Evaluation einer grafischen Entwicklungsumgebung für interaktive Physikanalysen

Thomas Münzer
RWTH-Aachen University

21. Januar 2010

Diplomarbeit

Erstgutachter:
Prof. Dr. Matthias Jarke.
Lehrstuhl Informationssysteme und Datenbanken.
Informatik 5.

Zweitgutachter:
Prof. Dr. Martin Erdmann.
III. Physikalisches Institut A.

Erklärung

Hiermit erkläre ich, dass ich ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

_____ Aachen, den 21.01.2009
Thomas Münzer

Danksagung

Hiermit möchte ich mich ganz herzlich bei allen Mitgliedern des VISPA-Teams für die große Unterstützung und die Geduld bedanken.
Besonders möchte ich mich bei Prof. Martin Erdmann bedanken, der mir die Gelegenheit gegeben hat, meine Diplomarbeit im Bereich der Informatik mit einem Themenbereich der Physik zu verbinden. Außerdem gilt mein Dank Prof. Matthias Jarke, der sich bereit erklärt hat, die Diplomarbeit als Erstgutachter aus dem Bereich der Informatik zu betreuen.

Inhaltsverzeichnis

1 Einführung

Das größte und aufwendigste Experiment, das es zur Zeit gibt, ist der Large Hadron Collider (LHC) bei Cern. Die Ausmaße dieses Teilchenbeschleunigers sind riesig. Er ist ca. 27 Km lang und befindet sich ein bis zu 175 Meter tiefen Tunnel. Ebenso groß ist auch der technische Aufwand, der betrieben werden muss. Die Magnete, die die Teilchen auf ihren Bahnen halten müssen nahe an den absoluten Nullpunkt herunter gekühlt werden. Die Temperatur der Magnete liegt bei ca. 1,9 Grad Kelvin. Auch die Detektoren für die Teilchenkollisionen habe riesige Dimensionen. Beispielsweise der CMS-Detektor. Er ist ca. 21 Meter lang, hat einen Durchmesser von 16 Metern und wiegt etwa 12500 Tonnen. Sein Aufbau ist mit 100 Millionen Einzelteilen unglaublich komplex. Die Physiker erhoffen sich am LHC neue fundamentale Entdeckungen, die das Verständnis von der Materie und deren Wechselwirkungen erweitern. Besonders wird am LHC nach dem Higgsteilchen und den supersymetrischen Teilchen gesucht. Ebenso groß, wie der technische Aufwand zum Bau und Inbetriebnahme des Teilchenbeschleunigers, werden die Daten sein, die die Teilchendetektoren produzieren. Ohne effiziente Software und Rechenkapazitäten wären diese Daten gar nicht zu verarbeiten und zu analysieren. Eines der Programme die den Physikern für ihre Analysen der Daten zur Verfügung gestellt werden, ist VISPA. Mit VISPA wird ein neuartiger Ansatz verfolgt, der der interaktiven grafischen Analyse. Mit VISPA sollen graphische Bedienelemente und klassische Programmierung zusammengeführt werden. Dies hat es bis jetzt auf dem Softwaregebiet der Hochenergiephysik so noch nicht gegeben. Dabei erhofft man sich Vorteile gegenüber der konventionellen Programmierung, welche Klassenbibliotheken aus dem Bereich der Hochenergiephysik verwendet. Aber auch auf dem Gebiet der Astroteilchenphysik soll VISPA die Physiker bei ihrer Arbeit unterstützen.

In der Diplomarbeit geht es nun um die Evaluation des Programms. In Kapitel 2 werden die Gründe der Motivation dazu erläutert. Das 3. Kapitel beschreibt die Anwendungsgebiete von VISPA und die Ziele der Software. Wie VISPA implementiert wurde und welche Programmiersprache, Softwarebibliotheken und Werkzeuge verwendet wurden, erfährt der Leser in Kapitel 4. Die eigentliche Evaluation ist in Kapitel 5 beschrieben. Eine Zusammenfassung findet sich in Kapitel 6. In Kapitel 7 wird ein Ausblick auf zukünftige Entwicklungen gewährt.

2 Motivation der Evaluation

Bei der Evaluation vom VISPA und PXL geht es hauptsächlich um eine kritische Sicht aus dem Blickwinkel der Informatik. Die Evaluation eines Programms soll generell helfen, dieses zu optimieren, zu verbessern und eventuell schwerwiegende Fehler zu entdecken.

2.1 Laufzeitmessungen und Speicherbedarf

Von großem Interesse bei der Evaluation VISPA und PXL ist die Geschwindigkeit, mit der die Analysen ausgeführt werden. Dabei geht es um die Fragen, ob es eventuell Engpässe bei den Analysen im Hinblick auf die Ausführungsgeschwindigkeit gibt oder ob sich die Geschwindigkeit bei der Analysedurchführung verbessern lässt. Ein weiterer wichtiger Aspekt im Zusammenhang mit der Geschwindigkeit ist die Optimierung der Analysen. Aus der Sicht der Informatik, aber auch für den Physiker, stellen sich dabei folgende Fragen: Wann lohnt es sich ein Analysemodul für VISPA in C++ oder in der Skriptsprache Python zu programmieren? Welche Laufzeitunterschiede hat dies zufolge? Bei welcher Art von Analysen ist dieser Unterschied wie groß? welche Vor- und Nachteile haben die beiden Möglichkeiten ansonsten, wie beispielsweise Austauschbarkeit der Analysen oder Mehraufwand?

Der Speicherbedarf einer Analyse ist ebenfalls von Bedeutung. Je nach Art der Analyse, können unter Umständen große Datenmengen entstehen, diese müssen dann in einer sinnvollen Weise abgespeichert werden. Den Speicherbedarf möglichst gering zu halten, ist dabei von großem Interesse, denn dadurch vereinfacht sich der Datenaustausch über das Internet. Analyseresultate können so leichter zwischen Forschern zirkulieren. Um den Speicherbedarf zu reduzieren bietet sich natürlich eine Komprimierung der Daten an. Hier stellt sich die Frage nach dem Verhältnis der Komprimierung zu Zeitaufwand.

2.2 Qualität der Software

Ein weiterer interessanter Punkt ist die Qualität der Software. Hierbei geht es um die Qualität des Quellcodes. Dies betrifft Fragen wie die Komplexität des Quellcodes und den damit verbundenen Aufwand zur Erstellung, Wartung und zur Erweiterung der Software. Ein weiterer wichtiger Punkt der mit der Qualität der Software in Verbindung steht ist die Fehleranfälligkeit und der Aufwand zum Testen der Software. In diesem Zusammenhang werden bestimmte Metriken des Quellcodes der Software untersucht. Um einen Vergleich mit anderen Softwarepaketen aus diesem Bereich der Physik zu bekommen, werden diese ebenfalls auf die gleichen Metriken untersucht.

2.3 Bedienbarkeit der Software

Mit der Untersuchung der Bedienbarkeit von VISPA soll die Interaktion zwischen dem Benutzer und der Software beurteilt werden. Von Bedeutung ist hierbei die Übersichtlichkeit der Analysestruktur und der damit verbundenen Drag and Dropoberfläche, die Möglichkeit Fehler in der Analyse zu finden und zu korrigieren, die Anzahl der Schritte

für einen Zyklus der Analyseentwicklung und die Übersichtlichkeit der Resultate. Dies soll helfen eventuelle Schwächen in der Bedienbarkeit zu finden und zu verbessern.

2.4 Zusätzliche Erweiterungen und alternative Implementierungen

Auf Grundlage der Evaluation ergeben sich eventuell sinnvolle Erweiterungen für VISPA und PXL. Ebenso können alternative Implementierungen diskutiert werden, wobei Vor- und Nachteile zur jetzigen Implementierung erörtert werden.

3 Visual Physics Analysis: Das VISPA Projekt

VISPA ist eine Entwicklungsumgebung zum Erstellen von physikalischen Analysen. Dabei eröffnet VISPA dem Physiker die Möglichkeit, Analysen mit Hilfe einer Kombination aus grafischer Benutzeroberfläche und klassischer Programmierung zu entwickeln . Mit der Benutzeroberfläche ist es dem Physiker möglich, interaktiv eine Analyse zu entwerfen, auszuführen und die Resultate zu überprüfen.

3.1 Einsatzgebiete für VISPA

Für VISPA gibt es zwei Einsatzgebiete. Das eine ist der Bereich der Hochenergiephysik, das andere der Bereich der Astroteilchenphysik.

3.1.1 Beispiele für Analysen in der Hochenergiephysik

Die Hochenergiephysik beschäftigt sich mit dem Aufbau und der Zusammensetzung der Materie durch Elementarteilchen sowie deren Wechselwirkung untereinander. Dazu bringt man mit Hilfe von großen Teilchenbeschleunigern, Teilchen auf sehr hohe Energien, indem sie bis an die Grenze der Lichtgeschwindigkeit beschleunigt werden. Die Energie der Teilchen liegt dann im Bereich von GeV oder TeV. In einem Teilchendetektor werden die Teilchen dann zur Kollision gebracht. Dabei entstehen und zerfallen neue Teilchen. Die Größenordnung in der diese Kollisionen in Beschleunigern untersucht werden ist etwa 10^3 bis 10^7 Ereignisse pro Sekunde, wobei ein Ereignis einer Kollision entspricht.

Der bekannteste und größte Teilchenbeschleuniger, der Large Hadron Collider (LHC), steht in der Schweiz bei Cern. Beim LHC werden Protonen in entgegen gesetzter Richtung beschleunigt. Die Energie, auf die die Protonen dabei gebracht werden können, entspricht jeweils 7 TeV. Daraus ergibt sich eine Gesamtenergie von 14 TeV. Beim LHC können in den Detektoren wie am CMS-Experiment bis zu 40 Millionen Protonenpakete in der Sekunde aufeinander treffen, wobei bis zu 1 Milliarde Ereignisse (Protonenkollisionen) pro Sekunde erwartet werden. Ein Triggersystem filtert die interessanten Ereignisse heraus und reduziert diese auf nicht mehr als einige hundert in der Sekunde, die dann offline gespeichert werden. Trotz dieser Reduktion wird erwartet, dass der CMS-Detektor etwa 50.000 Terabyte Daten im Jahr produziert [Col06][Heb08].

Einige Teilchen, die bei der Kollision entstehen, sind nur sehr kurzlebig, wie zum Beispiel das Top Quark, welches eine durchschnittliche Lebensdauer von nur ca. $4 \cdot 10^{-25}$ Sekunden hat. Es zerfällt sofort wieder in langlebigere Teilchen, die dann im Detektor Spuren hinterlassen und deren Eigenschaften wie Energie, Richtung oder Ladung gemessen werden können. Kurzlebige Teilchen, wie das Top-Quark, können vom Detektor nicht erfasst werden und müssen so über die Rekonstruktion von Teilchenzerfällen nachgewiesen werden. Ein aktuelles Beispiel ist die Suche nach dem postulierten Higgs-Teilchen, welches anderen Teilchen ihre Masse verleiht. Es zerfällt der Theorie nach über zwei Z-Bosonen in vier Myonen.

Mit Hilfe von VISPA ist es nun möglich Datenanalyse auf diesem Gebiet zu betreiben. Ein typisches Beispiel ist hier die Rekonstruktion von Teilchenzerfällen sowie deren

Darstellung oder die Messung von Eigenschaften der Teilchen, wie zum Beispiel die Energieverteilung. Die Daten, die für VISPA verwendet werden, sind dann zum Beispiel die offline abgespeicherten Ereignisse.

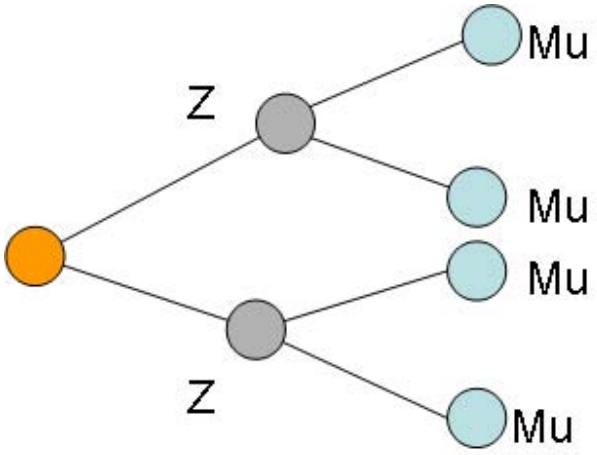


Abbildung 1: Zerfall eines Higgs Teilchens.

3.1.2 Beispiele für Analysen in der Astroteilchenphysik

Das zweite Teilgebiet in der Physik, bei der VISPA Anwendung findet ist die Astroteilchenphysik, speziell die Untersuchung von kosmischer Strahlung. Bei kosmischer Strahlung handelt es sich um hochenergetische Teilchenstrahlung aus dem Weltall mit einer Energie von $1000eV$ bis zu über $10^{20}eV$ [iD09]. Beim Auftreffen auf die Erdatmosphäre kollidieren die Teilchen der kosmischen Strahlung mit Atomen der Atmosphäre. Dabei wird eine Vielzahl von Sekundärteilchen erzeugt, die wiederum mit Atomen der Atmosphäre wechselwirken und neue Teilchen erzeugen. Dies geschieht so lange, bis die Energie des Primärteilchens aufgebraucht ist. Dadurch entsteht ein regelrechter Teilchenschauer, der auf der Erde ankommt.

Mit weltraumgestützten Experimenten, wie Detektoren in Satelliten, oder an hoch fliegenden Ballonen (ca. 40 km Höhe), lassen sich die Primärteilchen der kosmischen Strahlung direkt nachweisen. Die Wahrscheinlichkeit, dass der Detektor von einem Teilchen getroffen wird, sinkt aber mit steigender Energie des Teilchens. Kommen bei einer Energie von $10^{12}eV$ noch etwa 10 Teilchen pro m^2 und Minute an, sind es bei $10^{18}eV$ nur noch 65 pro km^2 und Jahr. Siehe Abbildung 2. Um auch diese Teilchen nachzuweisen, benötigt man sehr große Detektoren, die mit Satelliten oder Ballonen nicht realisierbar sind. Deshalb werden auch am Boden Detektoren aufgebaut. Ein Beispiel dafür ist die Anlage am Pierre-Auger-Observatorium, bei dem sich die Detektoren über eine Fläche von 3000 km^2 verteilen. Mit den bodengestützten Detektoren können nur die Teilchen aus den Schauern direkt nachgewiesen werden, aus denen dann aber Rückschlüsse auf das Primärteilchen gezogen werden können. Da die meisten Teilchen der kosmischen Strahlung, auf ihrem Weg durch den interstellaren Raum zur Erde, mehrfach von kosmischen Magnetfeldern abgelenkt werden, können keine oder nur schwer Rückschlüsse auf die Quelle gezogen werden. Ausnahme sind hierbei Teilchen mit extrem hohen Energien, bei denen man hofft die Quellen im Weltall zu finden.

Auch auf diesem Gebiet wird mit VISPA Datenanalyse betrieben. Mit VISPA kann der Physiker, beispielsweise anhand der gemessenen Teilchen, die Energieverteilung am

Himmel errechnen und in einer Himmelskarte darstellen oder Regionen von Interesse *Regions of Interest* lokalisieren. *Regions of Interest* können beispielsweise Regionen mit hoch hochenergetischen Ereignissen sein.

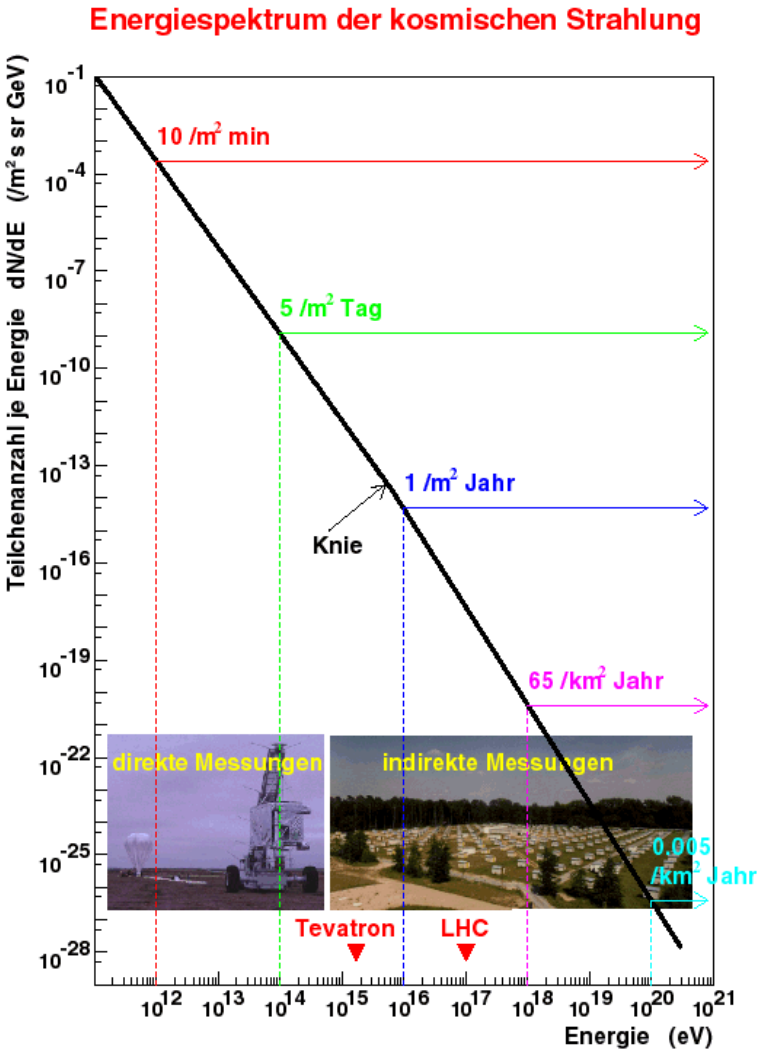


Abbildung 2: Energiespektrum der kosmischen Strahlung [iD09].

3.2 Ziele der Analyseumgebung VISPA

Herkömmliche Analysewerkzeuge wie ROOT, Pythia oder SPR [ROO09][Pyt09][Sta09] sind bis jetzt größtenteils Sammlungen von Klassenbibliotheken. Erstellt man eine Analyse mit Hilfe der herkömmlichen Klassenbibliotheken, so geschieht das meistens unter der Verwendung von C++ oder einer anderen Programmiersprache. Der Physiker entwickelt also für jede Analyse im Prinzip ein eigenständiges Programm. Dies hat zwar den Vorteil, dass er eine größtmögliche Freiheit bei der Entwicklung seiner Analyse hat, aber unter Umständen den Nachteil, dass der Physiker seine Analysen und Resultate nur schwer mit anderen Kollegen austauschen kann. Auch ist die Übersicht und die Nachvollziehbarkeit einer Analyse, die nur aus Quellcode besteht, sowie deren Verbesserung oder Korrektur schwierig und zeitaufwändig. Der Physiker verbringt viel Zeit sich mit technischen Problemen des Programmierens auseinanderzusetzen, anstatt mit der eigentlichen

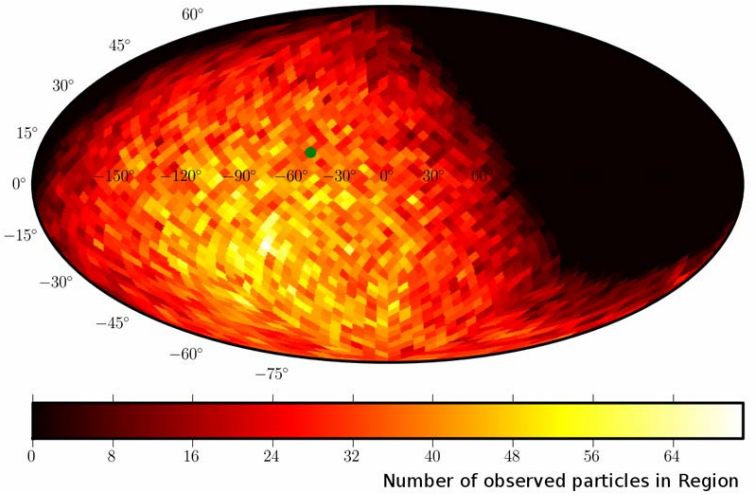


Abbildung 3: Simulation der Verteilung der Kosmischen Strahlung am durch das Auger Experiment beobachteten Himmel. Die Skala gibt die Anzahl der Ereignisse pro Pixel der Karte an. Der grüne Punkt markiert die Position von Centaurus A, einer möglichen Quelle der UHECR. (Quelle: Tobias Winchen, Private Communication)

Analyse. Durch die Nutzung der grafischen Benutzeroberfläche in VISPA soll nun das Entwickeln einer Analyse vereinfacht werden, so dass der Physiker sich wieder mehr auf seine eigentliche Arbeit, die physikalische Analyse, konzentrieren kann als auf die Probleme des Programmierens. Weiterhin bietet VISPA aber die Möglichkeit mit Hilfe der Skriptsprache Python oder mit C++, eigene Module einzubinden. Dadurch soll weiterhin eine größtmögliche Freiheit bei den Analysen gewährleistet werden.

3.2.1 Analysezyklus: Entwerfen - Ausführen - Überprüfen

Der Zyklus einer Analyseentwicklung in VISPA besteht hauptsächlich aus folgenden drei Schritten. Der Physiker erstellt seine Analyse, führt diese aus und überprüft anschließend die Resultate. Dabei kann der Physiker feststellen, ob er seine Analyse verbessern muss und den Zyklus erneut durchläuft. Dieser Analysezyklus mit den drei Schritten der Entwicklung soll möglichst schnell und unkompliziert ablaufen und damit eine Verbesserung gegenüber konventionellen Softwarepaketen bieten.

3.2.2 Drag and Drop-Oberfläche und grafische Übersicht

Ein weiteres wichtiges Ziel von VISPA ist es, die Übersicht gegen einer rein Quellcode basierten Analyse stark zu verbessern. Die Übersicht über die Analyse und die Resultate soll kompakt, schnell und leicht verständlich sein. Dabei verwendet VISPA ein Multifunktionsfenster, indem zum einen die Analysestruktur repräsentiert wird oder die Daten der Analyse dargestellt werden. Die einzelnen Elemente bzw. Module der Analyse, können per Drag and Drop von einem Nachbarfenster in das Analysefenster gezogen werden. Dort können sie verschoben, beliebig angeordnet und miteinander verbunden werden.

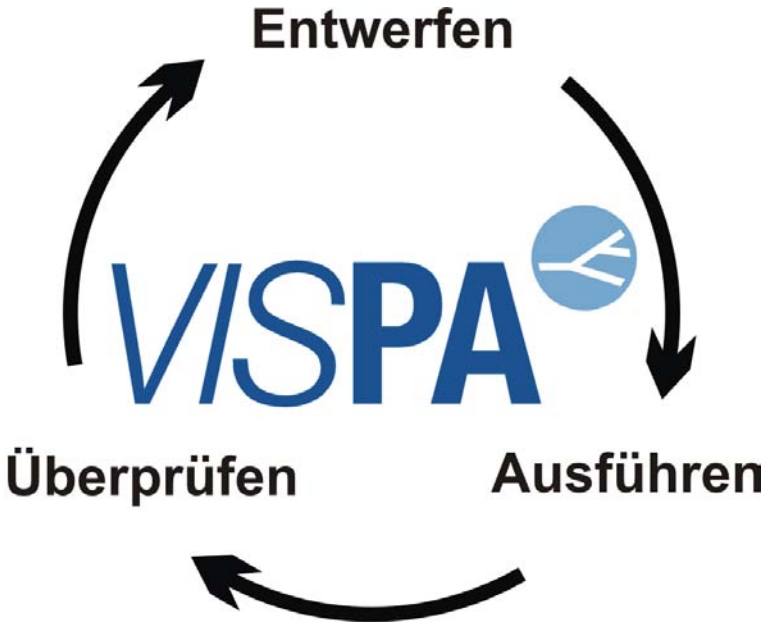


Abbildung 4: Entwerfen - Ausführen - Überprüfen.

3.2.3 Geschwindigkeit während der Durchführung von Analysen

Die Geschwindigkeit mit der die Analyse ausgeführt wird muss sich in einem für den Physiker akzeptablen Rahmen befinden, da er unter Umständen große Datenmengen zu bearbeiten hat. Wie schon erwähnt liegt beispielsweise in der Hochenergiephysik die Größenordnung der Ereignisse bei 10^3 bis 10^7. Dabei muss der Physiker, wenn er den Zerfall eines Teilchens rekonstruieren will, für ein einzelnes Ereignis mehrere mögliche Rekonstruktionen überprüfen. Die Größenordnung der möglichen Rekonstruktionen liegt hier bei etwa 10 bis 10^3 pro Ereignis. Das angestrebte Ziel ist es, auch auf nicht so leistungsfähigen Rechnern, eine Geschwindigkeit der Analyse zu erreichen, die für eine Rekonstruktion eines möglichen Teilchenzerfalls im Bereich von einer Millisekunde liegt. Dies natürlich unter der Berücksichtigung einer effizienten Programmierung.

3.2.4 Einfacher Austausch von Analysen und Resultaten

Die Möglichkeit Analysen und Resultate auszutauschen, hat den Sinn, dass sich Physiker dadurch gegenseitig kontrollieren und verbessern können. Auch für eine Teamarbeit ist dieser Aspekt wichtig, wenn mehr als ein Physiker an einer Analyse arbeitet. Probleme mit dem Austausch von Analysen mit konventionellen Analysewerkzeugen kann es geben, wenn zum Beispiel die Physiker unterschiedliche Entwicklungsumgebungen haben oder auf unterschiedlichen Plattformen wie Windows und Linux arbeiten. Werden dann die Quellcodedatein ausgetauscht, kann nicht garantiert werden, dass sie sofort kompiliert und ausgeführt werden können. Notfalls muss der Quellcode an die Entwicklungsumgebung oder an die jeweilige Plattform angepasst werden. Das ist besonders dann der Fall, wenn plattform- oder entwicklungsabhängige Funktionen verwendet werden. Mit VISPA soll nun der Austausch von Analysen erheblich vereinfacht werden. Mit einer Exportfunktion soll VISPA die Möglichkeit bieten, Analysen zwischen Physikern auszutauschen, unabhängig von der Entwicklungsumgebung und der Plattform. Die Analyse soll dann möglichst ohne Anpassung sofort gestartet werden können.

3.2.5 Verwendung von VISPA in der Lehre

Durch die Verlagerung von Quellcode hin zu einer grafischen Benutzeroberfläche, wird VISPA auch für die Lehre interessant. Physikstudenten sollen dort mit VISPA erste Physikanalysen in der Hochenergiephysik entwickeln und durchführen. Der Vorteil von VISPA durch die Verwendung einer Skriptsprache ist in dem Fall, dass die Physikstudenten keine tief gehenden Programmierkenntnisse besitzen müssen.

3.2.6 Verfügbarkeit und Plattformunabhängigkeit

VISPA soll für die drei großen Plattformen Windows, Linux, und MAC OS X zur Verfügung stehen, wobei unter Linux nur die Distribution von Debian unterstützt wird. Auf anderen Linuxdistributionen ist es auch möglich, VISPA zu installieren, aber die Funktionalität kann nicht garantiert werden. Ebenfalls ist vorgesehen, dass die Installation der Software möglichst einfach ist, um den Anwender nicht von Vornherein abzuschrecken.

4 Aufbau der VISPA 0.3.3- und PXL 2.5.5 -Programmpakete

VISPA wurde in der Skriptsprache Python entwickelt, unter der Verwendung von der PyQt Klassenbibliothek für die grafische Benutzeroberfläche. Für die Hochenergie- und Astropteilchenhysik bedient sich VISPA dem PXL Softwarepaket. Mit VISPA selbst wird die Analysestruktur erstellt und gespeichert, sowie die Resultate der Analyse visualisiert. Das Ausführen der Analyse, sowie das Abspeichern der Resultate geschieht dann mit PXL.

4.1 Grafische Darstellung der Analysen und Resultate

VISPA besitzt eine Oberfläche für das Entwerfen und Ausführen von Analysen sowie Visualisierung der Analysestruktur, einen Datenbrowser zum betrachten von Objektdaten und einen Editor zum Erstellen von Ereignissen.

Die Oberfläche für das Erstellen und Ausführen von Analysen besteht aus drei nebeneinander liegenden Fenstern. Das Linke beinhaltet die zur Verfügung stehenden Module. Diese können per Drag and Drop in das mittig platzierte Hauptfenster gezogen und dort frei angeordnet werden. Dort wird die Struktur der Analyse dargestellt. Die Module verfügen über Eingänge und/oder Ausgänge, die miteinander verbunden werden können. Über die Eingänge empfängt das Modul Objekte und über Ausgänge sendet es Objekte. *Objekt* ist hier ein allgemeiner Begriff für eine Datenstruktur. In der Hochenergiephysik ist es zum Beispiel meistens ein Ereignis, welches mit dem Datentyp *pxl:event* realisiert ist. In der Astroteilchenphysik wäre das Objekt vom Typ *pxl:BasisContainer*. Die Verbindung von Eingängen und Ausgängen werden über rote Linien angezeigt. Bei der Analyse wandert nun ein Objekt von seiner Quelle, einem Eingabe- oder Generatormodul, die Kette von Modulen entlang, die miteinander über Ein- und Ausgänge verbunden sind. In dem Fenster der rechten Seite werden die Eigenschaften der Module aufgelistet. Bei den Eigenschaften, die je nach Modul unterschiedlich sind, können dem Modul weitere Parameter übergeben werden. Wird die Analyse gestartet, öffnet sich ein viertes Fenster, welches eventuelle Meldungen von Modulen während der Analyse anzeigt.

Resultate der Analysen, wie zum Beispiel rekonstruierte Zerfälle von Teilchen, können als Ereignisdaten in einer Datei abgespeichert werden. Diese Ereignisdaten können mit VISPA ebenfalls mit Hilfe des Datenbrowsers betrachtet werden. Dabei stehen wieder drei Fenster zur Verfügung. Im Fenster links ist die Struktur des Ereignisses in einem Baum repräsentiert. Im Hauptfenster in der Mitte ist das Ereignis mit seinen Teilchenzerfällen grafisch angezeigt. Rechts befindet sich das Fenster, welches die Eigenschaften des Ereignisses bzw. der Teilchen anzeigt.

Mit dem Editor ist es möglich, manuell Ereignisdaten zu erstellen. Teilchenzerfallsmodelle können hier selbst entworfen werden. Dies ist nützlich für die Erstellung einer Steuerungsdatei, die das Autoprozessmodul benötigt.

4.2 Die Standardmodule in VISPA

VISPA verfügt über einige Standardmodule die im Folgendem vorgestellt werden.

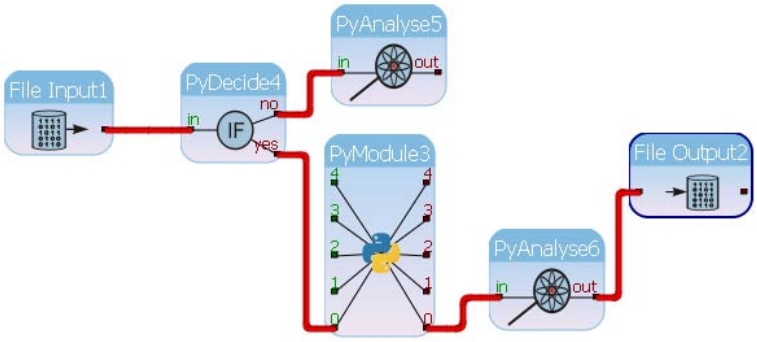


Abbildung 5: Aufbau bzw. Visualisierung der Analyse.

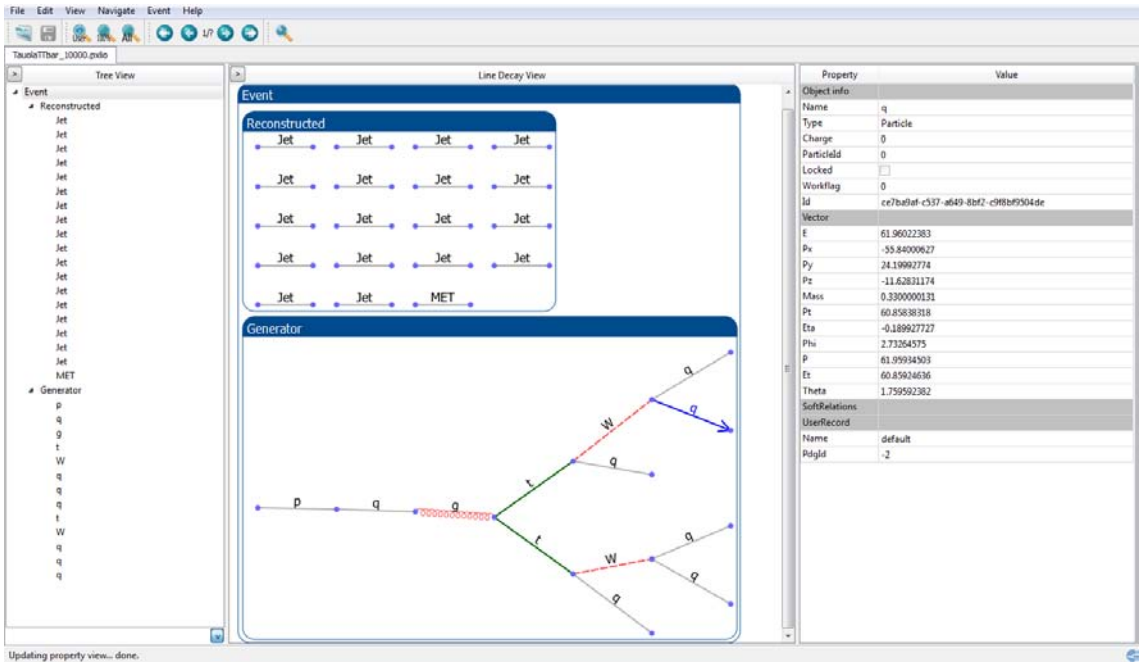


Abbildung 6: Ein Teilchenzerfall im Datenbrowser.

4.2.1 Die Ein- und Ausgabemodule

Mit dem Eingabemodul werden Dateien ausgelesen, die Objektdaten enthalten und diese dann an das nächste Modul weiter gesendet. Die Ereignisse sind in den Objektdateien seriell abgespeichert. Bei den Eigenschaften des Moduls kann der Startpunkt der Objektkette und der Endpunkt der Kette festgelegt werden, die über die Ausgabe gesendet wird. So kann sich der Anwender ein bestimmtes Objekt oder eine bestimmte Abfolge von Objekten herausgreifen. Das Modul verfügt über nur einen Ausgang.

Die Objekte, die an das Ausgabemodul gesendet werden, werden dort abgespeichert. Das Modul verfügt unter anderem über Einstellungsmöglichkeiten für die Kompression beim Abspeichern der Datei. Dieses Modul besitzt sowohl einen Eingang als auch einen Ausgang. Beide Module sind Bestandteil von PXL.

4.2.2 Das Pythonmodul

Das Pythonmodul bietet dem Anwender die Option, eigene Pythonskripts in die Analyse einzubinden. Hier hat der Physiker die Möglichkeit, die an den Eingängen ankommenden Objektdaten, beliebig auszuwerten, zu verarbeiten, zu manipulieren und schließlich an die Ausgänge weiterzuleiten. Das Skript wird für das Modul skelettartig generiert und in einer Datei gespeichert, die mit dem Modul assoziiert ist. Das generierte Skript hat drei Funktionen: *beginJob(parameters=None)*, *endJob()* und *process(object, sink)*. Die Funktion *beginJob(parameters=None)* wird einmal vor dem Beginn der Analyse aufgerufen, danach wird für jedes ankommende Objekt die Funktion *Process(object, sink)* aufgerufen der dann das Objekt (**object**) und die Nummer des Eingangs (*sink*) übergeben wird. In dieser Funktion sollen die Ereignisse verarbeitet werden. Nach der Analyse wird schließlich die Funktion *endJob()* ebenfalls einmal aufgerufen. Die Anzahl der Ein- und Ausgänge ist in diesem Modul über die Eigenschaften frei wählbar.

4.2.3 Das Switch-, Decide- und Analysemodul

Das Switchmodul verfügt über nur einen Eingang, aber über beliebig viele Ausgänge. Ansonsten verhält es sich wie das Pythonmodul. Nur ist der Name der Funktion hier *switch(object)* anstelle von *process(object,sink)*.

Das Decidemodul hat nur einen Eingang und zwei Ausgänge, ansonsten ist es analog zum Pythonmodul. Das Analysemodul verfügt über nur einen Ein- und Ausgang, ansonsten ist es ebenfalls analog zum Pythonmodul.

4.2.4 Das Autoprozessmodul

Dieses Modul rekonstruiert den Zerfall von Teilchen automatisch, unter der Verwendung einer Steuerungsdatei. Die Steuerungsdatei besteht aus einem Ereignis mit einer Ereignisansicht, die den zu rekonstruierenden Teilchenzerfall enthält. Diese Datei kann mit dem Ereigniseditor erstellt werden. Der Autoprozess erzeugt dabei alle möglichen Kombinationen eines Zerfalls, die sich aus den eingehenden Ereignisdaten ergeben können und fügt diese dem Ereignis hinzu. Mit der Option *Match* ist es möglich, die vom Autoprozess rekonstruierten möglichen Zerfälle mit einem vorgegeben Monte-Carlo-Zerfall zu vergleichen, um so den besten Treffer unter allen Kombinationsmöglichkeiten zu finden. Dieses Modul verfügt über einen Ein- und Ausgang. Das Autoprozessmodul ist ebenfalls Bestandteil von PXL.

4.3 Physics Extension Library (PXL)

PXL [EAo08] ist der Nachfolger von Physics Analysis eXpert (PAX) und ist der Softwareunterbau von VISPA.

4.3.1 Analysebasispaket in der Programmiersprache C++

Das PXL Softwarepaket ist eine Bibliothek von Klassen für Analysen in der experimentellen Hochenergiephysik sowie in der Astroteilchenphysik. Die Klassen wurden in der Hochsprache C++ entworfen und basieren auf dem Ansi C++ Standard [EAo08]. Dies ermöglicht eine Plattform übergreifende Nutzung. Auch soll durch die Wahl von C++ sichergestellt werden, dass der Code schnell genug ausgeführt wird, um bei Analysen eine akzeptable Geschwindigkeit zu erreichen.

PXL stellt alle nötigen Datenstrukturen für die Modellierung von Ereignissen zur Verfügung. Das reicht von Teilchen und deren Eigenschaften, bis hin zu Zerfallsbäumen und den damit verbundenen notwendigen Relationen. Außerdem besteht die Möglichkeit, beliebige Benutzerdaten zu den Informationen der Objekte hinzu zufügen. Weiterhin werden Algorithmen für die Vier-Vektor Analyse zur Verfügung gestellt sowie Funktionen zum Relationsmanagement. Ein weiterer wichtige Funktionalität, die in PXL existiert, ist der Autoprozess. Er ermöglicht eine automatisierte Rekonstruktion von Teilchenzerfällen unter Angabe einer Hypothese. Für das Kopieren von Ereignissen und deren kompletten Inhalt werden ebenfalls Funktionen zur Verfügung gestellt. Eine weitere Eigenschaft von PXL ist das Ein/Ausgabe-System zum Einlesen und Abspeichern von Objektdaten in Dateien.

Im Zusammenhang mit VISPA hat PXL auch die Funktion, die von VISPA erstellte Analysestruktur und die damit verbundene Analyse auszuführen. Dafür ist das Programm *Pxlrun* verantwortlich, welches von der Kommandokonsole oder aus VISPA heraus gestartet wird. Es liest die mit VISPA erstellte Analysestruktur, in Form einer XML-Datei, ein und führt dann die Analyse aus. Bei der Visualisierung von Daten, wie zum Beispiel Zerfallsbäume, greift VISPA ebenfalls auf die Datenstrukturen von PXL zurück.

Beim Kompilieren von PXL werden die für VISPA notwendigen DLL-Dateien (Windows) bzw. SO-Dateien (LINUX) erzeugt. Weiterhin wird die *Pxlrun* generiert. Ebenfalls werden die notwendigen Interface-Dateien erzeugt, die notwendig sind, um eine Schnittstelle zwischen VISPA und PXL bzw. zwischen Python und C++ zu haben.

4.3.2 Kompressionsbibliothek zlib

Je nach Analyse können große Datenmengen erzeugt werden. Deshalb bedient sich PXL der Kompressionsbibliothek zlib, um die Daten vor dem Speichern in Objektdateien zu komprimieren. Zlib verwendet den Deflate-Algorithmus zum Komprimieren und Dekomprimieren der Daten. Der Deflate-Algorithmus besteht aus einer Kombination der Huffman-Kodierung und dem LZ77-Algorithmus [DG96][Deu96].

Der LZ77-Algorithmus ist ein verlustfreies, auf einem Lexikon basierendes Verfahren zur Datenkompression [ZL77]. Bei diesem Algorithmus streicht ein Fenster, das so genannte *Sliding Window*, den Eingabestrom der Daten in eine Richtung entlang. Das Fenster besteht aus einem Such-Puffer, dem Lexikon und einem Vorschau-Puffer. Der Vorschau-Puffer enthält die noch zu komprimierenden Daten und der Such-Puffer die schon komprimierten Daten. Der Algorithmus sucht nun im Lexikon, also im Such-Puffer, nach einer größtmöglichen Übereinstimmungen mit dem Vorschau-Puffer. Als Ausgabe hat der Algorithmus ein Tripel (**Offset, Länge, Folgesymbol**). Das Offset ist die Position im Such-

Puffer zu Beginn der Übereinstimmung, Länge ist die Länge der Übereinstimmung und das Folgesymbol ist das erste Symbol, welches nach der Übereinstimmung im Vorschau-Puffer folgt. Der Algorithmus arbeitet nun wie folgt:

Voraussetzungen:
Die Größe des Such-Puffers sei m und die des Vorschau-Puffers ist n.
l = Länge maximaler Übereinstimmung ($2 \leq l \leq n$)

Zu Beginn ist der Such-Puffer leer und der Vorschau-Puffer mit den ersten n Symbolen des Eingabestroms gefüllt.

Algorithmus:
Vergleiche das erste Symbol im Vorschau-Puffer X mit allen Symbolen im Such-Puffer;
Wenn Übereinstimmung gefunden
. Überprüfe jeweils die Folgesymbole im Such- und Vorschau-Puffer
. nach Übereinstimmung;
. Wenn Länge Übereinstimmung $< l$;
. Durchsuche den gesamten Such-Puffer nach weiteren größeren
. Übereinstimmungen;
. O = Position im Such-Puffer Anfangs der größten
. Übereinstimmung;
. L = Länge der größten Übereinstimmung;
. S = Erstes Folgesymbol nach der größten Übereinstimmung im
. Vorschau-Puffer;
. Gebe Tripel aus (O, L ,S);
. Bewege Fenster um L Symbole über den Eingabestrom;
Sonst
. Gebe Tripel aus (0, 0, X);
. Bewege Fenster um ein Symbole über den Eingabestrom;
Wiederhole solange bis Ende des Eingabestroms;

Such-Puffer (Lexikon)																	**Vorschau-Puffer**										
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	1	2	3	4	5	6	7	8	9	10	Ausgabe
																		L	A	A	B	L	A	A	L	U	U	
																	L	A	A	B	L	A	A	L	U	U	␣	(0,0,L)
																L	A	A	B	L	A	A	L	U	U	␣	L	(0,0,A)
														L	A	A	B	L	A	A	L	U	U	␣	L	A	A	(18,1,B)
										L	A	A	B	L	A	A	L	U	U	␣	L	A	A	B	L	A	A	(15,3,L)
									L	A	A	B	L	A	A	L	U	U	␣	L	A	A	B	L	A	A	L	(0,0,U)
							L	A	A	B	L	A	A	L	U	U	␣	L	A	A	B	L	A	A	L	U	U	(1,1,␣)
L	A	A	L	U	U	␣	L	A	A	B	L	A	A	L	U	U	␣	L	A	A	B	L	A	A	.			(8,10,␣)
A	A	B	L	A	A	L	U	U	␣	L	A	A	B	L	A	A	.											(8,7,.)

Tabelle 1: Der Algorithmus LZ77 an einem Beispiel des Datenstroms
LAABLAALUU␣LAABLAALUU␣LAABLAA.

Der andere Teil des Deflate-Algorithmus ist die Huffman-Kodierung. Die Huffman-Kodierung ist eine bestimmte Form der Entropiekodierung [Huf51]. Die Idee der Huffman-

Kodierung basiert auf der unterschiedlichen Wahrscheinlichkeiten des Vorkommens von Symbolen in einem Datenstrom. Symbole mit hoher Wahrscheinlichkeit sollen möglichst mit einer kurzen Bitfolge kodiert werden, und Symbole mit geringer Wahrscheinlichkeit mit einer längeren Bitfolge. Dadurch sinkt die mittlere Länge der Bitfolgen und somit werden die Daten komprimiert. Eine wichtige Eigenschaft die für die Kodierung erfüllt werden muss, ist die präfixfreie Notation. Präfixfreie Notation bedeutet, keine der Bitfolgen die ein Symbol kodieren, darf am Anfang einer anderen Bitfolge stehen, die ein weiteres Symbol kodieren. Ein Beispiel für eine nicht präfixfreie Notation wäre **X = 10**, **Y = 110**, **Z = 11** und **A=0**. Die Folge **11010** kann für **YX** aber auch für **ZAY** stehen. Somit ist die Eindeutigkeit nicht mehr gegeben.

Um nun eine solche präfixfreie Notation zu bekommen, nimmt man einen Binärbaum zur Hilfe. Die Blätter des Binärbaums repräsentieren die zu kodierenden Symbole. Der Pfad von der Wurzel zu einem Blatt ist die Kodierung, also die Bitfolge für das Symbol. Angenommen es gilt folgenden Text zu kodieren: **ADAEABDBCAACADABAA** Die Häufigkeiten der Buchstaben ergeben sich wie folgt: **A=9, B=3, C=2, D=3, E=1**

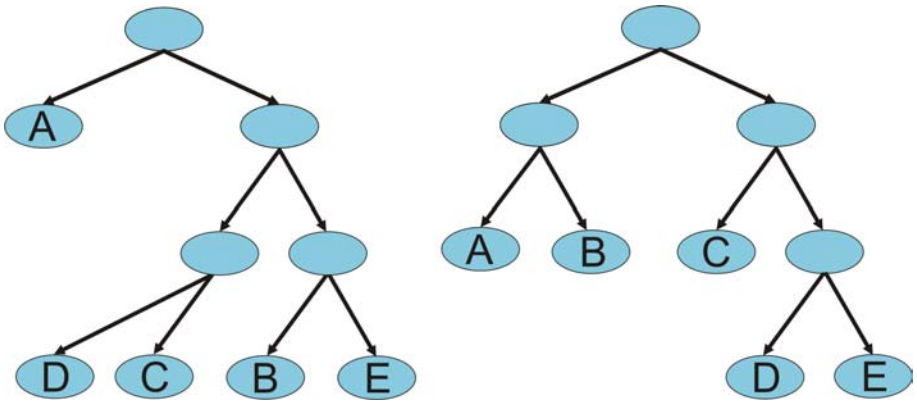


Abbildung 7: Zwei der Möglichkeiten den Baum aufzubauen.

Für die beiden Bäume würden sich folgende Kodierungen ergeben.

Kodierung Baum 1		**Kodierung Baum 2**	
A	0	A	00
B	110	B	01
C	101	C	10
D	100	D	110
E	111	E	111

Die Länge der Kodierung ist jedoch unterschiedlich. Beim ersten Baum wären es

$$L = \frac{1Bit \cdot 9 + 3Bit \cdot 3 + 3Bit \cdot 2 + 3Bit \cdot 3 + 3Bit \cdot 1}{18} \approx 1,88 \frac{Bit}{Symbol}$$

Beim zweiten Baum

$$L = \frac{2Bit \cdot 9 + 2Bit \cdot 2 + 2Bit \cdot 2 + 3Bit \cdot 3 + 3Bit \cdot 1}{18} \approx 2,11 \frac{Bit}{Symbol}$$

Es gilt also den Baum zu finden, der die optimale Kodierung hat, also so wenig Bit/Symbol wie möglich.

Die Huffman-Kodierung baut nun mit folgendem Verfahren einen Baum auf, der immer optimal ist [Huf51].

Voraussetzungen:
Ein Alphabet mit $a_1, a_2..., a_n$ Symbolen
Text mit n unterschiedlichen Symbolen
Wahrscheinlichkeiten P_i für das Auftreten eines Symbols von a_i für $i = (1, 2..., n)$ mit $P_1 + P_2 + ... + P_i = 1$

Algorithmus:

1. Erstelle für jedes Symbol aus dem Alphabet einen Baum mit einem Knoten.
2. Fasse zwei Bäume, deren Symbole die geringste Wahrscheinlichkeit haben, als Teilbäume mit neuer Wurzel, zu einem neuen Baum zusammen.
3. Berechne die Wahrscheinlichkeit des neuen Baums aus der Summe der beiden alten.
4. Wiederhole 2 und 3, bis nur noch ein Baum vorhanden.

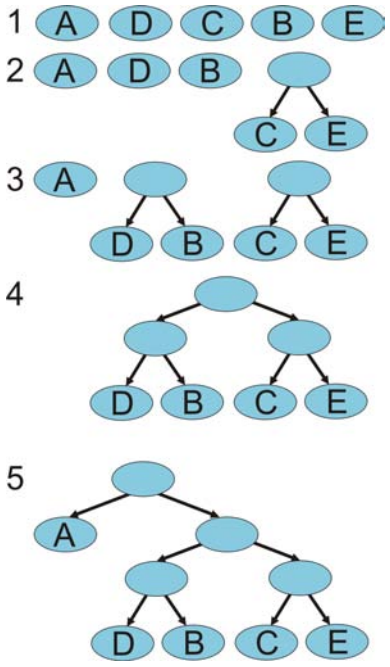


Abbildung 8: Beispiel für den Algorithmus.

Der Deflate-Algorithmus führt nun erst den LZ77-Algorithmus aus und anschließend die Huffman-Kodierung. Zur Kompression verwendet PXL die Funktion *compress2()* aus der zlib-Bibliothek.

4.4 Verwendete Programmiersprache und Bibliotheken für VISPA

Im Folgenden werden die Programmiersprachen und Klassenbibliotheken, die für VISPA verwendet wurden, vorgestellt.

4.4.1 Die Skriptsprache Python

Python ist eine objektorientierte, interaktive Interpretersprache [Fou09c]. Python zeichnet sich vor allem durch seine Einfachheit, Übersichtlichkeit und leichte Erlernbarkeit aus. Dafür ist die Geschwindigkeit von Python langsamer, als die von den meisten kompilierbaren Sprachen wie C oder C++ [Deb09]. Korrektheit und Übersicht geht vor Geschwindigkeit [Fou09b]. Dadurch kann in Python besonders einfach und schnell programmiert werden. Außerdem kommen die Vorteile der Übersichtlichkeit und der leichten Erlernbarkeit vor allem Programmieranfängern zugute, oder Anwendern, die nur geringe Programmierkenntnisse besitzen. Dies zeigt sich beispielsweise daran, dass Python im Gegensatz zu C++ über eine automatische Speicherbereinigung verfügt. Auch verfügt Python über keine statische Typüberprüfung wie C++.

Weil VISPA die rechenintensiven Analysen selbst nicht ausführt, sondern nur die Benutzeroberfläche für das Erstellen der Analysestruktur bereitstellt sowie die Resultate und Objektdaten anzeigt, fällt die Geschwindigkeit von Python an dieser Stelle nicht ins Gewicht. Außerdem besteht die Möglichkeit in Python, zeitkritische Funktionen oder Module in andere Sprachen wie C oder C++ auszulagern.

Eine weitere wichtige Eigenschaft von Python ist die plattformunabhängigkeit. Pythoninterpreter existieren für Windows, Linux und MAC OS X.

4.4.2 Verwendung der QT-Klassenbibliothek

Die QT-Klassenbibliothek dient hauptsächlich zur Entwicklung grafischer Benutzeroberflächen, aber auch für Datenbanken und XML-Anwendungen. Die Klassenbibliotheken wurden in C++ entwickelt. Mit PyQt bekommt man die für VISPA notwendige Anbindung an Python. Qt bzw. PyQt ist ebenfalls plattformübergreifend und steht für Windows, Linux und MAC-OS zur Verfügung.

4.4.3 Verwendung von XML zur Speicherung der Analysestruktur

Für die Speicherung der Analysestruktur wird die Extensible Markup Language (XML) verwendet. XML ist eine textbasierte Auszeichnungssprache für die Darstellung von hierarchischen Datenstrukturen. Da XML plattformunabhängig ist, eignet es sich besonders gut für den Austausch von Daten zwischen Computern, vor allem über das Internet [BP+08].

Das XML-Dokument, in dem eine Analysestruktur gespeichert wird, ist wie folgt aufgebaut: Das erste Element, also das Wurzelelement ist <*analysis*>, dieser Tag umschließt alle anderen Elemente. Für jedes Modul, das der Analysestruktur hinzugefügt wird, wird der Tag <*module*> eingefügt. Der Tag besitzt noch folgende Attribute, welche die Haupteigenschaften aller Module sind: *id* eindeutige Id-Nummer des Moduls, *name* Name des Moduls, *type* der Typ des Moduls und die Attribute x und y für die Koordinaten bei der Darstellung auf der Benutzeroberfläche im Hauptfenster. Das Eingabe- und das Generatormodul besitzen noch zusätzlich das Attribut *runIndex*, welches angibt in welcher Reihenfolge die Module ihre Objekte senden. Bei gleichem Index senden beide Module ihre Objekte parallel. Bei unterschiedlichen Indizes sendet das Modul mit dem kleinsten Index erst alle seine Objekte, bevor das mit dem nächst höheren Index, beginnt seine Objekte zu senden.

Für jede weitere Eigenschaft, die das Modul besitzt, wird der Tag <*option*> innerhalb des Tag <*module*> hinzugefügt. Dieser Tag besitzt zwei Attribute, *name* für den Namen

der Eigenschaft und *type* für den Typ der Eigenschaft. Sollte es sich um den Typ *string* handeln, so steht dieser als CDATA-Abschnitt innerhalb des <*option*> Tags.

Die Verbindungen zwischen den Modulen werden mit dem Tag <*connection*> abgespeichert. Innerhalb dieses Tags befinden sich immer die Tags <*source*> und <*sink*> für Aus- und Eingang mit der die Verbindung verknüpft ist. Diese beiden Tags haben jeweils die Attribute *id*, für die eindeutige Id-Nummer des Moduls, und *name*, für den Namen bzw. die Nummer des Ein- und Ausgangs innerhalb des Moduls.

Die Module und Verbindungen werden in der Reihenfolge in die XML-Datei geschrieben, in der sie auch in der Benutzeroberfläche in die Analysestruktur eingefügt werden.

4.5 Schnittstelle zwischen VISPA und PXL

Da VISPA in Python entwickelt wurde und PXL in C++, muss für die Benutzung der PXL-Klassen aus Python heraus eine Schnittstelle zwischen VISPA und PXL existieren. Dies ist mit SWIG realisiert.

4.5.1 SWIG: Python-Schnittstelle zu PXL

Der Simplified Wrapper and Interface Generator (SWIG) ist ein plattformübergreifendes Programm, das es ermöglicht, in C oder C++ geschriebene Module, mit einer anderen Skriptsprache zu verbinden. Damit stehen dann der Skriptsprache die C/C++ Module zur Verfügung. Die Skriptsprache, für die diese Verbindung hergestellt werden soll, muss allerdings schon von sich aus die Möglichkeit haben, C oder C++ Bibliotheken aufzurufen.

Um ein C/C++ Modul in Python einzubinden, geht man in der Regel wie folgt vor: Als erstes benötigt man die Interface-Datei als Eingabe für SWIG. In dieser Interface-Datei stehen die Funktionen oder die Headerdateien des C/C++ Moduls, das Python zur Verfügung gestellt werden soll, sowie eventuell weitere Befehle zur Anpassung. SWIG erzeugt daraufhin zwei Dateien. Die erste Datei ist eine C-Quellcodedatei mit Funktionen, die als Schnittstelle zwischen Python und den C/C++ Modulen dienen, den so genannten Wrappern. Diese C-Quellcodedatei muss noch mit dem Rest des C/C++ Moduls kompiliert werden. Die zweite Datei ist eine Python-Quellcodedatei, die dann in den Python-Quellcode importiert werden muss, um das Modul zu nutzen.

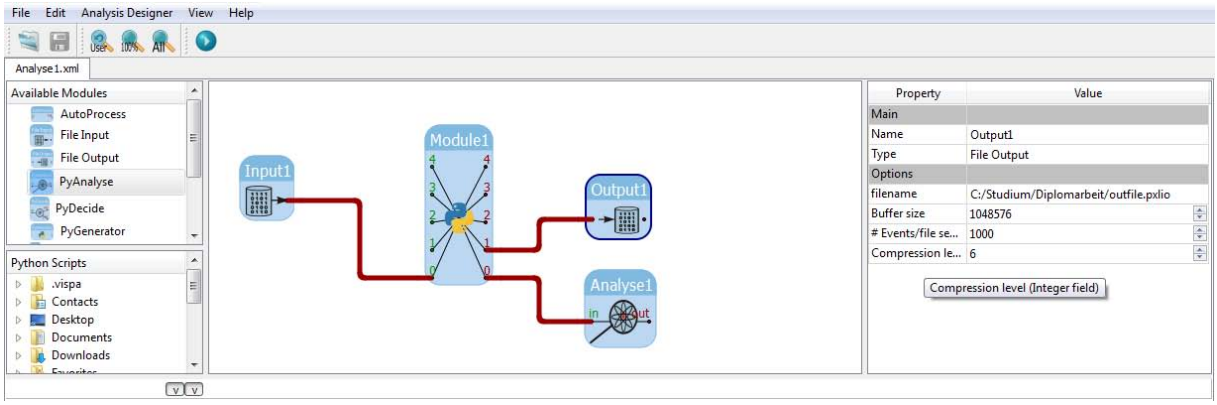
Nach dem Kompilieren erhält man die DLL-Datei für das C/C++ Modul und eine PYD-Datei für den Wrapper-Code unter Windows bzw. zwei SO-Dateien unter LINUX.

Wird nun eine Funktion des C/C++ Moduls aus einem Pythonskript aufgerufen, so wird erst die zuständige Funktion in der von SWIG generierten Python-Datei aufgerufen. Diese Funktion ruft ihrerseits die kompilierte Wrapper-Funktion auf, die dann letztendlich die Funktion des C++ Moduls aufruft.

Für die Nutzung von PXL durch VISPA wurden mehrere PXL-Klassen jeweils zu einem Modul zusammengefasst.

4.5.2 Einbindung von C++ Modulen

Wie schon erwähnt, besteht in VISPA die Möglichkeit, eigene Module in C++ mit Hilfe von PXL zu erstellen und einzubinden. Dafür stellt PXL eine Beispieldatei *example plugin.cpp* sowie das notwendige Makefile (Linux) bzw. die Projekt-Datei für VISUAL C++ (Windows) bereit. Diese Beispieldatei ist eine Art Skelett, ähnlich wie die generierten Python-Dateien für ein Pythonmodul. Die Datei enthält alle notwendigen Funktionen und Variablen, um alle Eigenschaften von Modulen zu bestimmen. Analog zu den generierten Python-Dateien für die Python-Module, besitzt auch das Skelett der Beispieldatei eine




```
<?xml version="1.0" ?>
<analysis file_format_version="0.3">
  <module id="f65c6d61-ed07-11de-955e-00269e5d5622" name="Input1" runIndex="0" type="File Input" xPos="30" yPos="74">
    <option name="filename" type="string">
<![CDATA[c:\program files (x86)\vispa\examples\analysisdesigner\zmumu_mc_10.pxlio]]>    </option>
    <option name="start" type="long">
      0
    </option>
    <option name="end" type="long">
      -1
    </option>
    <option name="Additional file names" type="string_vector">
      ()
    </option>
  </module>
  <module id="004ab1b0-ed08-11de-8ea4-00269e5d5622" name="Module1" type="PyModule" xPos="217" yPos="43">
    <option name="filename" type="string">
<![CDATA[c:\program files (x86)\vispa\examples\analysisdesigner\selection_script.py]]>    </option>
    <option name="script" type="string">
<![CDATA[]]>    </option>
    <option name="parameter" type="string">
<![CDATA[]]>    </option>
    <option name="sinks" type="long">
      5
    </option>
    <option name="sources" type="long">
      5
    </option>
  </module>
  <connection>
    <source module="f65c6d61-ed07-11de-955e-00269e5d5622" name="out"/>
    <sink module="004ab1b0-ed08-11de-8ea4-00269e5d5622" name="0"/>
  </connection>
  <module id="04f59c1e-ed08-11de-b30b-00269e5d5622" name="Output1" type="File Output" xPos="379" yPos="93">
    <option name="filename" type="string">
<![CDATA[outfile.pxlio]]>    </option>
    <option name="Buffer size" type="long">
      1048576
    </option>
    <option name="# Events/file section" type="long">
      1000
    </option>
    <option name="Compression level" type="long">
      6
    </option>
  </module>
  <connection>
    <source module="004ab1b0-ed08-11de-8ea4-00269e5d5622" name="1"/>
    <sink module="04f59c1e-ed08-11de-b30b-00269e5d5622" name="in"/>
  </connection>
  <module id="098b29cf-ed08-11de-98ec-00269e5d5622" name="Analyse1" type="PyAnalyse" xPos="378" yPos="189">
    <option name="filename" type="string">
<![CDATA[c:\program files (x86)\vispa\examples\analysisdesigner\sort_filter_particles_script.py]]>    </option>
    <option name="script" type="string">
<![CDATA[]]>    </option>
    <option name="parameter" type="string">
<![CDATA[]]>    </option>
  </module>
  <connection>
    <source module="004ab1b0-ed08-11de-8ea4-00269e5d5622" name="0"/>
    <sink module="098b29cf-ed08-11de-98ec-00269e5d5622" name="in"/>
  </connection>
</analysis>
```

Abbildung 9: Erstellte Analysestruktur visuell und als XML.

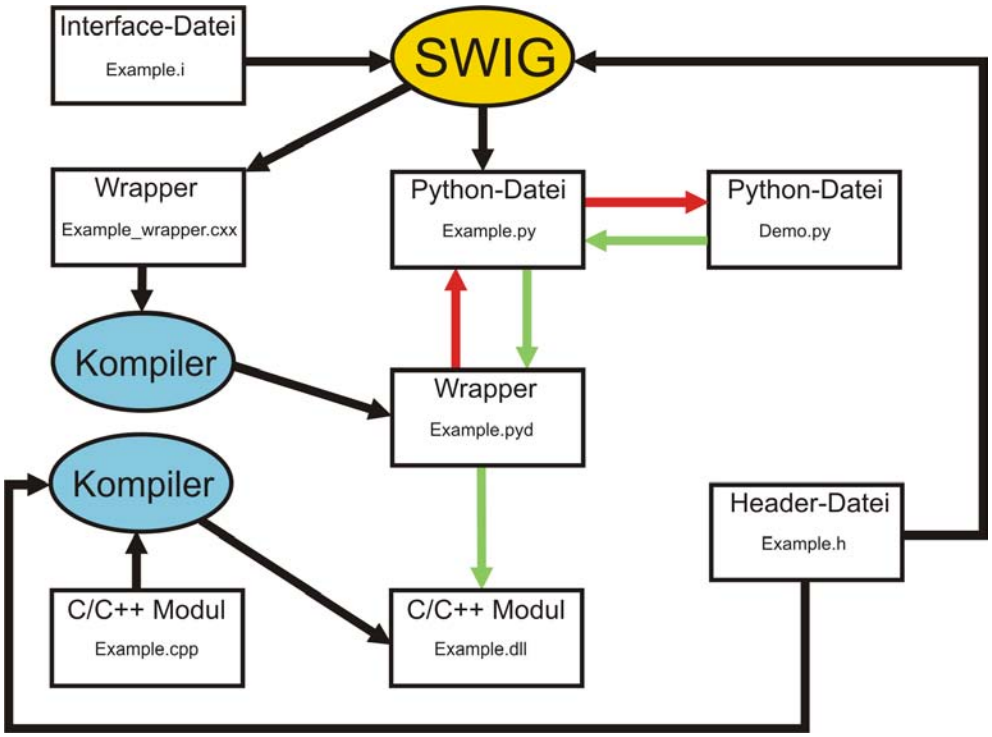


Abbildung 10: Erzeugung eines Interfaces zwischen C++ und Python mit Hilfe von SWIG.

Funktion *analyse()*, in die der Code für die Analyse implementiert wird. Diese wird ebenfalls, wie in der Python-Datei, für jedes Objekt aufgerufen. Ebenso gibt es die Funktionen *beginJob()* und *endJob()*. Weiterhin existieren Funktionen und Variablen zum Setzen der Anzahl der Ein- und Ausgänge des Moduls sowie zur Erzeugung der Eigenschaften (Options) und zur Bestimmung des Modulnamens. Der Inhalt der Funktionen des Skelettes kann mit Code gefüllt und der Wert der Variablen geändert werden, bzw. wenn es sich um Vektoren handelt, diese beliebig aufgefüllt werden. Die Deklaration, der bereits im Skelett vorkommenden Funktionen, muss unverändert bleiben. Fügt der Benutzer neue Funktionen hinzu, sind diese nur innerhalb der cpp-Datei nutz- und sichtbar.

Wenn die Datei kompiliert, wird eine DLL- oder SO-Datei erzeugt. Diese erzeugte Datei muss nun in den Plugin-Ordner von VISPA kopiert werden. Damit steht dann das Modul VISPA zur Verfügung. Eine explizite Interface-Datei muss nicht erzeugt werden. Die Schnittstelle für die erstellten Module und VISPA besteht aus einer universellen Interface-Datei, die einen PlugIn-Manager beinhaltet.

4.6 Ausführung einer Analyse mit VISPA und PXL

Betätigt der Anwender in VISPA den Startknopf für die Analyse, so wird per Shell von VISPA die Datei *Pxlrun* mit der XML-Datei als Parameter aufgerufen. *Pxlrun* liest die XML-Datei und startet die Analyse. Es wird eine Modulkette auf Grundlage der in der XML-Datei gespeicherten Analysestruktur aufgebaut. Beim Start der Analyse wird bei allen Modulen nacheinander die Funktion *StartJob()* aufgerufen. Jetzt werden von der oder den Quellen, Eingabe- oder Generatormodulen, die Objekte in die Modulkette geschickt. Dabei durchläuft das Objekt erst alle Module bis zum Ende der Kette, bevor das nächste von der Quelle gesendet wird. Ist das letzte Objekt abgearbeitet, wird bei allen Modulen

die Funktion *EndJob* aufgerufen. Alternativ kann die Analyse auch ohne VISPA, nur mit dem Aufruf von *Pxlrun* und der XML-Datei als Parameter, von der Kommandokonsole gestartet werden.

4.7 Verwendung des Softwarepakets ROOT

ROOT ist ein Rahmensoftwarepaket, welches Serviceklassen zur Datenanalyse und für Datenhistogramme im Bereich der Hochenergiephysik zur Verfügung stellt. Es existieren unter anderem Anbindungen für C++ und Python. In VISPA wird ROOT hauptsächlich verwendet, um Histogramme und Plots zu erstellen. Im Datenbrowser gibt es ein Betrachtungsmodus, welcher Standardhistogramme für Teilchen und kosmische Teilchen anzeigt.

4.8 Verfügbarkeit und Installation von VISPA und PXL

Für die drei Betriebssysteme Linux, MAC-OS und Windows existieren Installationspakete von VISPA und PXL, die online heruntergeladen werden können. Der Installer von PXL installiert auf allen Betriebssystemen PXL in der schon kompilierten Form, zusammen mit den notwendigen Interface-Dateien, welche mit SWIG generiert wurden und ebenfalls schon kompiliert sind.

Unter Linux wird die Installation mit dem deb-installer realisiert. Im Installer sind alle notwendigen Abhängigkeiten für VISPA aufgelistet. Dies betrifft PXL, Python, PyQt und ROOT. Sollten die Pakete nicht vorhanden sein, wird der Benutzer aufgefordert sie herunterzuladen und zu installieren.

Im Installationspaket für MAC-OS sind alle Programme schon enthalten, das bedeutet es muss nichts aus dem Netz nachgeladen werden.

Die Windows-Installation geschieht mit dem Installationspaket von Nullsoft. Der Installer beinhaltet VISPA selbst und PXL. Bei der Installation werden die anderen noch nicht installierten notwendigen Pakete angezeigt und zur Installation angeboten. Auch hier werden sie dann automatisch über das Netz heruntergeladen.

Für keine der drei Installationspakete besteht eine Updatefunktion. Wenn der Benutzer eine neue Version von VISPA installieren möchte, muss erst die alte Version deinstalliert werden, bevor die neue Version installiert werden kann.

5 Methodik der Evaluation

Die wichtigen Aspekte, die bei der Evaluation von VISPA und PXL aufgegriffen werden, sind die Handhabung, die Ausführungsgeschwindigkeit bei Analysen sowie der Speicherbedarf der Resultate, die Qualität der Software und die Beurteilung durch den Benutzer.

Im ersten Punkt wird anhand der Implementierung von VISPA und PXL erörtert, ob und in wie weit die Ziele, in Bezug auf die Handhabung und Bedienbarkeit von VISPA, umgesetzt wurden.

Im zweiten Punkt werden, mit Hilfe von typischen Analysen aus den beiden Bereichen Hochenergie- und Astroteilchenphysik für die VSPA konzipiert wurde, Laufzeitmessungen vorgenommen. Zum einen wird die absolute Geschwindigkeit bei der Durchführung von Analysen gemessen. Damit kann das Erreichen eines der Ziele von VISPA, das der Geschwindigkeit von mindestens einer Teilchenrekonstruktion im Millisekundenbereich, verifiziert werden. Die andere Messung ist die der relativen Geschwindigkeiten von Python- und C++ Modulen. Dies ist ein wichtiger Faktor bei der Entscheidung, ob der Physiker seine Analyse mit Pythonmodulen oder mit C++ Modulen entwickelt. Die Gründe für die Geschwindigkeitsunterschiede von Python und PXL werden ebenfalls untersucht. Schließlich wird noch der Speicherbedarf gemessen, der bei der Abspeicherung der Resultate entsteht. In diesem Zusammenhang wird auch die Effizienz der Komprimierung von den Daten der Resultate evaluiert.

Beim Punkt Softwarequalität werden verschiedene Metriken gemessen, um Schwächen im Quellcode zu finden. Um eine Referenz für die Metriken zu haben, werden sie mit anderen Softwarepaketen aus dem Bereich der Hochenergiephysik verglichen.

Im letzten Punkt wird eine Umfrage über VISPA ausgewertet und diskutiert.

5.1 Handhabung und Bedienbarkeit von VISPA und PXL

Die folgenden Aussagen über die Handhabung und Bedienbarkeit basieren ausschließlich auf Grundlagen der Implementierung von VISPA und PXL. Aussagen über Handhabung und Bedienbarkeit auf Grund von subjektiven Empfindungen, Eindrücken und Erfahrungen mit VISPA werden in Abschnitt 5.6 mit Hilfe einer Umfrage gemacht.

5.1.1 Mehraufwand beim Einbinden eigener C++ Module

Wenn der Anwender in VISPA eigene C++ Module einbinden möchte, entsteht ein Mehraufwand an Arbeit und Zeit im Zyklus der Analyseentwicklung. Muss bei einer Korrektur der Analyse die Funktion eines Moduls geändert werden, so geht das bei einem Pythonmodul, indem die zugehörige Skriptdatei editiert und gespeichert wird. Die Analyse kann sofort wiederholt werden. Bei einer Änderung eines C++ Moduls sind wesentlich mehr Schritte erforderlich. Außer dem Editieren der Quellcodedatei, muss die Datei noch kompiliert werden. Außerdem muss zum jetzigen Zeitpunkt das VISPA-Programm, in der Version 0.3.3, noch geschlossen und neu gestartet werden, damit das geänderte Modul in VISPA verfügbar ist. Grund hierfür ist, dass der Pluginmanager von VISPA beim Start des Programms, die Module lädt. Als nächster Schritt muss die Analysestruktur

nach dem Start von VISPA geladen werden, also die zugehörige XML-Datei geöffnet werden. Erst dann kann die Analyse wieder ausgeführt werden. Einen weiteren Mehraufwand hat der Anwender bei der Aufspürung von Fehlern. Wird beispielsweise ein dem Modul übergebenes Objekt falsch behandelt, so dass ein Laufzeitfehler entsteht, gibt das Ausgabefenster in VISPA die Zeile im Skript an, in der der Fehler aufgetreten ist. Wenn es sich um ein C++ Modul handelt, erscheint unter Umständen nur eine Systemfehlermeldung, aus der keine oder nur schwer Rückschlüsse auf den Fehler zu ziehen sind. Um eine gleichartige Fehlermeldung wie bei einem Pythonmodul zu bekommen, muss das C++ Modul schon mit Debug-Informationen kompiliert worden sein und es muss eine geeignete Entwicklungsumgebung zur Verfügung stehen, wie zum Beispiel Visual C++, welche die Debug-Informationen auslesen kann.

Mit einer Analyse, die nur aus Pythonmodulen und Standardmodulen besteht, kann ein Zyklus der Analyseentwicklung also wesentlich schneller durchlaufen werden, als Analysen die selbst geschriebene C++ Module haben.

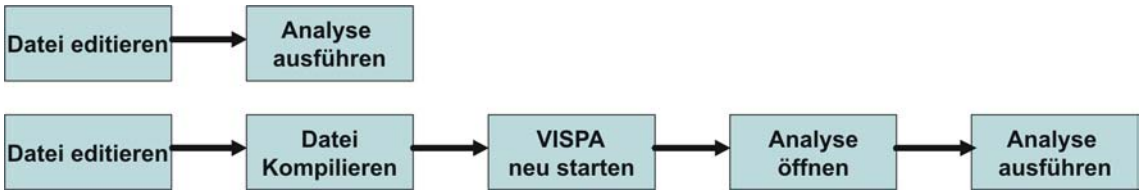


Abbildung 11: Schritte zum Ändern eines Moduls. Oben: ein Pythonmodul. Unten: Ein C++ Modul .

5.1.2 Austauschbarkeit von Analysen

Ein wichtiger Kernpunkt für die Ziele von VISPA ist die Austauschbarkeit von Analysen. Hier ist eine Analyse, die nur aus Standardmodulen und Modulen mit Pythonskripts besteht, wiederum im Vorteil. Mit der Exportfunktion werden alle nötigen Dateien, die XML-Datei, die Skriptdateien welche mit den Pythonmodulen assoziiert sind, die Objektdateien, welche als Eingabedateien für die Analyse dienen und eventuell die notwendigen Steuerungsdateien des Autoprozesses, zu einem ZIP-Archiv hinzugefügt. Dieses ZIP-Archiv kann auf einem anderen Rechner, unabhängig von der Plattform oder installierter Entwicklungsumgebung, geöffnet und in VISPA importiert werden. Die Analyse kann sofort gestartet werden. Mit vom Anwender selbst entwickelten C++ Modulen ist die Handhabung nicht so einfach. Haben die beiden Rechner, zwischen denen der Austausch stattfinden soll, zwei unterschiedliche Betriebssysteme, beispielsweise Linux und Windows, so ist es nicht möglich DLL-Dateien von Windows nach Linux zu portieren. Stattdessen muss die Quellcodedatei portiert und auf dem Zielrechner neu kompiliert werden. Selbst bei Rechnern mit gleicher Plattform, kann es zu Komplikationen kommen. Das ist dann der Fall, wenn auf den Rechnern unterschiedliche Entwicklungsumgebungen vorhanden sind, wie Visual Studio und die GNU Compiler Collection. Während Visual Studio eigene Projektdateien für Compilereinstellungen braucht, benötigt der GNU Compiler Makefiles. Diese Dateien müssen bei der Portierung eventuell erst angepasst oder erstellt werden.

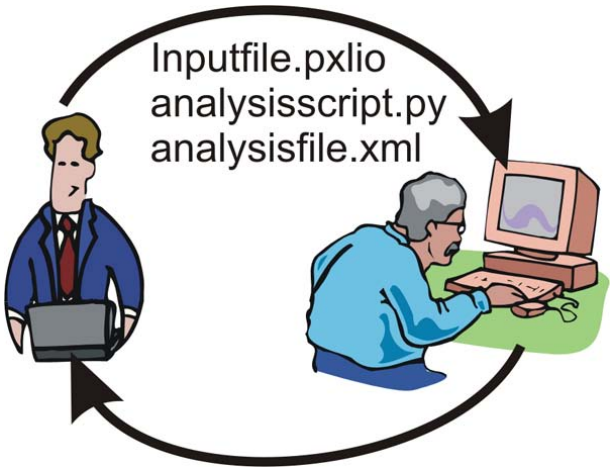


Abbildung 12: Austausch von Analysen mit den notwendigen Dateien.

5.1.3 Einschränkung der Pythonmodule

Die Pythonmodule besitzen, im Gegensatz zu den C++ Modulen, allerdings auch Einschränkungen. In den Pythonmodulen kann man zwar die ankommenden Objekte beliebig manipulieren, aber es ist in der gegenwärtigen Implementation nicht möglich das Hauptobjekt, welches alle anderen beinhaltet, zu kopieren und damit ein und dasselbe Objekt an zwei verschiedene Ausgänge des Moduls zu senden. Es ist also mit Pythonmodulen nicht möglich, ein Kopier- oder Klonmodul zu bauen. Möchte man mit einem Objekt zwei verschiedene Analysen durchlaufen, welche das Objekt auch verändern, dann geht das auch nur in zwei getrennten Analysestrukturen. Ein Beispiel hierfür wäre, wenn Teilchen herausgefiltert werden, indem sie aus dem Objekt bzw. Ereignis gelöscht werden, um Daten zu reduzieren, und die beiden Analysen unterschiedliche Filterkriterien haben. Eine weitere Einschränkung ist das Hinzufügen von Eigenschaften bzw. Optionsfeldern für ein Modul. Bei Pythonmodulen ist das nicht direkt möglich. Dort muss der Anwender, wenn er dem Pythonmodul Eigenschaften übergeben will, dies in Form von einer Parameterzeichenkette in der Option *Parameter* realisieren. Dies ist umständlicher und unübersichtlicher, als wenn es für jede Eigenschaft ein Optionsfeld gibt. Die C++ Module unterliegen nicht diesen Beschränkungen. Ein weiterer Nachteil von Pythonmodulen ist, dass Fehler im Skript erst zur Laufzeit auftauchen, während sie in C++ schon während des Kompilierens angezeigt werden.

5.1.4 Dokumentation von VISPA und PXL

Informationen über VISPA und PXL sind auf den jeweiligen Internetseiten verfügbar. Dort befinden sich auch jeweils die Klassendokumentationen, welche mit *Doxygen* [vH10] erstellt wurden. Die Klassendokumentationen sind besonders im Umgang mit Datenstrukturen und Funktionen von PXL hilfreich. Weiterhin gibt es ein Tutorial für VISPA im PDF-Format und es existiert ein Manual für PXL. Beide sind auf der Internetseite von VISPA verfügbar. Was fehlt, sind Beispielanalysen, und die dazugehörigen Erklärungen. Einige wenige Beispielanalysen befinden sich zwar bei der Installation von VISPA, allerdings ohne eine Verbindung mit einer physikalischen Erklärung. In der Benutzeroberfläche von VISPA fehlt eine Hilfe oder eine Dokumentation. Hat der Benutzer keinen Internetzugang, so ist für ihn diese Klassendokumentation nicht abrufbar.

5.2 Laufzeitmessungen und Speicherbedarf der Analysen

5.2.1 Verwendete Programme und Methoden zur Evaluation

Für die Laufzeitmessung der Analysen wurden im Prinzip zwei Methoden verwendet. Die Erste ist eine Zeitmessung mit Hilfe von Zeitfunktionen. Zu Beginn jeder Analyse wird in jedem Modul die Funktion *beginJob()* aufgerufen. Direkt am Anfang dieser Funktion wird dann die Zeit gemessen. Nach Beendigung der Analyse erfolgt der Aufruf der Funktion *endJob()* in jedem Modul. Dort wird ganz am Ende der Funktion die Zeit ein zweites Mal gemessen. Die Differenz der beiden Zeiten wird dann als Zeit angesehen, welche die Analyse während der Durchführung benötigt hat. Für die Zeitmessung wurden vier Zeitfunktionen verwendet. Die ersten Beiden sind die Funktion *time()* und *clock()* unter Python. Die Zeitfunktion *clock()* eignet sich besonders zur Laufzeitmessung und hat unter Windows eine Auflösung von 1 Mikrosekunde [Fou09a]. Die dritte Zeitfunktion ist die Windowsfunktion *timeGetTime()* unter C++. *timeGetTime()* besitzt eine Genauigkeit von 5 Millisekunden [Mic09]. Die letzte Zeitfunktion ist die C-Funktion *clock()*. Sie zeigt die verbrauchte CPU-Zeit für den Prozess an [Kub07]. Um zufällige Einflüsse von anderen Programmen, wie Virenscanner, Firewalls oder Systemprozessen, zu vermeiden, wurde die Laufzeitmessung für jede Analyse mehrfach wiederholt. Des Weiteren wurde grundsätzlich mit unterschiedlichen Zeitfunktionen, sowohl beim ersten als auch beim letzten Modul in der Analysekette, gemessen. Damit soll ausgeschlossen werden, dass einige Befehle, die bei den Ein- oder Ausgabemodule in den Funktionen *beginJob()* und *endJob()* stehen, nicht von der Laufzeitmessung erfasst werden. Bein keiner der Messreihen unterschieden sich die verschiedenen Zeitfunktionen, um mehr als 3% von der Gesamtzeit innerhalb einer Messung. Die Standardabweichungen vom Mittelwert einer Messreihe war bei allen Analysen nie größer als $2,5\%$ der Gesamtzeit. Die meisten Abweichungen waren jedoch unter 1%. Die Verwendung von Zeitfunktionen zur Laufzeitmessung ist auch in [Kub07] beschrieben.

Für die zweite Methode wurde das Programm *AQTime* verwendet. *AQTime* ist ein Analyseprogramm zur Messung von Laufzeit, Speicherbedarf und weiteren Resourcen eines Programms [Aut09]. Das Programm wird dazu verwendet, den relativen Zeitbedarf von Modulen und Funktionen zu messen. Solche Profiler-Programme wie *AQTime* finden oft in der Geschwindigkeitsnalyse von Programmen Verwendung [RSvdW09]. Für die Messungen mit *AQTime* wurde PXL mit Debuginformationen kompiliert, um eine möglichst genaue und detaillierte Analyse zu bekommen.

5.2.2 Verwendete Analysen

Die Analysen, die für die Laufzeittests verwendet wurden, sind aus den beiden Bereichen Hochenergie- und Astroteilchenphysik.

Bei den verwendeten Analysen in der Hochenergiephysik handelt es sich um die Rekonstruktion von Teilchenzerfällen, speziell mit der Hilfe der Vier-Vektoranalyse. Bei einem Zerfall eines Teilchens in andere Teilchen muss die Energie und der Impuls der Teilchen erhalten bleiben. Der Viererimpuls des ursprünglichen Teilchens ist gleich die Summe der Viererimpulse der Teilchen, in die es zerfällt. Als Beispiel, der Zerfall eines W-Teilchens (W-Boson) in zwei Quarks. Es gilt:

Energieerhaltung

$$E_w = E_{q1} + E_{q2} \tag{1}$$

Impuls eines Teilchens

$$\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{2}$$

Impulserhaltung

$$\vec{p}_w = \vec{p}_{q1} + \vec{p}_{q2} \tag{3}$$

$$\vec{p}_w^{\,2} = (\vec{p}_{q1} + \vec{p}_{q2})^2 \tag{4}$$

Viererimpuls eines Teilchens

$$p = \begin{pmatrix} \frac{1}{c}E \\ \vec{p} \end{pmatrix} = \begin{pmatrix} Gesamtenergie \\ Impulsvektor \end{pmatrix} \tag{5}$$

Viererimpulserhaltung

$$p_w = p_{q1} + p_{q2} \tag{6}$$

$$p_w^2 = (p_{q1} + p_{q2})^2 = m_w^2 c^2 \tag{7}$$

$$= \begin{pmatrix} \frac{1}{c}(E_{q1} + E_{q2}) \\ (\vec{p}_{q1} + \vec{p}_{q2}) \end{pmatrix}^2 \tag{8}$$

Daraus folgt:

$$m_w^2 c^2 = \frac{1}{c^2}(E_{q1} + E_{q2})^2 - (\vec{p}_{q1} + \vec{p}_{q2})^2 \tag{9}$$

$$m_w^2 = (E_{q1} + E_{q2})^2 - (\vec{p}_{q1} + \vec{p}_{q2})^2 \tag{10}$$

Hierbei ist c die Konstante der Lichtgeschwindigkeit, E_w Energie des W-Teilchens, p_w Impuls des W-Teilchens und m_w Masse des W-Teilchens. Entsprechend sind $E_{q1},\ E_{q2}$ und $p_{q1},\ p_{q2}$ jeweils Energie und Impuls der Quarks. Ein Beispiel mit konkreten Werten:

$$p_{q1} = \begin{pmatrix} \frac{1}{c}E_{q1} \\ \begin{pmatrix} \vec{p}_{x_{q1}} \\ \vec{p}_{y_{q1}} \\ \vec{p}_{z_{q1}} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{c}43,90\ GeV \\ \begin{pmatrix} -11,50\ GeV \\ -34,83\ GeV \\ 24,12\ GeV \end{pmatrix} \end{pmatrix} \tag{11}$$

$$p_{q2} = \begin{pmatrix} \frac{1}{c}E_{q2} \\ \begin{pmatrix} \vec{p}_{x_{q2}} \\ \vec{p}_{y_{q2}} \\ \vec{p}_{z_{q2}} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{c}61,96\ GeV \\ \begin{pmatrix} -55,84\ GeV \\ 24,20\ GeV \\ -11,63\ GeV \end{pmatrix} \end{pmatrix} \tag{12}$$

$$(13) \qquad p_w = \begin{pmatrix} \frac{1}{c}E_w \\ \begin{pmatrix} \vec{p}_{x_w} \\ \vec{p}_{y_w} \\ \vec{p}_{z_w} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} \frac{1}{c}43,90\ GeV \\ \begin{pmatrix} -11,50\ GeV \\ -34,83\ GeV \\ 24,12\ GeV \end{pmatrix} \end{pmatrix} + \begin{pmatrix} \frac{1}{c}61,96\ GeV \\ \begin{pmatrix} -55,84\ GeV \\ 24,20\ GeV \\ -11,63\ GeV \end{pmatrix} \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{c}105,86\ GeV \\ \begin{pmatrix} -67,34\ GeV \\ -10,63\ GeV \\ 12,49\ GeV \end{pmatrix} \end{pmatrix}$$

$$(14) \qquad m_w^2 = E_w^2 - \vec{p}_w^{\,2} = (105,86\ GeV)^2 - 4803.37 GeV^2 = 80.02\ GeV$$

Für die Laufzeittests wurden zwei Analysen erstellt. Die erste Analyse ist die Rekonstruktion eines W-Bosons aus zwei im Detektor gemessenen Quarks. Die zweite Analyse ist die Rekonstruktion eines Top-Quarks aus drei im Detektor gemessenen Quarks.

Eine Analyse für die Laufzeitmessung auf dem Gebiet der Astroteilchenphysik ist die Berechnung eines Sphärizitäts-Tensors, der Sphärizität und der Aplanarität. Die Sphärizität und Aplanarität sind Maße für die Abweichung der gemessenen kosmischen Strahlung am Himmel, von der erwarteten, isotrop verteilten kosmischen Strahlung. Die kosmische Strahlung, Ultra-High-Energy Cosmic Rays (UHECRs), ist angegeben mit ihrer Energie E und dem Richtungsvektor $\vec{p} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$. Der Sphärizitäts-Tensor berechnet sich aus den UHECRs wie folgt:

$$(15) \qquad S^{\alpha\beta} = \frac{\sum\limits_i p_i^\alpha p_i^\beta}{\sum\limits_i p_i}$$

für $\alpha, \beta = (1, 2, 3)$ wobei p_1 die x Komponente des Richtungsvektors ist und p_2, p_3 die y und z Komponente.

Die Sphärizität S und A Aplanarität berechnet sich aus den Eigenwerten $(\lambda_1, \lambda_2, \lambda_3)$ des Tensors die sich aus der Diagonalisierung von $S^{\alpha\beta}$ ergeben. Für die Eigenwerte gilt $\lambda_1 \geq \lambda_2 \geq \lambda_3$ und $\lambda_1 + \lambda_2 + \lambda_3 = 1$. Es gilt weiterhin:

$$(16) \qquad S = \frac{3}{2}(\lambda_2 + \lambda_3)$$

$$(17) \qquad A = \frac{3}{2}(\lambda_3)$$

Für isotrope kosmische Strahlung erwartet man $S = 1$ und $\lambda_1 = \lambda_2 = \lambda_3 = 1/3$, ansonsten $0 \leq S \leq 1$. Für die Aplanarität gilt: bei $A = 1/2$ ist die kosmische Strahlung isotrop, ansonsten $0 \leq A \leq 1/2$.

Eine weitere Analyse auf dem Gebiet der Astroteilchenphysik, die für den Laufzeitvergleich gewählt wurde, ist die Berechnung der *Regions of Interest*. Bei den *Regions of Interest* handelt es sich um Regionen am Himmel, bei denen die Energie der einfallenden Teilchenstrahlung sehr hoch ist. Diese Regionen werden auf folgende Art und Weise berechnet. Zuerst werden alle UHECR's, die einen Schwellwert S überschreiten, in einer Liste gespeichert. Weiterhin wird für jedes UHECR in der Liste eine Ereignisansicht angelegt. Anschließend wird die Liste abgearbeitet. Für jedes UHECR in der Liste, wird innerhalb eines bestimmten Kegelradius β, nach weiteren UHECRs gesucht. Wird ein solches UHECR gefunden, wird es der Ereignisansicht, in der sich das UHECR mit der hohen Energie befindet, hinzugefügt. Die Energie des hinzugefügten UHECRS's braucht den Schwellwert nicht zu überschreiten. Die Anzahl der hinzugefügten UHECR's und deren aufsummierte Energie werden jeweils in einem *UserRecord* der Ereignisansicht gespeichert. Die maximale Laufzeit ist hierbei $\Omega(n) = n + n^2$. Um zu berechnen, ob sich ein UHECR innerhalb des Kegelradius β eines anderen UHECR befindet, kann man folgende einfache Formel verwenden.

$$p_u p_E > cos(\beta) \tag{18}$$

p_u und p_E sind dabei die normierten Richtungsvektoren des UHECR.

5.2.3 Vergleich der Geschwindigkeit von Analysen in Python und C++

Die erste Analyse, die zum Laufzeitvergleich herangezogen wird, ist die Rekonstruktion des Zerfalls eines W-Boson in zwei Quarks. Die Analyse besteht aus drei Modulen, dem Eingabemodul, dem C++ oder Pythonmodul und dem Ausgabemodul. Die Eingabedatei besteht aus 3200 Objekten, in diesem Fall Ereignisse vom Typ *pxl:Event*. Jedes Ereignis enthält zwei Ereignisansichten vom Typ *pxl:EventView*. Die erste Ereignisansicht enthält einen zufälligen, nach dem Monte-Carlo-Verfahren generierten, Zerfallsbaum von Teilchen. Die zweite Ereignisansicht enthält einzelne zufällig generierte Teilchen, wie man sie anhand des generierten Zerfalls in einem Detektor messen würde. Das W-Boson wird jetzt aus Teilchen vom Typ *Jet*, die als Quarks interpretiert werden, aus der zweiten Ereignisansicht rekonstruiert. Die erste Ereignisansicht ist für die Analyse irrelevant und wird ignoriert. Bei der Rekonstruktion wird das Ereignis um eine Ereignisansicht erweitert, in die alle möglichen Rekonstruktionen, die sich ergeben, hinzugefügt werden. Anschließend wird das Ereignis an das Ausgabemodul geschickt und abgespeichert. Jede einzelne Rekonstruktion wird erstellt, indem für jedes beteiligte Teilchen ein Objekt vom Typ *pxl:Particle* erzeugt wird, und diese dann, nach der Vorschrift des Zerfalls, miteinander verbunden werden. Die Anzahl der möglichen Rekonstruktionen ist von der Anzahl der Jets abhängig. Da ein W-Boson in zwei Quarks bzw. Jets zerfällt, gibt es bei der Rekonstruktion, wenn n Jets in einem Ereignis gemessen wurden, $\binom{n}{2}$ mögliche Rekunstruktionen pro Ereignis. Jedoch rekonstruiert die verwendete Analyse doppelt so viele W-Bosonen, nämlich $n(n-1)$. Das ist nötig, weil die Analyse später mit dem Autoprozess verglichen werden soll, und der Autoprozess ebenfalls $n(n-1)$ mögliche Rekonstruktionen erstelt. Da das C++ Modul und das Pythonmodul gleich verfahren, hat das keine Auswirkungen auf den Vergleich beider Module.

Die Analyse wird auf einem Rechner mit folgender Konfiguration ausgeführt.(Tabelle 2).

Dabei wird die Analyse mit und ohne Ausgabemodul ausgeführt, jeweils mit einem C++ oder Pythonmodul. Es gibt also vier Kombinationen der Analyse. Alle werden jeweils

- Betriebsystem: Windows XP Professional 32 bit
- CPU: Intel Centrino M 1,73 GHz
- RAM: 1 GB
- Festplatte: 80GB Fujitsu MHV2080AT

Tabelle 2: Notebook 1.

12 Mal wiederholt. Der Kompressionsgrad im Ausgabemodul ist standardmäßig auf 6 eingestellt.

Weiterhin wird auch 12 Mal die Zeit gemessen, die das Eingabemodul mit der Eingabedatei benötigt, indem es als alleiniges Modul in einer Analyse ausgeführt wird. Der Mittelwert beträgt $5,24$ Sekunden

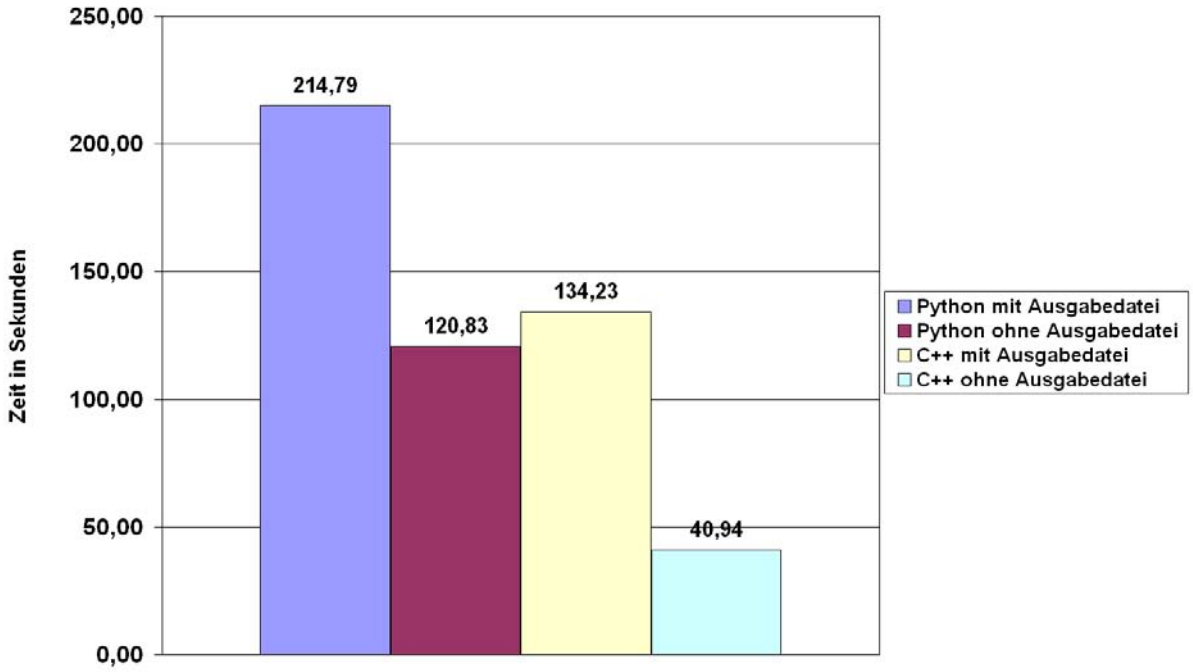


Abbildung 13: Laufzeitvergleich zwischen Python- und C++ Modulen. Rekonstruktion eines W-Boson aus zwei Quarks mit 783826 möglichen Rekonstruktionen.

Aus dem Laufzeitvergleich in Abbildung 13 kann man erkennen, dass das C++ Modul etwa um den Faktor 3 schneller ist, als das Pythonmodul. Zieht man die Zeit, die das Eingabemodul benötigt, ca. 5,2 Sekunden, von der Laufzeitmessung ab, ist es sogar ein Faktor von ca. 3,2.

Die totale Rate der einzelnen Zerfallsrekonstruktionen sieht man in Tabelle 3.

Die zweite Analyse ist ebenfalls eine Rekonstruktion eines Teilchenzerfalls. Diesmal ist

-	Pyth. mit Ausg.	Pyth. ohne Ausg.	C++ mit Ausg.	C++ ohne Ausg.
Zeit pro Rekonst.	274,0 μs	154,1 μs	171,2 μs	52,2 μs

Tabelle 3: Totale Rate der einzelnen Zerfallsrekonstruktion bei einem W-Bosonzerfall nach zwei Quarks.

der Zerfall etwas komplexer. Ein Top-Quark zerfällt in ein Quark und ein W-Boson. Das W-Boson zerfällt wiederum in zwei Quarks, siehe Abbildung 14

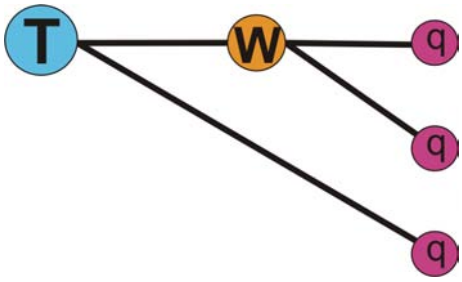


Abbildung 14: Zerfall eines Top Quarks.

Die Analyse besteht wiederum aus dem Eingabemodul, dem C++ oder Pythonmodul und dem Ausgabemodul. Die Eingabedatei ist die gleiche, wie bei der ersten Analyse und das Verfahren analog. Allerdings werden hier nur 300 Ereignisse anstatt der 3200 verwendet. Der Rechner ist ebenfalls wieder der gleiche. Bei der Rekonstruktion des Top-Quarks liefert die Analyse, für n Jets in einem Ereignis, welche auch wieder als Quarks angesehen werden, $n(n-1)(n-2)$ mögliche Rekonstruktionen pro Ereignis.

Die Analyse wird ebenfalls, in den vier Kombinationen, 12 Mal wiederholt. Der Kompressionsgrad im Ausgabemodul ist hier auch wieder standardmäßig 6.

Bei diesem Laufzeitvergleich (Abbildung 15) ist das C++ Modul etwa um den Faktor 2,7 schneller als das Pythonmodul.

-	Pyth. mit Ausg.	Pyth. ohne Ausg.	C++ mit Ausg.	C++ ohne Ausg.
Zeit pro Rekonst.	233,37 μs	139 μs	145,9 μs	51,2 μs

Tabelle 4: Totale Rate der einzelnen Zerfallsrekonstruktion bei einem Zerfall des Top-Quark. Ein Zerfall eines Top-Quark besteht aus zwei Einzelzerfällen.

Die nächste Analyse, die zum Laufzeitvergleich herangezogen wird, ist eine Abänderung der Analyse des W-Bosonzerfalls. Der entscheidende Unterschied ist, dass nicht jedes W-Boson rekonstruiert wird, sondern nur solche die eine Masse zwischen $80.0 - 81.0\ GeV$ haben. Dies wird mit Hilfe der Formel (14) berechnet. Es werden wieder $n(n-1)$ Möglichkeiten pro Ereignis überprüft. Jedoch führt diese Einschränkung zu lediglich 3286 Rekonstruktionen. Dies hat zur Folge, dass viel weniger Objekte erzeugt werden und die Datenmenge, im Gegensatz zur ursprünglichen Analyse, viel geringer ist. Des Weiteren wird das Ausgabemodul durch ein Pythonmodul ersetzt, welches ein Energiehistogramm der W-Bosonen erstellt. Wie aus Abbildung 16 erkannt werden kann, ist die Analyse mit dem C++ Modul ca. 4,4 mal schneller als die des Pythonmodul.

Weiterhin wird die Analyse zur Berechnung des Sphärizitäts-Tensors, der Sphärizität und der Aplanarität einer Laufzeitmessung unterzogen. Die Analyse besteht aus zwei Modulen, dem Eingabemodul und dem C++ oder Pythonmodul. Die Eingabedatei hat 10 Objekte, in diesem Fall BasisContainer, mit je 30000 UHECRs. Die Messungen werden

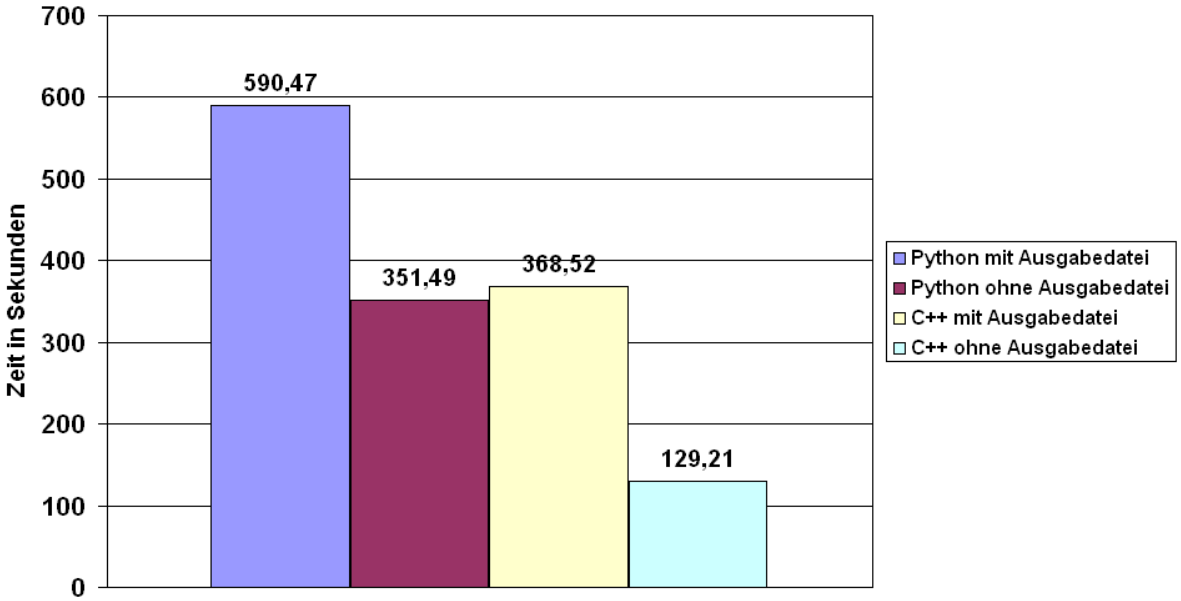


Abbildung 15: Laufzeitvergleich zwischen Python- und C++ Modulen. Rekonstruktion eines Top-Quarks aus einem W-Boson und Quarks mit 1.262.940 möglichen Rekonstruktionen.

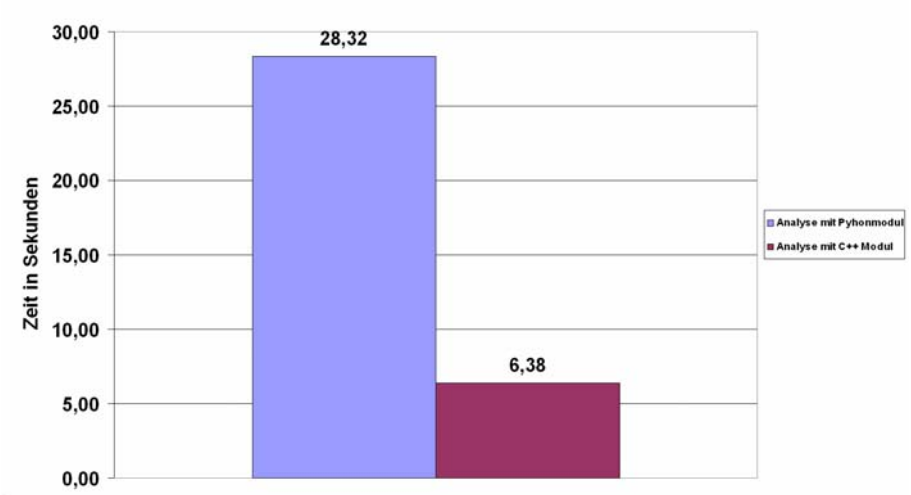


Abbildung 16: Laufzeitvergleich zwischen Python- und C++ Modulen. Modifizierte Analyse des W-Bosonzerfalls mit nur 3286 Rekonstruktionen.

ebenfalls 12 mal wiederholt. Die Konfiguration des Rechners bleit die gleiche, wie in den vergangenen Laufzeittests. Außerdem wird auch wieder die Zeit gemessen, die das Eingabemodul benötigt. Sie beträgt 11, 72 Sekunden.

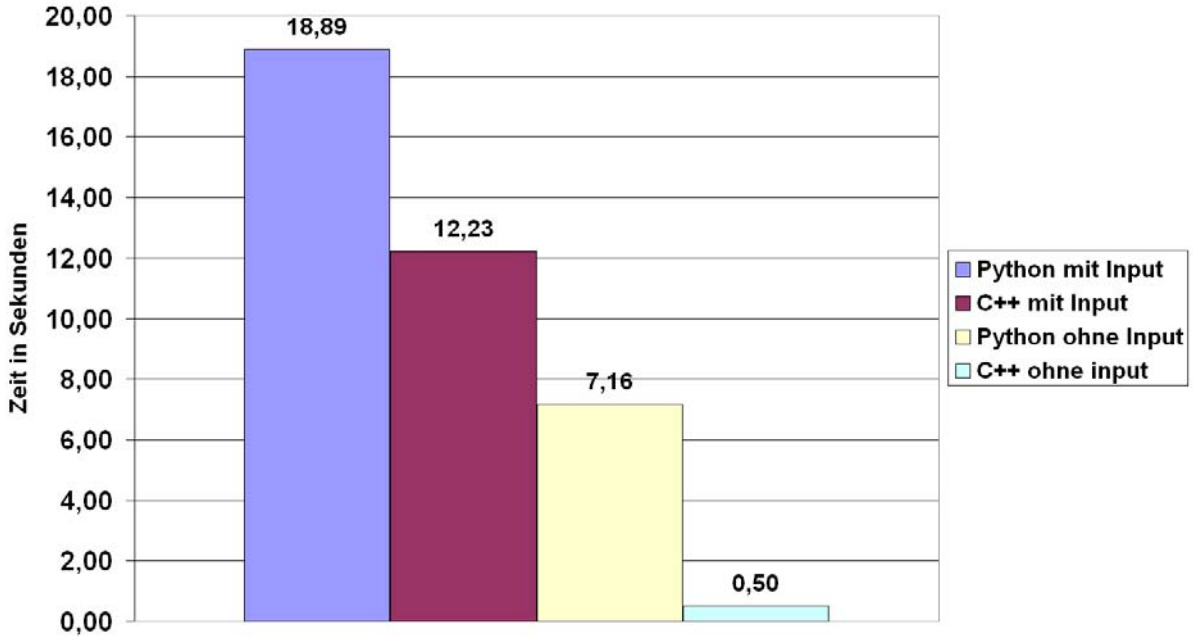


Abbildung 17: Laufzeitvergleich zwischen Python- und C++ Modulen. Berechnung des Sphärizitäts-Tensors, der Sphärizität und der Aplanarität. 10 solcher Berechnungen mit je 30000 UHECRs.

In Abbildung 17 kann man erkennen, dass die reine Geschwindigkeit des C++ Moduls etwa um den Faktor 14 schneller ist als die des Pythonmoduls. Wird jedoch die gesamte Analyse in Betracht gezogen, also bedeutet die Laufzeit des Eingabemodul mit eingerechnet, so ist der Faktor nur noch ca. 1,5. Wird noch ein Ausgabemodul hinzugefügt, sinkt dieser Faktor noch weiter auf ca 1,3. (Abbildung 18)

Als letzte Analyse für den Laufzeitvergleich von Python und C++ Modulen wird die Berechnung der *Regions of Interest* evaluiert. Die Analyse besteht aus drei Modulen, dem Eingabemodul, dem Python oder C++ Modul und dem Ausgabemodul. Die Kompressionsrate des Ausgabemoduls wurde hier auf 2 herabgesetzt. Die Bedingungen, wie Eingabedatei und Konfiguration des Rechners, sind die gleichen wie in der Analyse zuvor. Durchgeführt werden die Analysen mit einem Schwellwert der Energie von 50 EeV und Kegelradien von $0.10 - 0.30$ *Rad*. Wie in Abbildung 19 erkannt werden kann, weist das C++ Modul eine etwa 4 - 4,5 mal höhere Geschwindigkeit auf als das Pythonmodul.

5.2.4 Ursachen der Unterschiede in der Laufzeit von Analysen mit Python- und C++ Modulen

Die Unterschiede in der Laufzeit zwischen Pythonmodulen und C++ Modulen haben zwei Hauptursachen: Die erste Ursache ist, dass Python zu den Sprachen zählt, die von einem Interpreter ausgeführt werden. Das Pythonskript wird zur Laufzeit erst in einen Python-internen Bytecode übersetzt, welcher plattformunabhängig ist. Python muss allerdings nur bei der ersten Ausführung die Skriptdatei in Bytecode übersetzen. Danach ist der Bytecode in einer .pyc-Datei gespeichert. Dieser Bytecode wird dann von der *Python*

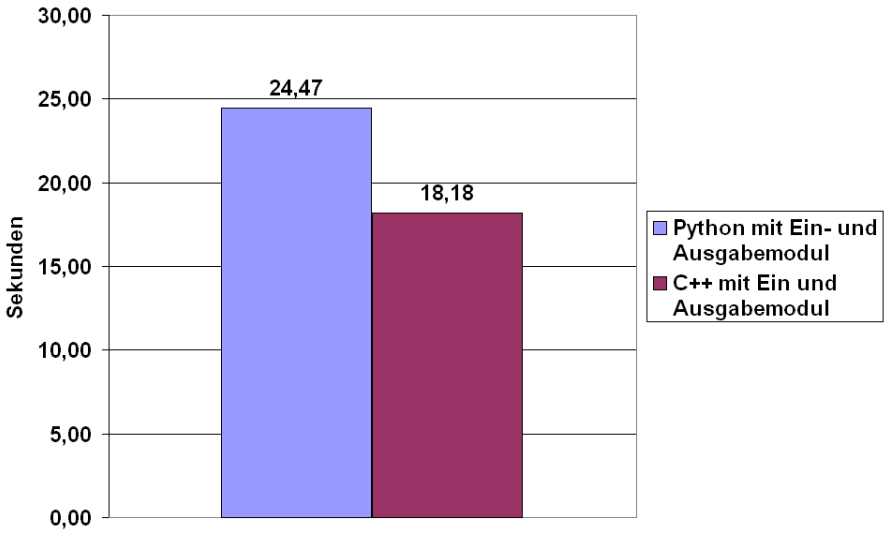


Abbildung 18: Laufzeitvergleich zwischen Python- und C++ Modulen. Analyse mit Ein- und Ausgabemodul. Berechnung des Sphärizitäts-Tensors, der Sphärizität und der Aplanarität.

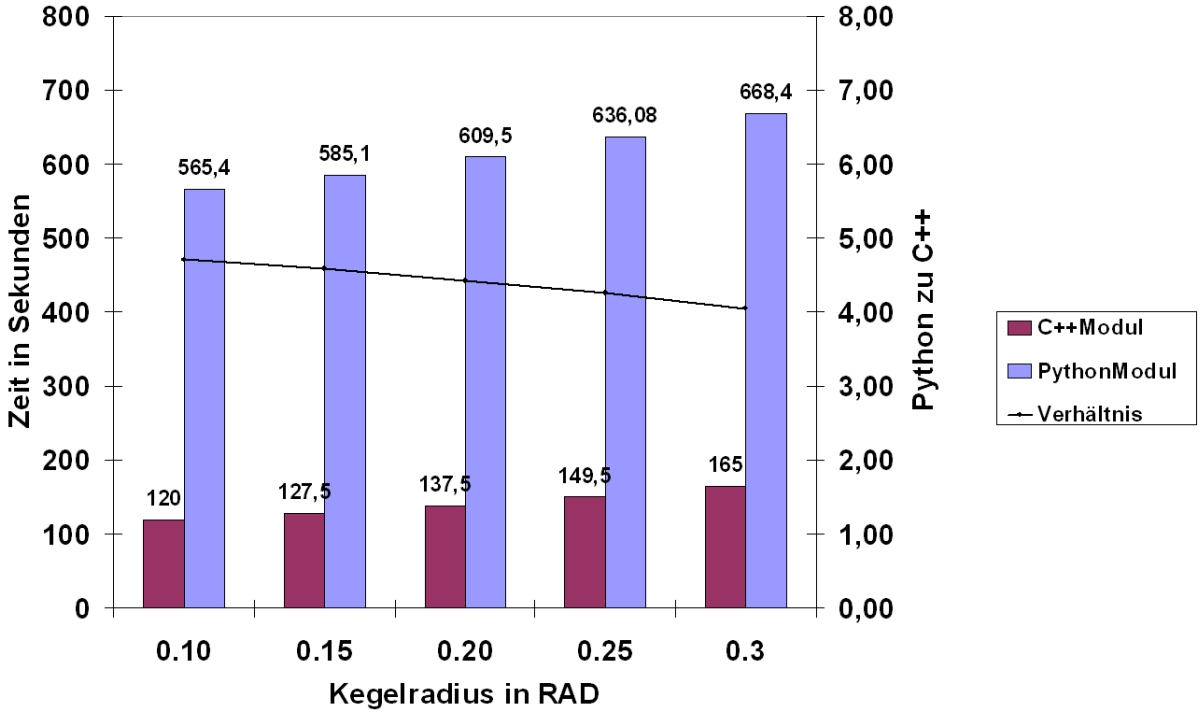


Abbildung 19: Laufzeitvergleich von Python- und C++ Modulen für das Aufspüren von *Regions of Interest*.

Virtual Machine (PVM) ausgeführt. Die PVM ist im Prinzip eine große Programmschleife, die über die Anweisungen im Bytecode iteriert und deren Befehle ausführt. C++ Compiler hingegen erzeuge, vor der Laufzeit den Maschinencode. Dieser Maschinencode kann vom Compiler für die jeweilige Plattform optimiert werden [Per]. Der Maschinencode wird direkt von der CPU ausgeführt, während der Bytecode erst von der PVM ausgeführt wird, was mehr Zeit kostet [LA07].

Die zweite Ursache für die längere Laufzeit ist, dass ein ein großer Teil der Zeit in der Schnittstelle zwischen Python und C++ verbraucht wird. Um heraus zu finden, wie groß dieser Zeitverbrauch ist, werden die vier Analysen aus Abschnit 5.2.3 während ihrer Ausführung mit dem Profiler-Programm *AQTime* überwacht. Vorher wird PXL mit Debuginformationen kompiliert, um genauere und detailliertere Ergebnisse mit AQTime zu bekommen. AQtime liefert den relativen Zeitbedarf für jede Klasse von PXL. Überwacht werden alle *shared Librarys* von PXL sowie die *pxlrun* selbst. Weiterhin werden alle von SWIG generierten .pyd-Dateien, die als Schnittstelle zwischen VISPA und PXL dienen, überwacht. Die zeitliche Verteilung, der bei dem Aufruf des Pythonmoduls beteiligten Funktionen, sieht man bei den Abbildungen 20, 21, 22 und 23.

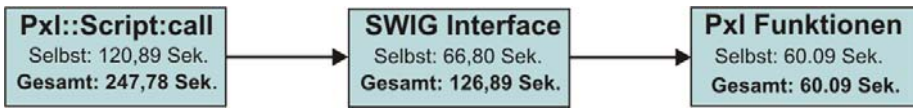


Abbildung 20: Zeitverteilung des Pythonmoduls bei der Analyse eines W-Bosonzerfalls aus Abschnitt 5.2.3. *Selbst:* bedeutet wie viel Zeit die Funktion selber benötigt hat. *Gesamt:* bezeichnet die Zeit, die die Funktion selbst und alle Unterfunktionen zusammen benötigt haben.

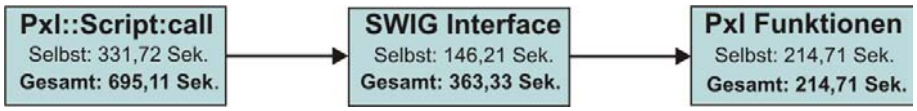


Abbildung 21: Zeitverteilung des Pythonmoduls bei der Analyse eines Top-Quarkzerfalls aus Abschnitt 5.2.3.

Abbildung 22: Zeitverteilung des Pythonmoduls bei der Analyse zur Berechnung der Sphärizität aus Abschnitt 5.2.3.

Bei der Zeit, die das Pythonmodul bei der Analyse des W-Bosonzerfalls benötigt, enfällt ca. 26% auf die von SWIG generierte Schnittstelle zwischen Pythonmodul und PXL. Bei der Analyse des Top-Quarkszerfalls sind es ca. 21%, bei der Analyse der Sphärizität und Region Of Interest sogar ca. 41%.

Abbildung 23: Zeitverteilung des Pythonmoduls bei der Analyse zur Berechnung der *Region of Interests* aus Abschnitt 5.2.3.

5.2.5 Zeitverbrauch der einzelnen Module

Wenn man die Zeit, die die einzelnen Analysen aus Abschnitt 5.2.3 benötigen, anhand der durchgeführten Laufzeitmessungen auf die Module umrechnet, ergibt sich folgendes Bild. Aus Abbildung 24 und 25 ist ersichtlich, dass das Ausgabemodul, welches die Daten in eine Datei schreibt, $40\% - 70\%$ der Zeit benötigt. Besonders wenn bei den Analysen große Datenmengen erzeugt werden, wie bei der Rekonstruktion von Teilchenzerfällen, beansprucht das Ausgabemodul viel Zeit. Deshalb lohnt es sich, das Speichern der Daten etwas genauer zu betrachten. Dazu nehmen wir im folgenden Abschnitt das Ausgabemodul etwas genauer unter die Lupe.

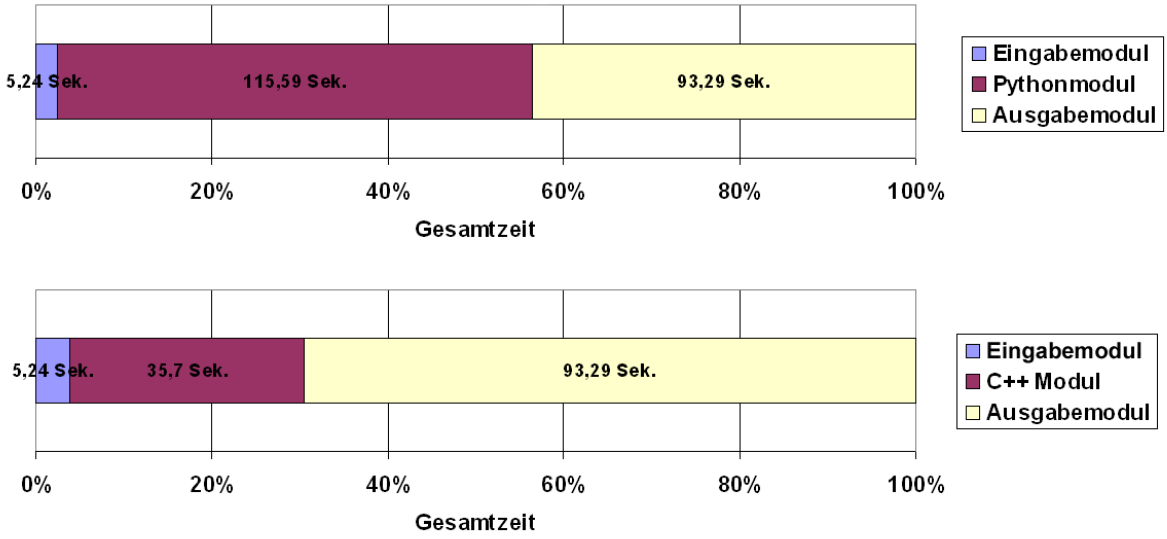


Abbildung 24: Zeit, die die einzelnen Module bei der Analyse des W-Bosonzerfalls benötigt haben.

5.2.6 Speicherung der Daten

Das Speichern der Daten in einer Datei übernimmt das Ausgabemodul. Dies geschieht in drei großen Schritten. Als erstes werden die Objekte serialisiert. Die serialisierten Daten werden zuerst in einen Puffer geschrieben. Ist die Puffergrenze erreicht, erfolgt die Kompression des Pufferinhaltes mit zlib. Als letztes werden dann die komprimierten Daten in die Datei geschrieben. Um den Zeitverbrauch der einzelnen Schritte zu bestimmen, werden wieder eine Reihe von Laufzeitmessungen durchgeführt. Für die Laufzeitmessungen werden die Analysen des W-Boson- und des Top-Quarkzerfalls aus Abschnitt 5.2.3 benutzt. Die Analysen bestehen aus dem Eingabemodul, dem C++ Modul und dem Ausgabemodul. Der Kompressionsgrad im Ausgabemodul ist standardmäßig auf 6 gesetzt.

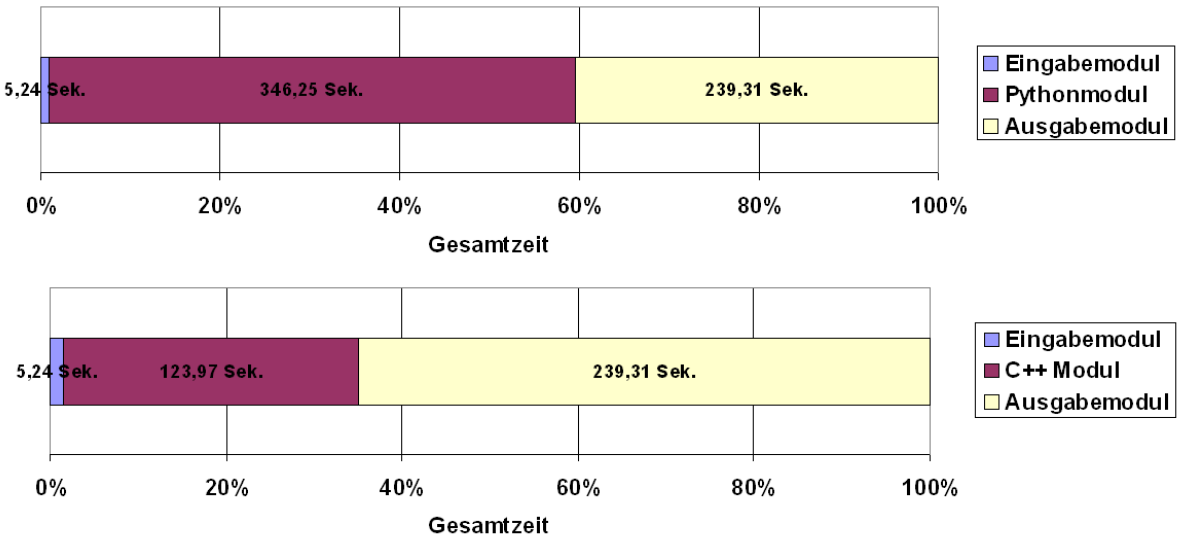


Abbildung 25: Zeit, die die einzelnen Module bei der Analyse des Top-Quarkzerfalls benötigt haben.

Der Quellcode von PXL wurde dahingehend verändert, dass alle Befehle, die das physikalische Schreiben in eine Datei betrafen, deaktiviert bzw. auskommentiert wurden. Nun wurde die Laufzeit der Analysen gemessen. Im nächsten Schritt wurde nun zusätzlich das Komprimieren der Daten auskommentiert, so dass das Ausgabemodul nur noch die eingehenden Objekte serialisiert hat. PXL wurde wieder neu kompiliert und die Laufzeittests mit den Analysen erneut ausgeführt. Auch hier wurden die Messungen jeweils 10 mal wiederholt. Aus der Differenz der gemessenen Laufzeiten ergab sich, wie man in Abbildung 26 und 27 sehen kann, folgende Aufteilung des Zeitverbrauchs der drei Schritte.

Bei der ersten Analyse benötigt das Ausgabemodul insgesamt 93, 29 Sekunden. Davon entfällt auf das physikalische Schreiben der Daten etwa 1, 67 Sekunden. Dies entspricht ca. 1, 8% der Gesamtzeit. Für das Serialisieren der Daten wurden 21, 57 Sekunden benötigt. Das sind etwa 23% der Gesamtzeit. Das Komprimieren der Daten nimmt mit 70, 05 Sekunden und 75% den größten Teil in Anspruch.

Bei der zweiten Analyse sieht es ähnlich aus. Die Zeit, die das Ausgabemodul bei dieser Analyse benötigt, beträgt 239, 31 Sekunden. Das physikalische Schreiben in eine Datei nimmt 5, 14 Sekunden in Anspruch, und damit 2, 2% der Gesamtzeit. Beim Serialisieren sind es 22, 8%. Den größten Anteil benötigt auch hier die Kompression mit ca. 75%.

Da die Kompression die meiste Zeit im Ausgabemodul in Anspruch nimmt, und sich eine Optimierung hier am meisten lohnen würde, betrachten wir die Kompressionsmethode etwas genauer.

5.2.7 Evaluation der Datenkompression im Ausgabemodul

Die Funktion mit der das Ausgabemodul die Daten komprimiert ist *compress2(Bytef *dest, uLongf *destLen, const Bytef *source, uLong sourceLen, int level)* aus der zlib-Bibliothek. Die Parameter der Funktion haben dabei folgende Bedeutung. **dest* Zielpuffer für die komprimierten Daten,**destLen* Länge des Zielpuffers, **source* Quellpuffer der zu komprimierenden Daten, *sourceLen* Länge des Quellpuffers und *level* der Kompressionsgrad. Dabei bedeutet ein Kompressionsgrad von 1 die schnellste Kompression und ein Kompressionsgrad von 9 die höchste Kompression. Ein Wert von 0 bedeutet keine

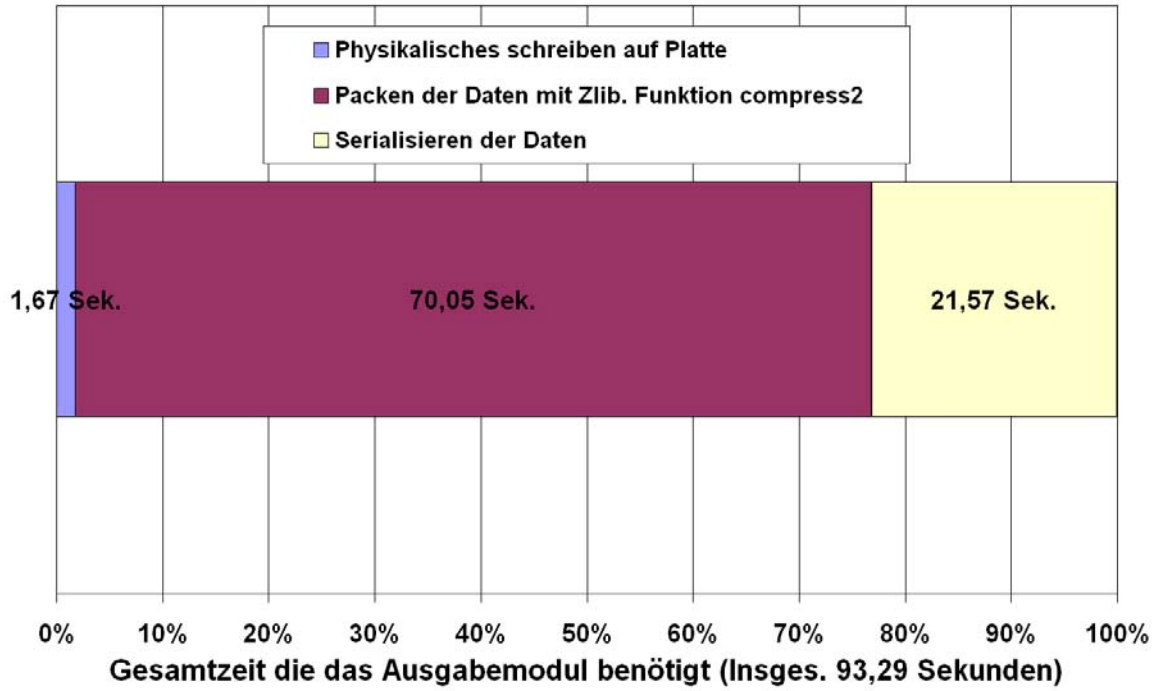


Abbildung 26: Zeitverteilung im Ausgabemodul mit Standardkompressionsgrad 6, bei der Analyse des W-Bosonzerfalls.

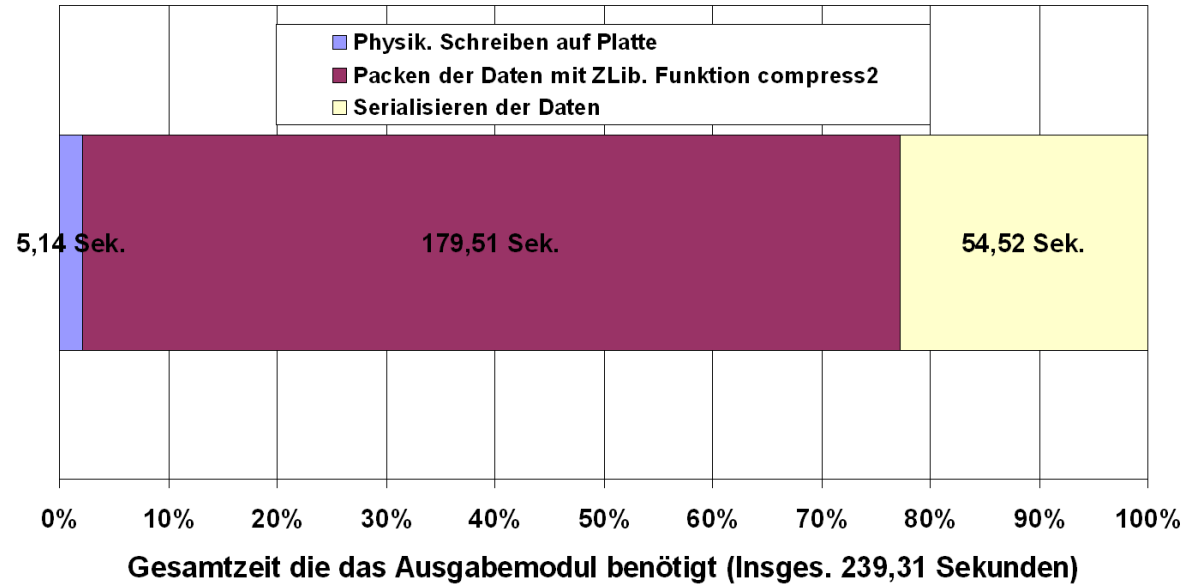


Abbildung 27: Zeitverteilung im Ausgabemodul mit Standardkompressionsgrad 6, bei der Analyse des Top-Quarkzerfall.

Kompression der Daten. Als erstes wird die Auswirkung des Kompressionsgrades, auf das Zeit-Kompressionsverhältnis evaluiert. Für alle 11 Einstellungen der Kompression, nämlich Kompressionsgrad 0 bis 9 und nicht ausgeführter Kompressionsbefehl, werden jeweils 10 Laufzeittests vorgenommen. Für die Laufzeittests werden die Analysen, des W-Bosonzerfalls und des Top-Quarkzerfalls aus Abschnitt 5.2.3 unverändert übernommen. Es handelt sich dabei jeweils um die Analysen mit C++ Modul. Von der dann gemessenen Zeit wird die Zeit abgezogen, die das Eingabemodul und das C++Modul benötigen. Die für die Serialisierung eingestellte Größe des Datenpuffers, der auch die Größe des Quellpuffers für die *compress()*-Funktion entspricht, liegt bei standardmäßig 1MB.

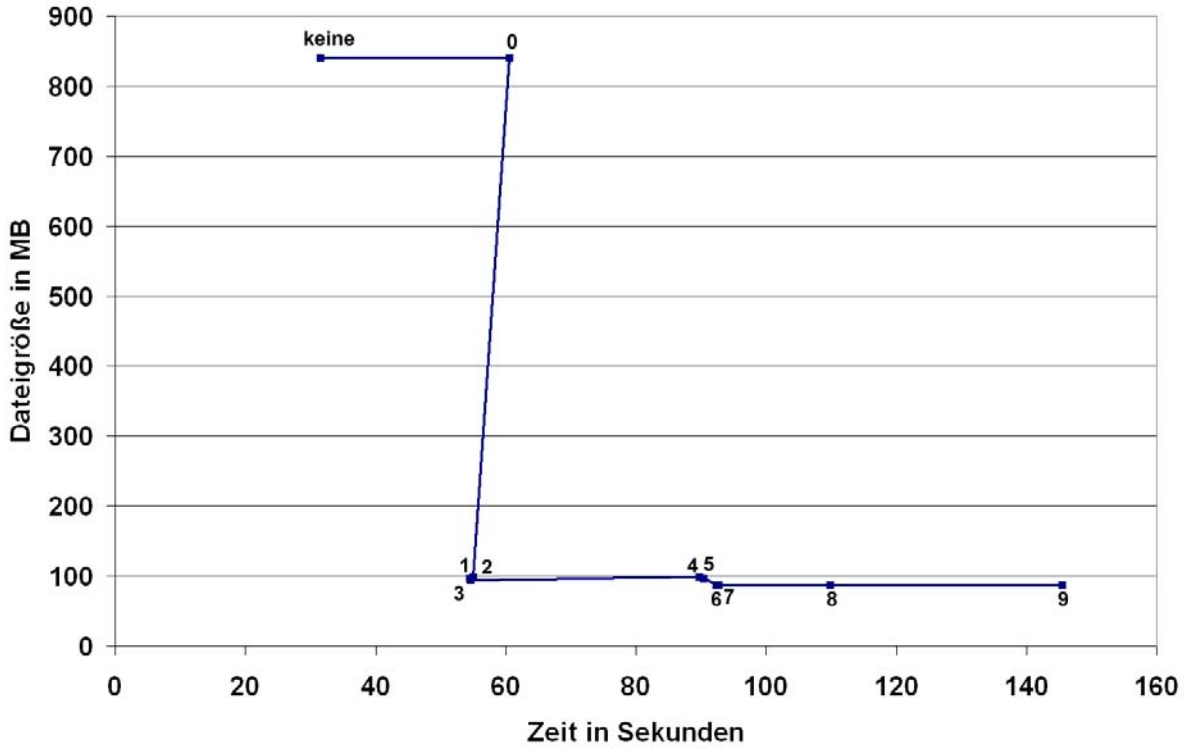


Abbildung 28: Zeit-Größenverhältnis der Komprimirung bei den erzeugten Daten der Rekonstruktuionsanalyse des W-Bosonzerfalls. Die Nummern an den Punkten geben den Kompressionsgrad an.

Aus den Abbildungen 28 und 29 ergibt sich, dass die optimalen Werte für den Kompressionsgrad, im Bezug auf den Zeit-Kompressionsfaktor für die beiden Analysen, zwischen 1 und 3 liegen. Als nächstes wird untersucht, wie groß die Auswirkungen, der Größe des Quellpuffers, auf den Kompressionsfaktor und auf die Laufzeit sind. Dazu werden wieder mit den gleichen Analysen Laufzeittests mit verschieden großen Puffern und Kompressionsgraden, vorgenommen. Es werden die Kompressionsgrade 2, 6, 9 mit Puffergrößen von jeweils 16KB, 32KB, 64Kb, 1MB und 10MB untersucht. Allerdings konnten hier keinen nennenswerten Unterschiede, sowohl in der Laufzeit als auch in der Dateigröße, festgestellt werden. Alle Unterschiede liegen im Rahmen der Messungenauigkeit von ca. 1-2 Sekunden. Der Unterschied der Größe der Datei liegt bei unter 2%.

Nun gibt es die Möglichkeit den Objekten, wie zum Beispiel *pxl:Event*, *pxl:EventViews* oder *pxl:Particle*, so genannte **UserRecords** anzuhängen. Ein **UserRecords** ist nichts weiter, als eine zweistellige Relation, mit der Informationen an das Objekt angehangen

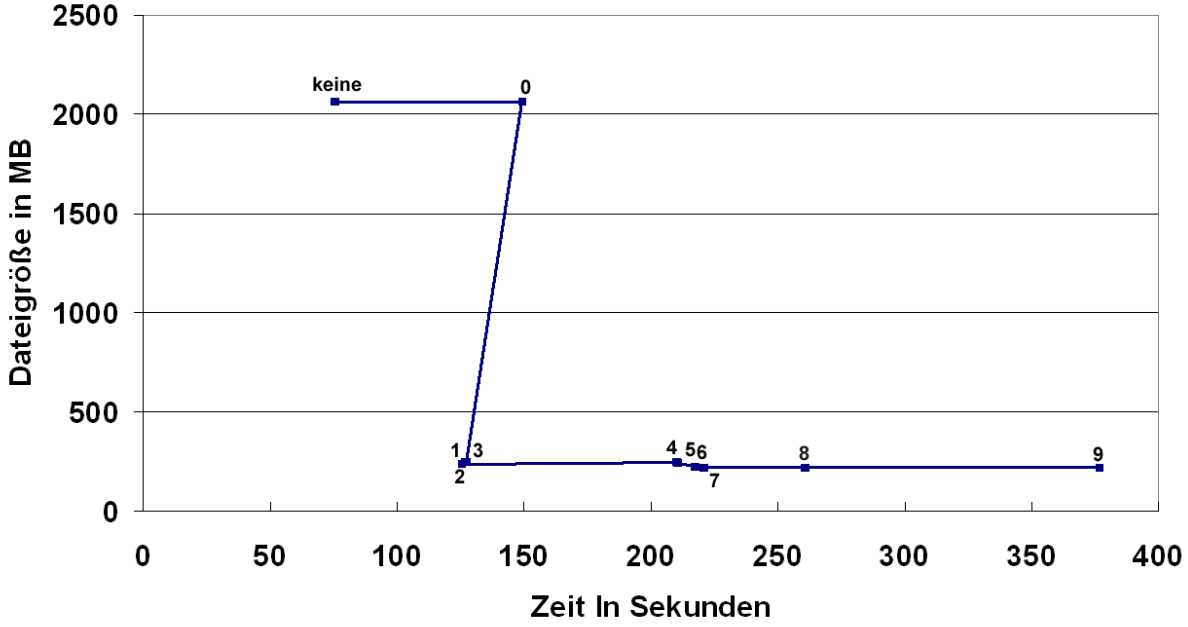


Abbildung 29: Zeit- Größenverhältnis der Komprimierung bei den erzeugten Daten der Rekonstruktuionsanalyse des Top-Quarkzerfalls.

werden können. Die beiden Stellen der Relation sind Name des *UserRecords* und Inhalt. Diese können beliebig gesetzt werden. Der Name des *UserRecords* darf innerhalb eines Objektes allerdings nur einmal auftauchen. Mit dem *UserRecord* ist es also möglich, die Redundanz der Daten zu erhöhen oder zu verkleinern. Die Frage ist nun, ob und in wie weit dies Auswirkungen auf den Zeit-Kompressionsfaktor und den Kompressionsgrad hat.

Um dies zu evaluieren, werden zwei Analysen erstellt. Die Analysen bestehen jeweils aus einem Generatormodul und einem Ausgabemodul. Bei der einen Analyse erzeugt das Generatormodul Objekte mit hoher Redundanz an Daten, bei der anderen Analyse Objekte mit möglichst kleiner Redundanz an Daten. Die Daten, die vom Generatormodul in der ersten Analyse erzeugt werden, sehen wir folgt aus.

- Erzeugte Objekte (Events) vom Typ *pxl:Event* 2000.
- Erzeugte Objekte (EventsViews) vom Typ *pxl:EventView* je Event 100, alle mit Namen *Autoprocess*.
- Erzeugte UserRecords je Eventview 10, Namen der UserRecords *UserRecord-i* (i = 1..20). Alle mit Inhalt der Zeichenkette *CMS-DetectorReconstructed*.
- Erzeugte Objekte (Particles) vom Typ *pxl:Particle* 3 je EventView.
- Erzeugte UserRecords pro Particle 25, Namen der UserRecords *UserRecord-i*, (i = 1..20). Alle mit Inhalt der Zeichenkette *CMS-DetectorReconstructed*.

Die beiden Analysen, werden auf einem Rechner mit folgender Konfiguration ausgeführt.(Tabelle 5).

Die Verlaufskurve des Zeit-Kompressionsfaktor, ist ähnlich wie in den beiden vorangegangen. Der beste Wert für den Kompressionsgrad ist wieder 1-3. Die Änderung der Puffergröße, auf 16KB, 32KB, 64Kb, 1MB und 10MB bei den Kompressionsgrade 2, 6, 9 bringen ebenfalls nur minimale Unterschiede. Die Unterschiede in der Laufzeit sind mit

- Betriebssystem Windows 7 Professional 64Bit
- CPU: Intel Core2 Duo P8700 2,53 GHz
- RAM: 4GB
- Festplatte: 500GB WDC WD5000BEVT-22ZAT0 ATA Device

Tabelle 5: Notebook 2.

nur 1-2 Sekunden wieder im Rahmen der Messungenauigkeit. Die Varianz der Dateigröße liegt unter 2%.

In der zweiten Analyse erzeugt das Generatormodul folgende Daten.

- Erzeugte Objekte (Events) vom Typ *pxl:Event* 2000.
- Erzeugte Objekte (EventsViews) vom Typ *pxl:EventView* je Event 100. Bei allen ist der Name aus Groß-und Kleinbuchstaben zufällig generiert und 11 Zeichen lang.
- Erzeugte UserRecords je EventView 10, Namen der UserRecords zufällig generiert. Inhalt zufällig generiert und 26 Zeichen lang.
- Erzeugte Objekte (Particles) vom Typ *pxl:Particle* 3 je EventView.
- Erzeugte UserRecords pro Particle 25, Namen der UserRecords zufällig generiert. Inhalt zufällig generiert und 26 Zeichen lang.

In Abbildung 31 ist die Verlaufskurve etwas anders als sonst. Der Kompressionsgrad 0 benötigt, relativ zu den anderen Kompressionsgraden, weniger Zeit, als bei den vorangegangenen Testdaten. Die Optimalen Werte des Kompressionsgrad für den Zeit- Kompressionsfaktor sind 1-4 und nicht 1-3 wie sonst. Die Änderung der Puffergröße bringt, wie in den vorangegangenen Testläufen, keinen messbaren Unterschied.

5.2.8 Parallelisierung von Analysen

Viele Prozessoren besitzen heute schon zwei oder mehr Kerne. Deshalb ist die Frage interessant, ob und wie dies mit VISPA genutzt werden kann. PXL, welches ja für die Ausführung der Analyse verantwortlich ist, unterstützt zur Zeit keine Mehrprozessorsysteme. Die einzige Möglichkeit mehrere Prozessoren zu nutzen, ist dann die Analyse aufzuteilen. Möchte der Anwender zum Beispiel in der Hochenergiephysik 1000 Ereignisse untersuchen, so kann er die Analyse, in zwei Analysen zu je 500 Ereignissen aufteilen. Diese zwei Analysen kann der Anwender dann parallel auf einem Rechner mit einem Mehrprozessorsystem laufen lassen. Enthalten die Analysen jedoch Ausgabemodule, so entstehen bei den Analysen auch zwei Ausgabedateien. Sollen die beiden Dateien in eine Datei zusammengeführt werden, so ist eine dritte Analyse notwendig, die dies ausführt. Um die Zeitersparnis, die bei der Aufteilung von Analysen gewonnen wird, zu erhalten, wird wieder die Analyse des W-Bosonzerfalls zur Evaluation herangezogen. Der Kompressionsgrad im Ausgabemodul wird allerdings von der Standardkompression 6 auf den Kompressionsgrad 2 umgestellt. Die Analyse wird einmal mit allen 3200 Ereignissen ausgeführt. Danach wird die Analyse in zwei Analysen zu je 1600 Ereignissen aufgeteilt, die dann parallel ausgeführt werden. Die parallele Ausführung wird mit einem kurzen C-Programm, unter Verwendung des *ShellExecute*-Befehls, realisiert. Damit ist ein nahezu

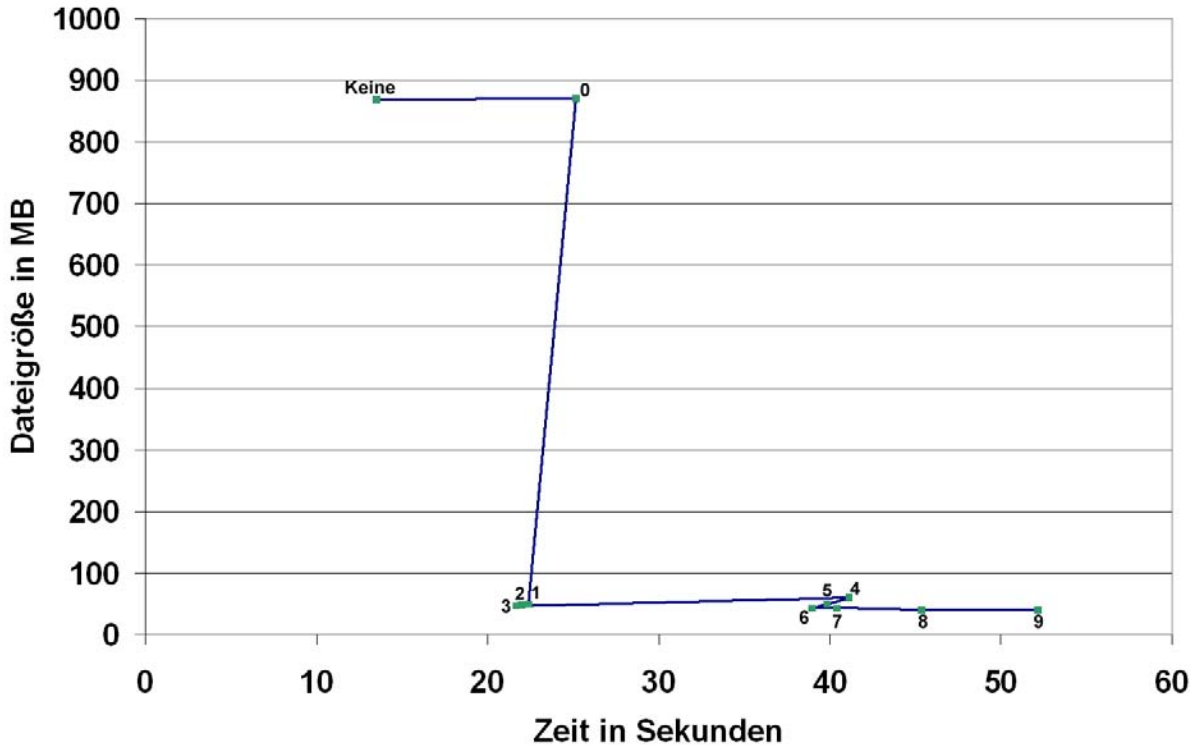


Abbildung 30: Zeit-Größenverhältnis der Komprimirung von Daten mit sehr viel Redundanz.

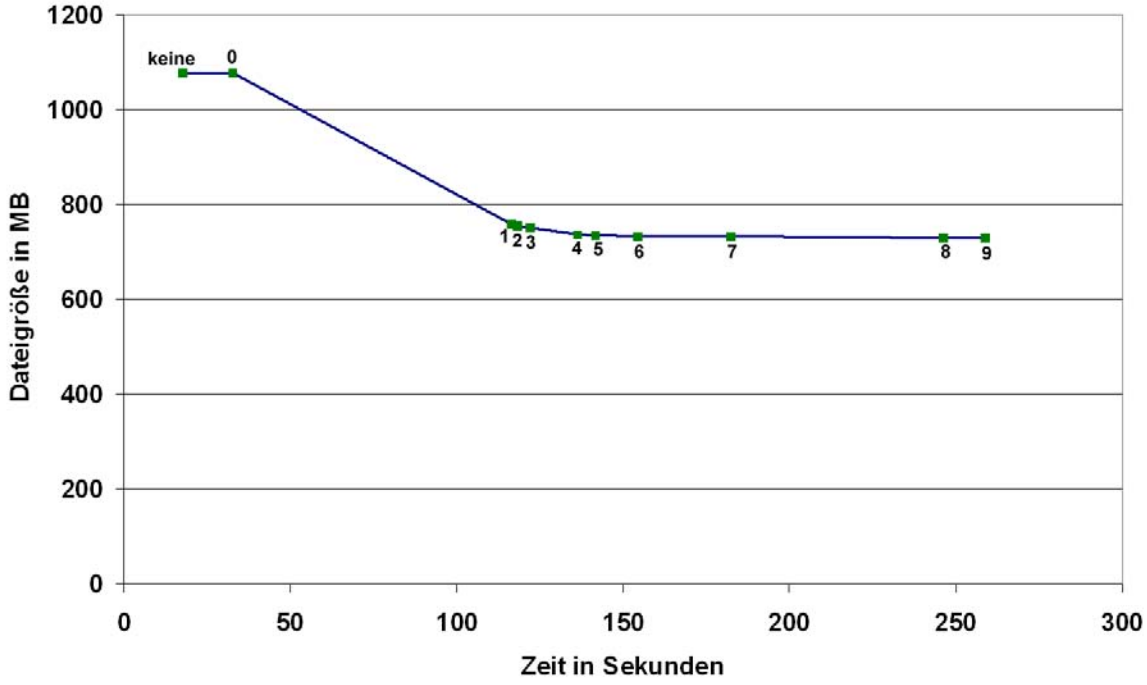


Abbildung 31: Zeit-Größenverhältnis der Kompression von Daten mit sehr wenig Redundanz.

gleichzeitiger Start der beiden aufgeteilten Analysen gewährleistet. Für die aufgeteilten Analysen, die parallel ausgeführt werden, erhält man logischerweise jedes mal zwei Ergebnisse der Laufzeitmessung. Dabei wird als Laufzeit die größere von beiden angesehen. Beide unterscheiden sich jedoch nie mehr als $3 \cdot 10^{-1}$ Sekunden.

Weiterhin wurde eine Laufzeitmessung von der Analyse durchgeführt, die beide Dateien, die bei der Aufteilung der Analysen des W-Bosonzerfalls entstehen, zu einer Datei zusammenführt. Diese Analyse besteht aus zwei Eingabemodulen mit den jeweiligen Dateien und dem Ausgabemodul. Siehe Abbildung 32.

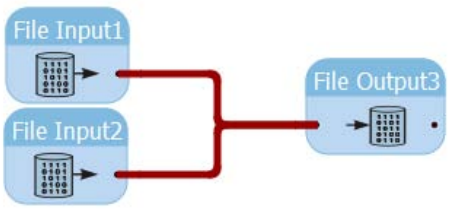


Abbildung 32: Analyse zum Zusammenführen von zwei Dateien.

Der Rechner auf dem die Laufzeitmessungen durchgeführt wird, hat folgende Konfiguration. (Tabelle 6).

- Betriebssystem: Windows XP 64 bit
- CPU: AMD Athlon64 X2 DualCore 5600 2,8GHz
- RAM: 4GB
- Festplatte: Western Digital WDC WD2500KS-00MJB0

Tabelle 6: Desktop 1.

Die Laufzeitmessungen wird jeweils 10 mal wiederholt. Wie in Abbildung 33 erkannt werden kann, halbiert sich fast die Zeit, wenn die Analyse in zwei Analysen aufgeteilt wird. Möchte man allerdings die beiden Dateien, die bei der Aufteilung der Analyse entstanden sind, in einer weiteren Analyse zusammenführen, so dauert das Zusammenführen schon mehr als doppelt solange, wie die ursprüngliche Analyse mit den 3200 Ereignissen benötigt hätte.

Eine weitere Analyse, an der diese Herangehensweise getestet wird, ist die Berechnung der *Regions of Interest*. Hier wird die Analyse wieder in zwei andere Analysen aufgeteilt. Die erste Analyse enthält die ersten 5 Objekte von Typ BasisContainer, die zweite Analyse die anderen 5 Objekte. Die Analysen enthalten jeweils ein Eingabemodul, dass C++ Modul zur Berechnung der *Regions of Interest* und ein Ausgabemodul. Es entstehen also wieder zwei Ausgabedateien. Die Parameter für die das Auffinden der *Regions of Interest* sind Schwellwert der Energie von 50 EeV und Kegelradius von 0.15 rad. Die beiden Analysen werden wieder parallel gestartet. Als Laufzeit der parallelen Ausführung der Analysen, wird die längere der beiden Laufzeiten der Analysen angesehen. Mit 44,38 Sekunden ist die Laufzeit, gegenüber der nicht aufgeteilten Analyse mit 83,13 Sekunden,

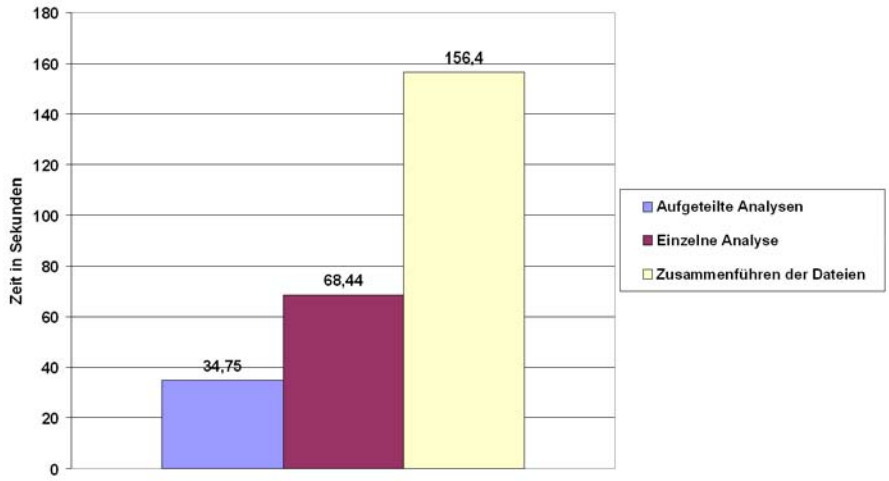


Abbildung 33: Laufzeit einer einzelnen Analyse und, einer aufgeteilten Analyse sowie einer Analyse für das Zusammenführen zu einer Datei.

fast halbiert. Das Zusammenführen der zwei Ausgabedateien zu einer Datei kosten 21,52 Sekunden an Laufzeit. Also ergibt sich hier eine Gesamtlaufzeit von 65,90 Sekunden. Damit ergibt sich eine Zeiterpsrniss von 17,23 Sekunden.

5.3 Evaluation der Erstellung und des Kopierens von Objekten

Bei vielen Analysen, besonders bei Rekonstruktionen von Teilchenzerfällen, ist es notwendig, neue Objekte wie Teilchen und Ereignisansichten anzulegen und diese untereinander zu verlinken. Deshalb lohnt es sich, einen Blick auf den Zeitverbrauch der dafür zuständigen PXL-Funktionen zu werfen.

5.3.1 Zeitverbrauch der zuständigen PXL-Funktionen

Betrachtet man den internen Zeitverbrauch des C++ Analysemoduls bei der Rekonstruktion des W-Bosonzerfalls oder des Top-Quarkzerfalls mit dem Programm AQTime, fällt auf, dass ein Großteil der Zeit für das Erstellen von Objekten, das Einfüllen von Objekten in andere Objekte und das Verlinken von Objekten benötigt wird. Insgesamt sind es bei beiden Analysen mehr als 96 % der Zeit, die das Analysemodul benötigt. Eine genaue Auflistung der einzelnen PXL-Funktionen und deren relativen Zeitverbrauch sieht man in Tabelle 7 und 8. Zum Erzeugen von Objekten werden in den beiden Analysen zwei verschiedene Methoden verwendet. Die Jets werden mit dem Konstruktor der Klasse *pxl::Particle* erstellt und dann mit der Funktion *EventView::setObject* in die Ereignisansicht eingefügt. Das W-Teilchen und das Top-Quark werden mit der Funktion *EventView::create<pxl::Particle>* direkt in der Ereignisansicht erstellt.

Um die totale Geschwindigkeit der Erstellung von Objekten herauszufinden, wird ein C++ Generatormodul entwickelt, welches eine bestimmte Anzahl von Objekten erzeugt. Dieses Modul wird dann einem Laufzeittest unterzogen. Für den Laufzeittest werden mehrere Analysen erstellt, die jeweils nur das C++ Generatormodul enthalten. Das Generatormodul erzeugt eine bestimmte Anzahl von Ereignissen *pxl::Event*, die eine gewisse Anzahl von Ergeignisansichten *pxl::Eventviews* enthalten. Diese Ereignisansichten enthalten wiederum eine bestimmte Anzahl von Teilchen *pxl::Particle*. Keines der Objekte enthält irgendwelche zusätzlichen Einträge wie *pxl::UserRecords* . Ausgeführt werden die Testläufe

Gesamtzeit des C++ Analysemoduls 71,23 Sekunden. (W-Bosonzerfall)	
PXL-Funktion	Zeit (Sek.)
pxl:Particle::Particle	26,66 s
pxl:ObjectOwner::create<pxl::Particle>	19,47 s
pxl::Particle::setP4FromDaugthers	10,13 s
pxl::ObjectManager::setObject	7,02 s
pxl::Realtive::linkDaugther	5,21 s
pxl::Relative::setName	0,34 s
pxl::EventView::EventView	0,11 s
Gesamtzeit der PXL-Funktionen	68,94 s

Tabelle 7: Zeitverbrauch der einzelnen PXL-Funktionen beim W-Bosonzerfall.

Gesamtzeit des C++ Analysemoduls 187,48 Sekunden. (Top-Quarkzerfall)	
PXL-Funktion	Zeit (Sek.)
pxl:ObjectOwner::create<pxl::Particle>	60,96 s
pxl:Particle::Particle	56,03 s
pxl::Particle::setP4FromDaugthers	28,53 s
pxl::ObjectManager::setObject	21,04 s
pxl::Realtive::linkDaugther	14,71 s
pxl::Relative::setName	0,90 s
pxl::EventView::EventView	0,01 s
Gesamtzeit der PXL-Funktionen	182,19 s

Tabelle 8: Zeitverbrauch der einzelnen PXL-Funktionen Top-Quarkzerfall.

auf dem *Notebook1*. Für die Erzeugung der Objekte wurden jeweils die Klassenkonstruktoren verwendet sowie die Funktion *pxl::ObjectManager::setObject* für das Einfüllen in andere Objekte. Verwendet man stattdessen die Funktion *pxl:ObjectOwner::create<pxl::Object>* ist die Geschwindigkeit der Erstellung etwa um 10 % höher. In Tabelle 9 kann die Lauftzeit des Generatormoduls in Abhängigkeit der Anzahl der generierten Teilchen abgelesen werden sowie die Rate der generierten Teilchen pro Sekunde. Es fällt auf, dass mit steigender Anzahl von Teilchen pro Ereignis, die Erstellungsrate von Teilchen sinkt. Eine Ursache dafür ist, dass erst nach jedem durchgereichten Ereignis der Speicher bereinigt wird. Des Weiteren werden die Teilchen innerhalb des Ereignis nach ihrer Id-Nummer sortiert. Eine größere Anzahl von Teilchen in einem Ereignis bedeuten einen größeren Zeitaufwand für das Sortieren.

Ereignisse	$\frac{Ereignisansicht}{Ereign.}$	$\frac{Teilchen}{ErgAns.}$	Teilchen gesamt	Zeit (Sek.)	$\frac{Teilchen}{Sek.}$
1000	1	500	$5 \cdot 10^5$	7,07 s	70721
1000	1	1000	10^6	15,16 s	65963
1000	1	2000	$2 \cdot 10^6$	15,16 s	62972
1000	1	5000	$5 \cdot 10^6$	85,11 s	58748
1000	1	10000	10^7	175,87 s	56860
1000	1	20000	$2 \cdot 10^7$	361,41 s	55339

Tabelle 9: Zeitverbrauch beim Generieren von Teilchen.

5.3.2 Laufzeit eines Kopiermoduls

Ein Kopiermodul zu erstellen ist aus zwei Gründen interessant. Der erste ist, dass es nur so möglich ist in einer Analyse eine Kopie von Objekten wie Ereignissen herzustellen und diese unabhängig voneinander zu bearbeiten. Der zweite Grund ist, dass man etwas generell über die Geschwindigkeit des Kopierens von Objekten erfährt. Das Kopiermodul macht eine exakte Kopie aller Objekte, ihres Inhalts und ihrer Verlinkungen. Einzig die eindeutige Id-Nummer der Objekte ändert sich. Zur Messung der Laufzeit werden zwei Analysen erstellt. Sie bestehen aus dem C++ Generatormodul und dem Kopiermodul. Beide erzeugen 1000 Ereignisse mit je einer Ereignisansicht und 1000 Teilchen je Ereignisansicht. In der einen Analyse enthält jedes Objekt noch 10 zusätzlichen Einträge (*pxl::UserRecords*) mit einer Länge von 25 Zeichen. Das entspricht etwa 250 MB an zusätzlichen Daten. Ohne die zusätzlichen Einträge benötigt das Kopiermodul 14,88 Sekunden. Mit den zusätzlichen Einträgen beträgt die Kopierzeit 16,41 Sekunden. Der Grund für den geringen Zeitunterschied ist, dass der *UserRecord* einen *Copy-on-Write*-Mechanismus besitzt. Das bedeutet, dass nur die Zeiger auf den *UserRecord* kopiert werden. Der *UserRecord* merkt sich die Anzahl der Zeiger, die auf ihn verweisen. Erst wenn der Inhalt des *UserRecords* von einem Zeiger geändert wird, wird eine Kopie von ihm angelegt.

5.4 Evaluation des Autoprozessmoduls

Der Autoprozess erzeugt ähnliche Daten wie die Analysen der Teilchenrekonstruktionen in Abschnitt 5.2.3. Einziger Unterschied ist, dass die möglichen Rekonstruktionen der Teilchenzerfälle nicht alle in eine Ereignisansicht geschrieben werden, sondern für jede mögliche Rekonstruktion wird eine eigene Ereignisansicht angelegt.

5.4.1 Geschwindigkeit des Autoprozessmoduls

Für die Evaluation der Geschwindigkeit des Autoprozessmoduls wird wieder die Rekonstruktion des W-Bosonzerfall gewählt. Die erstellte Analyse für den Laufzeittest besteht aus dem Eingabemodul, dem Autoprozessmodul und dem Ausgabemodul. Damit man einen genauen Vergleich zwischen dem Autoprozess und der Analyse des W-Bosonzerfalls aus Abschnitt 5.2.3 hat, wurde diese Analyse leicht modifiziert. Sie erzeugt jetzt, genau wie der Autoprozess, eine Ereignisansicht pro möglicher Rekonstruktion und damit die selben Daten. Die Laufzeit der Analysen wird wieder mit und ohne Ausgabemodul gemessen. Die Option **Match** im Autoprozessmodul, zum Auffinden der besten Rekonstruktion, wird deaktiviert. Die Eingabedatei und der Rechner, auf dem die Evaluation durchgeführt wird, sind die gleichen wie in Abschnitt 5.2.3.

Aus der Abbildung 34 kann erkannt werden, dass der Autoprozess eine leicht längere Laufzeit besitzt. Im Vergleich zur modifizierten Analyse des W-Bosonzerfalls, ist die Analyse mit dem Autoprozess um den Faktor $1,12$ langsamer mit Ausgabemodul, und um den Faktor $1,28$ langsamer ohne Ausgabemodul. Würden alle möglichen Rekonstruktionen, wie in der originalen Analyse des W-Bosonzerfalls, in eine Ereignisansicht geschrieben, wäre der Autoprozess um die Faktoren $1,34$ und $1,88$ langsamer. Ist die Option **Match** im Autoprozessmodul aktiviert, verschlechtert sich die Laufzeit in der Analyse mit Ausgabemodul nur um ca. 2 Sekunden. Da bei der Analyse mit dem Autoprozessmodul, wie es zu erwarten war, auch wieder das Ausgabemodul die meiste Zeit in Anspruch nimmt, wird auch hier der beste Kompressionsgrad für das Ausgabemodul in bekannter Weise evaluiert.

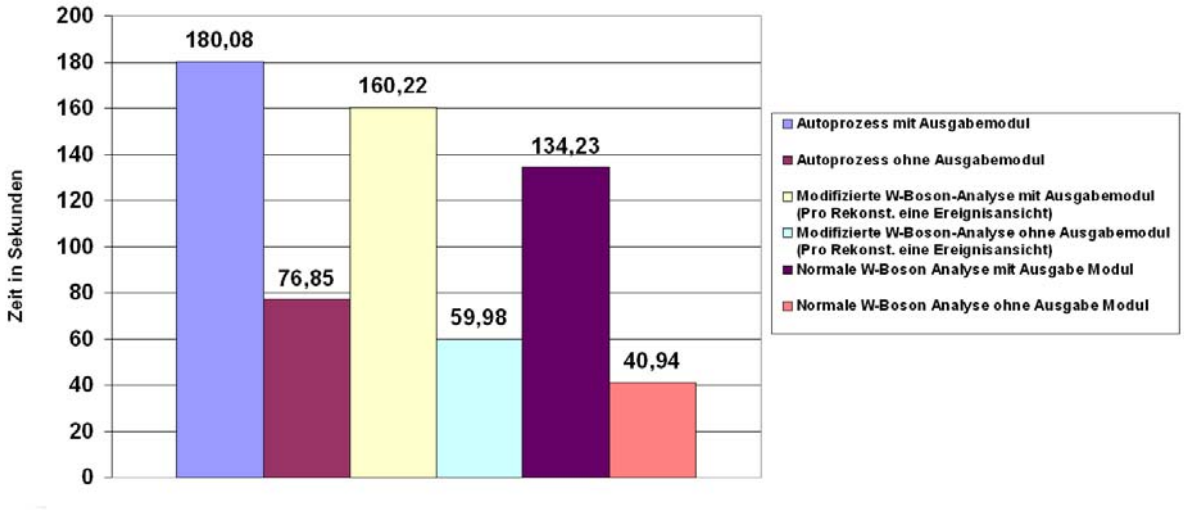


Abbildung 34: Laufzeit des Autoprozess bei der Rekonstruktion eines W-Bosonzerfalls. Im Vergleich die modifizierte und die originale Analyse aus Abschnitt 5.2.3. Alle Analysen jeweils mit und ohne Ausgabemodul.

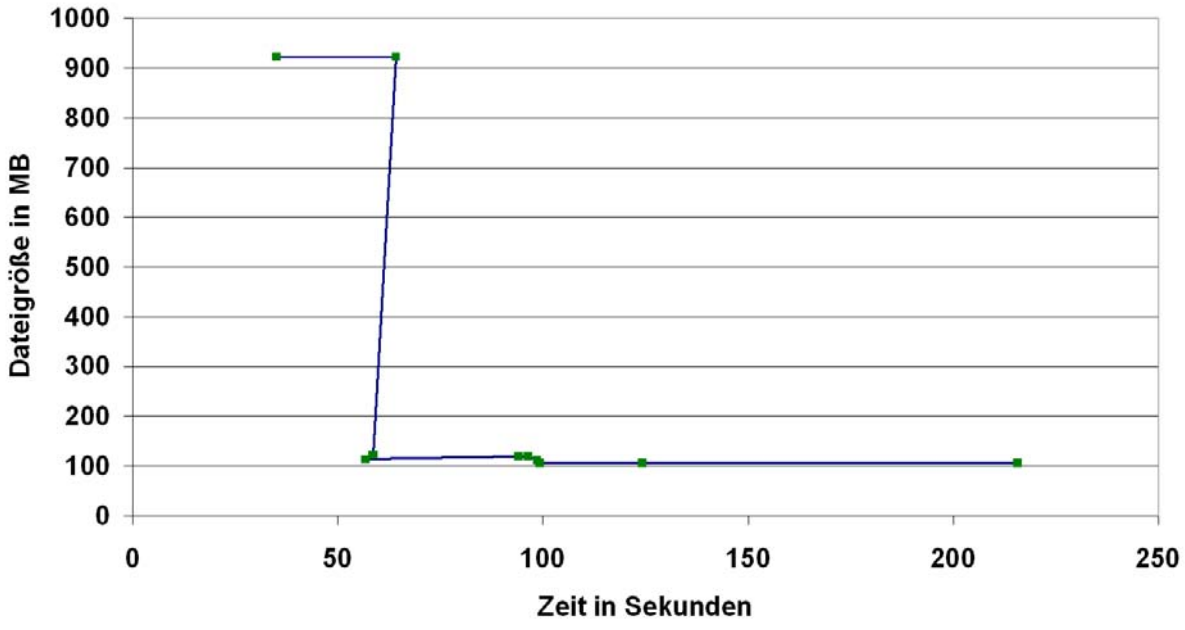


Abbildung 35: Zeit-Größenverhältnis der Kompression von Daten des Autoprozesses bei der Rekonstruktion eines W-Bosonzerfalls.

Aus der Abbildung 35 folgt, dass auch hier die ausgewogensten Werte für den Kompressionsgrad, bezüglich des Zeit-Kompressionsfaktors, 1, 2 und 3 sind. Auch der Zerfall des Top-Quarks wird mit Hilfe des Autoprozessmoduls analysiert und zeitlich erfasst. Hier liegt die Laufzeit bei $467,24$ Sekunden mit Ausgabemodul und bei $208,31$ Sekunden ohne Ausgabemodul. Die Eingabedatei, der Rechner und die Anzahl der Ereignisse sind dabei analog zur Analyse des Top-Quarkzerfalls aus Abschnitt 5.2.3.

5.4.2 Komplexität der möglichen Teilchenzerfälle

Ein wichtiger Punkt ist die Komplexität der möglichen Rekonstruktion eines Teilchenzerfalls. Die Komplexität beschreibt wie viele Möglichkeiten es gibt, aus den im Detektor gemessenen Teilchen, ein anderes bestimmtes Teilchen zu rekonstruieren. Als Beispiel nehmen wir hier den W-Bosonzerfall. Das W-Boson zerfällt zu zwei Quarks. $(W \Rightarrow q_1 + q_2)$. Werden im Detektor drei Jets gemessen, q_1, q_2, q_3 gibt es also folgende Möglichkeiten einer Rekonstruktion von Teilchen. $(W \Rightarrow q_1 + q_2)$ $(W \Rightarrow q_1 + q_3)$ $(W \Rightarrow q_2 + q_3)$. Zwischen $(W \Rightarrow q_1 + q_3)$ und $(W \Rightarrow q_3 + q_1)$ wird nicht unterschieden, die Reihenfolge der Zerfallsteilchen wird also nicht berücksichtigt. Es gibt also, wenn n Quarks gemessen werden, $\binom{n}{2}$ mögliche Rekonstruktionen. Der Autoprozess jedoch rekonstruiert zur Zeit beide Varianten $(W \Rightarrow q_1 + q_3)$ und $(W \Rightarrow q_3 + q_1)$. Dies führt, in diesem Beispiel, zu doppelt so vielen Rekonstruktion wie nötig und damit auch zu einer fast doppelten Menge an erzeugten Daten. Beim Top-Quarkzerfall $(top \Rightarrow (W \rightarrow q_1 + q_2) + q_3)$ erzeugt der Autoprozess ebenfalls die doppelte Menge an möglichen Rekonstruktionen, nämlich $n(n-1)(n-2)$, für n im Detektor gemessene Quarks. Nötig wären aber nur $\binom{n}{2}$ $(n-2)$. Bei der Rekonstruktion eines Higgsteilchenzerfalls $(H \Rightarrow (Z_0 \rightarrow mu_1 + mu_2) + (Z_0 \rightarrow mu_3 + mu_4))$ erzeugt der Autoprozess 8 mal mehr Rekonstruktionen, als der Benutzer unter Umständen benötigt. Denn bei diesem Zerfall müssen bei n Myonen nur $\binom{n}{4} \cdot 3$ Teilchen rekonstruiert werden. Es gibt $\binom{n}{4}$ Möglichkeiten, die n Myonen auf die vier Myonen im Zerfall zu verteilen $(H \Rightarrow (Z_0 \rightarrow mu_1 + mu_2) + (Z_0 \rightarrow mu_3 + mu_4))$ und drei Möglichkeiten je Verteilung verschiedene Z-Teilchen zu rekonstruieren.
$(H \Rightarrow (Z_0 \rightarrow mu_1 + mu_2) + (Z_0 \rightarrow mu_3 + mu_4))$,
$(H \Rightarrow (Z_0 \rightarrow mu_1 + mu_3) + (Z_0 \rightarrow mu_2 + mu_4))$ und
$(H \Rightarrow (Z_0 \rightarrow mu_1 + mu_4) + (Z_0 \rightarrow mu_2 + mu_3))$
Die anderen Varianten des Zerfalls können als symmetrisch angesehen werden. Der Autoprozess erzeugt jedoch $n(n-1)(n-2)(n-3)$ Rekonstruktionen.

$$\binom{n}{4} \cdot 3 = \frac{n!}{4!(n-4)!} \cdot 3 = \frac{n(n-1)(n-2)(n-3)...1}{4 \cdot 3 \cdot 2 \cdot 1(n-4)(n-5)...1} \cdot 3$$

$$= \frac{n(n-1)(n-2)(n-3)}{8}$$

Entsprechend der mehr erzeugten Rekonstruktionen, ist sowohl die Laufzeit einer Analyse bei einem Autoprozessmodul länger, als auch die Menge an erzeugten Daten.

5.5 Optimierung der Analysen

Die in Abschnitt 5.2.3 gezeigte Analysen der Rekonstruktion des W-Bosonzerfalls und des Top-Quarkzerfalls sind noch stark optimierbar. In einem ersten Schritt wird dafür gesorgt, dass die überflüssigen Rekonstruktionen gar nicht erstellt werden. In einem zweiten Schritt wird der Kompressionsgrad von der Standardkompression 6 auf den Kompressionsgrad 2 reduziert. Im letzten Schritt werden nur Teilchen rekonstruiert, bei denen mindestens

50% der erwarteten Ruhemasse erzielt wird. Die 50% entsprechen der Messungenauigkeit in einem Detektor. Optimiert werden hierbei C++ Module. Wie man in Abbildung 36 erkennen kann, wird die Geschwindigkeit der Analyse um den Faktor 4 bis 6 verbessert.

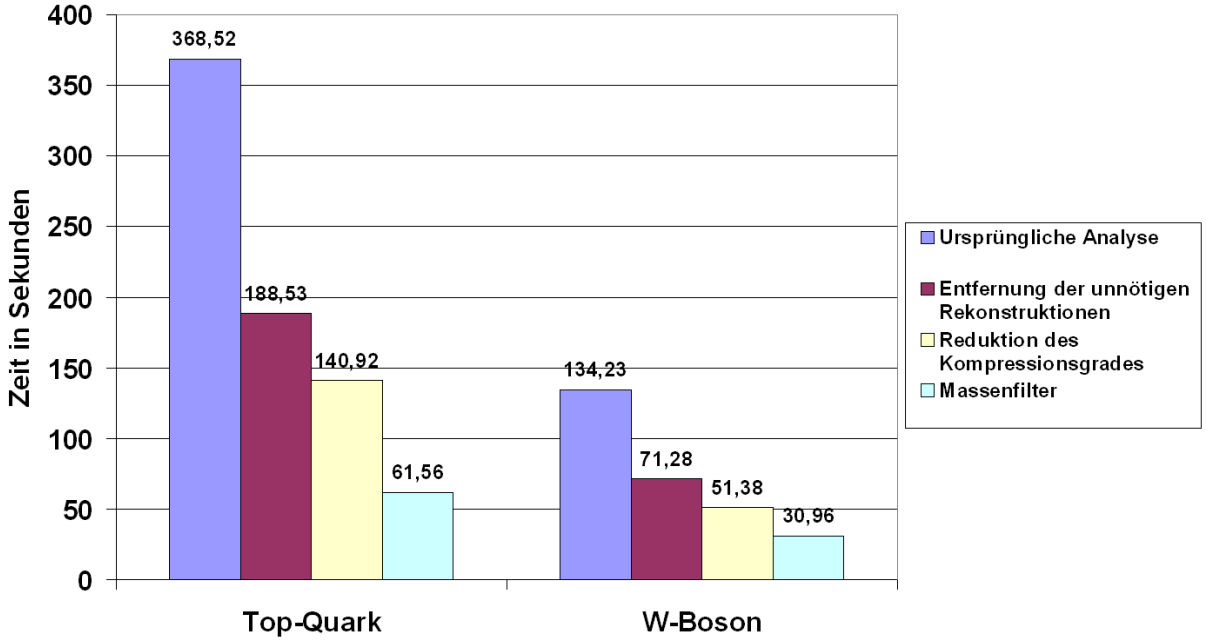


Abbildung 36: Optimierung von Analysen und Auswirkungen auf die Laufzeit.

Auf einem leistungsfähigeren Rechner, zum Beispiel *Notebook 2*, kann die Rechenzeit mit 32, 1 Sekunden bei der Analyse des Top-Quarkzerfalls und 18, 5 Sekunden bei dem W-Bosonzerfall fast halbiert werden. Die Optimierung hat auch Auswirkungen auf die Dateigröße der Ausgabedatei. Abbildung 37 zeigt, dass die Dateigröße um den Faktor 2,5 - 4 verkleinert wird.

5.6 Softwaremetriken

Wenn der Anwender mit VISPA eigene Module, sowohl Pythonmodule als auch C++Module erstellt, greift er dabei sehr stark auf die Klassen und Funktionen von PXL zurück. Deshalb ist für die Bedienbarkeit von PXL und damit auch von VISPA die Qualität der PXL-Software von Bedeutung. Aber auch wenn der Anwender PXL für eigene Zwecke erweitern möchte, hängt die Erweiterbarkeit von PXL von der Softwarequalität ab.

Es gibt jedoch auch eine zweite Sichtweise für die die Softwarequalität wichtig ist, die Sichtweise der Entwickler. Für sie stellt sich ebenfalls die Frage nach der Erweiterbarkeit und zusätzlich auch nach der Wartbarkeit der Software.

Die Bedienbarkeit, Erweiterbarkeit und Wartbarkeit hängen von weiteren Faktoren, wie Verständlichkeit und Änderbarkeit des Quellcodes, Fehleranfälligkeit der Software und Testbarkeit ab, siehe Abbildung 38.

Um Eigenschaften, wie Fehleranfälligkeit, Testbarkeit, Änderbarkeit, Verständlichkeit und Aufwand zum Erstellen einer Software zu kennen, ist es nötig, die Qualität und Komplexität einer Software zu bestimmen. Komplexere Software ist in der Regel fehleranfälliger und benötigt größeren Aufwand bei Tests sowie zur Wartung. Um Maße für die Qualität und Komplexität einer Software zu bekommen, bedient man sich so genannter Metriken [FP98] [Mun03] [Zus98]. PXL wird nun im Folgenden mit verschiedene Metriken untersucht und mit anderen Softwarepaketen verglichen.

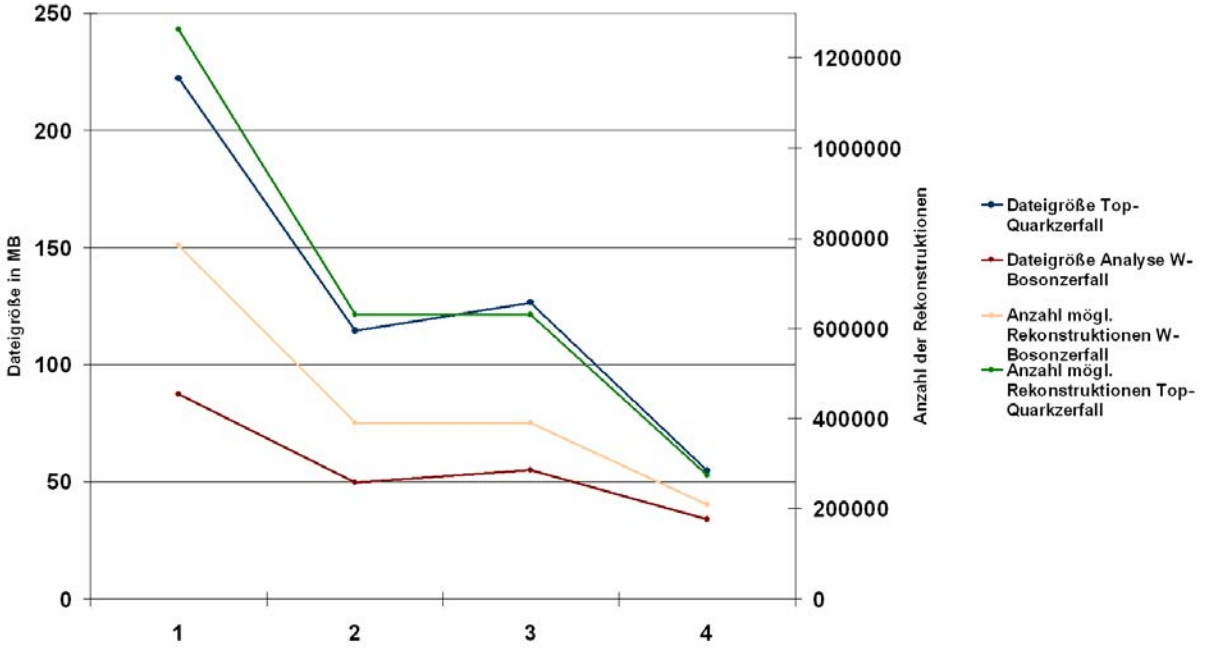


Abbildung 37: Optimierung von Analysen: Auswirkungen auf Dateigröße und Anzahl der möglichen Rekonstruktionen. Auf der y-Achse sind die Schritte der Optimierung wiedergegeben: 1. Keine Optimierung, 2. Entfernung der unnötigen Rekonstruktionen, 3. Reduktion des Kompressionsgrades auf 2, 4. Massenfilter.

5.6.1 Verwendete Metriken

Für die Beurteilung der Komplexität und Qualität von PXL wird unter anderem die Zeilenmetrik *Lines of Code (LOC)*, LOC per Lines of Comment, die McCabe-Metrik und die Informationsfluss-Metrik *IF4* verwendet.

Lines of Code(LOC) Die Zeilenmetrik *Lines of Code (LOC)* ist die einfachste und gebräuchliste Form eines Maßes der Komplexität einer Software. Sie misst die Anzahl der Zeilen Code einer Software. Kommentare im Quellcode werden nicht zu LOC hinzugerechnet. LOC dient zur Messung des generellen Umfangs einer Software und ist nur ein grobes Maß für die Komplexität [FP98].

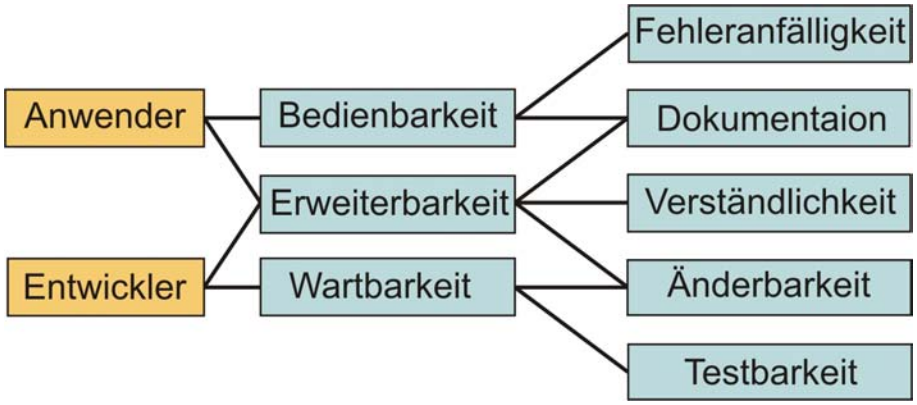


Abbildung 38: Die einzelnen Aspekte der Softwarequalität.

LOC per Lines of Comment Die Anzahl der LOC pro Kommentarzeilen gibt an, wie gut der Quellcode dokumentiert ist. Die Kommentarzeilen fließen bei PXL in die Klassendokumentation von *Doxygen* mit ein. Da der Anwender bei der Verwendung von Pythonskripts auf die Klassen von PXL zugreift, ist diese Klassendokumentation für eine gute Übersicht über die Klassen und Funktionen von PXL sehr wichtig. Ein gutes Verständnis der Klassen und Funktionen von PXL ist für die Benutzerfreundlichkeit, wie auch für die Erweiterbarkeit von PXL wichtig. Je niedriger der Wert dieser Metrik ist, desto besser ist der Quellcode dokumentiert [Lit01].

McCabe-Metrik Die McCabe-Metrik ist ein weiteres Maß für die Komplexität eines Programms oder einer Funktion. Sie wurde von Thomas McCabe 1976 eingeführt [McC76]. Der Wert $v(G)$ der McCabe-Metrik gibt die Komplexität des Kontrollflusses im Code an und ist gleich der linear unabhängigen Pfade im Kontrollflussgraphen. Der Wert ist gleichzeitig eine obere Schranke für die Anzahl benötigter Testfälle des Programms oder der Funktion. Mit ihm lässt sich etwas über die Fehleranfälligkeit und Verständlichkeit eines Programms aussagen sowie über den Aufwand des Testens. Ein größerer Wert bedeutet eine größere Komplexität des Programms, eine größere Fehleranfälligkeit, eine geringere Verständlichkeit des Codes und einen größeren Aufwand des Testens. Für den Wert der McCabe-Metrik $v(G)$ gilt:

$$v(G) = e - n + 2p \tag{19}$$

- n: Knoten in dem Kontrollflussgraphen
- e: Kanten im Kontrollflussgraphen
- p: Anzahl der zusammenhängenden Komponenten im Graphen
- 2: Für die Anzahl der Ein- und Ausgänge des Programms oder der Funktion

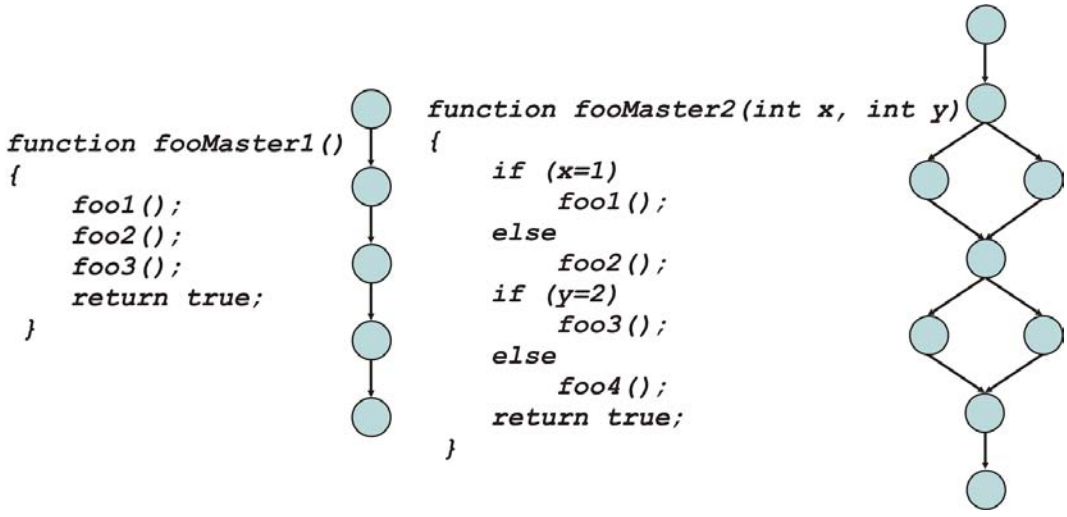


Abbildung 39: **Links:** Kontrollflussgraph einer Funktion mit nur sequentiellen Anweisungen ohne Verzweigung. **Rechts** Kontrollflussgraph einer Funktion mit zwei binären Verzweigungsanweisungen.

In Abbildung 39 ist der Wert der McCabe-Metrik für die linke Funktion $v(G) = 4 - 5 + 2 = 1$. Für die rechte Funktion ergibt sich der Wert $v(G) = 11 - 10 + 2 = 3$. Daraus folgt: Bei jeder binären Verzweigung des Programms oder der Funktion erhöht sich die Anzahl der linear unabhängigen Pfade im Kontrollflussgraphen um 1. Damit erhöht sich auch der Wert der McCabe-Metrik um 1. Eine Funktion, welche nur eine Reihe von Anweisungen besitzt, hat eine McCabe-Metrik von 1. Besitzt die Funktion allerdings keinen Ausgang, zum Beispiel keine *return*-Anweisung, so ist die McCabe-Metrik 0. Als binäre Verzweigungen gelten *if*-Anweisungen, Schleifenbefehle wie *while* oder *for* und *case*-Anweisungen innerhalb von *switch*-Befehlen.

Für die McCabe-Metrik einer Funktion wird ein Wert empfohlen, der nicht über 10 liegen sollte, in Ausnahmefällen nicht über 15 [MW96] [Lig02]. Die Fehleranfälligkeit ergibt sich aus Tabelle 10 [Inc09].

Wert der McCabe-Metrik	Fehleranfälligkeit
1 - 10	Einfache Funktion. Geringe Fehleranfälligkeit
10 - 20	Komplexere Funktion. Mittlere Fehleranfälligkeit
20 - 50	Komplexe Funktion. Hohe Fehleranfälligkeit
51+	Unstabile Funktion. Extrem hohe Fehleranfälligkeit

Tabelle 10: Werte der McCabe-Metrik im Zusammenhang mit der Fehleranfälligkeit.

Informationsfluss IF4 Der Informationsfluss *IF4* nach Henry und Kafura beschreibt die Koppelung von Klassen untereinander [Lit01] [HKH81]. Das heißt, wie sehr ist die Klasse abhängig von anderen Klassen ist und wie viel Einfluss die Klasse auf andere Klassen ausübt. Wird beispielsweise in einer Klasse **A** eine Änderung vorgenommen oder tritt ein Fehler in dieser Klasse auf, so hat dies Auswirkungen auf alle anderen Klassen, auf die die Klasse **A** Einfluss hat. Mit dieser Metrik lässt sich also etwas über die Änderbarkeit und die Fehleranfälligkeit einer Software aussagen. Der Informationsfluss basiert dabei auf zwei Werten: *FanIn* und *FanOut*.

FanIn beschreibt den Informationsfluss, der in eine Klasse hineinführt. Der Wert entspricht der Anzahl der verschiedenen anderen Klassen, die der Klasse übergeben werden oder die die Klasse benutzt.

FanOut bezeichnet den Informationsfluss, der aus einer Klasse herausführt. Der Wert gibt an, von wie vielen andere Klassen die Klasse benutzt wird. Der gesamte Informationsfluss $IF4$ berechnet sich nun folgendermaßen [Lit01].

$$IF4 = (FanIn \cdot FanOut)^2 \tag{20}$$

Je höher dieser Wert für eine Klasse ist, um so größer ist die Kopplung der Klasse mit anderen Klassen. Daraus folgt, dass der damit verbundene Wartungsaufwand für die Software und die Fehleranfälligkeit ebenfalls steigen. Ein Beispiel für die Berechnung sehen wir in Abbildung 40. Die Klasse **A** hat einen $FanIn = 2$ und $FanOut = 1$ und somit gilt $IF4_A = (2 \cdot 1)^2 = 4$. Für Klasse **B** gilt $FanIn = 2$ und $FanOut = 2$, damit gilt $IF4_B = (2 \cdot 2)^2 = 16$. Schließlich gilt für Klasse **C** $FanIn = 1$, $FanOut = 2$ und $IF4_C = (1 \cdot 2)^2 = 4$. Der Grund, warum das Produkt aus $FanIn$ und $FanOut$ quadriert wird, ist dass die Metrik auch in Kombination mit anderen Metriken verwendet wird. Zum Beispiel zusammen mit der Zeilenmetrik LOC. Beide bilden zusammen ein weiteres Maß

```
class ClassA
{
  ClassC c;

  ClassA::ClassA(ClassB b1)
  {
  }

  int ClassA::foo1(ClassB b1)
  {
  }
}
```

```
class ClassB
{
  ClassB::ClassB(ClassC c1)
  {
  }

  int ClassB::foo1(ClassA a1,
                   ClassC c1)
  {
  }
}
```

```
class ClassC
{
  ClassC::ClassC(ClassB b1)
  {
  }

  int ClassC::foo1()
  {
  }
}
```

Abbildung 40: Beispiel des Informationsflusses zwischen drei Klassen.

der generellen Komplexität einer Software. $Komlex = LOC \cdot IF4_S$. Mit der Quadrierung erhält der Informationsfluss dann ein stärkeres Gewicht.

Wenn in der Klasse **B** nun etwas geändert wird oder dort ein Fehler auftritt, so hat das Auswirkungen auf die Klassen **A** und **C**. Für den Informationsfluss des Gesamtsystems IF_S, also der gesamten Software, werden die Werte des Informationsflusses der einzelnen Klassen aufaddiert.

$$IF4_S = \sum_{i=0}^{i=n} (FanIn_i \cdot FanOut_i)^2 \tag{21}$$

5.6.2 Verwendete Programme zur Evaluation

Zur Messung der Softwaremetriken werden zwei verschiedene Programme verwendet. Das erste Programm heißt *CodeAnalyzer*[Cod05]. Mit diesem Programm wird die Zeilenmetrik (LOC) und die Zeilenmetrik *LOC per Lines of Comment* gemessen. Das zweite Programm zur Messung der Metriken ist das *CCCC*-Softwaretool von Tim Littlefair [Lit01]. Mit *CCCC* werden die Metriken IF4 und McCabe bestimmt.

5.6.3 Vergleich der Metriken mit anderen Softwareprojekten

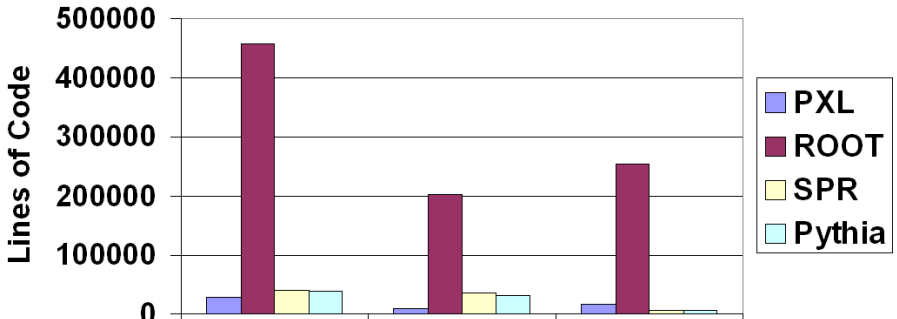
Um für PXL einen Vergleich mit anderen Softwarepaketen aus dem Bereich der Hochenergiephysik zu haben, wurden verschiedene Softwarepakete ebenfalls auf die gleichen Softwaremetriken untersucht. Dabei handelt es sich um die Softwarepakete ROOT, SPR und Pythia [ROO09][Pyt09][Sta09].

Das ROOT-Software Paket ist schon in Abschnitt 4.7 beschrieben.

StatPatternRecognition (SPR) Das Softwarepaket implementiert eine Vielzahl von Werkzeugen für die Kategorisierung und Klassifizierung von multivariaten Daten, wie zum Beispiel *Boosted-Decision-Trees* und *MulticlassLearner*. Diese dienen unter anderem zur Herausfilterung von bestimmten Signalen aus einer Reihe von Hintergrundsignalen. SPR wurde in C++ realisiert [Nar05].

Pythia Das Programm dient zur Simulation von Teilchenzerfällen, wie sie bei Teilchenkollisionen im Detektor auftreten. Das Programm wurde ebenfalls in C++ entwickelt [SMS06].

Bei allen Messungen der Metriken werden generierten Quellcodedateien nicht berücksichtigt, da die Entwickler keinen direkten Einfluss auf den Inhalt haben. In der Regel werden diese Dateien auch nicht editiert. Als erstes wird bei den Softwarepaketen die Zeilenmetrik LOC gemessen und miteinander verglichen, um einen ungefähren Eindruck von der Größe von PXL zu anderen Softwarepaketen zu bekommen.



	Total	c / cc / cpp	h / hh
PXL	28078	10327	17751
ROOT	457082	202369	254713
SPR	40291	35015	5286
Pythia	38483	31616	6867

Abbildung 41: Lines of Code.

Wie in Abbildung 41 erkannt werden kann, ist ROOT das mit Abstand größte Softwarepaket. Die Softwarepakete SPR und Pythia sind nach LOC gemessen etwa gleich umfangreich. PXL liegt an letzter Stelle.

Als zweite Metrik wird das Verhältnis von Codezeilen zu Kommentarzeilen gemessen. Nach dieser Metrik haben, wie man in Abbildung 42 erkennen kann, Pythia und ROOT die ausführlichste Dokumentation des Quellcodes. Die geringste Dokumentation hat dagegen PXL.

Die nächste Metrik, die gemessen wird, ist die McCabe-Metrik. Für diese Messung wurde das *CCCC*-Program von Tim Littlefair verwendet. In Abbildung 43 kann man die Anzahl der Funktionen erkennen, die einen bestimmten Wert der McCabe-Metrik haben. Die Funktionen sind dabei aufgeteilt in Werte von $0-10$, $11-20$, $21-50$ und über 51. Abbildung 44 zeigt die Verteilung der Funktionen auf die Werte der McCabe-Metrik in Prozent. In beiden Abbildungen zeigt sich, dass bei allen Softwarepaketen die Empfehlung, den Wert der McCabe-Metrik aus maximal 10 innerhalb einer Funktion zu begrenzen, zu fast 90% eingehalten wurde. PXL liegt hier mit über 98% der Funktionen, die eine McCabe-Metrik von 10 oder kleiner haben, an der Spitze. Auch gibt es, im Gegensatz zu den anderen Softwarepaketen, in PXL keine Funktion die nach Tabelle 10 als instabil angesehen wird.

Als letzte Metrik wurde der Informationsfluss der Klassen untersucht. Dabei wurde ebenfalls das *CCCC*-Programm von Tim Littlefair verwendet. In Abbildung 45 wird der

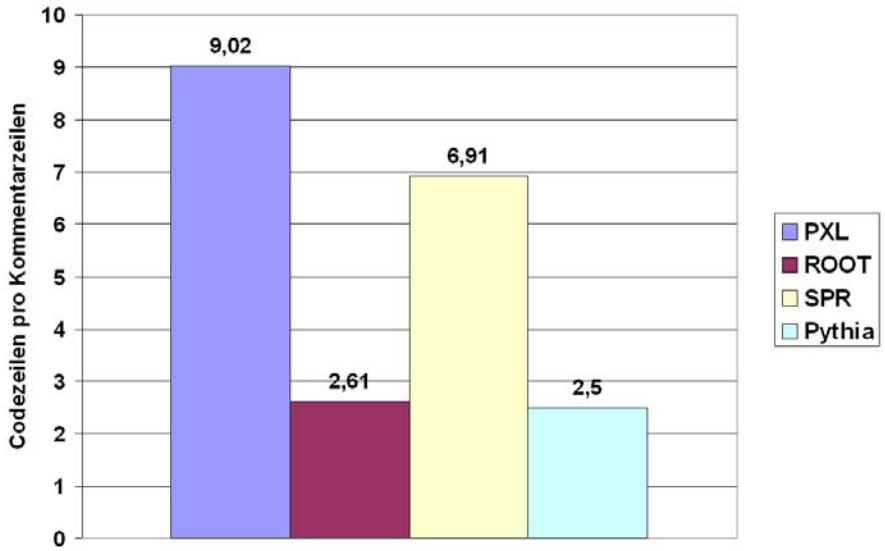


Abbildung 42: Verhältnis von Codezeilen zu Kommentarzeilen. Je kleiner der Wert ist, desto besser ist der Quellcode kommentiert.

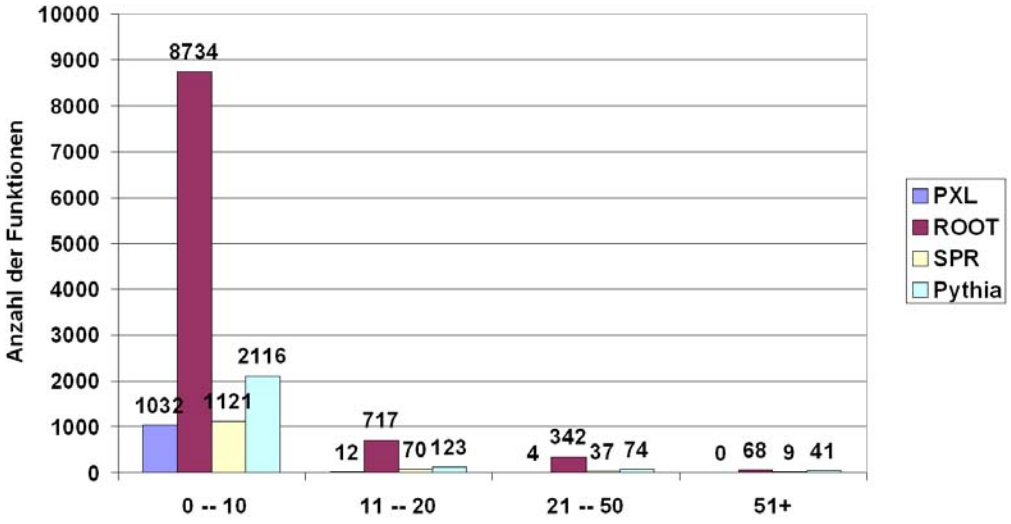
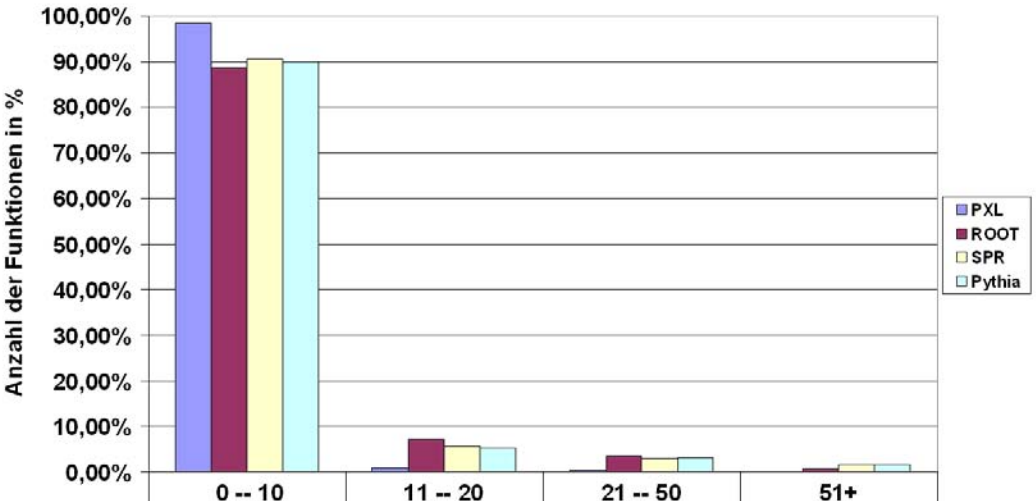


Abbildung 43: Anzahl der Funktionen mit einem bestimmten Wert der McCabe-Metrik.



	0 -- 10	11 -- 20	21 -- 50	51+
PXL	98,47%	1,15%	0,38%	0,00%
ROOT	88,57%	7,27%	3,47%	0,69%
SPR	90,62%	5,66%	2,99%	1,74%
Pythia	89,89%	5,23%	3,14%	1,74%

Abbildung 44: Anzahl der Funktionen mit einem bestimmten Wert der McCabe-Metrik in Prozent.

durchnittliche Informationsfluss pro Klasse dargestellt. Es fällt auf, dass es bei dem Softwarepaket ROOT einen extrem hohen Wert gegenüber den anderen Paketen gibt. Dies liegt daran, dass in ROOT einige Basistypen mit Klassen definiert werden, die in vielen anderen Klassen gebraucht werden und so einen extrem hohen Wert erzeugen. Ein Beispiel hierfür ist *TString*. Dieser Typ wir in 499 anderen Klassen verwendet, benötigt aber selbst einige Klassen von ROOT, wie *TSubString* oder *TBuffer*. Für die Klasse *TString* mit einem $FanIn = 18$ und $FanOut = 499$ ergibt sich $(18 \cdot 499)^2 = 80676324$. PXL weist hier gegenüber den anderen Softwarepaketen den geringsten Informationsfluss pro Klasse auf.

5.7 Umfrage über VISPA

Eine weitere Möglickeit, die Bedienbarkeit und die Handhabung einer Software zu evaluieren, ist unter den Benutzern der Software eine Umfrage zu starten [Nie93][STL00]. Im Sommersemester 2009 wurde VISPA in die Übungen für Studenten im Fach Experimentalphysik integriert. Einige in den Übungen gestellte Aufgaben mussten unter der Verwendung von VISPA gelöst werden. Am Ende des Semester wurden die Studenten, die mit VISPA gearbeitet haben, befragt. An der Befragung haben insgesamt ca. 20 Anwender teilgenommen, darunter 19 Studenten und ein wissenschaftlicher Mitarbeiter.

5.7.1 Verwendete Software zur Umfrage

Für die Umfrage wurde das Programm *testMaker* verwendet. Das Programm *testMaker* ist eine webbasierte Software, mit der der Benutzer webbasierte Fragebogen und Tests erstellen und durchführen kann. Es wurde zwar speziell für psychologische Eignungstests entwickelt, dank seiner Mächtigkeit ist es aber auch für Befragungen und Test anderer Art

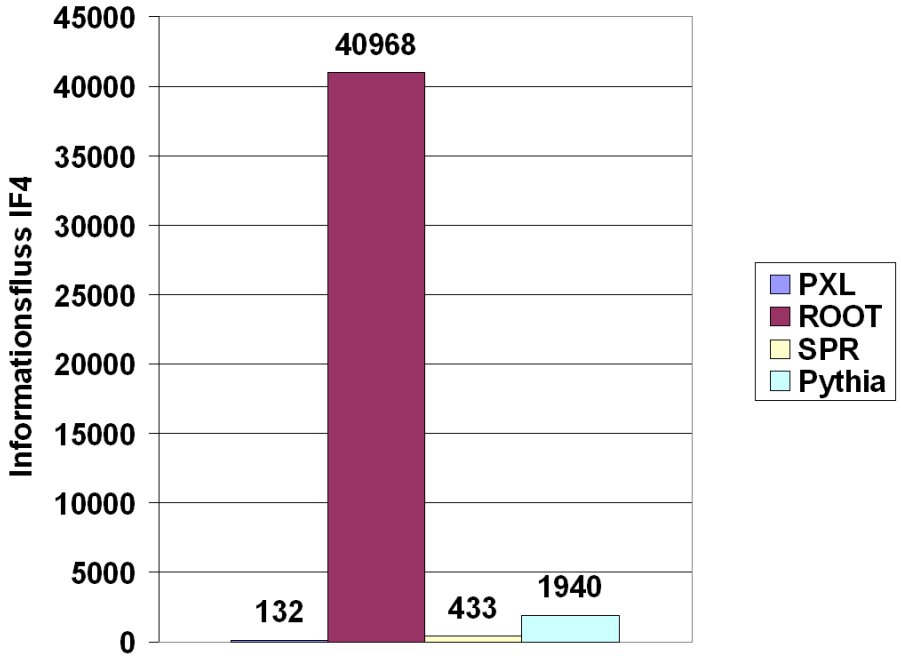


Abbildung 45: Der durchschnittliche Informationsfluss pro Klasse.

geeignet [Har10]. Der Quellcode des Projekts ist frei verfügbar und die Hauptentwicklung findet am Lehrstuhl für Betriebs- und Organisationspsychologie der RWTH Aachen statt.

5.7.2 Fragekategorien und Struktur der Umfrage

Die Umfrage zu VISPA besteht grundsätzlich aus drei Teilen. Im ersten Teil werden kurz Fragen zur Person, Fragen zum verwendeten Betriebssystem und Fragen zur Erfahrung mit Python und ROOT gestellt. Im zweiten Teil der Umfrage beziehen sich die Fragen hauptsächlich auf die Übersichtlichkeit, Bedienbarkeit und die Handhabung von VISPA. Der dritte Teil der Umfrage bezieht sich auf den Support und die Dokumentation von VISPA. In der Umfrage existieren zwei grundsätzliche Arten von Fragen. Die erste Art ist eine, bei der aus mehreren Antwortmöglichkeiten eine Antwort ausgewählt werden kann. Die zweite Art von Fragen ist eine, bei der die Antwort in Form einer freien Texteingabe gegeben werden kann. Im zweiten und dritten Teil der Umfrage bestehen die Fragen, bei denen es mehrere Antwortmöglichkeiten gibt, immer aus Feststellungen, mit denen der Benutzer übereinstimmen kann oder nicht. Seine Übereinstimmung kann er auf von einer Skala von 1 - 6 ausdrücken. Die Antwortmöglichkeiten dafür sind:

- Trifft voll und ganz zu
- Trifft überwiegend zu
- Trifft eher zu
- Trifft eher nicht zu
- Trifft überwiegend nicht zu
- Triff überhaupt nicht zu

Eine neutrale Antwortmöglichkeit kann nicht gewählt werden, weil der Befragte sich schon für ein positive oder negative Antwort entscheiden soll. Damit soll ausgeschlossen werden, dass zu viele neutrale Antworten gegeben werden und die Umfrage so an Aussagekraft verliert. Ebenfalls existieren im zweiten und dritten Teil Fragen, bei denen eine Freitexteingabe möglich ist. Fragen mit den Antwortmöglichkeiten *Ja / Nein* existieren ebenfalls, sie stehen jedoch immer vor einer Frage mit einer Freitexteingabe und dienen nur dazu, diese eventuell zu überspringen.

5.7.3 Selektion der aussagekräftigen Antworten

Nicht alle Beantwortungen der Umfrage sind zu verwerten. Der Befragte muss sich mit der Umfrage ernsthaft genug befasst haben, damit die Antworten eine gewisse Aussagekraft haben. Ein Indiz dafür ist die Zeit, die der Befragte benötigt hat, um die Umfrage zu beantworten. Alle Umfragen, die weniger als 180 Sekunden gedauert haben, werden verworfen. Es werden auch nur vollständig beantwortete Umfragen berücksichtigt. Umfragen, die abgebrochen und nicht vollständig beantwortet wurden, werden verworfen. Nach diesen Selektionskriterien bleiben noch 17 brauchbare Beantwortungen der Umfrage übrig.

5.7.4 Ergebnisse

In Abbildung 46 bis 49 werden einige exemplarisch ausgewählte Fragen gezeigt. Die meisten Fragen zur Bedienbarkeit und Handhabung von VISPA sind überwiegend positiv bis sehr positiv beantwortet wurden. Besonders positive Antworten gibt es nach den Fragen der Übersichtlichkeit Abbildung 46, der Handhabung Abbildung 47 rechts und nach dem Zyklus der Analyseentwicklung Abbildung 49 rechts. Einzige Ausnahme ist die Frage nach dem Auffinden von Fehlern in VISPA, Abbildung 47 links. Dort sind vermehrt negative Bewertungen vorhanden. Eine Erklärung hierfür ist, dass bei der Version für Windows im ROOT Softwarepaket ein Fehler aufgetreten ist. Bei dem Füllen eines Histogramms mit vielen Daten verursachte ROOT einen Fehler, der sich auf das gesamte Programm von VISPA auswirkte. Dies führte zum Einfrieren des Programms. Dass der Fehler in ROOT lag, war jedoch nicht sofort sichtbar. Ebenfalls nicht ganz so positive ist die Frage nach der Dokumentation von VISPA. Hier treten auch vermehrt negative Bewertungen auf. Bei den Antworten mit Freitexteingabe ist hervorzuheben, dass sich die Benutzer vor allem eine bessere Dokumentation und mehr Beispielanalysen wünschen.

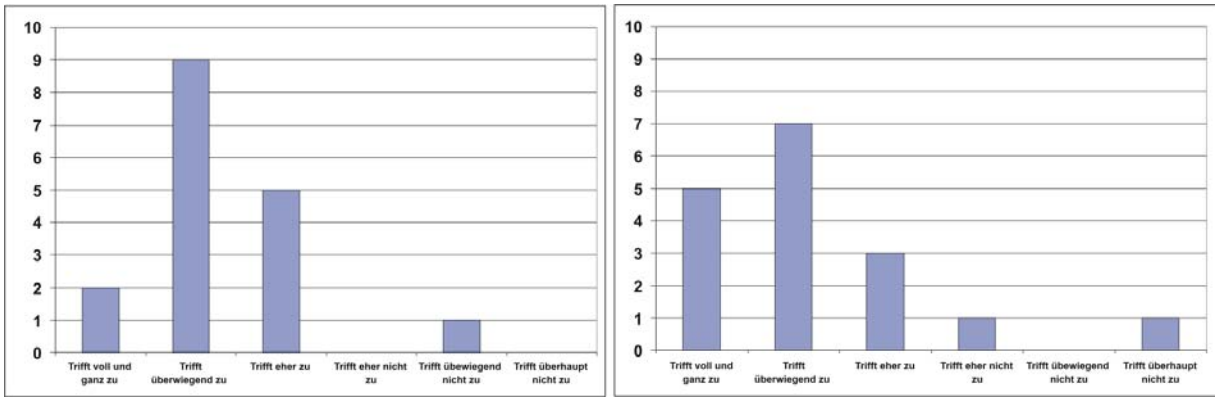


Abbildung 46: Beantwortung auf folgende Fragen: **Links:** *Die Benutzeroberfläche von VISPA ist übersichtlich aufgebaut.* **Rechts:** *Der Aufbau der Analyse wird übersichtlich und leicht verständlich dargestellt.*

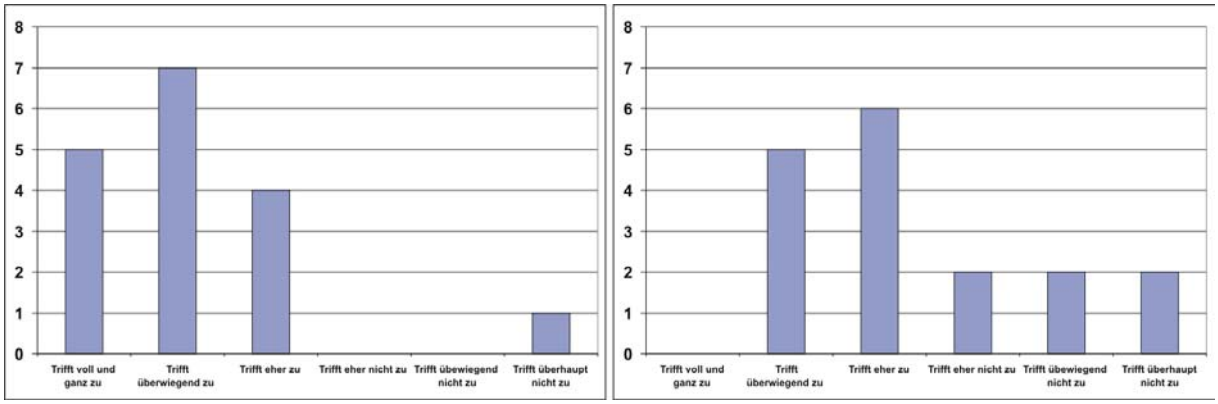


Abbildung 47: Beantwortung auf folgende Fragen: **Links:** *Das Erstellen der Analyse bzw. die Handhabung der Drag and Drop Oberfläche ist einfach und unkompliziert.* **Rechts:** *Fehler in der Analyse sind durch die Visualisierung von VISPA leicht zu finden.*

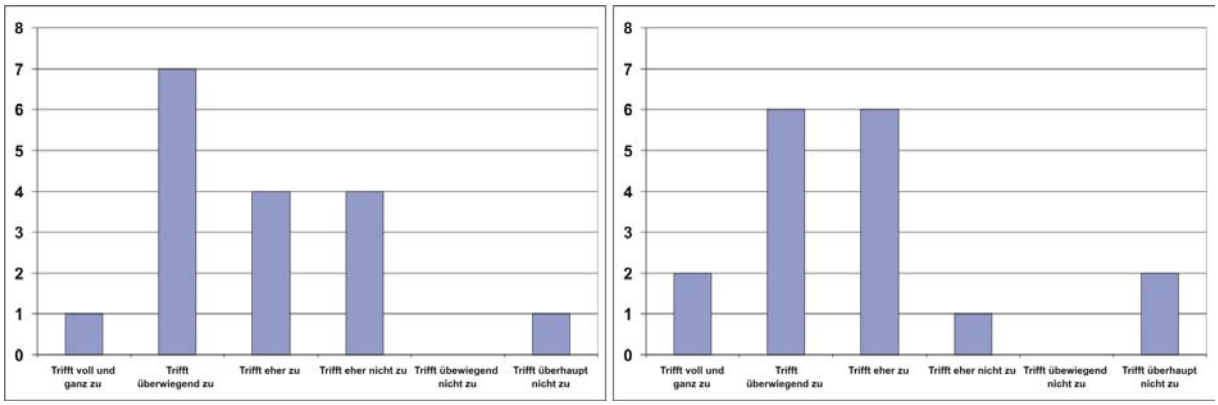


Abbildung 48: Beantwortung auf folgende Fragen: **Links:** *Die Darstellung von Ereignissen im Browser ist übersichtlich.* **Rechts:** *Die Inspektion einzelner Objekte eines Ereignisses ist einfach und unkompliziert.*

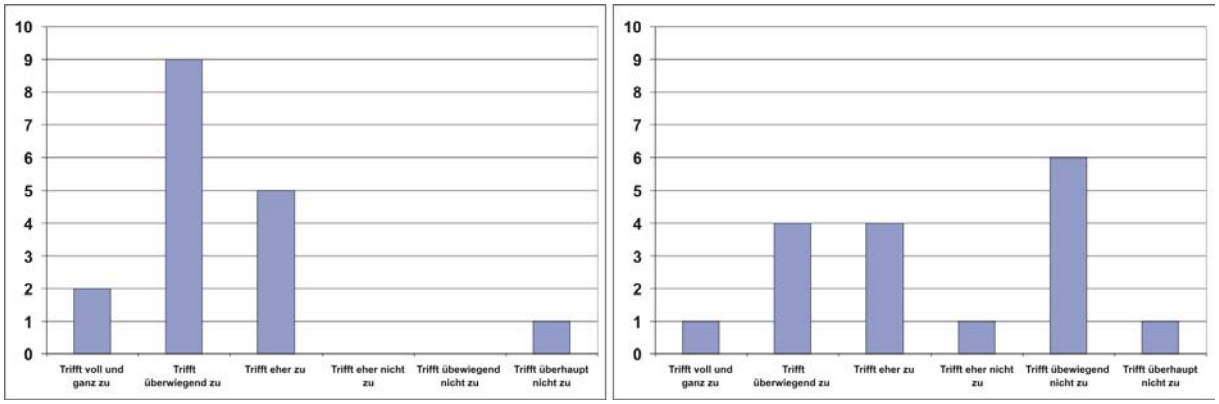


Abbildung 49: Beantwortung auf folgende Fragen: **Links:** *Die Iteration einer Analyseentwicklung wird in wenigen Schritten durchgeführt.* **Rechts:** *Die Dokumentation für VISPA ist verständlich und umfassend.*

6 Zusammenfassung

6.1 Inwieweit sind die Ziele von VISPA erreicht

Aufgrund der Erkenntnisse aus Kapitel 5 kann nun erörtert werden, inwieweit die einzelnen Ziele von VISPA erreicht wurden.

6.1.1 Analysezyklus des Entwerfens - Ausführens - Überprüfens

Das Durchlaufen des Entwicklungszyklus für eine Analyse in VISPA benötigt nur sehr wenige Schritte. Sofern die Analyse nur aus Standard- und Pythonmodulen besteht, sind es: Analyse ausführen, Ergebnis prüfen, eventuell die Skriptdatei für das Pythonmodul editieren und Analyse erneut starten. Sollten sich jedoch selbst geschriebene C++ Module in der Analyse befinden, so sind mehr Schritte notwendig, wie in Abbildung 11 dargestellt. Der Entwicklungszyklus mit C++ Modulen benötigt also mehr Schritte und ist deshalb langwieriger. Auch die Ergebnisse der Umfrage zeigen, dass der Entwicklungszyklus von Analysen in wenigen Schritten ausgeführt wird und keine unnötigen Schritte enthält. Das Ziel von VISPA, in Bezug auf ein schnelles Durchlaufen des Entwicklungszyklus einer Analyse wird also erreicht.

6.1.2 Bedienbarkeit und Übersicht

Sowohl die Softwaremetriken als auch die Umfragen ergeben in diesem Punkt ein positives Bild. Die McCabe-Metrik ist bei über 98 % der Funktionen von PXL nicht über einem Wert von 10. Damit ist die Fehleranfälligkeit sehr gering, was sich positiv auf die Benutzbarkeit der Funktionen von PXL auswirkt. Weil in den Skripten für die Pythonmodule die Benutzung von PXL-Funktionen unumgänglich ist, hat dies auch eine positive Auswirkung auf die Bedienbarkeit von VISPA. Auch der durchschnittliche Informationsfluss der Klassen von PXL ist geringer als bei vergleichbaren Softwarepaketen. Dies wirkt sich ebenfalls positive auf die Benutzbarkeit von PXL Klassen aus. Allerdings gibt es hier auch einen negativen Punkt der Metrik. Das Verhältnis von Kommentarzeilen zu Codezeilen ist geringer als bei den anderen Softwarepaketen. In den Umfragen fällt die Dokumentation ebenfalls als einziger Punkt negativ auf. Nach der Umfrage zu urteilen, sind die Beispielanalysen an sich sehr verständlich, aber die Anzahl wird als zu gering eingestuft. Alle anderen Punkte der Umfrage, wie Übersicht und Bedienbarkeit der Benutzeroberfläche, werfen ein postives Licht auf VISPA.

6.1.3 Austausch von Analysen

Analysen, die nur aus Standardmodulen und Pythonmodulen bestehen, sind problemlos austauschbar und plattformübergreifend. Die Analysen können sofort gestartet werden. Bei Analysestrukturen, die C++ Module enthalten, ist dies prinzipiell auch möglich. Allerdings muss der Quellcode erst neu kompiliert werden und die Compilereinstellungen eventuell an die Plattform angepasst werden. Der Austausch von Resultaten ist ebenfalls prinzipiell möglich. Die bei der Analyse erzeugten Dateien sind alle im PXL-Format und plattformübergreifend austauschbar. Hierbei ist allerdings zu beachten, dass die Dateien

unter Umständen sehr groß werden können, etwa mehrere hundert MB. Dies erschwert einen Austausch über das Internet. Der Benutzer sollte also darauf achten, dass die Ausgabedateien möglichst klein gehalten werden. Histogramme unter ROOT können in einem ROOT-eigenen Format abgespeichert werden. Da ROOT plattformunabhängig ist, kann dieses Format problemlos ausgetauscht werden. Weiterhin bietet bietet ROOT die Möglichkeit das Histogrammbild im PDF-Format zu speichern.

6.1.4 Geschwindigkeit und Speicherbedarf der Analysen

In dem Bereich der Hochenergiepysik liegt die Geschwindigkeit, mit der VISPA ein Ereignis abarbeiten kann, auf dem *Notebook1* bei etwa 10ms. Vorausgesetzt, es wird effizient programmiert. Zum Beispiel ist besonders wichtig, dass unnötige und doppelte Rekonstruktionen ausgeschlossen werden, denn das Anlegen von neuen Zerfallsbäumen in den Ereignisansichten kostet viel Zeit und verbraucht viel Speicher in der Ausgabedatei. Auch muss der Kompressionsgrad des Ausgabemoduls möglichst optimal gewählt werden. Ebenso sollte ein C++ Modul für rechenintensive Teile der Analyse verwendet werden. Eine weitere Voraussetzung muss sein, dass der Zerfall sehr einfach ist, wie Beispielsweise der W-Bosonzerfall in zwei Quarks. Bei komplizierteren Zerfällen, wie beim Top-Quark, steigt die benötigte Zeit schon auf etwa 200 ms pro Ereignis an. Auch dann, wenn effizient Programmiert worden ist. Die Rate mit der der Autoprozess die Ereignisse abarbeitet, liegt bei dem W-Bosonzerfall bei 56 ms pro Ereignis und beim Top-Quarkzerfall bei 1557 ms pro Ereignis. Der Grund für den hohen Wert des Autoprozesses ist, dass er beim Top-Quarkzerfall im Schnitt ca. 4200 Hypothesen pro Ereignis erstellt. Diese Zeiten beziehen sich alle auf das *Notebook1*. Mit einem schnelleren Rechner, wie dem *Notebook2*, kann die benötigte Zeit pro Ereignis in etwa halbiert werden. Das Ziel von VISPA bzw. PXL, dass die Rekonstruktion eines Ereignises im Milisekundenbereich zu verarbeiten, wird bei einfachen Teilchenzerfällen und effizienter Programmierung erreicht.

Bei allen Analysen mit VISPA zeigte sich, dass der optimale Kompressionsgrad des Ausgabemoduls, bezüglich der Zeit-Kompressionsrate, den Wert von 1 - 3 besitzt. Der Speicherbedarf der Analysen liegt, in den durchgeführten Beispielen, zwischen mehreren zehn Megabyte bis hin zu mehreren hundert Megabyte. Durch eine effiziente Programmierung kann der Speicherbedarf jedoch um den Faktor 2,5 - 4 gesenkt werden.

6.2 Pythonmodul vs. C++ Modul

Eine der wichtigen Fragen für den Anwender ist, ob er die Analyse mit einem Python- oder C++ Modul erstellt. Dazu müssen die Vor- und Nachteile der jeweiligen Module berücksichtigt werden.

Der Geschwindigkeitsvorteil von C++ Modulen wirkt sich nur dann aus, wenn die in der Analyse erzeugten zusätzlichen Daten, gering sind. Je mehr Daten, wie zum Beispiel Zerfallsbäume von Teilchen erzeugt werden, desto mehr Zeit benötigt das Ausgabemodul, um die Daten in eine Datei zu schreiben. Dies macht den Geschwindigkeitsvorteil wieder zu Nichte. Besonders groß wird der Geschwindigkeitsvorteil, je mehr Aufrufe von PXL Funktionen in der Analyse vorkommen. Denn wie in Abschnitt 5.2.4 beschrieben, verlieren die Pythonmodule viel Zeit in der von SWIG generierten Schnittstelle zwischen Python und PXL. Der Geschwindigkeitsvorteil des C++ Moduls muss mindestens den Mehraufwand an Zeit decken, der bei der Erstellung des Moduls entstanden ist. Dies kann dann der Fall sein, wenn der Analyseteil, der vom Modul ausgeführt wird, sehr lange dauert oder wenn das Modul auch für andere Analysen gebraucht wird, einen häufigen

Pythonmodul	
Vorteile	Nachteile
• Einfache Skriptsprache. • Wenige Schritte im Zyklus der Analyseentwicklung. • Erleichtert den Austausch von Analysen.	• Geringere Geschwindigkeit. • Umständliche Übergabe von Eigenschaften. • Kein Senden von ein und dem selben Objekt an verschiedene Ausgänge möglich. Damit auch kein Kopieren von Ereignissen möglich.

Tabelle 11: Vor -und Nachteile des Pythonmoduls in der aktuellen Implementierung.

C++ Modul	
Vorteile	Nachteile
• Hohe Geschwindigkeit. • Erstellung von eigenen Optionen • Senden eines Objektes an mehrere Ausgänge. Damit Kopieren von Ereignissen möglich.	• Schwierigere Programmiersprache. • Mehr Schritte nötig im Zyklus der Analyseentwicklung. • Größerer Aufwand beim Austausch von Analysen.

Tabelle 12: Vor -und Nachteile des C++ Moduls.

Wiederverwendungswert hat. Ein Beispiel für einen hohen Wiederverwendungswert hat das Modul zur Berechnung der *Regions of Interest*. Ein weiterer Grund, sich für ein C++ Modul zu entscheiden ist, wenn der Benutzer dem Modul eine Reihe von Eigenschaften geben will. In C++ Modulen geht das mit der Erstellung von Optionsfeldern angenehmer als bei Pythonmodulen, bei denen eine Parameterzeichenkette übergeben werden muss. Allerdings ist dieser Nachteil in der gegenwärtigen Implementation von VISPA v. 0.3.3 nicht beabsichtigt. Er kann in zukünftigen Versionen behoben werden. Ebenso der Nachteil, dass man mit einem Pythonmodul kein Objekt an zwei verschiedene Ausgänge senden kann. Für ein Pythonmodul spricht, wenn die Analyse nur kurz ist, so dass sich ein Mehraufwand an Zeit nicht lohnt. Bei der Erzeugung von großen Datenmengen während der Analyse, spricht man sich ebenfalls eher für ein Pythonmodul aus. Generell sollte jedoch bei einem Pythonmodul bedacht werden, dass der Benutzer unnötige Funktionsaufrufe von externen Klassen und Paketen wie PXL vermeiden sollte, weil ansonsten sehr viel Zeit in der Schnittstelle zwischen Python und C++ verbraucht wird. Eine weitere Strategie bei einer Analyseentwicklung kann sein, eine Art Prototyp mit einem Pythonmodul zu entwickeln um auf kleinen Datenmengen zu testen. Später, bei großen Datenmengen, kann dann das Modell des Prototyps auf ein C++ Modul übertragen werden.

6.3 Parallelisierung von Analysen

PXL nutzt von sich aus kein Mehrprozessorsystem. Die einzige Möglichkeit, mehrere Prozessoren für PXL nutzbar zu machen, ist die Aufteilung einer Analyse auf mehrere kleinere. An sich lässt sich die Laufzeit der Analyse damit fast halbieren, jedoch problematisch ist anschließend das Zusammenfügen der einzelnen dabei entstandenen Dateien. Dies kann unter Umständen länger dauern als die Analyse ohne Aufteilung auszuführen. Dies ist besonders dann der Fall, wenn die Menge an erzeugten Daten sehr groß ist und so die Ein- und Ausgabemodule beim Zusammenführen der Dateien viel Zeit benötigen. Ein gutes Beispiel dafür ist die Rekonstruktion von Teilchenzerfällen Eine Aufteilung lohnt sich immer dann, wenn das Analysemodul, im Gegensatz zu den Ein- und Ausgabemodulen, viel Zeit beansprucht. Die Analyse zum Auffinden der *Regions of Interest* wäre hierfür ein Beispiel.

6.4 Wartbarkeit und Aufwand für Erweiterungen

Nach den Softwaremetriken, die in Abschnitt 5.6 gemessen wurden, ist der Aufwand für die Wartbarkeit von PXL geringer als bei den anderen Softwarepaketen. Ebenso verhält es sich mit dem Aufwand für Erweiterungen. Einzig negativ wirkt sich die Dokumentation von PXL aus, die geringer ist als bei den anderen Softwarepaketen.

6.5 Betrachtung aus dem Blickwinkel der Informatik

Aus der Sicht der Informatik wurden eine Reihe von wichtigen Punkten der Software von VISPA und PXL evaluiert. Der erste Punkt ist die Funktionalität und die Bedienbarkeit des Programms auf Grundlage der durchgeführten Implementation. Unter die Implementation fallen die gewählten Programmiersprachen, Softwarepakete, Softwarewerkzeuge und der Aufbau des Programms. Der nächste Punkt der untersucht wurden ist, ist die Geschwindigkeit und der Speicherbedarf des Programms. Daraus ergaben sich Aufschlüsse über die Datenmenge, die in einem bestimmten Zeitraum verarbeitet werden kann und über die Datenmenge, die produziert wird. Die Verwendung von Zeitfunktionen, wie *clock()* oder *timeGetTime()*, waren wirksame Methoden zur Bestimmung der Laufzeit von Analysen. Sie lieferten immer reproduzierbare und nachvollziebare Ergebnisse. Weiterhin ergaben sich Erkenntnisse über die Vor- und Nachteile der Skriptsprache Python gegenüber der Hochsprache C++. Besonders im Bezug auf die Geschwindigkeit und die Handhabung in Verbindung mit VISPA. Auf dieser Grundlage können nun in Zukunft effizientere Analysen entwickelt werden. Ebenso hilfreich war die Analyse mit dem Profilerprogramm AQTime. Hier konnte der Ablauf des Programms sowie die relative Zeitverteilung auf jede Funktion nachvollzogen werden. So konnte ausgeschlossen werden, dass die Software keine übermäßig zeitkritischen oder zeitintesiven Funktionen enthält. Damit haben sich Profilerprogramme, wie AQTime, als wirkungsvolles Mittel für die Evaluation von Software erwiesen. Weiterhin wurden Softwaremetriken untersucht, die ein wesentliches Maß für die Qualität einer Software sind und damit auch zur Bedienbarkeit des Programms wesentlich beitragen. Weiterhin lassen sich für den Informatiker daraus auch Rückschlüsse auf eventuelle Verbesserungen und Optimierung des Quellcodes ziehen. Auch hier gelangt man zu der Schlussfolgerung, dass Programme wie *CCCC*, die Metriken Messen, einen sinnvollen Beitrag zur Evaluation von Software leisten können. Als letztes wurden, wie in [STL00][Nie93] vorgeschlagen, Befragungen der Benutzer durchgeführt. Dadurch wurde die Zufriedenheit der Benutzer von VISPA festgestellt und weitere

Erkenntnisse über die Bedienbarkeit erlangt.

Generell kann man sagen, dass die Evaluation einer Software zur Verbesserung und zur Optimierung dieser beiträgt. Zusätzlich kann anhand der Evaluation überprüft werden, ob und in wie weit die Ziele, die mit der Software erreicht werden sollten auch erreicht wurden sind. Es ist jedoch immer zu beachten, was evaluiert werden soll. So war es Beispielsweise in dem Fall von VISPA für die Entwickler sehr wichtig, die Geschwindigkeit des Programms in der Abhängigkeit der Art der Analyse zu kennen. Deshalb wurde auch viel Zeit in diesen Teil der Evaluation gesteckt.

7 Ausblick

7.1 Mögliche Verbesserungen

Die Ziele von VISPA sind größtenteils erreicht. Dennoch gibt es Platz für Verbesserungen und Ergänzungen.

7.1.1 Verbesserungen des Autoprozessmoduls

Eine sehr wichtige Verbesserung des Autoprozessmoduls wäre, eine Option einzufügen, die die unter Umständen unnötige Rekonstruktionen von Teilchenzerfällen unterbindet. Je nach Zerfall kann an das 8-fache an Rekonstruktionen eingespart werden. Dies würde eine erhebliche Menge an Zeit während der Analyse einsparen. Ebenso wäre der Speicherbedarf der Ausgabedateien reduziert. Eine solche Implementierung ist im Autoprozessmodul jedoch nicht einfach zu bewerkstelligen. Die Schwierigkeit ist, herauszufinden, ab wann der Zerfallsbaum symmetrisch ist. Weiterhin muss erkannt werden, ob eine zu rekonstruierende Möglichkeit eines Zerfalls symmetrisch zu einer anderen bereits rekonstruierten Möglichkeit ist. Bei einem W-Bosonzerfall der Art, $W \Rightarrow q_1 + q_2$ ist die Symmetrie $W \Rightarrow q_2 + q_1$ noch leicht zu überprüfen. Bei einem komplexeren Zerfall, wie $g \Rightarrow (Top \Rightarrow (W \rightarrow q_1 + q_2) + q_3) + (Top \Rightarrow (W \rightarrow q_4 + q_5) + q_6)$ ist dies jedoch nicht mehr so trivial zu implementieren. Eine weitere Möglichkeit das Autoprozessmodul effizienter zu gestalten, wäre die Option eines Filter. Der Filter würde zum Beispiel dafür sorgen, dass nur solche Teilchen rekonstruiert werden, bei denen mindestens 50 % der erwarteten Ruhemasse erzielt wird. Damit könnte dafür gesorgt werden, dass ein weiterer Teil der Rekonstruktionen eingespart wird, was sich wiederum positiv auf die Laufzeit der Analyse und auf den Speicherbedarf der Ausgabedatei auswirkt.

7.1.2 Verbesserung der Dokumentaion

Das Ergebnis der Umfrage und die Auswertung der Softwaremetriken zeigen, dass die Dokumentation von VISPA und PXL noch verbesserungswürdig ist. Die Anzahl der Beispielanalysen könnte erhöht werden. Im Bereich der Astroteilchenphysik gibt es beispielsweise kein einziges Analysebeispiel. Auch wäre eine Verlinkung in der Hilfe von VISPA zur Online-Klassendokumentation von PXL vorteilhaft. Noch besser wäre, die Klassendokumentation von PXL mit zu installieren. So hat man erstens eine Klassendokumentation, die zur installierten Version von PXL passt, und zweitens muss der Benutzer keinen Onlinezugang haben, um die Dokumentation nutzen zu können. Der Quellcode von PXL könnte noch besser dokumentiert werden. Dies würde sich positiv auf die Klassendokumentation auswirken und damit positiv auf die Bedienbarkeit von VISPA.

7.1.3 Parallelisierung der Analysen

Eine sinnvolle Ergänzung für PXL wäre die Nutzung von Mehrprozessorsystemen bei der Durchführung der Analyse. Eine solche Verbesserung könnte die Ausführungsgeschwindigkeit von Analysen, je nach Anzahl der zur Verfügung stehenden Prozessoren, vervielfachen. Eine Realisierung könnte so aussehen, dass während des Durchlaufens der Analysekette

eines Objektes ein eigener Thread gestartet wird. Dies ist jedoch mit einem nicht zu unterschätzenden Umbau der Analyseausführung verbunden. Eine weitere Schwierigkeit ist, dass dafür gesorgt werden muss, dass die Objekte in der richtigen Reihenfolge in eine mögliche Ausgabedatei geschrieben werden. Wenn beispielsweise Objekt 3 schneller abgearbeitet wurde als Objekt 2, muss Objekt 3 solange warten bis Objekt 2 in die Ausgabedatei geschrieben wurde.

7.2 Weitere Erweiterungen von VISPA und PXL

7.2.1 International Linear Collider ILC

Der ILC ist ein geplantes Projekt eines Elektron-Positron Beschleunigers. Dort sollen die Elementarteilchen mit einer Energie von 500 GeV bis 1000 GeV zur Kollision gebracht werden. Das Projekt befindet sich zur Zeit noch in der Planungsphase. Jedoch befindet sich beispielsweise die Software für die Detektoren schon in der Entwicklung. Damit VISPA und PXL am zukünftigen ILC genutzt werden können, wird ein so genannter Datenumfüller implementiert. Der Datenumfüller ist ein Programm, der die Daten des Detektors in das PXL Format transformiert.

7.2.2 CMS

Für den CMS-Detektor am LHC existiert bereits ein Datenumfüller. Damit ist die Datenanalyse am CMS-Detektor mit VISPA bzw. PXL möglich und wird bereits jetzt von einigen Wissenschaftlern genutzt.

7.3 Mögliche alternative Implementierung

7.3.1 Internetseite

Eine Alternative zur aktuellen Implementierung von VISPA ist die Implementation über eine Internetseite, VISPA@web. Die grafische Benutzeroberfläche zum Aufbau der Analysestruktur und die Ansicht von Objekten, wie Ereignissen, ist nun in Javascript realisiert. Für die grafische Benutzeroberfläche wird das Javascript-Paket ExtJS verwendet. Die Erstellung der Analysestruktur findet nun innerhalb eines Internetbrowsers statt. Weiterhin befindet sich auf dem Server eine Python-Installation so wie die zur Analyseausführung notwendigen dll-Dateien. Pxlrun befindet sich hingegen nicht auf dem Server. Der wesentliche Unterschied ist, dass, nicht wie bisher die Analyse in einer XLM-Datei abgespeichert und dann von Pxlrun eingelesen und ausgeführt wird, sondern dass bei jeder Veränderung der Analysestruktur ein Request an den Server übermittelt wird. In diesem Request ist die Veränderung der Analysestruktur enthalten. Ein Pythonskript bearbeitet diesen Request und führt unter Verwendung der Funktionen der Modulchain-Klasse von PXL sofort eine Modifikation der Analysekette durch. Damit das Pythonskript die PXL-Klassen nutzen kann, sind natürlich auch die von SWIG generierten Schnittstellen auf dem Server nötig. Auch meldet das Pythonskript, welches die Anfrage bearbeitet, Informationen an das Javascript zurück. Auch das Ausführen der Analysekette übernimmt ein Pythonskript. Weil nun mehrere Benutzer auf die Internetseite zugreifen können, ist eine Benutzerverwaltung Pflicht. Die Benutzerverwaltung muss regeln, welche Analyse zu welchem Benutzer gehört und Analysen vor fremdem Zugriff schützen. Auch muss die Benutzerverwaltung dafür sorgen, dass Rechenzeiten den verschiedenen Benutzern bzw. Analysen zugewiesen werden, dabei handelt es sich um das bekannte Schedulingproblem. Weiterhin muss das

Hochladen von Eingabedateien und Skriptdateien geregelt werden. Hier gibt es auch noch zusätzlich Sicherheitsaspekte. Bei den hoch geladenen Pythonskripts muss sichergestellt werden, dass diese keinen Schaden auf dem Server anrichten können. Es müssen eine Reihe von sicherheitskritischen Funktionen in Python für die Benutzer deaktiviert werden. Für die Benutzerverwaltung bietet sich eine Datenbank an, in der die Benutzer gespeichert und die Analysen so wie Resultate archiviert werden.

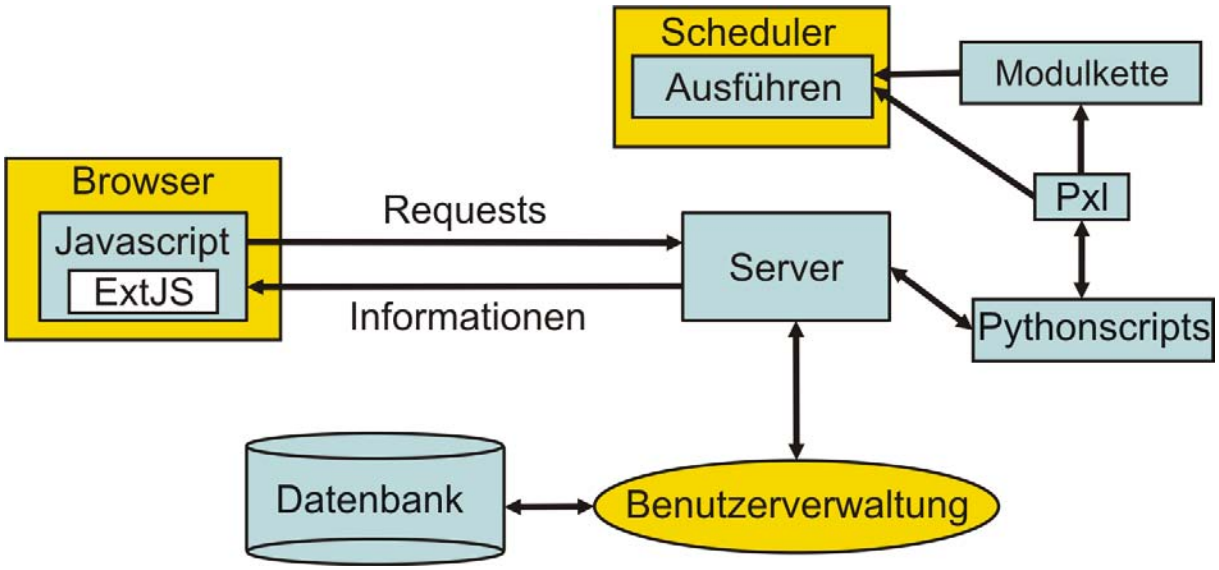


Abbildung 50: Alternative Implementation über einer Internetseite.

7.3.2 Vor- und Nachteile der Implementierung

Die Vorteile der Implementierung wären, dass der Benutzer VISPA, PXL und eine Reihe anderer nötiger Softwarepakete nicht bei sich installieren muss. Er braucht sich nur auf der Internetseite registrieren und kann sofort mit VISPA@web starten. Der Benutzer könnte seine Analysen und Resultate auch für Andere freigeben. Dort könnten andere Benutzer die Analysen und Resultate kontrollieren und verbessern, ohne dass die Analysen zwischen den Physikern zirkulieren müssen.

Jedoch birgt die Implementation auch Nachteile für den Benutzer. Aus Speicherplatzgründen wird es nicht möglich sein Analysen durchzuführen, die große Mengen an Daten erzeugen. Jedem Benutzer kann nur eine gewisse Kapazität an Festplattenspeicher zur Verfügung gestellt werden. Des weiteren ist es aus Sicherheitsgründen nicht möglich eigene C++ Module hochzuladen. Es können nur Pythonskripts verwendet werden. Da die zur Verfügung stehende Rechenleistung auf die Benutzer aufgeteilt werden muss und die Anzahl der Benutzer schwanken kann, ist eine konstante Laufzeit von Analysen nicht gegeben. Wenn viele Benutzer gleichzeitig Analysen ausführen, steigt die Laufzeit der Analysen an. Auch für die Implementierung ergibt sich ein zusätzlicher Aufwand. Die Anforderung an die Benutzerverwaltung und die Sicherheitsaspekte sind nicht zu unterschätzen. Außerdem ist die optimale Strategie für das Schedulingproblem nicht einfach zu finden.

Ideal wäre die Implementation für den Einsatz in der Lehre. Studenten könnten online ihre Aufgaben lösen und ihre Analysen ausführen. Solche Analysen, die bei den Aufgaben erstellt und ausgeführt werden müssen, sind nicht sehr rechenaufwendig und verbrauchen kaum Speicherplatz.

Literatur

[Aut09] AutomatedQA. http://www.automatedqa.com/products/aqtime/, 2009. [Online; Stand 26.Dezember 2009].

[BP+08] Tim Bray, Jean Paoli, et al. http://www.w3.org/tr/xml, 2008. [Online; Stand 26.November 2008].

[Cod05] CodeAnalyzer. Codeanalyzer - multi-platform java code analyzer, 2005. [Online; Stand 2.Januar 2010].

[Col06] The CMS Collaboration. *CMS Physics Technical Design Report: Detector Performance and Software*, volume 1. 2006. CERN/LHCC 2006-001 CMS TDR 8.1.

[Deb09] Debian. http://shootout.alioth.debian.org/u32/benchmark.php?, 2009. [Online; Stand 18.Dezember 2009].

[Deu96] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3 (RFC151).* Aladdin Enterprises, 1996.

[DG96] P. Deutsch and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3 (RFC150).* Aladdin Enterprises, 1996.

[EAo08] Martin Erdmann, Oxana Actis, and others. *Physics eXtension Library (PXL) User Guide*, 2008. http://pxl.sourceforge.net/manual.pdf.

[Fou09a] Python Software Foundation. http://docs.python.org/library/time.html, 2009. [Online; Stand 26.Dezember 2009].

[Fou09b] Python Software Foundation. http://www.python.org/dev/culture/, 2009. [Online; Stand 18.Dezember 2009].

[Fou09c] Python Software Foundation. http://www.python.org/doc/faq/general/, 2009. [Online; Stand 18.Dezember 2009].

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 1998.

[Har10] Verena Hartwig. http://www.global-assess.rwth-aachen.de/testmaker-wiki/de/index.php/hauptseite, 2010. [Online; Stand 2.Januar 2010].

[Heb08] Thomas Hebbecker. http://www.weltderphysik.de/de/6443.php, 2008. [Online; Stand 18.Dezember 2009].

[HKH81] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 81–88, New York, NY, USA, 1981. ACM.

[Huf51] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E*, 40:1098–1101, 1951.

[iD09] Astroteilchenphysik in Deutschland. http://www.astroteilchenphysik.de/ topics/cr/cr.htm, 2009. [Online; Stand 9.Dezember 2009].

[Inc09] Persistent Designs Inc. http://persistentdesigns.com/wp/?p=260, 2009. [Online; Stand 1.Januar 2010].

[Kub07] Sven Kubiak. *Antwort- und Laufzeitmessung: Prinzip, Implementierung und Experiment*. GRIN Verlag, 2007.

[LA07] Mark Lutz and David Ascher. *Einführung in Python*. O´Reilly, 2 edition, 2007. S.16 - S.20.

[Lig02] Peter Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. Spektrum. Akademischer Verlag, 2002.

[Lit01] Tim Littlefair. *An investigation into the use of software code metrics in the industrial software development einvironment*. PhD thesis, At the Faculty of Communications, Health and Science, Edith Cowan University, Mount Lawley Campus., 2001.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING VOL. SE-2, NO.4*, pages 308–320, 1976.

[Mic09] Microsoft. http://msdn.microsoft.com/en-us/library/dd757629(vs.85).aspx, 2009. [Online; Stand 26.Dezember 2009].

[Mun03] John C. Munson. *Software Engineering Measurement*. Auerbach Publications, 2003.

[MW96] Thomas J. McCabe and Arthur H. Watson. A testing methodology using the cyclomatic complexity metric. Technical report, The National Institute of Standards and Technology (NIST), 1996.

[Nar05] I. Narsky. Statpatternrecognition: A c++ package for statistical analysis of high energy physics data, 2005.

[Nie93] Jakob Nielsen. *Usability Engeneering*. Morgan Kaufmann, 1993.

[Per] Technical report on c++ performance. Technical Report IEC TR 18015:2006(E), ISO.

[Pyt09] Pythia. http://home.thep.lu.se/ torbjorn/pythia.html, 2009. [Online; Stand 23.Dezember 2009].

[ROO09] ROOT. http://root.cern.ch/drupal/content/about, 2009. [Online; Stand 23.Dezember 2009].

[RSvdW09] Michiel Roza, Mark Schroders, and Huub van de Wetering. A high performance visual profiler for games. In *Sandbox '09: Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, pages 103–110, New York, NY, USA, 2009. ACM.

[SMS06] Torbjorn Sjostrand, Stephen Mrenna, and Peter Skands. Pythia 6.4 physics and manual. *JHEP*, 0605:026, 2006.

[Sta09] StatPatternRecognition. http://statpatrec.sourceforge.net/, 2009. [Online; Stand 23.Dezember 2009].

[STL00] Peter Schenkel, Sigmar-Olaf Tergan, and Alfred Lottmann. *Qualitätsberurteilung multimedialer Lern- und Informationssysteme. (Evaluationsmethoden auf dem Prüfstand).* BW Bildung und Wissen. Verlag und Software GmbH Nürnberg, 2000.

[vH10] Dimitri van Heesch. http://www.stack.nl/ dimitri/doxygen/index.html, 2010. [Online; Stand 2.Januar 2010].

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, pages 337–343, 1977.

[Zus98] Horst Zuse. *A Framework of Software Measurement.* de Gruyter, 1998.

Abbildungsverzeichnis

Tabellenverzeichnis