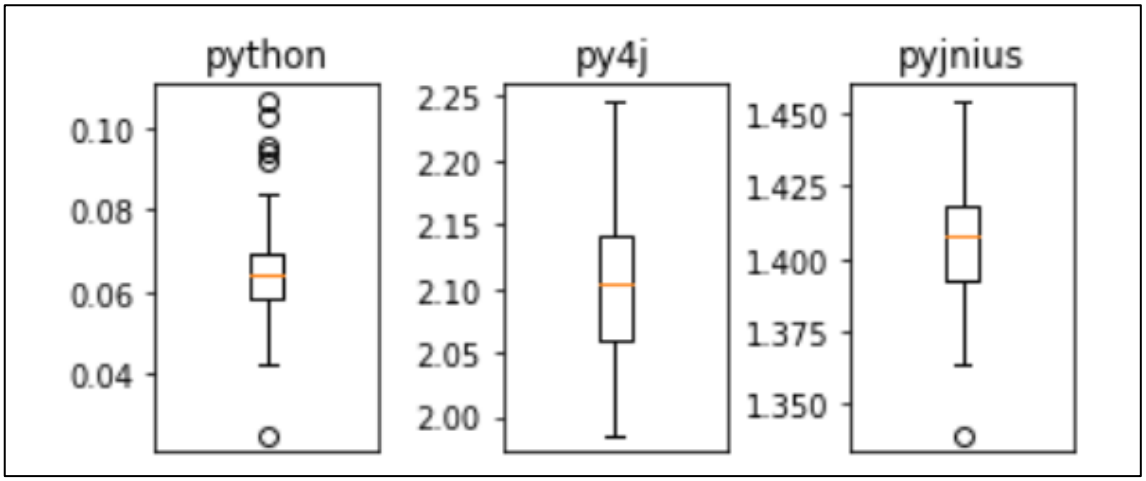

Abbildung 3: Implementierung eines automatisierten Tests für die *time_stamp* Methode