MASTERARBEIT

Anforderungen, Schnittstellen und deren Prototypen für eine Datenbank zur Erfassung der Berechnungsergebnisse und Einstellparameter aus Elektronenoptik-Simulationen von Röntgenröhren

vorgelegt im November 2019 von
Thu Khang Cao

1. Gutachter: Prof. Dr. Udo van Stevendaal
2. Gutachter: Dipl. Ing. Steffen Holzapfel

in Zusammenarbeit mit
Philips Medical Systems DMC GmbH

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Fakultät Life Sciences
Studiengang Biomedical Engineering

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Anforderungen, Schnittstellen und deren Prototypen für eine Datenbank zur Erfassung der Berechnungsergebnisse und Einstellparameter aus Elektronenoptik-Simulationen von Röntgenröhren

Masterarbeit vorgelegt von
Thu Khang Cao

Inhaltsverzeichnis

1 Einführung

Als Wilhelm Conrad Röntgen die Röntgenstrahlen im Jahr 1895 entdeckte, wurde für Mediziner ein Traum wahr: In den Körper blicken, ohne ihn öffnen zu müssen. Die dadurch möglich gewordenen Bilder des Körperinnern revolutionierten die medizinische Welt und dienen in der heutigen Zeit als wichtiges und unverzichtbareres Hilfsmittel sowohl in der Diagnostik als auch in der Therapie von verschiedenen Erkrankungen. Die Röntgendiagnostik ist einer der am häufigsten angewandten bildgebenden Verfahren in der Medizin, bei dem ein Körper unter Verwendung eines Röntgenstrahlers durchstrahlt wird. Diese Technik wird eingesetzt, um zum Beispiel Knochenbrüche, Kopfverletzungen und frühzeitigen Krebs zu diagnostizieren [1, 2].

Röntgenstrahlen werden in einer sogenannten Röntgenröhre erzeugt und dringen bei einer Aufnahme durch das zu untersuchende Körperteil. Die Strahlen werden je nach Beschaffenheit des bestrahlten Gewebes unterschiedlich stark absorbiert, welche schlussendlich auf dem Röntgenfilm sichtbar werden [1]. Innerhalb der Röhre wird mittels einer angelegten Spannung ein Elektronenstrahl erzeugt, wodurch eine Abbremsung der Elektronen zur Bildung von Röntgenstrahlen führt. Während des Entwicklungsprozesses einer solchen Röntgenröhre gibt es die Möglichkeit, eine Simulationssoftware einzusetzen, um die gesamte Röhre mit ihren Einzelkomponenten und den berechneten Elektronenverlauf nachzubilden. Diese Methode bringt viele Vorteile für den Entwicklungsprozess der Röhre mit sich, welche im späteren Abschnitt näher erläutert werden. Um die Elektronenoptik einer Röntgenröhre simulieren zu können, braucht das Programm gewisse Einstellparameter, die die Geometrie sowie die Betriebsbedingungen abbilden. Neben den Geometriedaten, die den Aufbau der Röhre bestimmen, werden zusätzlich Berechnungsparameter bzw. physikalische Größen benötigt, um den Elektronenverlauf zu berechnen und zu simulieren. Verschiedene Einstellparameter führen hierbei zu unterschiedlichen Berechnungsergebnissen. Um das optimale Ergebnis zu erzielen, erfordert das Simulationsprogramm eine Vielzahl an Simulationsdurchläufen.

Heutzutage gibt es eine große Anzahl an Unternehmen, die mithilfe von Simulationssoftwares komplette Röntgensysteme und einzelne Komponenten entwickeln. Die Firma Philips Medical Systems Development and Manufacturing Centre (DMC) GmbH, Teilbereich von Philips Healthcare, ist eine von denen und gehört zu den marktführenden Produktionsstätten für diagnostische Bildgebung und bildgestützte Therapie. In der Geschäftseinheit Imaging Components (IC) werden Einzelkomponenten wie Röntgenröhren, Generatoren und weitere Bestandteile für bildgebende Röntgensysteme entwickelt und gefertigt [3].

Dabei wird ein Simulationsprogramm seit Jahren als ein unabdingbares Werkzeug für die Mitarbeiter der Röhrenentwicklung angesehen. In dem momentanen Arbeitsablauf werden die benötigten Parameter in unterschiedlichen Arten von Dateien abgespeichert, auf die das Programm zugreift, um die entsprechende Röhre zu simulieren. Mit der Zeit stieg die Zahl der Simulationen und die dazugehörigen Dateien stetig an, was zukünftig Nachteile für den Arbeitsprozess bei Philips mit sich bringen kann. Bei einer großen Anzahl an Dateien besteht das permanente Risiko von Absenz. Für eine erfolgreiche Simulation sind Dateien mit bestimmten Parametern essenziell. Da die Simulationssoftware auf mehrere Dateien angewiesen ist, würde die Simulation bei Unvollständigkeit der Informationen fehlschlagen. Ein weiteres Problem ist die lokale Datenhaltung. Jeder Anwender besitzt einen Simulationsrechner, auf dem nur die Daten gespeichert sind, die auf dem Rechner simuliert worden sind. Für die aktuelle Arbeitsweise bedeutet das, dass nicht jeder Anwender Zugriff auf alle Simulationsergebnisse und die dazugehörigen Informationen hat. Im Falle einer Projektübergabe müssten die benötigten Daten zwischen den Anwendern ausgetauscht werden, was Zeit in Anspruch nimmt. Dementsprechend sind Parameter und Ergebnisse von Simulationen sehr schwer miteinander zu vergleichen, da diese Art von Datenhaltung keine Relationen zwischen den Parametern aufzeigt. In Tabelle 1 sind die aufgeführten Nachteile des bisherigen Arbeitsablaufs kurz zusammengefasst.

• Mögliches Abhandenkommen von Daten
• Beschränkter Zugriff auf Daten aufgrund lokaler Datenhaltungen
• Aufwendige Auswertung bzw. Vergleich der Daten

Tab. 1: Nachteile der Datenhaltung im bisherigen Arbeitsablauf (Eigendarstellung)

Für eine Lösung dieser Probleme entstand die Idee einer Datenbank, auf die alle Nutzer des Simulationsprogramms Zugriff haben können. Die Datenbank hat den Zweck, die Anzahl der Dateien zu reduzieren und eine einheitliche Sicherung der Parameter zu ermöglichen.
Ziel der vorliegenden Arbeit ist es, den jetzigen Workflow bei Philips durch eine Zentralisierung der Datenhaltung zu verbessern. Hierbei werden zum einen die Anforderungen für eine passende Datenbank ausgearbeitet und zum anderen die Schnittstellen zwischen Datenbank und Simulationssoftware durch erste Prototypen realisiert.

Um das Ziel zu erreichen, wird im ersten Schritt der bisherige Arbeitsablauf mit der Simulationssoftware untersucht. Diesbezüglich soll eine informative Befragung der Anwender des Simulationsprogramms durchgeführt werden, um die routinierte Arbeitsweise der Nutzer besser zu verstehen. Die verschiedenen Arten der Parameter und die aktuelle Datenhaltung sind in dem Fall wichtige Informationen. Anhand von Simulationsbeispielen werden zugleich Schwachstellen und mögliche zukünftige Probleme in der Arbeitsfolge aufgezeigt. Nachfolgend wird eine passende Datenbank ausgewählt, um das Ziel der Zentralisierung der Datenhaltung näher zu kommen. Hierbei wird eine Anforderungsanalyse durchgeführt und anhand des Anforderungsprofils nach einem geeigneten Datenbankverwaltungssystems recherchiert. Danach werden eine Analyse und Bewertung der Datenbanken auf dem Markt hinsichtlich ihrer technischen Besonderheiten durchgeführt. Die Idee Datenbankstruktur wird mithilfe von Microsoft Access, ein Programm zur Erstellung und Verwaltung von Datenbankanwendungen, aufgebaut und in die ausgewählte Datenbank implementiert. Da die Erfassung von Berechnungsergebnissen und Einstellparametern der Simulationen ein wichtiges Ziel dieser Arbeit ist, werden schlussendlich Prototypen von Schnittstellen zwischen der Datenbank und dem Simulationsprogramm programmatisch verwirklicht.

2 Grundlagen und Simulation in der Röntgendiagnostik

In der vorliegenden Arbeit wird ein Konzept für die Datenhaltung der Parameter und Ergebnisse für das Simulationsprogramm erstellt und verwirklicht. Zumal in der Röhrenentwicklung das simulierte Produkt der Röntgenstrahler ist, werden zunächst die theoretischen Grundlagen einer Röntgenröhre und die Entstehung von Röntgenstrahlung vorgestellt. Darauffolgend wird die Simulation der Elektronenoptik von Röntgenröhren thematisiert.

2.1 Aufbau und Funktion der Röntgenröhre

Röntgenstrahlen sind elektromagnetische Wellen, die für das menschliche Auge unsichtbar sind. Diese Strahlen liegen im elektromagnetischen Spektrum zwischen dem UV-Licht und der Gamma-Strahlung. Um mit Röntgenstrahlen Bilder produzieren zu können, wird eine Röntgenstrahlenquelle (Röntgenröhre) und eine Aufnahmeeinheit (fotografischer Film) benötigt. Bei der Untersuchung befindet sich der Patient zwischen Quelle und Film, während Röntgenstrahlen durch ihn hindurchtreten. Die unterschiedlichen Dichten der Gewebe (Weichteilgewebe, Knorpel, Knochen) und Hohlräume im menschlichen Körper führen zu unterschiedlichen Abschwächungen der Strahlen, welche schließlich auf den Röntgenfilm treffen. Je strahlendurchlässiger die Strukturen, desto dunkler erscheinen diese auf dem Röntgenfilm. Strahlendichte Strukturen hingegen stellen sich hell dar. Somit entsteht mithilfe von Röntgenstrahlen ein sichtbares Bild. Die Quelle, an der die Erzeugung der Strahlen stattfindet, wird Röntgenröhre genannt [1, 4].

Die Röntgenröhre ist die bedeutendste Strahlungsquelle der Radiologie, welche seither zur Leistungsfähigkeit weiterentwickelt wurde, dennoch ihr Grundkonzept beibehalten hat. Außerdem ist diese historisch gesehen die erste Quelle für die künstliche Erzeugung von ionisierender Strahlung. Als ionisierende Strahlung wird jede elektromagnetische Strahlung bezeichnet, die in der Lage ist, Elektronen aus Atomen oder Molekülen zu entfernen. Durch diesen Vorgang werden die Atome zu reaktiven Ionen, welche in lebendem Gewebe großen Gesundheitsschaden anrichten können [5].

Die Beschreibung des Röhrenaufbaus geschieht im weiteren Verlauf in Bezugnahme an Abbildung 1. Generell besteht jede Röntgenröhre aus zwei Elektroden: einer heizbaren Kathode (1) und einer Anode (2), die sich unter einem hohen Vakuum befinden. Das Prinzip der Röhre zur Erzeugung der Strahlen basiert auf der Beschleunigung der Elektronen, weshalb diese auch zur Kategorie der Gleichspannungsbeschleuniger angehört. Doch bevor diese beschleunigt werden, müssen sie erst einmal erzeugt werden. Dies geschieht an der Kathode, welche als Elektronenquelle dient.

Durch das Erhitzen der Kathode mit einer sogenannten Heizspannung werden Elektronen (3) emittiert bzw. freigesetzt. Es wird eine Hochspannung (auch Anodenspannung) im Bereich von 30-150kV zwischen den beiden Elektroden angelegt, wodurch es zur Beschleunigung der Elektronen zur Anode führt. Der Ort, auf den die Elektronen auftreffen, wird als Brennfleck (4) bzw. Fokus bezeichnet. An dieser Schnittstelle wird Energie freigesetzt, wobei ein großer Teil in Wärme umgewandelt und ein sehr kleiner Teil in Röntgenstrahlung (5) umgesetzt wird. Die entstandene Röntgenstrahlung teilt sich wiederum in zwei Komponenten auf: die kontinuierliche Bremsstrahlung und die diskrete charakteristische Strahlung [2, 5].

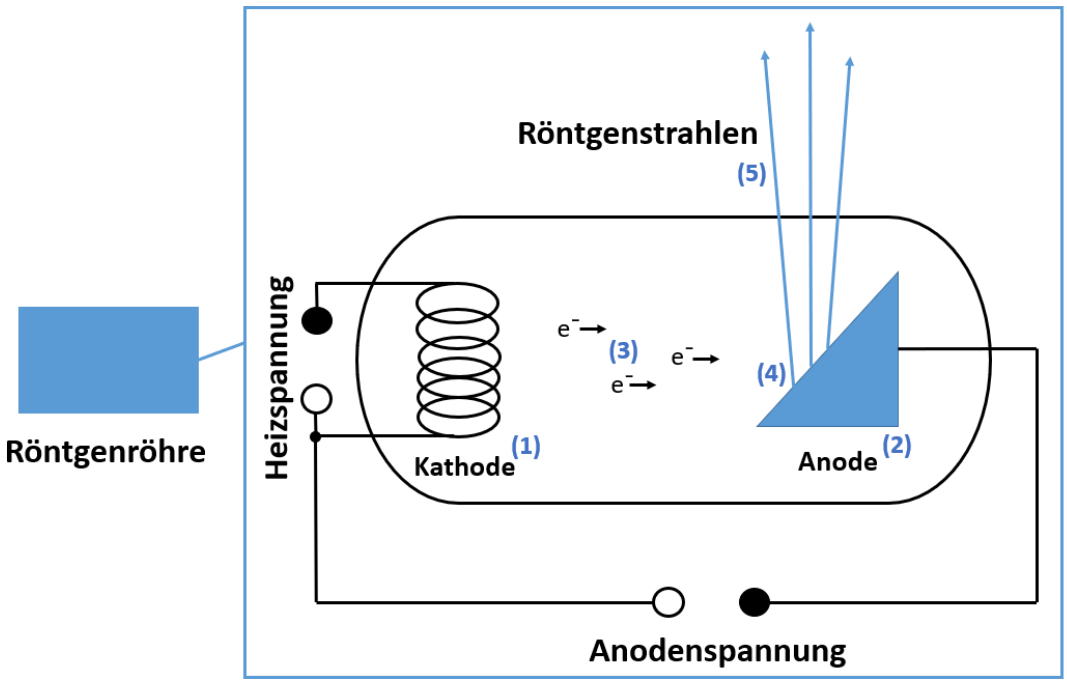


Abb. 1: Aufbau der Röntgenröhre (Eigendarstellung)

Die Bremsstrahlung resultiert aus der Wechselwirkung zwischen dem beschleunigten Elektron und einem Atomkern des Target-Materials. Jede Geschwindigkeitsänderung eines geladenen Teilchens erzeugt Strahlung. In Abbildung 2 entspricht der rote Pfeil die Flugbahn des Elektrons, die dicht am positiv-geladenen Kern bzw. Nucleus (blau) liegt. Durch die Abbremsung des Elektrons im elektrischen Feld des Atomkernes, wird ein Teil der Energie in Form von sogenannter Bremsstrahlung (grüner Pfeil) abgegeben [5, 6].

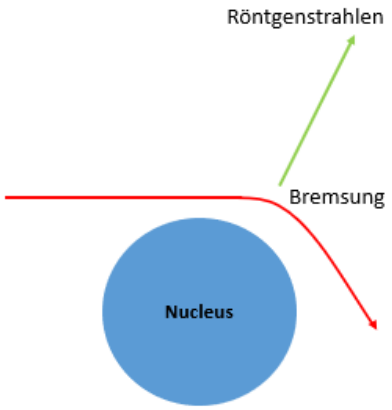


Abb. 2: Entstehung von Bremsstrahlung (Eigendarstellung)

Die charakteristische Strahlung ist die Folge einer Interaktion zwischen einem freien beschleunigten Elektron und einem in der Atomhülle gebundenen Elektron. Mithilfe des Schalenmodells, welches in Abbildung 4 dargestellt ist, lässt sich die Entstehung der charakteristischen Strahlung wie folgt erklären: Das gebundene Elektron (2) wird durch das Freie (1) aus seiner Schale herausgestoßen. Zur Folge füllt ein Elektron aus einer höheren Schale (3) diese Lücke aus. Durch die Energiedifferenz zwischen den beiden Schalen entstehen dabei die charakteristischen Röntgenstrahlen (4) [6].

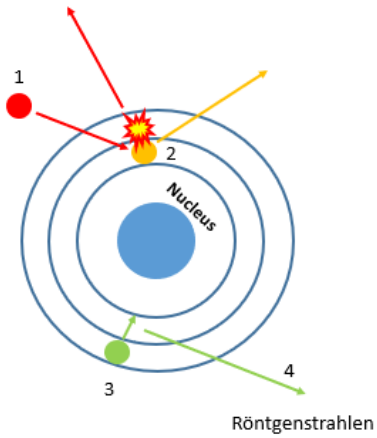


Abb. 3: Entstehung von charakteristischer Röntgenstrahlung (Eigendarstellung)

Für spezifische Bereiche des Körpers werden bestimmte Strahlenqualitäten benötigt. Es können weiche und harte Strahlungen erzeugt werden, um ungleich dichte Gewebe zu durchdringen. Weiche Strahlungen lassen Weichstrahlaufnahmen entstehen, die feinste Gewebeunterschiede auf dem Röntgenfilm darstellen können. Je weicher die Strahlung ist, desto größer ist der Anteil der vom Gewebe absorbierten Strahlung. Nachteil hierfür ist die mögliche Folge der hohen Strahlenbelastung. Hartstrahlaufnahmen besitzen im Gegensatz zu Weichstrahlaufnahmen einen geringeren Kontrast, aber dafür auch eine kleinere Strahlenbelastung [2].

Die Bildgebung mit ionisierender Strahlung macht es demnach möglich, Abbildungen hoher Auflösung und Kontrast zu erstellen. Die Bildschärfe nimmt mit zunehmendem Abstand zwischen Röntgenfilm und dem darzustellenden Objekt ab. Das bedeutet je näher das Objekt am Röntgenfilm ist, desto besser ist die Bildqualität. Der Kontrast ist abhängig von der Strahlendosis, Filter und Streustrahlenreduktion. Bei einer guten Röntgenbildgebung besteht die Schwierigkeit, den Kompromiss zwischen Strahlenschutz des Patienten und zufriedenstellender Bildqualität zu finden [6, 7].

Je nach Einstellungen der verschiedenen Parameter an der Röntgenröhre, ergeben sich unterschiedliche Resultate hinsichtlich der Strahlenqualität, welche im direkten Bezug mit der Bildqualität steht. Bei der Parameterwahl spielt die Brennfleckgröße eine wichtige Rolle, denn dieser beeinflusst sehr stark die Strahlengeometrie und ist damit mitbestimmend für die Bildschärfe. Grundlegend wird der Brennfleck in drei Kategorien aufgeteilt: elektronischer Brennfleck, thermischer Brennfleck und optischer Brennfleck. Der elektronische Brennfleck wird von der Normenausschuss Radiologie (NAR) im DIN EN 60336 als Schnittfläche des Elektronenstrahlbündels mit der Anodenoberfläche definiert. Der thermische Brennfleck ist der vom Elektronenstrahlbündel getroffene Anteil auf der Anode. Bei Festanoden sind beide genannten Brennflecke gleich groß und bei Drehanoden sind diese aufgrund der Drehung des Anodentellers unterschiedlich. Der optische Brennfleck entspricht der senkrechten Projektion des elektronischen Brennflecks und ist im rechten Winkel zum Zentralstrahl angeordnet. Er befindet sich demnach auf der Bildempfängerebene [5, 6].

Röntgenröhren kommen mit unterschiedlich großen Brennflecken zum Einsatz. Je kleiner der Fokus desto besser ist die Zeichenschärfe und je größer der Fokus, desto höher ist die Belastbarkeit der Röntgenröhre. Ein kleiner Brennfleck kommt zum Einsatz, um feinere Strukturen darzustellen, wohingegen ein großer Brennfleck für dickere Objekte verwendet wird [7].

Zusammenfassend lässt sich sagen, dass die Eigenschaften des Röntgenstrahls in direkter Beziehung zum Brennfleck stehen, da die Strahlen vom Fokus aus starten. Das bedeutet, dass die Aufnahmeparameter bestmöglich auf den Brennfleck abgestimmt sein müssen, um einen optimalen Strahl und damit das ideale Ergebnis zu erzielen.

2.2 Simulation der Elektronenoptik von Röntgenröhren

Bekanntlich sind Simulationen in der Luftfahrt- und Automobilindustrie sehr stark vertreten. Diese haben auf vielfältige Weise zur Entstehung und Entwicklung der Simulation in der Medizin bzw. Medizintechnik beigetragen [8]. Doch wo werden Simulationen in der Medizintechnik verwendet?
Die gängigste Vorgehensweise ein komplexes System zu untersuchen, ist dieses zu simulieren. Bei einer Simulation werden Experimente an einem Modell durchgeführt, um Erkenntnisse über das reale System zu gewinnen. Der Ablauf des zu simulierenden Systems mit bestimmten Werten oder Parametern wird als Simulationsexperiment bezeichnet. Schließlich können dann Aussagen und Entscheidungen über die daraus entstehenden Ergebnisse vollzogen werden. Als Modellierung wird die Entwicklung eines neuen Modells bezeichnet. Ist das Modell für das zu lösende Problem geeignet, müssen lediglich die Parameter eingestellt werden [8, 9].
In der Röhrenentwicklung bei Philips ist der Einsatz von einer Simulationssoftware unabdingbar. Diese wird genutzt, um die Thermik, Mechanik, Elektronik und Elektronenoptik von Röntgenröhren nachzubilden. Zu Beginn des Simulationsablaufs wird ein Modell aufgebaut. Dies wird ermöglicht, indem das Programm Informationen über die Geometrie der Röhre übermittelt bekommt und diese dann verarbeitet. Getrennt von dem Modellaufbau, werden zusätzlich Randbedingungen in Form von sogenannten Berechnungsparametern in das Programm eingelesen, um den Elektronenstrahl zwischen Kathode und Anode nachzuahmen. Diese Parameter versorgen das Modell mit physikalischen Größen wie Temperatur, Spannung, Strom und viele mehr, was dazu führt, dass die Röhre ihre Funktion erlangt, Röntgenstrahlen zu erzeugen.
Die Anwendung einer Simulationssoftware spielt eine große Rolle bei der Entwicklung der Elektronenoptik. Primär für den Designprozess ist eine derartige Software mit sehr vielen Vorteilen verbunden. In erster Linie ist es nützlich, die Fokussierung und Tiefensteuerung von Elektronenstrahlen zu verstehen. Es können die Ergebnisse nach jeder Berechnungsschleife und die Daten von Prototyp-Röhren gespeichert werden. Zur Folge werden sowohl Entwicklungszeit als auch Kosten für die Hardwarerealisierung gespart. Darüber hinaus wird im Hinblick auf die Nachversorgung des Produktlebenszyklus bei alten und neuen Produkten die Untersuchung von Problemen vereinfacht und unterstützt. Die genannten Vorteile sind in Tabelle 2 nochmal aufgeführt.

• Verständnis des Zusammenhangs zwischen Elektronenstrahl, Brennfleck und Röntgenstrahl
• Reduzierung der Entwicklungskosten durch Einsparung von Prototypen
• Verbesserung von alten und neuen Produkten

Tab. 2: Vorteile von Simulationen in der Elektronenoptik von Röntgenröhren (Eigendarstellung)

3 Aktueller Arbeitsablauf mit der Simulationssoftware Opera

In diesem Kapitel wird der momentane Arbeitsablauf, die Röntgenröhre zu simulieren, erläutert. Die Mitarbeiter aus der Röhrenentwicklung arbeiten dabei mit der Simulationssoftware „Opera“. Diese basiert auf der Finite-Elemente-Methode (FEM), ein numerisches Verfahren, das bei unterschiedlichen physikalischen Aufgabenstellungen angewendet wird.

Opera gilt heutzutage als ein wichtiges Werkzeug für industrielle und wissenschaftliche Anwendungen. Die Software besitzt umfangreiche Unterstützungsfunktionen, um konkrete Modelle zu erzielen und bietet genaue numerische Lösungen für reale Probleme in Bereichen wie Elektrostatik, Magnetostatik, dynamische Elektromagnetik, Hochfrequenz-Elektromagnetik, thermische Analyse und geladene Partikel [10].

In der Röhrenentwicklung von Philips wird das letztgenannte Ladungsmodul für die Simulierung von Röntgenröhren und Elektronenverläufe eingesetzt. Mit diesem Modul ist es möglich, die Wechselwirkung geladener Teilchen in elektrostatischen und magnetostatischen Feldern zu berechnen. Die Maxwell-Gleichungen, die den Zusammenhang von elektrischer und magnetischer Felder untereinander sowie elektrischer Ladungen und elektrischem Strom unter gegebenen Randbedingungen beschreiben, wird durch Verwendung der FEM gelöst. Während der Erzeugung der Partikel, werden sogar die Änderung der Potentialverteilung berücksichtigt, die durch ihre Ladung verursacht wird [11].

In einigen Simulationsaufgaben ist die Berechnung von magnetischen Linsen notwendig. Deren Magnetfeld kann optional ausgewertet und in die Simulation einbezogen werden.

Die Röntgenröhren werden basierend auf dem Konzept der sogenannten ELOP (elektrooptisch) Toolbox, ein Standardwerkzeug des Designprozesses der Elektronenoptik, simuliert. Die ELOP Toolbox ist eine Bibliothek mit verschiedenen Röhrenkomponenten, welche in der Befehlssprache von Opera verfasst sind. Die Möglichkeit besteht, das Modell mit neuen oder geänderten Komponenten zu erweitern, was das Simulieren von verschiedenen Röhren sehr flexibel gestaltet. Außerdem besitzt es zusätzliche Tools zur Berechnung des Brennflecks, das Ergebnis der Simulationen. Mithilfe der Integration der ELOP Toolbox in ein Versionskontrollsystem, können die Änderungen innerhalb der Toolbox somit verwaltet werden.

Die Abbildung 4 veranschaulicht den groben Arbeitsablauf mit der Simulationssoftware Opera. Für eine Simulation, also einer elektrooptischen Berechnung, wird ein Input benötigt, der hier als Einstellparameter abgebildet ist. Daraus ergibt sich ein Output bzw. die Ergebnisse, die im nachfolgenden Schritt ausgewertet werden können. Die ELOP-Kalkulation wird in dem Fall als Black Box dargestellt. Das bedeutet, dass nur das äußere Verhalten betrachtet wird und die Untersuchung sich nur auf die Input-Output-Beziehung beschränkt. Je nach Anforderungen werden die Einstellparameter so angepasst, dass die Ergebnisse mit den Anforderungen übereinstimmen. Hierbei wird nach jeder Änderung des Einstellparameters eine neue ELOP-Rechnung durchgeführt bis diese ein zufriedenstellendes Resultat liefert.

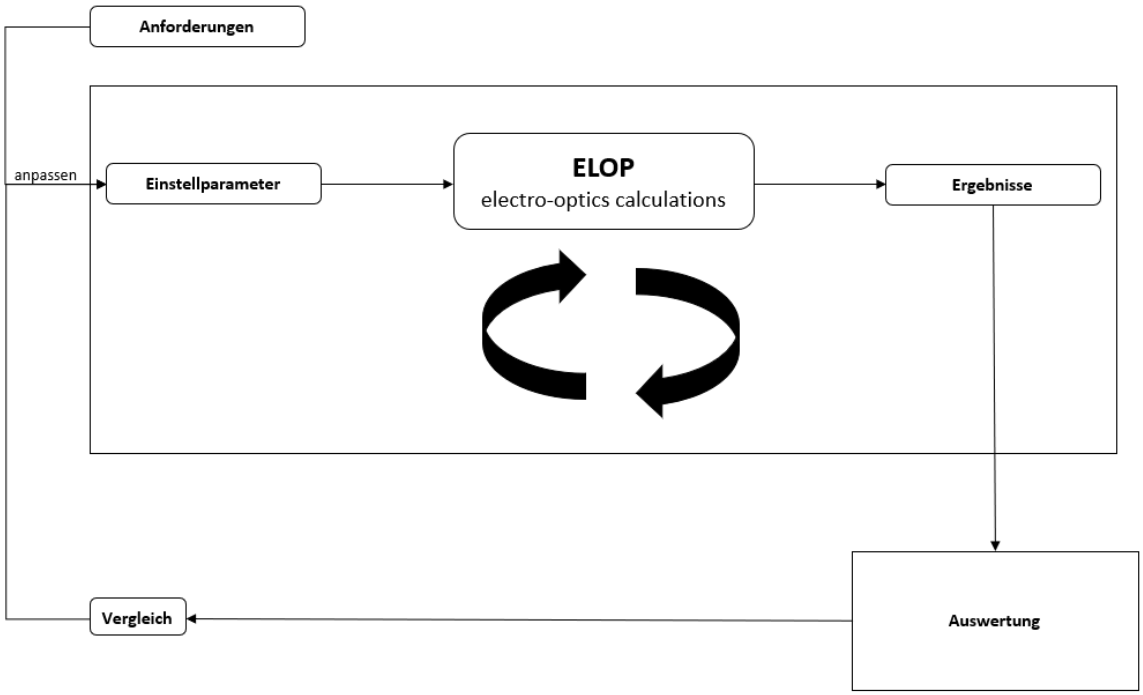


Abb. 4: Aktueller Arbeitsablauf mit dem Simulationsprogramm Opera (Eigendarstellung)

Im Weiteren werden die Bausteine „Einstellparameter“, „ELOP“, „Ergebnisse“ und „Auswertung“ aus dem Ablaufdiagramm detaillierter wiedergegeben.

Bevor eine Rechnung durchgeführt werden kann, müssen Einstellparameter definiert werden. Dies geschieht in bestimmten Textdateien, die für jede neue Berechnung neu angelegt werden. Die Parameter sind in den Dateien hinterlegt und werden von Opera für die Simulation benötigt.

Jede dieser Dateien besitzt ihre eigene ID-Nummer, die im Dateinamen festgelegt ist. In der SC-Datei wird mittels der Angabe der ID auf eine bestimmte SG-Datei verwiesen.

Wie in der Abbildung 5 zu sehen, lassen sich die Parameter in zwei Hauptdateien untergliedern: SG-, SC-Dateien. SG-Dateien beinhalten Geometrieparameter, die für die Modellierung der Röhrenkomponenten verantwortlich sind. In diesen Dateien stehen ebenfalls die Konfigurationen der einzelnen Teile, sprich welche Bestandteile miteinander verknüpft sind. Jede Komponente besitzt definierte Parameter, die einen Wert enthalten. Eine Änderung der Werte führt zu einer Modifikation der Geometrie am Modell.

SC-Dateien bestehen aus physikalischen Größen, wie Strom, Spannung, Temperatur, usw. Diese werden auch als Berechnungsparameter bezeichnet, weil die Röhre dadurch ihre Funktion erlangt, Elektronen von der Kathode zur Anode zu beschleunigen. Zur modellierten Röntgenröhre wird demzufolge ein Elektronenverlauf simuliert, womit man schließlich die Größe und Lage des Brennflecks erhält. Jede SC-Datei enthält einen Verweis auf die verwendete Geometrie. TG- und TC-Dateien enthalten ebenfalls Geometrie- und Berechnungsparameter, allerdings beziehen sich diese auf das Magnetfeld. Die Verwendung der TG- und TC-Parameter sind optional und werden nur für bestimmte Röhrentypen eingesetzt.

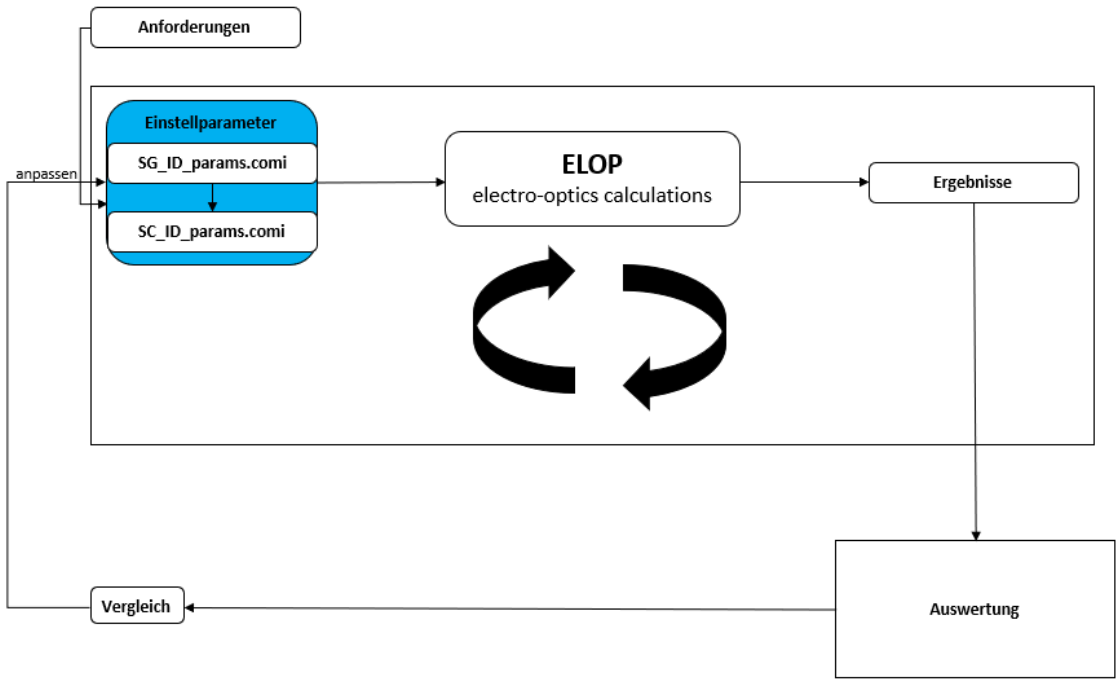


Abb. 5: Unterteilung der Einstellparameter in „*_params.comi“-Dateien (Eigendarstellung)

Sind Geometrie- und Berechnungsparameter in den „*_params.comi“-Dateien definiert, können diese von Opera für die Simulationsrechnungen zur Verfügung gestellt werden. Die Berechnung der Röhren werden in der Röhrenentwicklung bei Philips auch als ELOP-Rechnung bezeichnet. Der Block ELOP enthält zwei Bausteine: Modeller und Solver (siehe Abb. 6).

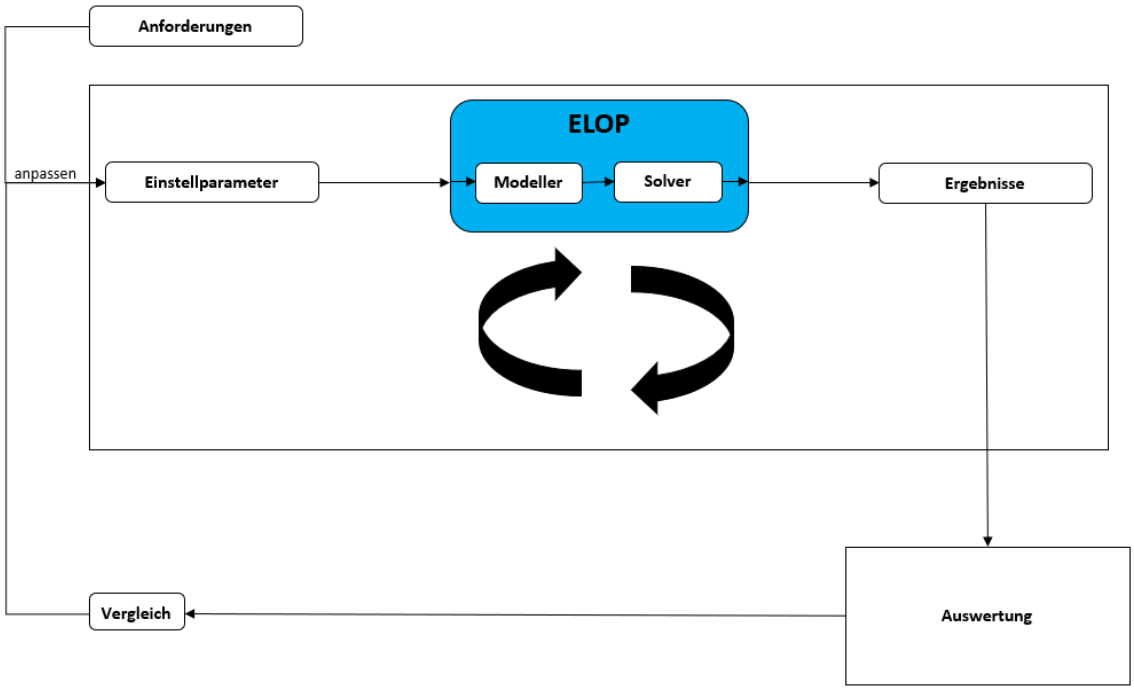


Abb. 6: Opera als Modellierer (Modeller) und Löser (Solver) (Eigendarstellung)

Zu Beginn des Simulationsablaufs werden durch den Modeller die Einzelkomponenten der Röhre aufgebaut und verknüpft (siehe Abb. 7, links). SG und SC werden im Modeller benutzt, um eine Datenbank mit den finiten Elementen aufzubauen, welche der Solver lösen kann. Auf diese Weise kann der Verlauf der beschleunigten Elektronen simuliert werden, welcher in Abbildung 7 rechts zu sehen ist.

Abb. 7: Modell der Röntgenröhre (links) und Verlauf der Elektronen zwischen Kathode und Anode (rechts) (Eigendarstellung)

Wichtig hierbei zu erwähnen ist, dass nicht der eigentliche Röntgenstrahl simuliert wird, sondern nur der Elektronenverlauf zwischen Kathode und Anode. Da der elektronische Brennfleck eine direkte Beziehung zum Röntgenstrahl besitzt, werden die Einstellparameter auf diesen abgestimmt.
Aufgrund der großen Bedeutung des Brennflecks, sind die Bezeichnung des Nennwertes, die zulässige Abweichung von Nennwert und die dazugehörigen Messmethoden normiert [6]**.**
In dem Dokument IEC 60336 „Medizinische elektrische Geräte – Röntgenstrahler für medizinische Diagnostik – Kennwerte von Brennflecken“ sind die Prüfmethoden zur Bewertung der Merkmale von Brennflecken festgelegt und die notwendigen Einrichtungen beschrieben. Es beinhaltet die Grundlage zur Bestimmung der Größe des Brennflecks und deren Toleranzen. Zweck der Normierung ist die Einhaltung der Sicherheitsanforderungen und des Strahlschutzes.
Während einer Opera-Simulation entstehen eine Menge Daten, die in verschiedenen Dokumenten abgespeichert werden. In Abbildung 8 wird illustriert, welche Dateien durch die ELOP-Rechnung erstellt werden. Der Modeller kommt zweimal zum Einsatz. Wie bereits beschrieben, wird zunächst die SG-Datei verarbeitet. Daraus ergibt sich ein sogenanntes „*.opcb“ Dokument, das als wichtigen Bestandteil über die Geometrie aufgebaute Maschengitter beinhaltet. Durch das Setzen der Randbedingungen erhält man zur Folge zwei Dateien: opc und op3.

Die Information in der opc-Datei umfasst die Geometriedaten und Randbedingungen, wohingegen die op3 eine Datenbank mit den finiten Elementen und deren Gleichungen ist. Im Anschluss daran wird der Solver verwendet, welcher die bisher ungelöste op3 Datenbank löst. Das Ergebnis einer ELOP Rechnung ist neben der gelösten Datenbank eine tracks-Datei, welche die Pfade der Elektronenbahnen beinhaltet. Aus den Elektronenbahnen ist es schließlich möglich, Kennwerte über den entstandenen Brennfleck zu erhalten. Die Tabelle 3 fasst alle entstandenen Dateiformate mit den entsprechenden Inhalten zusammen.

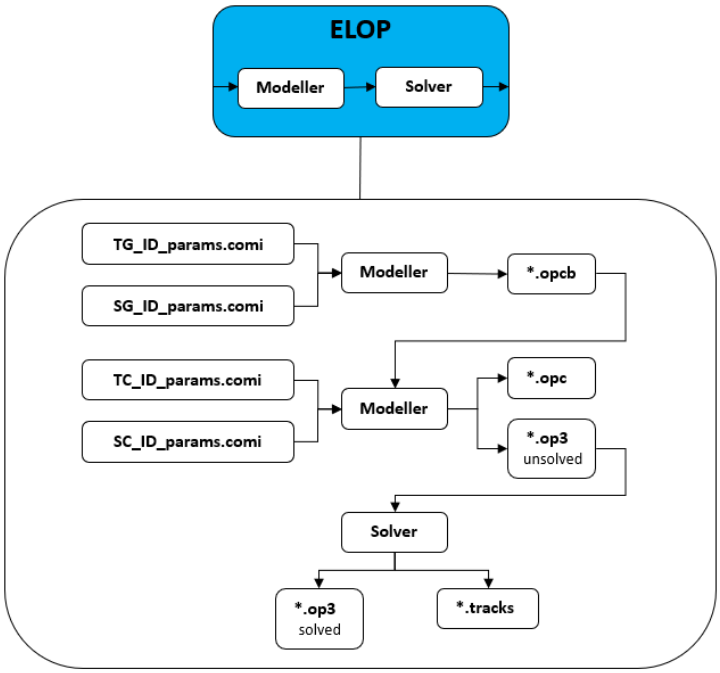


Abb. 8: Entstehung von verschiedenen Dateien während der ELOP-Kalkulationen (Eigendarstellung)

*.opcb	Gitter/Netz/Geometrie
*.opc	Geometrie + Grenzen/Randwerte
*.op3	Datenbank mit Ergebnissen der Feldberechnungen
*.tracks	Ergebnisse des Elektronenverläufe

Tab. 3: Überblick der generierten Dateiformate mit entsprechendem Inhalt (Eigendarstellung)

In der Auswertung des bisherigen Arbeitsprozesses werden aus den Ergebnissen, also aus den Informationen des Elektronenverlaufs, die Größe und Lage des Brennflecks herausgefiltert. In diesem Schritt entsteht sozusagen ein Brennfleck Protokoll in png-Format (siehe Abb. 9).

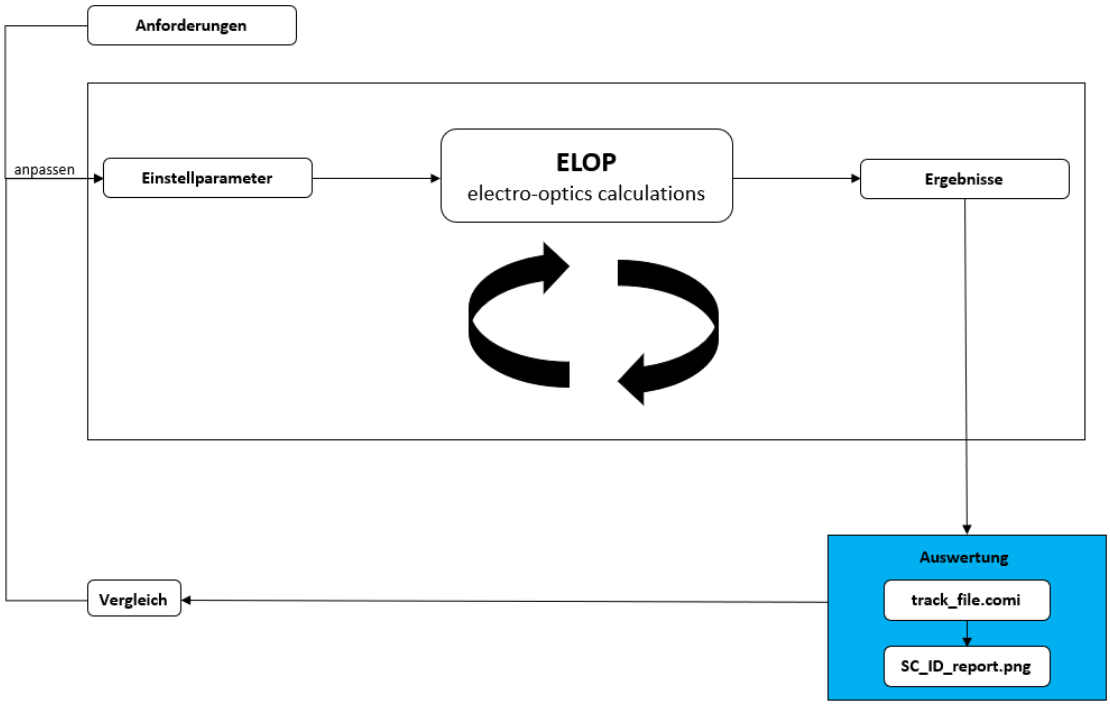


Abb. 9: Entstehung des Brennfleck-Protokolls aus der „track_file.comi“-Datei nach der Auswertung des Elektronenverlaufs (Eigendarstellung)

Zweifellos entsprechen Simulationen nicht immer der Realität und weisen erfahrungsgemäß Toleranzen auf. In der Abbildung 10 ist der Vergleich zwischen gemessenem und simuliertem Brennfleck dargestellt. Anhand der vorliegenden Ergebnisse lässt sich sagen, dass sich der simulierte von dem gemessenem Fokuspunkt kaum unterscheidet. Die Röhrenentwicklung bei Philips baute das Simulationsprogramm so weit aus, dass die simulierten Röhren und deren Ergebnisse sehr realitätsnah sind und den praktischen Messungen entsprechen.

Abb. 10: Vergleich von gemessenen(links) und simulierten Brennfleck (rechts) einer Röntgenröhre mit zwei unterschiedlichen Foki (Eigendarstellung)

Schlussfolgernd lässt sich aus dem bisherigen Arbeitsprozess mit Opera entnehmen, dass sehr viele verschiedene Dateien aus den einzelnen Bausteinen entstehen oder benötigt werden. Wie bereits beschrieben, sind Einstellparameter in Form von Textdateien gespeichert, auf die Opera für die Simulationen zugreift. Daraus entstehen wiederum mehr Dateien, die schließlich in der Auswertung weiterverarbeitet werden. In diesem Prozessschritt werden letztendlich auch wieder neue Dokumente generiert. Die Abbildung 11 zeigt auf, in welchen Prozessschritten neue Dateien entstehen.

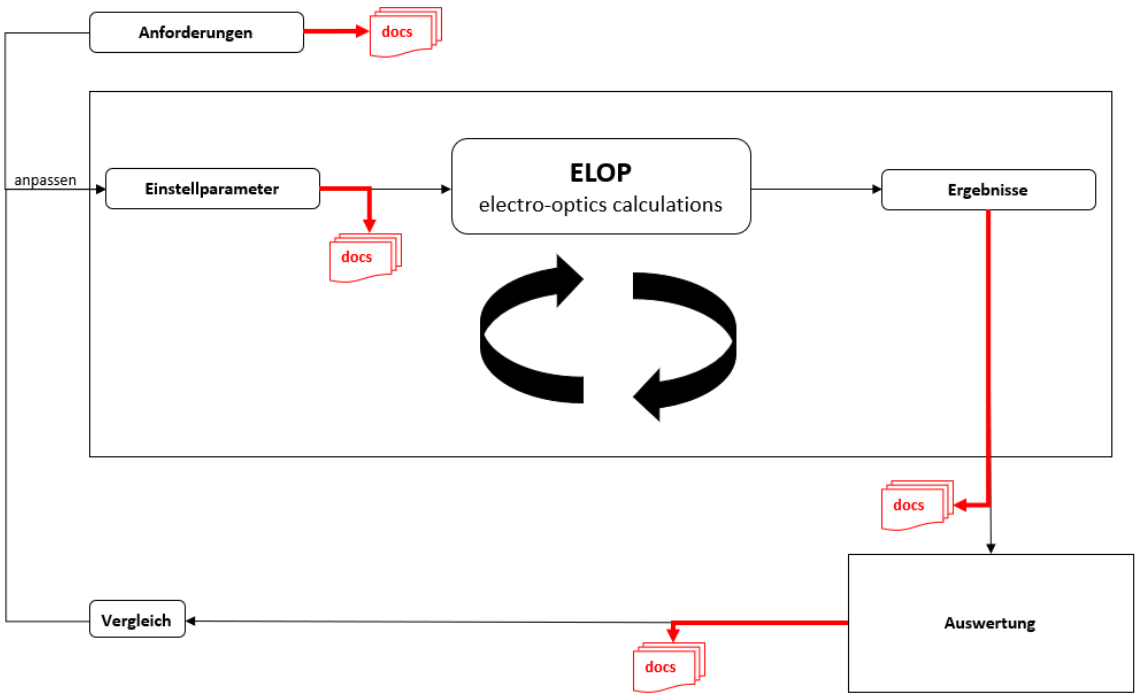


Abb. 11: Entstehung von vielen Dateien während des Simulationsablaufs (Eigendarstellung)

Aufgrund der Nachteile (siehe Kapitel 1), die durch die bisherige Datenhaltung entstehen, kam die Idee auf, eine Datenbank zur Speicherung von Einstellparametern und Ergebnissen von Opera zu implementieren.

3.1 Arten von Parametern

Die vorliegende Arbeit legt in erster Linie den Fokus auf das Einpflegen der Einstell- und Ergebnisparameter in eine Datenbank. Die Einstellparameter, also die Input-Daten für Opera, sind bekanntlich in Textformat gespeichert. Diese Arbeit konzentriert sich zunächst nur auf die Parameter aus den SG- und SC-Dateien, da TG- und TC-Parameter nur optional in bestimmten Röhren auftauchen. Generell haben die Dateien, die Einstellparameter enthalten, immer den gleichen Aufbau. Der Name der Datei trägt das Kürzel „SG“ und dahinter eine einzigartige fünfstellige ID-Nummer. Die Abbildung 15 veranschaulicht ein Beispiel einer solchen Textdatei und heißt „SG_66666_params.comi“. Der Inhalt fängt mit der Konfiguration der Röhre an (siehe Abb. 12 Zeile 2), die ähnlich wie eine Baumstruktur festlegt, welche Einzelkomponenten bzw. Module miteinander verknüpft sind. Nach dieser Angabe folgen die Einstellparameter für jedes einzelne Modul, welches nur eine gewisse Anzahl an bestimmten Parametern besitzt. In der gleichen Zeile sind ebenfalls der Wert und die Beschreibung des zugehörigen Parameters abzulesen. Zusätzlich zu den Parametern der Module gibt es in jeder SG-Datei noch feste Parameter, die in jeder dieser Dateien vorhanden sein müssen. Der Parameterwert dazu kann jedoch variieren.

SG_66666_params.comi

```
1  /Configuration
2  $STR CONFIG MRC200_Cathode-Cathode_Slit-<Cathode_Slit...
3
4  /Global
5  $CON #MSGLOBAL        1.0        'Mesh size global scale factor'
6  $CON #MSERROR         0.9        'Mesh size scale factor on mesh error'
7
8  /MRC200 Cathode
9  $CON #CATHDIA        34.0        'Cathode, cup diameter'
10 $CON #CATHFC          6.0        'Cathode, fase distance from top to fase edge'
11 $CON #CATHANG        45.0        'Cathode, fase angle (=0 no fase)'
12 ...
13
14 /Cathode Slit
15 $CON #CATHSDX1        0.0        'Cathode Slit 1, X-Displacement to Cathode-CS'
16 $CON #CATHSDY1       12.6+5.0    'Cathode Slit 1, Y-Displacement to Cathode-CS'
17 $CON #CATHSDZ1        1.0        'Cathode Slit 1, Z-Displacement to Cathode-CS'
18 ...
19
20 /Cathode Slit
21 $CON #CATHSDX2     #CATHSDX1     'Cathode Slit 2, X-Displacement to Cathode-CS'
22 $CON #CATHSDY2    -#CATHSDY1     'Cathode Slit 2, Y-Displacement to Cathode-CS'
23 $CON #CATHSDZ2     #CATHSDZ1     'Cathode Slit 2, Z-Displacement to Cathode-CS'
24 ...
```

Abb. 12: Beispiel einer „SG_IdNr_params.comi“-Datei (Eigendarstellung)

Die gleiche Dateistruktur befindet sich in der SC-Datei. Wie in der folgenden Darstellung zu sehen, verweist diese Textdatei auf ein SG-Dokument (siehe Abb. 13 Zeile 3). In diesem Fall werden die Betriebswerte, wie zum Beispiel für Strom und Spannung, gesetzt.

SC_66666_params.comi

```
1  // Opera-3d Modeller ### Parameters ###
2
3  $STR GEOID SG_66666
4
5  ##UT=75.0      'kV, tube voltage'
6  ##T1=2561.0*1  'K,  Filament 1 temperature'
7  ##T2=2578.0*0  'K,  Filament 2 temperature'
8  ##U1=49.0      'V,  Emitter 1 voltage offset'
9  ##U2=115.0     'V,  Emitter 2 voltage offset'
10
11 /Global
12 $CON #BCCATHODE    -500.0*##UT   'Potential of Cathode'
13 $CON #BCANODE       500.0*##UT   'Potential of Anode'
14 $CON #BCGROUND        0.0        'Potential of Ground'
15 $CON #BCOFFSET     0-#BCCATHODE  'Potential offset'
16 $CON #EMTEMPLIMIT  1900.0        'Emitter lower temperature limit'
17
18 /Scala
19 $CON #SCREL           0.35       'Under relaxation factor (SCALA)'
20 $CON #SCTOL           0.001      'Convergence tolerance (SCALA)'
21 $CON #SCIT           21.0        'Max number of iterations (SCALA)'
22 ...
```

Abb. 13: Beispiel einer „SC_IdNr_params.comi“-Datei (Eigendarstellung)

Neben den Parametern in den SG- und SC-Dateien existieren noch weitere Arten von Parametern, die für die Modellierung der Röhre eine bedeutende Rolle spielen. Je nach Röhrentyp, werden zusätzliche Informationen z.B. über die Rotationskonturen und Schnittebene benötigt. Der Dateiname beinhaltet hierbei auch den Parameternamen, z.B. steht „SG_66666_CathFrame_rot.txt“ für den Parameter „CathFrame“. Die Abbildung 14 zeigt ein Beispiel eines SG-Ordners auf, welcher zusätzlich neben der „*_params.comi“-Datei noch weitere Dateien beinhaltet.

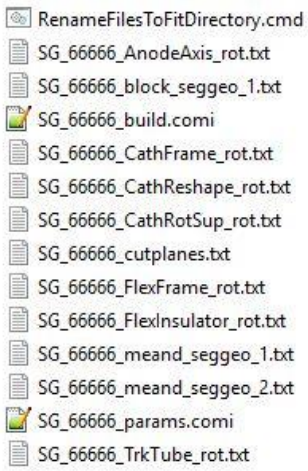


Abb. 14: Beispiel eines SG-Ordners (Eigendarstellung)

Der Inhalt dieser Dateien, der ebenfalls für Opera lesbar ist, unterscheidet sich von den Geometrie- und Berechnungsdateien und wird in Form einer Zahlenmatrix widergespiegelt (siehe Abb. 15).

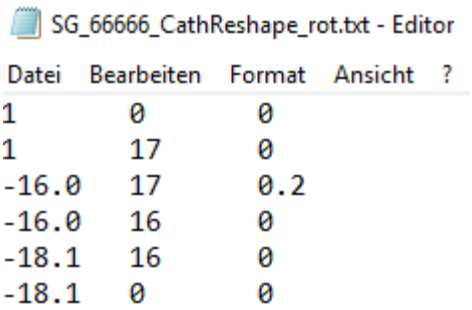


Abb. 15: Beispiel für den Inhalt einer Rotationskontur-Datei (Eigendarstellung)

Um die Daten später in die Datenbank importieren zu können, ist es wichtig, den Überblick über alle Arten der Parameter zu haben. In erster Linie ist es wesentlich alle Parametertypen, die im momentanen Arbeitsablauf bekannt sind, zu sammeln. Darüber hinaus unterscheidet sich, abhängig vom Parametertypen, die Speicherform des Parameternamens und die dazugehörigen Werte. Aus diesem Grund gibt die Tabelle 4 eine Zusammenfassung der verschiedenen Parameterarten und einen Überblick über die unterschiedlichen Speicherformen der Namen und Werte wieder.

Parametertyp	**Speicherform des Parameternamens und Werts**
Value	Parametername und Werte in Datei enthalten
RotSolid	Parametername im Dateinamen, Werte in Datei enthalten
Cutplane	Parametername im Dateinamen, Werte in Datei enthalten
block_seggeo	Parametername im Dateinamen, Werte in Datei enthalten
meand_seggeo	Parametername im Dateinamen, Werte in Datei enthalten
blending	Parametername im Dateinamen, Werte in Datei enthalten
OPC	Parametername im Dateinamen, Werte in Datei enthalten

Tab. 4: Überblick der verschiedenen Arten der Parameter (Eigendarstellung)

3.2 Aktuelle Datenhaltung

Derzeit werden die gesamten SG- und SC-IDs in einer Excel-Tabelle dokumentiert und abgespeichert. Diese Datei liegt in einem geteilten Netzwerkordner mit Zugriffskontrolle. Die Tabelle 5 protokolliert die bisherigen Opera-Berechnungen und gibt den Anwendern einen Überblick über die Simulationen mit den entsprechenden Parametern.

#	family	#SG	#SC	#TG	#TC	PC	comment

Tab. 5: Protokollierung der SG- und SC-IDs für Opera-Simulationen in einer Excel-Tabelle (Eigendarstellung)

Für jede neue Simulation wird in die Tabelle manuell von dem Anwender eine Rechnungsnummer (#) eingefügt. Des Weiteren wird eine Röhrenfamilie (family) zugeordnet, für die die Kalkulation bestimmt ist.

Wie bereits erwähnt, sind die Parameter mit den entsprechenden Werten in Textdateien abgespeichert, die jeweils einer bestimmten Dokumentennummer zugewiesen sind. Jeder Entwickler, der das Simulationsprogramm Opera benutzt, ist im Besitz eines Simulationsrechners. Auf diesen Rechnern befinden sich die lokal abgespeicherten Geometrie- (SG) und Berechnungsdaten (SC), die Opera für die Simulationen benötigt. Neben SG und SC gibt es bekanntlich noch TG und TC, welche auf Magnetfeldberechnungen referenzieren. Die zuletzt genannten Parameter werden allerdings nur zu bestimmten Simulationen herangezogen und werden in dieser Arbeit erstmal nicht beachtet.

Jeder Simulationsrechner enthält zwei lokale Ordner, die für die Opera-Simulation entscheidend sind. In dem einen Ordner befinden sich Vorlagen für die Textdateien hinsichtlich Geometrie- und Berechnungsparametern, da für jeden Röhrentyp unterschiedliche Parameter verwendet werden. Diese Vorlagen werden von dem Anwender für die Simulation bearbeitet und mit Werten gefüllt.

In dem zweiten Ordner werden schließlich alle zur Simulation benötigten Dateien und Ergebnisse abgelegt. Jeder Rechner besitzt somit unterschiedliche Geometrie- und Berechnungsdaten aufgrund unterschiedlich durchgeführter Simulationen.

Die Computerkennzahl (PC), auf dem die Simulation stattfindet, wird ebenfalls in der Tabelle notiert. Demzufolge können Anwender genauestens nachverfolgen, auf welchem Rechner welche Geometrie- und Berechnungsdaten zu finden sind.

Zum Schluss werden Schlagwörter für die entstandenen Ergebnisse oder Mittel in die Kommentarsektion (comment) aufgenommen, um die Simulationsrechnung kurz zu beschreiben.

Die Datenstruktur auf jedem Simulationsrechner lässt sich in der folgenden Abbildung demonstrieren (siehe Abb. 16). Ein Ordner besitzt die gesamten SG- und SC-Vorlagen für die verschiedenen Röhrentypen. Das bedeutet, dass der Anwender vor einer Simulation die Geometrie- und Berechnungsparameter in Textdateien nicht neu erstellen muss, sondern lediglich die Werte definiert oder umändert.

Daraufhin werden in einem zweiten Ordner die für die Simulation gefertigten Dateien mit neu angelegter ID abgesichert. Diese IDs werden in der bereits erwähnten Excel-Tabelle zur Dokumentation festgehalten.

Aufgrund der großen Anzahl an Dateien und Ordnern ist eine Dezentralisierung der Datenhaltung für diese Arbeitsweise von großem Vorteil. Eine Einführung einer Datenbank würde die Menge der Dateien reduzieren und einen besseren Überblick über die gesamten Parameter verschaffen. Die Vollständigkeit der Informationen und gleichzeitiger Zugriff wird damit gegeben.

Das nächste Kapitel beschäftigt sich mit der Auswahl eines geeigneten Datenbankverwaltungssystems, um die momentane Datenhaltung zu ersetzen und die Arbeitsweise infolgedessen zu vereinfachen.

Wie in Abbildung 16 zu sehen, enthält Ordner 1 die Vorlagen für die Geometrie- und Berechnungsparameter der verschiedenen Röhrentypen und in Ordner 2 werden diese mit entsprechender IDs abgespeichert.

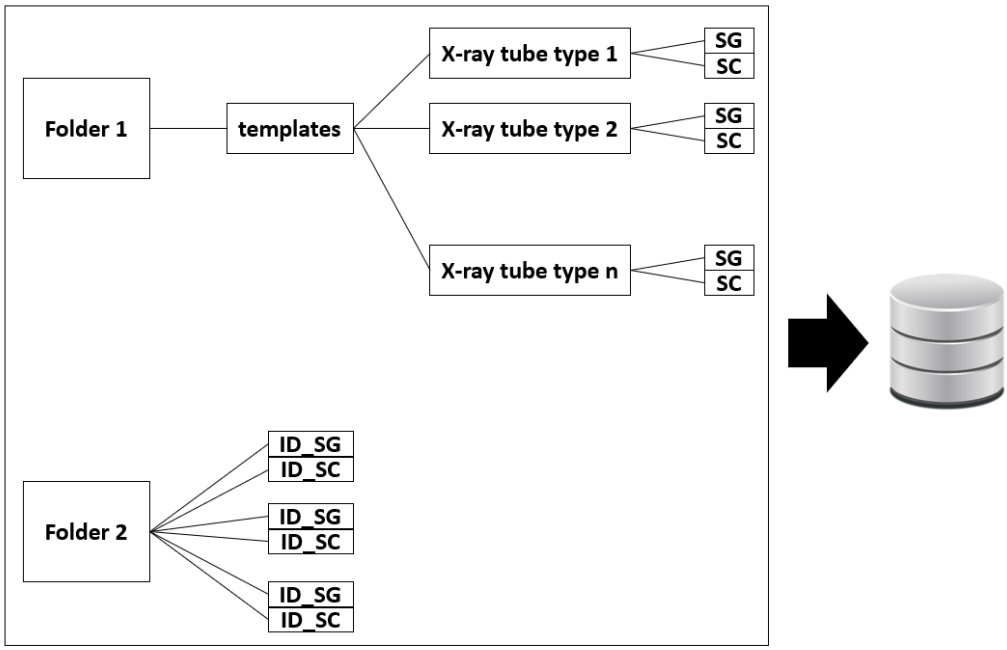


Abb. 16: Vorlagen für Geometrie- und Berechnungsparameter der verschiedenen Röhrentypen (Folder 1) und Speicherung der Parameter mit entsprechender IDs (Folder 2) (Eigendarstellung)

4 Auswahl der passenden Datenbank

Eine Anforderung kann sowohl als Forderung verstanden werden, die an ein System gestellt wird, oder auch als eine Eigenschaft, die erfüllt werden soll. Um eine geeignete Datenbank auszuwählen, die den Anforderungen gerecht ist, werden zunächst die Vor- und Nachteile der SQL- und NoSQL-Datenbankansätze abgewogen und sich anschließend für ein Datenbankmodell entschieden.
Nach dem Beschluss des Datenbankmodells, welches entweder auf dem relationalen oder nicht-relationalen Datenbanksystem beruht, werden die gebräuchlichsten Datenbankenverwaltungssysteme des entschiedenen Modells aufgelistet, mit den Anforderungen abgeglichen und bewertet.
Abschließend wird eine Entscheidung gefällt, welches Datenbankmanagementsystem hinsichtlich bestimmter Kriterien am besten geeignet ist. Der Auswahlprozess ist in Abbildung 17 veranschaulicht.

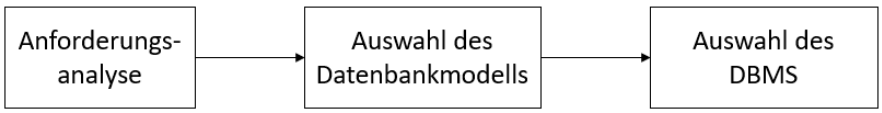


Abb. 17: Schritte zur Auswahl eines geeigneten Datenbankmanagementsystems (Eigendarstellung)

4.1 Anforderungsanalyse

Der Zweck des Systems, in dem Fall die Datenbank, besteht darin, das bestehende Problem zu lösen und die momentane Arbeitsweise zu verbessern. Das System ist dann eine Lösung, wenn es gewisse Anforderungen erfüllt. Generell werden bei einer Anforderungsanalyse Wünsche und Bedürfnisse von bestimmten Personen unter Berücksichtigung organisatorischer und technischer Gegebenheiten systematisch erfasst, in konkrete Anforderungen übersetzt und schlussendlich überprüft [12].
Die Personengruppen, die ein berechtigtes Interesse am Verlauf oder Ergebnis dieser Lösung haben, werden auch als „Stakeholder“ bezeichnet. Diese werden in die Anforderungsanalyse miteinbezogen, da sie gegenwärtig, oder auch zukünftig, direkt, oder auch indirekt, mit dem Ausgang der Arbeit betroffen sind [12].
Die Anforderungsanalyse wurde in Form einer Excel-Liste festgehalten und ist im Anhang 1 dieser Arbeit zu finden. Die Tabelle startet mit der Anforderung die Geometrie- und Berechnungsdateien zu verwalten. Die jetzige Arbeitsweise erlaubt den Anwendern bei einer neuen Simulation neue SG- und SC-IDs in eine Excel-Tabelle einzutragen. Die Generierung dieser IDs ist auch bei Einführung einer

Datenbank unausweichlich und sollte zukünftig automatisch erfolgen. Zusätzlich sollte man die Relation zwischen SG- und SC-ID verwalten können, d.h. die SC-ID sollte auf eine SG-ID verweisen können. Bisher gab es bei doppelter SC-ID keine Meldung, was in der Zukunft durch die Datenbank programmatisch implementiert werden sollte. Hinsichtlich der Programmierbarkeit ist eine essenzielle Anforderung eine Schnittstellenprogrammierung durch Python. Aufgrund der in Opera eingebetteten Python-Schnittstelle, sollte das Programm in der Opera Umgebung ausführbar sein. Um sich stärker von den Textdateien weg zu bewegen, sollen die Parameter mit ihren Werten in die Datenbank gespeichert werden. Das Sammeln der Simulationsdaten soll später auch als Auswertung dienen, daher sind das Filtern und Extrahieren der Daten ebenfalls in der Anforderungsliste. Die Datenbank sollte verschiedene Dateiformate abspeichern können, da Einstell- und Berechnungsparameter in Text- und Bildformate gespeichert sind. Die Menge der Daten wird sich maximal im Gigabyte-Bereich aufhalten, welches gleichzeitig als Anforderung hinsichtlich der Speicherkapazität gilt. Die Überlegung über eine Zugriffssicherung nur für Opera-Anwender bleibt vorerst außer Acht, da sich die Datenbank erstmal im Netzwerksystem von Philips befinden wird und unbefugten Zugriff von externen Personen einschränkt.

Anhand der Auflistung der wichtigsten Anforderungen an das Datenbanksystem, wird im nächsten Schritt eine Datenbankrecherche erfolgen, um ein passendes Datenbankmodell und schließlich eine bestimmte Datenbank zu finden.

4.2 Datenbankrecherche

Jedes Computerprogramm generiert in seiner Laufzeit Informationen oder greift auf Daten zurück, die zuverlässig, fehlerfrei und dauerhaft gespeichert werden müssen. Dies geschieht in einem sogenannten Datenbanksystem. Ein Datenbanksystem ist eine Software zur Datenverwaltung, dessen Aufgaben die Beschreibung, Speicherung und Abfrage von Daten sind. Das System besteht aus einer Speicherungs- und Verwaltungskomponente. Die Speicherungskomponente umfasst alle Daten und deren Beziehungen, welche in einer organisierten Form abgelegt werden sollen. Daneben dient die Verwaltungskomponente zur Pflege, Auswertung und Veränderung der Informationen sowie des Zugriffs- und Bearbeitungsrechte der Nutzer. Zur Verarbeitung und Bereitstellung der Daten enthält die Verwaltungskomponente der Datenbank eine bestimmte Abfragesprache [13].

Heutzutage werden in der Praxis größtenteils Datenbanken eingesetzt, die von der Abfragesprache Structured Query Language (SQL) unterstützt werden. Diese Art von Datenbanken unterliegt dem sogenannten relationalen Datenbankmodell, das auf der Speicherung von Informationen in verschiedenen Tabellen basiert, die untereinander über Beziehungen verknüpft werden können. Vor allem große und heterogene Datenbestände, die in Echtzeit abgewickelt werden müssen, stellen eine bedeutsame Herausforderung für dieses Datenbankmodell dar. Aus diesem Grund kommen neben den gängigen SQL-Datenbanken vermehrt NoSQL-basierte Systeme zur Anwendung, dessen Vorteile im nächsten Abschnitt aufgezählt werden. Es ist für die Auswahl der passenden Datenbank entscheidend, die Vor- und Nachteile von relationalen sowie nicht-relationalen Datenbankmodellen gegenüberzustellen und mit den Anforderungen abzugleichen [13, 14].

4.2.1 SQL vs. NoSQL

Relationale SQL-Datenbanksysteme gelten in der heutigen Zeit als ein etablierter Standard für betriebliche Datenbankanwendungen und werden von kleinen bis zu großen Unternehmen und Organisationen eingesetzt. Mit dem stetigen Anstieg von digitalen Informationen, ist dieses Datenbankmodell für einige Bereiche nicht mehr die optimalste Lösung. Zur Bewältigung dieser Schwierigkeit wurden NoSQL-Datenbanken entwickelt, welche hinsichtlich Big Data viele Vorteile mit sich bringen. Doch welches dieser zwei Datenbankmodelle eignet sich für diese Arbeit am besten [13]?

Eine relationale Datenbank ist eine auf Tabellen basierende Datenbank, zudem die Daten mit einem eindeutigen Schlüssel ausgestattet sind und in einer Beziehung miteinander stehen. Das tabellarische Ordnungsschemata besteht aus waagerechten Zeilen und senkrechten Spalten, die eine geordnete Strukturierung der Daten ermöglicht. Jede Zeile der Tabelle wird als ein Tupel bzw. Datensatz bezeichnet und jedes Tupel besteht aus einer Reihe von Attributs- bzw. Merkmalswerten, den Spalten der Tabelle. Die Relation beschreibt dabei eine Menge von Tupeln. Wie bereits erwähnt, arbeiten Relationale Datenbanken, mit Schlüsseln, die auch Identifikationsschlüsseln genannt werden. Diese werden eingesetzt, um die Datensätze innerhalb der Tabelle eindeutig zu identifizieren, d.h. verschiedene Tupel dürfen keine identischen Schlüssel aufweisen. In Bezug auf die Architektur des relationalen Datenbanksystems ist eine große Datenunabhängigkeit gewährleistet. Die Speicherungs- und Verwaltungskomponenten ermöglichen eine Trennung der Daten und Anwendungsprogramme. Änderungen in den Datenbanken haben somit keinen Einfluss auf die entsprechenden Anwendungsprogramme [14–16].
Relationale Datenbanken verwenden die weitverbreitetste Abfrage- und Manipulationssprache SQL, welche durch das ANSI (American National Standards Institute) und durch die ISO (International Organization for Standardization) genormt wurde. Eine Abfrage bezieht sich auf eine oder mehrere Tabellen und erzeugt als Resultat immer eine Tabelle. Diese Arbeitsweise ist ein wesentlicher Vorteil, weil eine einzige SQL-Abfrage eine ganze Reihe von Aktionen im Datenbanksystem auslösen kann. Zusätzlich besitzt die Sprache bezüglich Datenschutzes und Datensicherung auch Dienstfunktionen für die Wiederherstellung von Datenbeständen im Falle eines Fehlers. SQL setzt für die Definition, Selektion und Manipulation der Informationen deskriptive Formulierungen ein, d.h. es wird genau beschrieben, welche Daten man als Ergebnis erhalten möchte. Der Vorteil hierbei ist, dass die dafür erforderlichen Rechenschritte nicht von dem Anwender, sondern selbst von dem Datenbanksystem anhand eigener Such- und Zugriffsmethoden, berechnet wird [13, 14].

Neben dem relationalen SQL-Datenbanksystem, existiert noch das NoSQL-Datenbanksystem. Dieses verfolgt einen nicht-relationalen Ansatz und lässt sich für Big-Data-Anwendungen einsetzen. Als Big Data werden umfangreiche Datenmengen bezeichnet, die mit herkömmlichen Methoden und Softwarewerkzeugen kaum mehr zu bewältigen sind. Die Daten sind beispielsweise zu groß, komplex oder meistens unstrukturiert [13].

Hinter der Abkürzung NoSQL verbirgt sich der englische Begriff „Not only SQL“, was bedeutet, dass nicht nur relationale Datentechnologien zum Einsatz kommen, sondern auch alternative Datenbankmodelle verwendet werden. Während sich die Datenspeicherung von relationalen Datenbanken auf Tabellen mit Spalten und Zeilen stützen, nutzen NoSQL-Datenbanken Wertepaare, Objekte, Dokumente oder Listen und Reihen für die Organisation der Daten. NoSQL-Systeme sind auf bestimmte Anwendungen optimiert und stellen eine Ergänzung zu relationalen Datenbanken dar. Mit dem Aufkommen des Internets und einer Vielzahl von webbasierten Anwendungen haben nicht-relationale Datenkonzepte an Gewicht gewonnen. Dank der fehlenden Schemata sind NoSQL-Systeme sehr flexibel und eignen sich für große Datenmengen. Nicht-relationale Datenbankmodelle lassen sich in vier unterschiedlichen Kategorien einteilen, welche sind: Dokumentenorientierte Datenbanken, Key-Value-Datenbanken, Graphendatenbanken und spaltenorientierte Datenbanken. Heutzutage ist es sehr schwierig Big Data Anwendungen mit relationaler Datenbanktechnologie zu bewältigen [13, 14, 17].

Neben den bereits aufgeführten Unterschieden zwischen relationaler und nicht-relationaler Datenbanken, bestehen noch weitere bedeutende Unterscheidungskriterien. Während SQL-Datenbanken vertikal skalieren, werden NoSQL-Datenbanken horizontal erweitert. Unter der Skalierbarkeit wird die Fähigkeit zur Größenveränderung eines Systems verstanden. Bei der vertikalen Skalierung erhöht man die Kapazität durch das Aufrüsten von vorhandenen Servern, wohingegen die horizontale Skalierbarkeit die Fähigkeit ist, die Kapazität durch das Hinzufügen und Verknüpfen mehrere Hardware- oder Softwarekomponenten zu vergrößern [13, 17].

Eine wichtige Aufgabe von Datenbankmanagementsystemen ist die Gewährleistung der Integrität der Daten. Unter dem Begriff Integrität, oder auch Konsistenz, versteht man den Zustand widerspruchsfreier Daten. Das bedeutet, dass die Korrektheit der Daten während Änderungsoperationen bzw. Transaktionen bestehen bleiben sollen [13, 14].

Probleme können diesbezüglich bei einem Mehrbenutzerbetrieb entstehen, denn in den meisten Fällen greifen mehrere Benutzer gleichzeitig auf eine Datenbank, was zu Konfliktsituationen und gegenseitiger Behinderung führen kann. Transaktionsverwaltungen haben die Aufgabe, die Entstehung von Konsistenzverletzungen zu vermeiden und mehreren Benutzern ein konfliktfreies Arbeiten zu ermöglichen [13, 14].

Um die Integritätsbedingungen zu schützen, folgen Transaktionen in relationale Datenbankmanagementsysteme dem sogenannten „ACID“-Prinzip. ACID ist eine Abkürzung für Atomarität, Konsistenz, Isolation und Dauerhaftigkeit und gibt die Grunderwartungen eines Transaktionsbetriebs wieder. Jede Transaktion erfolgt atomar, mit anderen Worten zusammenhängend. Das bedeutet, dass die gesamte Transaktion entweder vollständig oder überhaupt nicht ausgeführt wird. Hierbei werden keine Zwischenzustände oder Teile einer bestimmten Transaktion erzeugt. Neben der Atomarität muss die Konsistenz der Daten gewährleistet sein. Das bedeutet, dass eine Transaktion nach Beendigung einen konsistenten Datenbankzustand hinterlassen muss. Durch das Prinzip der Isolation wird verhindert, dass gleichzeitig ablaufende Transaktionen sich nicht gegenseitig beeinflussen, um so vor ungewollten Seiteneffekte geschützt zu bleiben. Die Dauerhaftigkeit verlangt ein langfristig und dauerhaft sicheres Ablegen der Daten. Datenbankzustände müssen so lange erhalten bleiben, bis sie von Transaktionen verändert werden. Im Falle eines Rechnerabsturzes oder Systemfehlers müssen die verlorenen Daten wiederhergestellt werden können [13, 14, 16].

Anders als die relationalen Datenbankverwaltungssysteme, charakterisieren nicht-relationale Systeme das „BASE“-Prinzip, das Gegenstück zu den strengen ACID Kriterien. Bei NoSQL Datenbanken strebt man nicht in jedem Fall nach Konsistenzforderung, sondern schwerwiegender auf Verfügbarkeit und Ausfalltoleranz. Solche Systeme bevorzugen eine weichere Konsistenzforderung, was erlaubt, dass zwischenzeitlich unterschiedliche Datenversionen gehalten und erst zeitlich verzögert aktualisiert werden. BASE steht für „basically available, soft state, eventual consistency“, was zusammengefasst bedeutet, dass die Konsistenz erst nach einer gewissen Zeitspanne der Inkonsistenz sichergestellt ist. Aus dem Grund werden bei webbasierten Anwendungen, bei denen die Weiterarbeit bei einem Ausfall einer Netzwerkverbindung wichtig ist, NoSQL Systeme bevorzugt [13, 14]. Anhand der aufgelisteten Unterschiede in Tabelle 6, wird die Frage nach dem geeigneten Datenbankmodell im weiteren Verlauf beantwortet.

SQL Datenbank (relational)	**NoSQL Datenbank (nicht-relational)**
Tabellen (festes Datenbankschema)	Sammlungen (dynamisches Datenbankschema)
Datenschema mit Relationen	Zusammenhanglose Datensammlungen
Vertikale Skalierung	Horizontale Skalierung
ACID	BASE

Tab. 6: Unterschied zwischen relationaler (SQL) und nicht-relationaler (NoSQL) Datenstruktur (Eigendarstellung)

In Hinblick auf die Aufgabe dieser Arbeit ist die Skalierung des Datenbanksystems nicht von großer Bedeutung. Aufgrund der Simulationen in den vergangenen Jahren, lässt sich mit Sicherheit sagen, dass die Menge der generierten Daten den Gigabyte-Bereich nicht übersteigen werden. Aus dem Grund ist eine horizontale Skalierung nicht notwendig und höchstens eine Aufrüstung des Speichers absolut ausreichend.

NoSQL Datenbanken liegen anlässlich ihres personalisierten Datenbankschemas im Vorteil, wenn man verschiedene Arten von Daten zusammenhangslos abspeichern möchte. Das Sichern von Daten in einer SQL Datenbank muss durch ein vorgegebenes Schema erfolgen. Für diese Arbeit ist jedoch das Aufzeigen der Beziehungen zwischen den Daten eine Anforderung von hoher Priorität. Zusätzlich ermöglicht die Datenbanksprache SQL dem Anwender durch wenig Aufwand die Relationen der Daten zu erfassen und sich die notwendigen Informationen herauszuziehen.

Aufgrund dessen fiel die Entscheidung, eine relationale Datenbank einzusetzen. Im nächsten Kapitel werden die infrage kommenden Datenbankenverwaltungssysteme mit deren Eigenschaften in einer Liste aufgeführt und anhand der Anforderungen beurteilt.

4.2.2 Analyse und Bewertung von Datenbanken

In diesem Kapitel werden die Eigenschaften von SQL-Datenbankmanagementsysteme, die sich auf dem Markt bewiesen haben, analysiert und bewertet. Im Anhang 2 befindet sich eine Tabelle mit den recherchierten Datenbankverwaltungssysteme und die Einteilung der technischen Besonderheiten. Diese lassen sich in Kapazität, Sicherheit, Skalierbarkeit, Kosten, Mehrbenutzerbetrieb, Speichern von verschiedenen Datentypen und Programmiersprache gliedern. In Tabelle 7 sind die Mindestanforderungen für das Datenbankmanagementsystem aufgeführt.

	Kapazität	**Konsistenz**	**Skalierbarkeit**	**Kosten**	**Mehrbenutzer**	**Datentypen**	**Python API**
Optimale DB	GB-Bereich	ACID	vertikal	open source	server/serverlos?	Werte & Bilder	verfügbar

Tab. 7: Auflistung der Mindestanforderungen für ein passendes Datenbankmanagementsystem (Eigendarstellung)

Wie im Anhang 2 zu sehen, werden im Ganzen sieben relationale Datenbankmanagementsysteme mit ihren Eigenschaften aufgelistet: MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft SQL Server Express, SQLite und MariaDB. Alle Systeme erfüllen die Mindestkriterien, allerdings werden bei einigen eine kostenpflichtige Version benötigt, um bestimmte Funktionen nutzen zu können. Die aufgelisteten Datenbankverwaltungssysteme sind in der Lage, deutlich mehr als die geforderte Speichergröße zu erbringen. Außerdem ist es für die genannten Systeme möglich, durch die vorausgesetzte Python-Schnittstelle verschiedene Arten von Dateien abzuspeichern.
Hinsichtlich des Mehrbenutzerzugriffs auf die Datenbank, gibt es Vor- und Nachteile für das Aufsetzen eines Servers. Wenn mehrere Endanwender Zugriff auf dieselbe Datenbank haben, greifen die Anwendungsprogramme nicht unmittelbar auf die Daten. Es herrscht eine sogenannte Client-Server-Umgebung, welche in den meisten Datenbankensystemen implementiert ist. Hierbei nimmt ein Client-Rechner Kontakt zu einem Server auf, der meist ein Rechner mit großer Eingangs-Ausgangs-Kapazität ist und ihnen die Dienste bereitstellt. Diese Dienste werden dann von den Client-Rechnern ausgeführt. Programme, die auf die Datenbank zugreifen möchten, kommunizieren mit dem Server über eine Interprozesskommunikation, um Anforderungen an den Server zu senden und um Ergebnisse zu erhalten. Ein Vorteil für den Einsatz eines Servers ist die Zentralisierung der Ressourcen, was bedeutet, dass der Server als Zentrum des Netzwerks fungiert und für die Verwaltung der Ressourcen verantwortlich ist. Dadurch ist die Steuerung des Datenbankzugriffs und ein besserer Schutz vor Fehlern in der Clientanwendung gewährt. Außerdem bleibt das Netzwerk ausbaufähig, denn Clients

können somit entfernt und hinzugefügt werden, ohne den Betrieb des Netzwerks zu beeinträchtigen [15, 18].

Das Aufsetzen eines Servers bringt allerdings auch Nachteile mit sich. Das Einrichten und das Betreiben erfordern sehr viel Aufwand und Verantwortung. Zumal der Server das Zentrum des Netzwerkes ist, gilt es gleichzeitig als das schwache Glied im Client-Server-Netzwerk. Ein Ausfall des Servers führt zu einem Totalausfall des Netzwerks, da dieses um ihn herum aufgebaut ist. Es müssen eine Menge Sicherheitsmaßnahmen hinsichtlich Konfiguration der Server-Anwendungen vorgenommen werden, um eine hohe Ausfallsicherheit zu erhalten [15, 18].

Bei einem serverlosen System hingegen, umgeht man die umfangreiche Client-Server-Architektur und verringert folglich die Aufgaben der Infrastrukturverwaltung für die Bereitstellung von Servern. Ebenfalls werden Gedanken zum Verwalten und Betreiben von Servern vermieden. Der Prozess, der auf die Datenbank zugreift, liest und schreibt direkt aus den Datenbankdateien auf der Festplatte, d.h. es gibt keinen zwischengeschalteten Serverprozess. Weitere Vorteile eines serverlosen Systems sind die flexible Ressourcenverwaltung und die schnelle Bereitstellung der Ressourcen aufgrund des Fehlens einer zentralen Verwaltung [18, 19].

Im Hinblick auf die vorliegende Aufgabe, fällt die Entscheidung auf ein serverloses Datenbankmanagementsystem. In diesem Fall ist es nicht unbedingt notwendig einen Server zu errichten, um die Kontrolle über alle Prozesse zu besitzen. Des Weiteren ist der Kauf von weiterer Hardware, welches durch die technische Komplexität des Servers bedingt, weitaus teurer als die Verwendung einer serverlosen Datenbank. Bei einer derart niedrigen Zahl von zukünftigen Betreibern ist es sinnvoller, sich den Aufwand der Serverkonfiguration zu sparen.

4.2.3 Datenbankentscheidung

Mithilfe der Internetrecherche und Abgleichung mit dem aufgeführten Anforderungsprofil fiel der Beschluss auf SQLite, ein relationales und serverloses Datenbankmanagementsystem.
Die Besonderheit an diesem SQL-System ist, dass SQLite keinen separaten Serverprozess besitzt, der konfiguriert werden muss. Es ist nicht erforderlich, dass ein Administrator eine neue Datenbankinstanz oder Benutzerzugriffsrechte zuweist. Aus dem Grund wird SQLite auch als eine konfigurationsfreie Datenbank bezeichnet [18].
Eine SQLite-Datenbank ist eine einzelne ordinäre Festplattendatei, die sich an einer beliebigen Stelle im Verzeichnis befinden kann. Um den Zugriff auf die Datenbank für mehrere Nutzer zu gewähren, gibt es somit die Möglichkeit, die Datei in einen Netzwerkordner abzulegen.
Im Gegensatz zu den anderen aufgelisteten Datenbanken, ist das gleichzeitige Schreiben in eine SQLite-Datenbank begrenzt. SQLite unterstützt nur einen Schreibprozess pro Datenbankdatei. Allerdings dauert eine Schreibtransaktion in den meisten Fällen nur Millisekunden, sodass sich mehrere Schreibvorgänge abwechseln können. Client-Server-Datenbanksysteme können hingegen in der Regel weitaus mehr Schreibzugriffe ausführen, was bei so einer niedrigen Anzahl an Betreibern nicht notwendig ist [18].
Hinsichtlich der Python-Kompatibilität gibt es das sogenannte „sqlite3“-Modul, das eine SQL-Schnittstelle bietet, die der in PEP 249 beschriebenen Schnittstellenspezifikation entspricht [20].
Zusammengefasst lässt sich erschließen, dass SQLite für einen lokalen Speicher mit wenig Schreibzugriff und weniger als einem Terabyte Dateninhalt in dem Fall die optimalste Lösung ist [18].
Während der Arbeit wird der DB Browser für SQLite verwendet, welcher eine kostenlose Software Applikation zum Erstellen, Entwerfen und Datenanzeigen von SQLite-kompatiblen Datenbanken ist. Für den späteren Betrieb wird der DB Browser allerdings nicht benötigt.

5 Datenbankstrukturierung

Kapitel 5 beschäftigt sich mit der Strukturierung der Datenbank und auf welche Weise die Parameter gespeichert werden. Mithilfe von Microsoft Access, ein Tool zum Entwerfen und Bereitstellen von Datenbankanwendungen, wurden die dafür notwendigen Tabellen erstellt und in einem Relationsschaubild dargestellt (siehe Anhang 3). Insgesamt gibt es 13 Tabellen, die untereinander durch Primär- und Fremdschlüsseln miteinander verknüpft sind. Diese Tabellen sind in Tabelle 8 und detaillierter im Anhang 4 zu finden.

Module
Module_SC_Params
Module_SG_Params
ParamType
SG
SG_Params
SG_Files
SC
SC_Params
SC_Files
Results_Params
Results_Value
Results_Picture

Tab. 8: Übersicht aller Datenbanktabellen (Eigendarstellung)

Zwecks der SC- und SG-Tabellen können durch die Generierung von ID-Nummern, neue Berechnungen bzw. Simulationen festgehalten werden. Die SG-Tabelle beinhaltet dabei eine SG-ID, die gleichzeitig der Primärschlüssel und somit einzigartig ist. Zusätzlich besitzt diese Tabelle die Spalte „SG_description“, um die Geometrie der Röhre zu beschreiben. Ergänzend dazu wird der Aufbau der Röhre durch die Konfiguration der Einzelmodule erfasst. Die SC-Tabelle speichert SC-IDs für neue Berechnungen und verweist dabei auf eine SG-ID. Durch die Implementierung dieser zwei Tabellen kann die bisher geführte Exceltabelle zur Dokumentation der Geometrie- und Berechnungs-IDs ersetzt werden.

In der Module-Tabelle werden den verschiedenen Einzelkomponenten eigene IDs vergeben, welche auch den Primärschlüssel bilden. Die Idee dahinter ist, dass die IDs Im Fall einer Änderung der Modulnamen gleichbleiben können. Zugleich lässt sich aus dieser Tabelle in der Spalte „MultipleUse“ herauslesen, ob das Modul in einer Röhre mehrmals oder alleinig benutzbar ist.

Die Parameter der Module bzw. Einzelkomponenten der Röhre sind in den beiden Tabellen „Module_SC_Params“ und „Module_SG_Params“ aufgeteilt. Zweck dieser beiden Tabellen ist das Zuordnen der Parameter zu den entsprechenden Modulen, da jedes Modul bekanntermaßen vordefinierte Parameter besitzt. In beiden Fällen setzt sich der Primärschlüssel aus der Modul-ID und dem Parameternamen, also dem SC_ParamName/SG_ParamName, zusammen. Einige von ihnen werden für das Modul als optional angesehen und müssen nicht eingesetzt werden. Zugleich ist in dieser Tabelle die Angabe des Parametertyps notwendig, um die Flexibilität der Parameterspeicherung beizubehalten.

Die Geometrie- und Berechnungsparameter werden zunächst unter Werte und Dateien kategorisiert. Die Einstellparameter, die dem Parametertyp „Value“ angehören, werden in die Tabellen „SG_Params“ und „SC_Params“ gespeichert. Alle anderen Parametertypen werden in „SG_Files“ und „SC_Files“ gesichert. Die Tabellen „SG_Params“ und „SC_Params“ beinhalten jeweils die Informationen über die IDs, Parameternamen und Parameterwerte. Ebenso sind in „SG_Files“ und „SC_Files“ die Spalten für ID und Parametername enthalten. Hierbei entstand der Beschluss, die Dateien auf der Festplatte zu speichern und lediglich die dazugehörigen Prüfsummen in die Datenbank aufzunehmen. SQLite besitzt zwar die Funktion, verschiedene Arten von Dokumenten direkt in die Datenbank zu importieren, doch aufgrund einer möglichen zukünftigen Überlastung des Speichers, werden diese bevorzugt auf der Festplatte gelagert. Die Prüfsumme ist ein Wert, mit dem die Integrität von Daten überprüft werden kann. Er ist im Prinzip wie ein Fingerabdruck, der die Eindeutigkeit einer Datei bestimmt. Die Idee bei alledem ist, dass nur Dateien mit unterschiedlichem Inhalt in dem Ordner existieren dürfen, denn exakt identische Dokumente würden wiederum den Speicher unnötig beeinträchtigen. Für diese Arbeit wird das „Message-Digest Algorithm 5 (MD5)“ verwendet, eine bestimmte Art von Prüfsummen, die aus 32 Hexadezimalzahlen besteht. In Kombination mit der Dateigröße wird die Einmaligkeit der im Ordner befindlichen Datei erhöht. In allen vier Tabellen wird ihre Einzigartigkeit durch den Zusammenschluss zwischen SG-ID/SC-ID und Parametername als Primärschlüssel festgelegt [21, 22].

Die Struktur der Tabellen für die Ergebnisse laufen nach dem gleichen Prinzip, wie die der Einstellparameter. Auch in diesem Fall gibt es eine Tabelle für das Abspeichern der Werte mit dem Parametertyp „Value“ und eine andere für das Sammeln von Dokumenten in Form von Prüfsummen. Die eigentlichen Dateien werden andererseits in den Datenbankordner auf der Festplatte eingeführt.

Allerdings trägt der Name der zuletzt genannten Tabelle den Namen „Results_Pictures“, da diese Dateien nur auf Bilder beschränkt sind.

Generell gibt es zwei Möglichkeiten diese Datenbankstruktur zu implementieren. Der erste Weg wäre die Anfertigung direkt in der Datenbank, denn SQLite DB Browser bietet dem Anwender eine triviale Benutzeroberfläche zur Erstellung der Tabellen und ihre Relationen. Die andere Möglichkeit lässt sich programmatisch durch Python, in Kombination mit SQL-Befehlen, verwirklichen.

6 Prototypen der Schnittstellen zwischen der Datenbank und der Simulationssoftware

Dieser Teil der Masterarbeit stellt die Prototypen der Schnittstellen zwischen der SQLite Datenbank und der Simulationssoftware Opera vor. Es wurden insgesamt vier Programme mit der Programmiersprache Python in der Entwicklungsumgebung PyCharm geschrieben. Der erste Prototyp hat die Aufgabe für jede neue Simulation, die dazugehörige Geometrie- und Berechnungs-ID in der Datenbank zu erstellen. Zwei weitere Programme haben die Funktion, die Geometrie- und Berechnungsparameter in die Datenbank zu importieren und zu exportieren. Der letzte Prototyp sorgt für den Import der Simulationsergebnisse.

6.1 Generierung einer neuen Simulation in der Datenbank

Wie bereits in Kapitel 3.2 beschrieben, werden als erster Schritt im Arbeitsablauf mit Opera die IDs für die Geometrie- und Berechnungsdaten manuell erstellt und in einer Exceldatei festgehalten. Da die Datenbank zukünftig ebenfalls diese Aufgabe übernehmen soll, wurde ein Programm mittels Python entwickelt, welches die IDs für die Berechnung und Geometrie verwaltet. Im Anhang 5 befindet sich ein Python Source Code, der dem Anwender durch ein Menü in der Konsole erlaubt, SG- und SC-IDs automatisch zu generieren.

Wie in der Abbildung 18 zu erkennen, gestattet das Menü dem Benutzer drei Möglichkeiten:

1. Generierung von SG ID und zusätzlicher Beschreibung
2. Generierung von SC ID und zusätzlicher Beschreibung
3. Beenden

Je nachdem, welche Option der Anwender wählt, wird automatisch in die entsprechende Tabelle der Datenbank eine neue und einzigartige ID erstellt.

```
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]:
```

Abb. 18: Menü für die Verwaltung von Berechnung & Geometrie (Eigendarstellung)

Beschließt sich der Benutzer eine neue Geometrie-ID in die Datenbank generieren zu lassen, wählt dieser im Menü die erste Option „1. Generate SG ID and add description“. Daraufhin wird der Nutzer aufgefordert eine Beschreibung hinzuzufügen. In der folgenden Abbildung wird der Ablauf, eine neue SG-ID zu generieren, in der Konsole von PyCharm angezeigt (vgl. Abb. 19). In dem aufgeführten Beispiel wird die ID 3530 erzeugt, welche der nächstgrößeren ID in der SG-Tabelle entspricht. Die vom Anwender eingetippte Beschreibung „SG description TEST“ wird ebenfalls in der Datenbank vermerkt (vgl. Abb. 20).

```
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 1
Please add a description for the new SG ID.
SG_description: SG description TEST
The generated SG ID: 3530
Description: SG description TEST
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 3
Exit
```

Abb. 19: Generierung einer neuen SG-ID (Eigendarstellung)

Tabelle: SG

	SG_ID	SG_description	Configuration
	...	Filtern	Filtern
1	3530	SG description TEST	*NULL*
2	3529	*NULL*	*NULL*
3	3528	*NULL*	*NULL*

Abb. 20: Erweiterung der SG-Tabelle durch eine neue SG-ID (Eigendarstellung)

Die zweite Auswahl im Menü hat die Funktion, eine neue SC-ID in der Datenbank zu generieren, die zugleich auf eine SG-ID verweist (siehe Abb. 21). Der Nutzer gibt dabei die zu referenzierte SG-ID in die Eingabeaufforderung ein. Die eingetippte ID 3530 wird darauffolgend hinsichtlich ihrer Existenz in der SG-Tabelle überprüft. Falls diese vorhanden ist, wird eine neue Berechnungs-ID erzeugt und in die Datenbank gespeichert. Hierbei zeigt die Abbildung 22, dass die neue SC-ID 9349 hinzugefügt wurde und auf die Geometriedaten der SG-ID 3530 referenziert.

```
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 2
Please enter a description for the calculation: SC description TEST
Please select a SG_ID for this calculation: 3530
The ID 3530 is existing in the DB! It will be used for the calculation.
The generated SC_ID: 9349
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 3
Exit
```

Abb. 21: Generierung einer neuen SC-ID (Eigendarstellung)

Tabelle: SC

	SC_ID	SG_ID	SC_description
	...	...	Filtern
1	9349	3530	SC description TEST
2	9348	3529	Ref
3	9347	3528	Ref (mit "Offsets")
4	9346	3527	Nachbildung CTR17xx; Kopf-Nr. 261629
5	9345	3505	XS80 80kV 2800K 3,2A, -1045V, LF (wie SC_09340, nur mit Gridspannung von 140kV)

Abb. 22: Erweiterung der SC-Tabelle durch eine neue SC-ID (Eigendarstellung)

Hinsichtlich der Eingabeaufforderung für die SG-ID besteht die Gelegenheit, eine nicht existierende ID einzugeben. Unter dieser Voraussetzung enthält der Anwender eine Fehlermeldung und zugleich eine Auswahlmöglichkeit, eine neue Geometrie-ID anzulegen. Das bedeutet, dass durch den zweiten Menüpunkt sowohl eine neue SC-ID als auch SG-ID generiert werden kann. In Abbildung 23 ist das Auswahlmenü mit den Eingaben des Anwenders zu sehen. Die Abbildung 24 zeigt die generierten SG- und SC-IDs in den jeweiligen Tabellen der Datenbank auf.

```
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 2
Please enter a description for the calculation: SC description TEST 2
Please select a SG_ID for this calculation: 3531
The entered SG_ID is not existing in the DB. Do you want to generate a new SG_ID? [y/n]
y
Please add a description for the new SG_ID!
Description for new SG_ID: SG description TEST 2
SG_ID 3531 is generated. It will be used for the calculation.
The generated SC_ID: 9350
------------------------------ Verwaltung von Berechung & Geometrie ------------------------------
1. Generate SG ID and add description
2. Generate SC ID and add description
3. Exit
--------------------------------------------------------------------------------------------------
Please select[1-3]: 3
Exit
```

Abb. 23: Generierung einer neuen SC- und SG-ID (Eigendarstellung)

Tabelle: SG

	SG_ID	SG_description	Configuration
	...	Filtern	Filtern
1	3531	SG description TEST 2	NULL
2	3530	SG description TEST	NULL
3	3529	NULL	NULL
4	3528	NULL	NULL
5	3527	NULL	NULL

Tabelle: SC

	SC_ID	SG_ID	SC_description
	...	...	Filtern
1	9350	3531	SC description TEST 2
2	9349	3530	SC description TEST
3	9348	3529	Ref
4	9347	3528	Ref (mit "Offsets")
5	9346	3527	Nachbildung CTR17xx; Kopf-Nr. 261629

Abb. 24: SG- (links) und SC-Tabelle (rechts) in der Datenbank (Eigendarstellung)

Der dritte Punkt „3. Exit“ besitzt lediglich die Aufgabe das laufende Programm zu beenden, nachdem eine neue ID angelegt wird.

6.2 Import der Einstellparameter in die Datenbank

Ein Ziel der Arbeit ist es, die enorme Anzahl der bestehenden Dateien durch die Einführung der Datenbank zu verringern und das Anwachsen dieser Dokumente zu vermeiden. Daraufhin wurde ein Konzept einer möglichen Datenhaltung entwickelt, die es ermöglicht, die Einstellparameter für Opera-Simulationen zu speichern. Für diese Aufgabe wurde ein Prototyp ausgearbeitet, welcher die Funktion besitzt, Einstellparameter in die Datenbank zu importieren. Hinsichtlich der Programmierung ist der Python-Code an die Arbeit angehängt (siehe Anhang 6). Im weiteren Verlauf des Textes wird zuerst der Ablauf des Imports beschrieben und im Anschluss daran das Prozedere des Exports vorgestellt.
Bevor Daten in die Datenbank gespeichert werden können, muss im ersten Schritt eine Verbindung zur Datenbank hergestellt werden. Daraufhin wird die gesuchte Geometrie-ID aus dem Dateinamen der „*_params.comi“-Datei herausgelesen, welche in diesem Beispiel aus der Nummer „66666“ besteht. Existiert dieses Dokument nicht, erhält der Nutzer eine Fehlermeldung. Andernfalls kann der Ablauf fortgeführt werden. Im nächsten Schritt wird im Inhalt der Datei nach der Konfiguration der Module gesucht, um die entsprechenden Modulnamen, mithilfe der „Module“-Tabelle aus der Datenbank, durch ihre ID-Nummer zu ersetzen. Die Baumstruktur, die nur noch aus den umgewandelten Modul-IDs und deren Verbindungen besteht, wird dann mit der dazugehörigen SG-ID in die Tabelle „SG“ in der Datenbank festgehalten (siehe Abb. 25). Hinter dieser Vorgehensweise steckt ein bestimmter Gedanke, welcher im nachfolgenden Schritt näher erläutert wird.

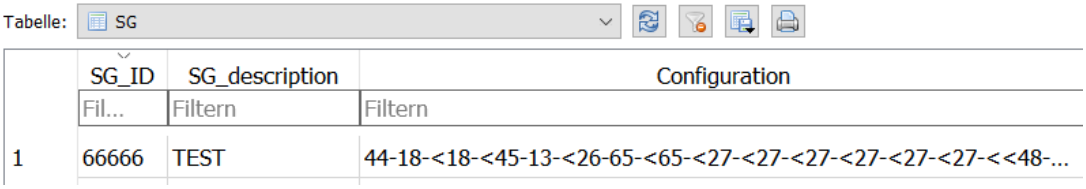


Tabelle: SG

	SG_ID	SG_description	Configuration
	Fil...	Filtern	Filtern
1	66666	TEST	44-18-<18-<45-13-<26-65-<65-<27-<27-<27-<27-<27-<27-<<48-...

Abb. 25: Import der SG_ID und Konfiguration der Module aus der „*_params.comi“-Datei (Eigendarstellung)

In der Datenbank gibt es zwei Tabellen, die die Namen „Module_SG_Params“ und „Module_SC_Params“ tragen. Wie bereits im Kapitel 6 geschildert, ist der Zweck dieser zwei Tabellen, die Modul-IDs mit den dazugehörigen Parametern zuzuordnen.
Anhand der umgeformten Konfiguration wird in der Datenbank somit überprüft, ob die erwarteten Parameter im „*_params.comi“-Dokument vorhanden sind. Mit anderen Worten, das Programm hangelt sich am Konfigurationspfad entlang und importiert jeden erwarteten Parameter, falls dieser in der Datei existiert. Falls ein Parametername fehlt, zu viel auftritt oder nicht wie in den zwei genannten

Tabellen dargestellt ist, wird eine Fehlermeldung ausgelöst. Während der Überprüfung des Vorhandenseins der Parameter in dem Dokument, muss zusätzlich die Optionsmöglichkeit beachtet werden. Das bedeutet, dass bestimmte Parameter nicht in jedem Fall verwendet werden müssen. Folglich würde kein Hinweis auf das Fehlen dieser Parameter hervorgerufen werden. Ein weiterer wichtiger Punkt ist die Mehrfachnutzung gewisser Module. Manche von ihnen werden in Röntgenröhren vielfach benutzt, was ebenfalls in der „Module“-Tabelle zu entnehmen ist. Folglich würden Parameter mit denselben Namen in die Datenbank gespeichert werden. Um diese Angelegenheit zu umgehen, werden Parameter von mehrfachbenutzten Modulen mit einem Index für ihre Verwendungshäufigkeit gekennzeichnet. Wenn die aufgeführten Voraussetzungen erfüllt sind, kann der Speicherprozess der Einstellparameter in die Datenbank durchgeführt werden. Wie schon in Kapitel 3.1 aufgeführt, gibt es verschiede Arten von Parametern, was ebenfalls in den Tabellen „Module_SG_Params“ und „Module_SC_Params“ vorgegeben ist. Diese werden auf unterschiedliche Weise in die Datenbank importiert. Betrachtet man den Parametertyp „Value“, werden diese Parameter zusammen mit ihrer SG_ID und Werte in die „SG_Params“-Tabelle gesichert (vgl. Abb. 26). Die Parameter, die sich in der Datei „*_params.comi“ befinden, gehören dem Typ „Value“ an. Das gleiche Importprinzip gilt ebenfalls für die Betriebsparameter des Typs „Value“, welche in die Tabelle „SC_Params“ mit ihrer SC_ID und Werte abgespeichert werden.

Tabelle: SG_Params

	SG_ID	SG_ParamName	ParamValue
	Fil...	Filtern	Filtern
1	66666	mserror	0.9
2	66666	msglobal	1
3	66666	cathsb1	10
4	66666	cathsds1	0
5	66666	cathsdx1	0
6	66666	cathsdy1	17.6
7	66666	cathsdz1	1
8	66666	cathsl1	10
9	66666	cathst1	-10
10	66666	cathsb2	15
11	66666	cathsds2	0.8
12	66666	cathsdx2	6

Tabelle: SC_Params

	SC_ID	SC_ParamName	ParamValue
	Fil...	Filtern	Filtern
1	66666	sctol	0.001
2	66666	screl	0.35
3	66666	scit	21
4	66666	geoid	66666
5	66666	flinoilsig	0
6	66666	flinoileps	2.25
7	66666	flincersig	0
8	66666	flincereps	9.7
9	66666	emwf2	4.52
10	66666	emwf1	4.52
11	66666	emtype2	1
12	66666	emtype1	1

Abb. 26: Ausschnitt der importierten Einstellparameter mit dem Parametertyp "Value" in die Tabellen „SG_Params“ (links) und „SC_Params“ (rechts) (Eigendarstellung)

Im Gegensatz zu dem zuletzt genannten Parametertypen werden alle anderen in einem sogenannten Datenbankordner importiert. Die restlichen Parametertypen sind Dateien, dessen Parameternamen im Dokumentennamen enthalten sind. Zweck dieses Vorgehens ist die Vermeidung der möglichen Überlastung des Datenbankspeichers. Die SQLite-Datenbank ist gewiss in der Lage, Dateien abzuspeichern, doch die zukünftige Ansammlung von Daten würde die Leistung der Datenbank beeinträchtigen. Aus dem Grund wurde beschlossen, einen Ordner zu erstellen, der zum Sammeln der Dateien dient und auf diese Weise als ein Teil der Datenbank zählt.

In Bezug auf den Import dieser Dateien wird ebenfalls zunächst das Vorhandensein kontrolliert. Wenn dies existiert, wird von jedem zu importierendem Dokument die Prüfsumme berechnet. Die resultierende Zahl wird mit den bereits vorhandenen Prüfsummen in der Datenbank verglichen. Falls die Summe noch nicht existiert, wird diese zusammen mit der dazugehörigen ID und der Dateigröße in die Tabellen „SG_Files“ und „SC_Files“ der Datenbank übertragen und zeitgleich das Dokument in den Datenbankordner eingefügt. Dabei wird der Dateiname durch die berechnete Prüfsumme ersetzt (siehe Abb. 27 und 28).

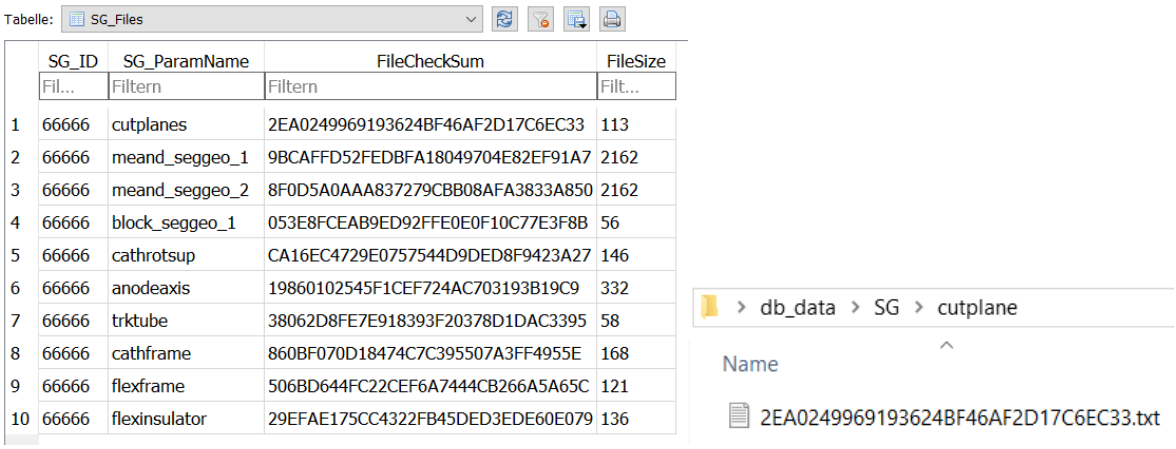


Abb. 27: Import der Prüfsummen in die Datenbank (links) und Import der „cutplane“-Datei in den Datenbankordner (rechts) (Eigendarstellung)

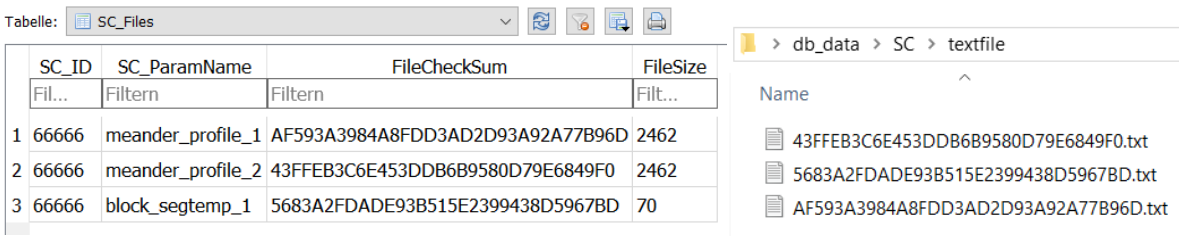


Abb. 28: Import der Prüfsummen in "SC_Files" in der Datenbank (links) und Import der Dateien in den Datenbankordner (rechts) (Eigendarstellung)

Sollte hingegen die Prüfsumme schon in der Datenbank vorliegen, gibt es womöglich bereits eine Datei mit gleichem Inhalt. Um dies zu überprüfen wird daraufhin die Prüfsumme der bereits existierenden Datei im Datenbankordner bestimmt, da dessen Inhalt sich eventuell geändert haben könnte. In diesem Zusammenhang hat der Anwender nun die Möglichkeit diese Datei durch die Neue zu ersetzen. Der Zweck des dargestellten Ablaufs ist die Vermeidung von mehrfach-existierenden Dokumenten im Datenbankordner, um den Speicher nicht unnötig zu belasten. Durch die Berechnung der Prüfsumme in Kombination mit der Datengröße, steigt der Wert der Einzigartigkeit der Datei. Diese Vorgehensweise wird für alle anderen Parametertypen vollzogen, die als Dateien behandelt werden.
Zusammengefasst lässt sich sagen, dass das beschriebene Programm in der Lage ist, die Einstellparameter für Opera in die Datenbank zu importieren. In Verbindung mit dem Datenbankordner kann eine zukünftige Belastung der Datenbank-Performance und Speicherplatz vermieden werden.

6.3 Export der Einstellparameter in die Datenbank

Im vorherigen Kapitel ist der Programmablauf zum Parameterimport in die Datenbank beschrieben. Im bisherigen Arbeitsablauf werden die Parameter für Opera durch die Dateien bereitgestellt. Opera sollte zukünftig die Einstellparameter aus der Datenbank entnehmen, um diese für Simulationen einzusetzen. Aus dem Grund wurde ein Prototyp entwickelt, der die Funktion besitzt, die Parameter in der Datenbank zu exportieren. Der dazugehörige Python-Code befindet sich im Anhang 7.
Das Programm beginnt wieder mit der Verbindungsherstellung zur SQLite-Datenbank. Danach wird in der Datenbank überprüft, ob die gesuchte SG-ID aus der Tabelle „SG“ existiert. Ist diese nicht vorhanden, erscheint eine Fehlermeldung. Wird diese Geometrie-ID gefunden, ist der nächste Schritt, die Konfiguration der Röhre zu exportieren. Dabei wird ebenfalls wieder in der „SG“-Tabelle nachgesehen, ob diese bestehend ist. Da Opera für die Modellierung der Röhre nicht die Modul-IDs verwendet, müssen diese zurück in die ursprünglichen Modulnamen transformiert werden. Die Umformung der Röhren-Konfiguration mit der SG-ID 66666 ist in der folgenden Abbildung dargestellt, wobei diese aufgrund der Länge unvollständig zu sehen ist (siehe Abb. 29).

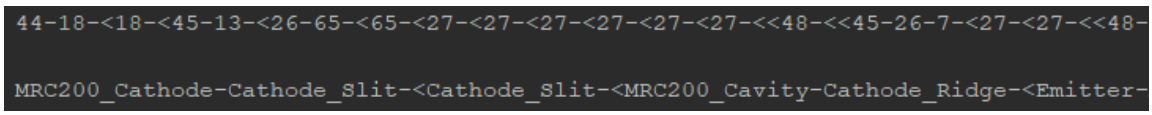


Abb. 29: Umformung der Konfiguration, bestehend aus Modul-IDs, in Modulnamen (Eigendarstellung)

Im nächsten Schritt werden alle Parameter aus den Tabellen „SG_Params“ und „SG_Files“ mit der entsprechenden SG-ID, in dem Fall die Nummer „66666“, zwischengespeichert. Diese werden am Ende des Programms auf Vollständigkeit überprüft, indem die Parameter mit den erwarteten Parametern verglichen werden. Anhand der Konfigurations-ID werden die zu erwarteten Parameter mithilfe der Tabelle „Module_SG_Params“ ebenfalls in einem Zwischenspeicher abgelegt. Je nachdem, ob das Modul mehrfach benutzt wird, wird den dazugehörigen Parameternamen zusätzlich ein Index für die Häufigkeit hinzugefügt.

Nun können die zwei zwischengespeicherten Parameter auf Gesamtheit verglichen werden. Ist der erwartete Parameter ebenfalls in den Tabellen „SG_Params“ oder „SG_Files“ enthalten, können diese exportiert werden. Da Opera noch nicht direkt auf die Datenbank und nur auf die Dateien zugreifen kann, wird ein Ordner errichtet, wo die exportierten Parameter in Form von Dateien erstellt werden können (vgl. Abb. 30, links). Parameter mit dem Parametertyp „Value“, die in „SG_Params“ stehen, werden mit der Röhren-Konfiguration in eine „*_params.comi“-Datei gespeichert (vgl. Abb. 30, rechts). Die Parameter aus „SG_Files“ werden hinsichtlich ihrer Prüfsumme mit den Dateien im Datenbankordner verglichen. Ist diese enthalten, wird die Datei in den Exportordner kopiert, wobei der Dateiname zu einer Verknüpfung aus der ID und dem Parameternamen wird. Der gleiche Ablauf des Programms gilt für den Export der SC-Parameter, nur dass die Parameter aus den Tabellen „SC_Params“ und „SC_Files“ entnommen werden.

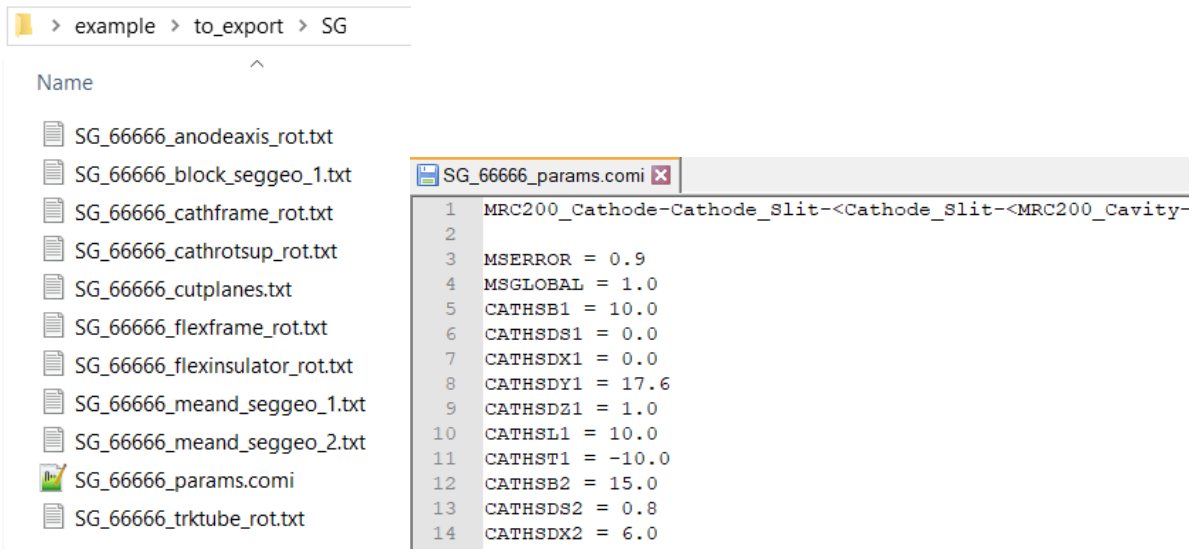


Abb. 30: Export der Einstellparameter aus Datenbank in einen Export-Ordner (links) und Einfügen der Konfiguration und Einstellparameter mit dem Parametertyp "Value" in eine Datei (rechts) (Eigendarstellung)

6.4 Import der Ergebnisse in die Datenbank

Neben den Einstellparametern bestehend aus Geometrie- und Betriebsparametern, sollten die Simulationsergebnisse ebenfalls in die Datenbank gespeichert werden. Aus dem Grund wird in diesem Kapitel der Prototyp für den Import der Ergebnisse vorgestellt. Im Anhang 8 befindet sich der dazugehörige Python-Code, wobei der Ablauf des Programms zum Importverfahren der Einstellparameter vergleichbar ist.

Am Anfang wird die Berechnungs-ID und der Ordnerpfad der Ergebnisdateien definiert. In diesem Beispiel wird die SC-ID „66666“ verwendet. In dem Ordner befindet sich eine Textdatei „SC_66666_results_values.txt“ mit den Ergebniswerten und eine Bilddatei „SC_66666_fs_image_bmp“ des Brennflecks. Die Namen und Inhalte der Dateien sind in Abbildung 31 zu entnehmen.

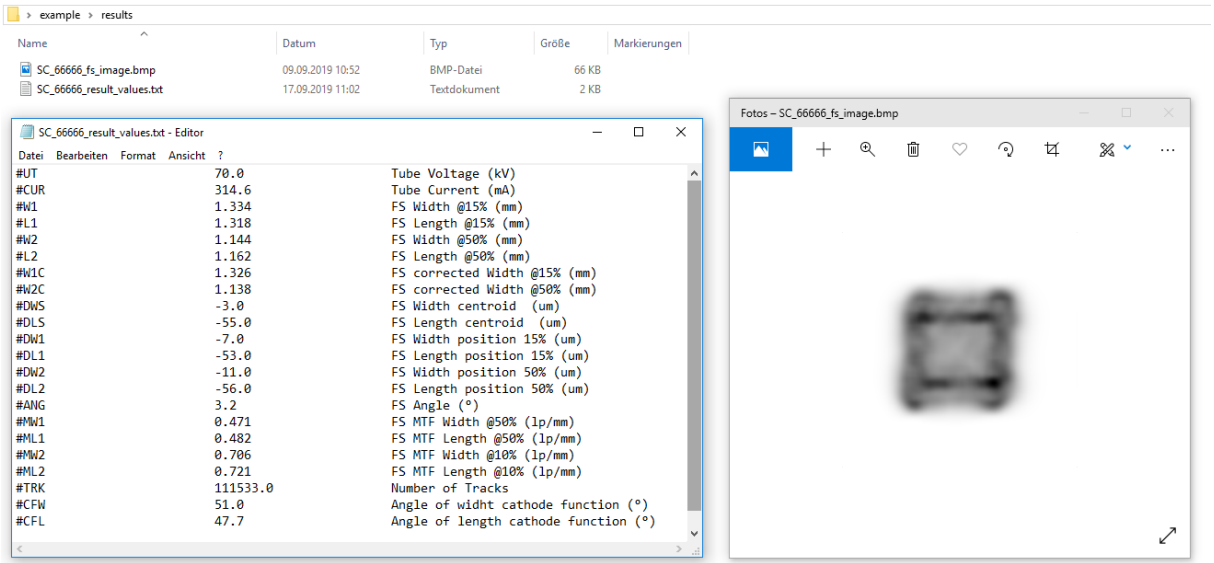


Abb. 31: Import der Ergebniswerte aus der Textdatei und Bild vom Brennfleck in die Datenbank (Eigendarstellung)

Das Programm beginnt mit der Verbindungsherstellung zur SQLite-Datenbank. Da die Dateinamen der Ergebnisse immer dieselbe Struktur aufweist und sich nur von der SC-ID-Nummer unterscheidet, wird in dem entsprechenden Ordner das Vorkommen der beiden Dokumente überprüft. Wird eine der Dateien nicht gefunden, taucht eine automatische Fehlermeldung auf. Existieren beide Dateien, ist das nächste Ziel die enthaltenden Daten in die Datenbank zu importieren. Je nach Dateiformat werden diese wiederum unterschiedlich abgespeichert. Hierbei werden die einzelnen Ergebnisparameter mit deren Werten aus der Textdatei in die Datenbank übertragen. Dies geschieht, indem zuerst die

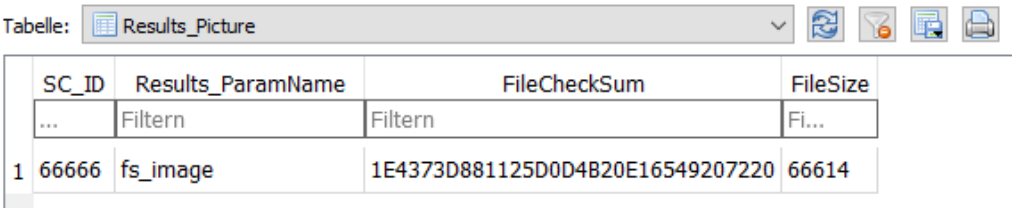
Vollzähligkeit der Parameter mit der Tabelle „Results_Params“ in der Datenbank verglichen wird. Werden alle erwarteten Parameter in der Textdatei gefunden, werden die Ergebnisparameter zusammen mit der dazugehörigen SC-ID und Wert in die Tabelle „Results_Value“ gesichert (vgl. Abb. 32).

Tabelle: Results_Value

	SC_ID	Results_ParamName	ParamValue
	...	Filtern	Filtern
1	66666	ut	70
2	66666	cur	314.6
3	66666	w1	1.334
4	66666	l1	1.318
5	66666	w2	1.144
6	66666	l2	1.162
7	66666	w1c	1.326
8	66666	w2c	1.138
9	66666	dws	-3
10	66666	dls	-55
11	66666	dw1	-7
12	66666	dl1	-53
13	66666	dw2	-11
14	66666	dl2	-56
15	66666	ang	3.2
16	66666	mw1	0.471
17	66666	ml1	0.482
18	66666	mw2	0.706
19	66666	ml2	0.721
20	66666	trk	111533
21	66666	cfw	51
22	66666	cfl	47.7

Abb. 32: Import der Ergebnisse in die Tabelle "Results_Value" (Eigendarstellung)

Anders als bei der Textdatei wird das Bild des Brennflecks in den Datenbankordner importiert und die Prüfsumme mit der Dateigröße in der Tabelle „Results_Picture“ vermerkt (siehe Abb. 33).



	SC_ID	Results_ParamName	FileCheckSum	FileSize
	...	Filtern	Filtern	Fi...
1	66666	fs_image	1E4373D881125D0D4B20E16549207220	66614

Abb. 33: Import der Prüfsumme des Brennfleckbilds in die Tabelle "Results_Picture" (Eigendarstellung)

Der Name der importierten Datei im Datenbankordner wird durch die Prüfsumme ersetzt, um diesen mit dem entsprechenden Datenbankeintrag so konform wie möglich zu machen. Anhand der Abbildungen 33 und 34 lassen sich die Prüfsummen und Dateigrößen in der Datenbank und im Datenbankordner vergleichen.

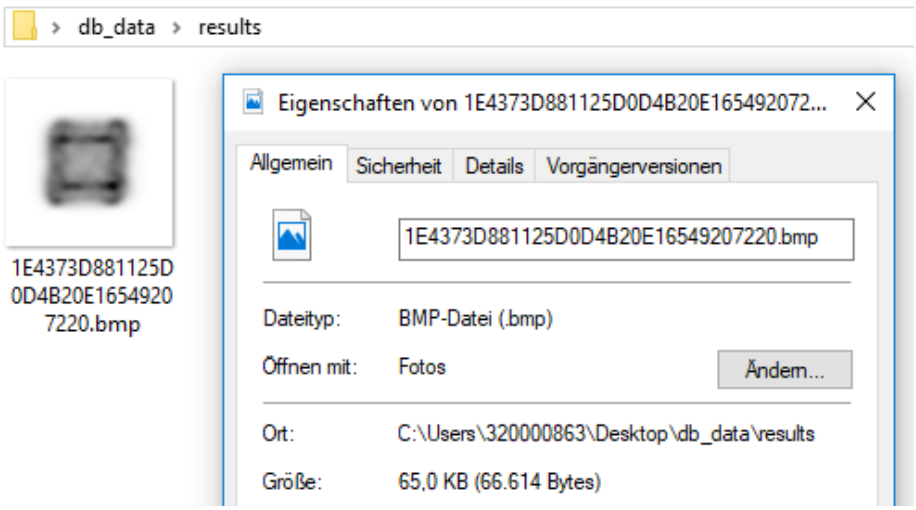


Abb. 34: Importierte Datei im Datenbankordner (Eigendarstellung)

7 Fazit und Ausblick

Die Masterarbeit hatte das Ziel, den jetzigen Arbeitsablauf bezüglich des Simulationsprogramms Opera durch eine Zentralisierung der Datenhaltung zu verbessern. Hierzu sollte eine passende Datenbank eingesetzt werden, mit der es möglich ist, die Einstellparameter für die Simulation von Röntgenröhren mit deren Ergebnissen abzuspeichern. Um diese Aufgabe zu bewältigen, wurde die Arbeit in drei Tätigkeitsbereiche untergliedert. Die erste Aufgabe war es, mithilfe eines selbst erstellten Anforderungsprofils nach einer geeigneten Datenbank zu suchen. Anhand der Anforderungsliste fiel der Entschluss, eine relationale SQL-Datenbank für die Aufgabenstellung der vorliegenden Masterarbeit zu verwenden. Mithilfe der Abfragesprache SQL können alle Operationen, vom Anlegen einer Datenbankstruktur bis zur gezielten Suche im Datenbestand, durchgeführt werden. Der Hauptgrund für den Beschluss auf SQLite war, dass dieser serverlos arbeitet und in der Standardausstattung bereits vorintegrierte Schnittstellen zu Python enthält. Auf diese Weise konnte man ohne zusätzliche Konfiguration unverzüglich mit der Programmierung der Schnittstellen starten. Nachdem die relationale Datenbankstruktur durch einen Programmcode angelegt wurde, wurde der Inhalt der Excel-Tabelle, die bisher zur Dokumentation der durchgeführten Simulationen diente, ebenso durch einen Programmcode in die Datenbank transferiert.

Der zweite Schritt war die Definition der Schnittstellen, welche die entsprechenden Parametertypen importieren und exportieren können. Die letzte Aufgabe widmete sich der der Realisierung der im zweiten Schritt definierten Schnittstellen mit Python , wodurch die Generierung von neuen Simulationen möglich gemacht wurde. Daneben wurden zusätzlich noch die Prototypen der Schnittstellen für den Import und Export der Einstellparameter und Ergebnissen entwickelt. Insgesamt wurden vier Funktionen erstellt:

1. Generierung von neuen Berechnungs-IDs
2. Import der Einstellparameter in die Datenbank
3. Export der Einstellparameter aus der Datenbank
4. Import der Berechnungsergebnisse in die Datenbank

Die Generierung einer neuen Simulations-ID würde zukünftig wie im Kapitel 6.1 erfolgen. Diese erstellte Funktion ist in der Lage die Excel-Tabelle zu ersetzen. Das manuelle Erzeugen der IDs kann bereits durch eine automatische Generierung ausgetauscht werden, was die Fehlerrate bzgl. Tippfehler oder Eingabe von bereits vorhandener ID verringern würde. Im momentanen Arbeitsablauf muss vor der manuellen Eingabe im Excel-Dokument vom Nutzer eine Aktualisierung erfolgen, um eine doppelte ID zu verhindern. Das Risiko einer Simulation mit derselben ID würde zukünftig durch die automatische Generierung wegfallen.

In den Kapiteln 6.2 bis 6.4 sind die Programme, die für den Import und Export der Einstellparameter und deren Ergebnisse zuständig sind, beschrieben. Aufgrund der verschiedenen Arten von Parametern bzw. Dateiformate, wurde ein Datenbankordner erstellt, womit bestimmte Dateien abgespeichert werden können. Dadurch wird der Datenbankspeicher zukünftig nicht belastet und die Performance aufrechterhalten.

In der vorliegenden Arbeit wurde der gleichzeitige Zugriff auf die Datenbank nicht getestet. Dies kann als nächster Anknüpfungspunkt der Arbeit fortgeführt werden. Damit die Nutzer auf die gleiche Datenbank und den dazugehörigen Datenbankordner zugreifen können, muss diese in einem geteilten Netzwerkordner mit Zugriffskontrolle liegen, da SQLite wie bekannt auf einem serverlosen Datenbankmanagementsystem basiert.

Außerdem ist noch zu beachten, dass die Kommunikation zwischen Datenbank und Opera hergestellt werden muss, denn bekanntlich erhält Opera bei Simulationen alle erforderlichen Daten aus den Textdateien. Das bedeutet, dass, bevor die Datenbank in den jetzigen Arbeitsablauf implementiert werden kann, muss das Simulationsprogramm in der Lage sein, auf die Parameter in der Datenbank zuzugreifen. Zur Folge muss das Programm so umgeändert werden, dass es sowohl die Parameter aus der Datenbank entnehmen kann als auch die aus dem Datenbankordner.

Zusammengefasst lässt sich festhalten, dass die Ziele der Arbeit erreicht worden sind. Durch die Datenbankrecherche und Ausarbeitung der Prototypen erwiesen sich keinerlei technische Grenzen im Hinblick auf die eingesetzte Datenbank und die damit verbundenen Ziele. Die Datenbank bietet dem Anwender einen guten Überblick und die volle Kontrolle über die Parameter und Ergebnisse. Bei Implementierung des in dieser Arbeit entworfenen Datenbankverwaltungssystems, würde die Röhrenentwicklung bei Philips von einer zentralen Datenhaltung profitieren, auf die alle Nutzer Zugang hätten. Dies würde zu einer erheblichen Erleichterung des Arbeitsablaufs mit Opera führen. Schlussfolgernd kann man sagen, dass SQLite sich bisher als eine optimale Lösung für die Datenhaltung und Aufgaben dieser Masterarbeit erwies.

Literaturverzeichnis

1. M. Galanski KL. Röntgenuntersuchung. 30.07.2019. https://www.krebsgesellschaft.de/onko-internetportal/basis-informationen-krebs/diagnosemethoden/roentgenuntersuchung.html. Accessed 31 Jul 2019.
2. Wetzke M BL. Röntgendiagnostik. http://www.gesundheits-lexikon.com/Medizingeraetediagnostik/Konventionelle-Roentgendiagnostik/Roentgendiagnostik-.html. Accessed 31 Jul 2019.
3. Philips. Über Philips. https://www.philips.de/a-w/ueber-philips/unternehmensprofil.html. Accessed 23 Aug 2019.
4. Schulthess GKv. Röntgen, Computertomografie & Co: Wie funktioniert medizinische Bildgebung? 1st ed. Berlin, Heidelberg, s.l.: Springer Berlin Heidelberg; 2017.
5. Krieger H. Strahlungsquellen für Technik und Medizin. 3rd ed. Berlin, Heidelberg: Springer Spektrum; 2018.
6. Schlegel W, Karger CP, Jäkel O, editors. Medizinische Physik: Grundlagen - Bildgebung - Therapie - Technik. Berlin, Heidelberg: Springer Berlin Heidelberg; 2018.
7. Juergen Schacht. Info Röntgen. http://juergen-schacht.de/inforntgen3.htm. Accessed 24 Sep 2019.
8. St. Pierre M, Breuer G, editors. Simulation in der Medizin: Grundlegende Konzepte - Klinische Anwendung. 2nd ed. Berlin, Heidelberg: Springer Berlin Heidelberg; 2018.
9. Bundesverband Geothermie. SIMULATION (PHYSIK). https://www.geothermie.de/bibliothek/lexikon-der-geothermie/s/simulation-physik.html. Accessed 23 Aug 2019.
10. Dassault Systemes. Opera Simulation Software. https://operafea.com/product/. Accessed 19 Sep 2019.
11. Dessault Systemes. Opera Simulation Software. https://operafea.com/charged-particle-devices/. Accessed 19 Sep 2019.
12. Prof. Dr. Jean-Paul Thommen. Anspruchsgruppen. https://wirtschaftslexikon.gabler.de/definition/anspruchsgruppen-27010. Accessed 30 Jun 2019.
13. Meier A, Kaufmann M. SQL- & NoSQL-Datenbanken. 8th ed. Berlin, Heidelberg: Springer Vieweg; 2016.
14. Meier A. Werkzeuge der digitalen Wirtschaft: Big Data, NoSQL & Co: Eine Einführung in relationale und nicht-relationale Datenbanken. Wiesbaden: Springer Vieweg; 2018.
15. Unterstein M, Matthiessen G. Relationale Datenbanken und SQL in Theorie und Praxis. 5th ed. Berlin: Springer Vieweg; 2012.
16. Schicker E. Datenbanken und SQL: Eine praxisorientierte Einführung mit Anwendungen in Oracle, SQL Server und MySQL. 5th ed. Wiesbaden: Springer Vieweg; 2017.
17. Stefan Luber NL. Was ist NoSQL? 12.06.2017. https://www.bigdata-insider.de/was-ist-nosql-a-615718/. Accessed 9 Oct 2019.
18. SQLite. SQLite. https://www.sqlite.org/serverless.html. Accessed 14 Oct 2019.
19. Rene Büst. „Serverless Infrastructure“: Der schmale Grat zwischen Einfachheit und Kontrollverlust. https://www.crisp-research.com/serverless-infrastructure-der-schmale-grat-zwischen-einfachheit-und-kontrollverlust/. Accessed 15 Oct 2019.
20. Python Software Foundation. PEP 249. https://www.python.org/dev/peps/pep-0249/. Accessed 24 Oct 2019.
21. Jan-Dirk in IT-Talents. Was sind Prüfsummen (Checksums)? 16.04.2016. https://www.it-talents.de/blog/it-talents/was-sind-pruefsummen-checksums. Accessed 18 Oct 2019.
22. Marcel Peters. MD5 Checksum. 28.03.2014. https://praxistipps.chip.de/md5-checksum-wozu-braucht-man-sie_29059. Accessed 24 Oct 2019.

Anhang

Anhang 1: Anforderungsliste für die Datenbank

Anhang 2: Überblick der passenden Datenbankmanagementsysteme auf dem Markt

Anhang 3: Datenbankstruktur

Anhang 4: Auflistung der Tabellen in der Datenbank

Anhang 5: Python-Code für das Menü zur Generierung neuer IDs

Anhang 6: Python-Code für den Einstellparameter-Import

Anhang 7: Python-Code für den Einstellparameter-Export

Anhang 8: Python-Code für den Ergebnis-Import

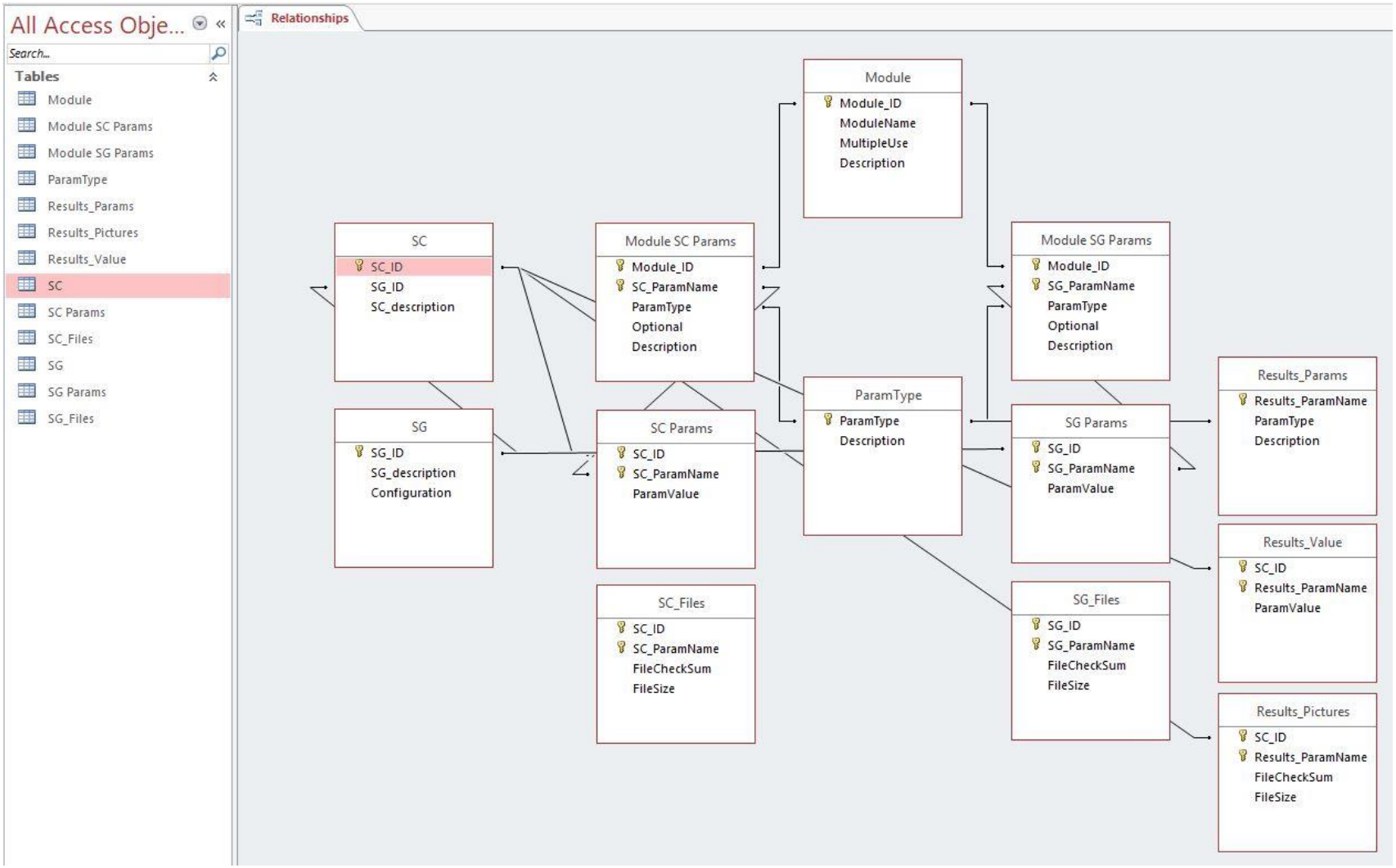
Anhang 1: Anforderungsliste für die Datenbank

Herkunft (Dokument/Norm Referenz wenn möglich)	Kategorie	Anforderung	Spezifikation (wenn möglich)	essential/nice-to-have	comment
Engineer X-Ray Tube	Funktionalität	Verwalten von Geometriedateien	SG-IDs generieren + Beschreibung	essential	
Engineer X-Ray Tube	Funktionalität	Verwalten von Berechnungsdateien	SC-IDs generiern + Beschreibung	essential	
Engineer X-Ray Tube	Funktionalität	Verwalten der Relation von Berechnung und Geometrie	Relation zwischen SC- und SG-ID kann hergestellt werden	essential	
Engineer X-Ray Tube	Funktionalität	Schnittstellen durch Python programmierbar	Parameter aus SG- und SC-Dateien importieren/exportieren	essential	Röhrenentwicklung spricht Opera durch die Programmiersprache Python an
Engineer X-Ray Tube	Funktionalität	Geometrie- und Berechnungsdaten extrahieren	Parameter aus Datenbank auslesbar	essential	
Engineer X-Ray Tube	Funktionalität	Geometrie- und Berechnungsdaten filtern	Informationen filtern	nice-to-have	
Engineer X-Ray Tube	Security	Garantie einer Zugriffssicherung	nur Opera-Anwender dürfen Zugriff auf die Datenbank haben	essential	wird erstmal nicht berücksicht, da die Datenbank sich im Netzwerksystem von Philips befindet
Engineer X-Ray Tube	Funktionalität	Möglichkeit zur Abspeicherung von verschiedenen Arten von Dateien	Text- und Bildformat	essential	to be discussed: direkt in die Datenbank oder lokal abspeichern?
Engineer X-Ray Tube	Funktionalität	Abspeichern einer bestimmten Menge an Daten	Gigabyte-Bereich	essential	
Engineer X-Ray Tube	Funktionalität	Skalierbarkeit der Datenbank	Speichererweiterung	nice-to-have	

Anhang 2: Überblick der passenden Datenbankmanagementsysteme auf dem Markt

Relationale Datenbank-managementsystem / Eigenschaften	opt. DB	MySQL	Oracle	PostgreSQL	Microsoft SQL Server	Microsoft SQL Server Express	SQLite	MariaDB	MS Access
Kapazität									
max. Database/row/column size	max. Größe der DB: Gigabyte-Bereich	determined by operating system constraints on file sizes, not by MySQL internal limits: Operating System File-size Limit Win32 w/ FAT/FAT32 2GB/4GB Win32 w/ NTFS 2TB (possibly larger) Linux 2.2-Intel 32-bit 2GB (LFS: 4GB) Linux 2.4+ (using ext3 file system) 4TB Solaris 9/10 16TB MacOS X w/ HFS+ 2TB	max. Spalten: 1000 pro Tabelle	max. Größe der DB: unbeschränkt max. Größe einer Tabelle: 32 TB max. Größe eines Datensatzes: 1.6 TB max. Größe einer Zelle: 1 GB max. Anzahl Zeilen pro Tabelle: unbeschränkt max. Anzahl der Spalten pro Tabelle ist abh. von den verwendeten Datentypen und liegt zwischen 250 und 1600	max. Größe der DB: 524272 TB max. Zeilen: 30.000	max. Größe der DB: 10 GB	max. Größe der DB: 140 TB max. Größe eines Datensatzes: 2GB max. Zeilen einer Tabelle: 2^64 Standarteinstellung für max. Spalten: 2000 (max. 32767)	max. Größe der Datenbank: 64 TB max. Spalten: 1000 pro Tabelle	max. Größe der DB: 2 GB
Security									
consistency/integrity	ACID	ACID	ACID	ACID	ACID	ACID	ACID	ACID	ACID
Skalierbarkeit									
vertikal	x	x	x	x	x	x	x	x	
horizontal	x								
Kosten									
open source	x	x		x		x	x	x	
cost			x		x				
Multiple Users									
server/no server		Server Betriebssysteme: FreeBSD, Linux, OS X, Solaris, Windows	Server	Server	Server	Server	Serverlos	Server	Serverlos
versch. Datentypen									
values	möglich	möglich	möglich	möglich	möglich	möglich	möglich	möglich	möglich
pictures	möglich	möglich	möglich	möglich	möglich	möglich	möglich	möglich	möglich
rotation contours									
Programmiersprache									
Python	x	MySQLdb	cx_Oracle	x	pyodbc		sqlite3	MySQL Connector/Python	pyodbc

Anhang 3: Datenbankstruktur



Anhang 4: Auflistung der Tabellen in der Datenbank

Name	Typ	Schema
Module		CREATE TABLE "Module" ("Module_ID" INTEGER, "ModuleName" TEXT, "MultipleUse" INTEGER, "Description" TEXT, PRIMARY KEY("Module_ID"))
Module_ID	INTEGER	"Module_ID" INTEGER
ModuleName	TEXT	"ModuleName" TEXT
MultipleUse	INTEGER	"MultipleUse" INTEGER
Description	TEXT	"Description" TEXT
Module_SC_Params		CREATE TABLE "Module_SC_Params" ("Module_ID" INTEGER, "SC_ParamName" TEXT, "ParamType" INTEGER, "Optional" INTEGER, "Description" TEXT, PRIMARY KEY("Module_ID","SC_ParamName"), FOREIGN KEY("Module_ID") REFERENCES "Module"("Module_ID"), FOREIGN KEY("ParamType") REFERENCES "ParamType"("ParamType"))
Module_ID	INTEGER	"Module_ID" INTEGER
SC_ParamName	TEXT	"SC_ParamName" TEXT
ParamType	INTEGER	"ParamType" INTEGER
Optional	INTEGER	"Optional" INTEGER
Description	TEXT	"Description" TEXT
Module_SG_Params		CREATE TABLE "Module_SG_Params" ("Module_ID" INTEGER, "SG_ParamName" TEXT, "ParamType" TEXT, "Optional" INTEGER, "Description" TEXT, PRIMARY KEY("Module_ID","SG_ParamName"), FOREIGN KEY("ParamType") REFERENCES "ParamType"("ParamType"), FOREIGN KEY("Module_ID") REFERENCES "Module"("Module_ID"))
Module_ID	INTEGER	"Module_ID" INTEGER
SG_ParamName	TEXT	"SG_ParamName" TEXT
ParamType	TEXT	"ParamType" TEXT
Optional	INTEGER	"Optional" INTEGER
Description	TEXT	"Description" TEXT
ParamType		CREATE TABLE "ParamType" ("ParamType" TEXT, "Description" TEXT, PRIMARY KEY("ParamType"))
ParamType	TEXT	"ParamType" TEXT
Description	TEXT	"Description" TEXT
Results_Params		CREATE TABLE "Results_Params" ("Results_ParamName" TEXT, "ParamType" INTEGER, "Description" TEXT, PRIMARY KEY("Results_ParamName"), FOREIGN KEY("ParamType") REFERENCES "ParamType"("ParamType"))
Results_ParamName	TEXT	"Results_ParamName" TEXT

Name	Typ	Schema
ParamType	INTEGER	"ParamType" INTEGER
Description	TEXT	"Description" TEXT
Results_Picture		CREATE TABLE "Results_Picture" ("SC_ID" INTEGER, "Results_ParamName" TEXT, "FileCheckSum" INTEGER, "FileSize" INTEGER, PRIMARY KEY("SC_ID","Results_ParamName"), FOREIGN KEY("SC_ID") REFERENCES "SC"("SC_ID"))
SC_ID	INTEGER	"SC_ID" INTEGER
Results_ParamName	TEXT	"Results_ParamName" TEXT
FileCheckSum	INTEGER	"FileCheckSum" INTEGER
FileSize	INTEGER	"FileSize" INTEGER
Results_Value		CREATE TABLE "Results_Value" ("SC_ID" INTEGER, "Results_ParamName" TEXT, "ParamValue" INTEGER, PRIMARY KEY("SC_ID","Results_ParamName"), FOREIGN KEY("SC_ID") REFERENCES "SC"("SC_ID"))
SC_ID	INTEGER	"SC_ID" INTEGER
Results_ParamName	TEXT	"Results_ParamName" TEXT
ParamValue	INTEGER	"ParamValue" INTEGER
SC		CREATE TABLE "SC" ("SC_ID" INTEGER, "SG_ID" INTEGER, "SC_description" TEXT, PRIMARY KEY("SC_ID"), FOREIGN KEY("SG_ID") REFERENCES "SG"("SG_ID"))
SC_ID	INTEGER	"SC_ID" INTEGER
SG_ID	INTEGER	"SG_ID" INTEGER
SC_description	TEXT	"SC_description" TEXT
SC_Files		CREATE TABLE "SC_Files" ("SC_ID" INTEGER, "SC_ParamName" INTEGER, "FileCheckSum" TEXT, "FileSize" INTEGER, PRIMARY KEY("SC_ID","SC_ParamName"))
SC_ID	INTEGER	"SC_ID" INTEGER
SC_ParamName	INTEGER	"SC_ParamName" INTEGER
FileCheckSum	TEXT	"FileCheckSum" TEXT
FileSize	INTEGER	"FileSize" INTEGER
SC_Params		CREATE TABLE "SC_Params" ("SC_ID" INTEGER, "SC_ParamName" TEXT, "ParamValue" INTEGER, PRIMARY KEY("SC_ID","SC_ParamName"), FOREIGN KEY("SC_ID") REFERENCES "SC"("SC_ID"))
SC_ID	INTEGER	"SC_ID" INTEGER
SC_ParamName	TEXT	"SC_ParamName" TEXT
ParamValue	INTEGER	"ParamValue" INTEGER
SG		CREATE TABLE "SG" ("SG_ID" INTEGER, "SG_description" TEXT, "Configuration" TEXT, PRIMARY KEY("SG_ID"))

Name	Typ	Schema
SG_ID	INTEGER	"SG_ID" INTEGER
SG_description	TEXT	"SG_description" TEXT
Configuration	TEXT	"Configuration" TEXT
SG_Files		CREATE TABLE "SG_Files" ("SG_ID" INTEGER, "SG_ParamName" INTEGER, "FileCheckSum" TEXT, "FileSize" INTEGER, PRIMARY KEY("SG_ID","SG_ParamName"))
SG_ID	INTEGER	"SG_ID" INTEGER
SG_ParamName	INTEGER	"SG_ParamName" INTEGER
FileCheckSum	TEXT	"FileCheckSum" TEXT
FileSize	INTEGER	"FileSize" INTEGER
SG_Params		CREATE TABLE "SG_Params" ("SG_ID" INTEGER, "SG_ParamName" TEXT, "ParamValue" INTEGER, PRIMARY KEY("SG_ID","SG_ParamName"), FOREIGN KEY("SG_ID") REFERENCES "SG"("SG_ID"))
SG_ID	INTEGER	"SG_ID" INTEGER
SG_ParamName	TEXT	"SG_ParamName" TEXT
ParamValue	INTEGER	"ParamValue" INTEGER

Anhang 5: Python-Code für das Menü zur Generierung neuer IDs

menu_id_generator.py

```
import sqlite3


def print_menu():

    print(30 * "-", "Verwaltung von Berechung & Geometrie", 30 * "-")
    print("1. Generate SG ID and add description")
    print("2. Generate SC ID and add description")
    print("3. Exit")
    print(98 * "-")


loop = True

while loop:
    conn = sqlite3.connect(r"C:\Users\320000863\Desktop\CalcDB.db", timeout=10)
    c = conn.cursor()
    print_menu()
    choice = input("Please select[1-3]: ")

    # SG: Generate
    if choice == "1":
        int_choice = 1
        choice = ''
        print("Please add a description for the new SG ID.")
        while len(choice) == 0:
            choice = input("SG_description: ")
            c.execute("INSERT INTO SG(SG_description) VALUES(?)", (choice,))
            c.execute("SELECT SG_ID FROM SG WHERE SG_ID = (SELECT max(SG_ID) FROM SG)")
            lastSGID = c.fetchone()[0]
            conn.commit()
            print("The generated SG ID: " + str(lastSGID) + "\n" + "Description: " + choice)

    # SC: Generate
    elif choice == "2":
        int_choice = 2
        choice = ''
        choice2 = ''
        while len(choice) == 0:
            choice2 = input("Please enter a description for the calculation: ")
            choice = input("Please select a SG_ID for this calculation: ")
            c.execute('''SELECT SG_ID FROM SG WHERE SG_ID = ?''', (choice,))
            exists = c.fetchall()

    # SG not existing in DB
            if len(exists) == 0:
                conn.commit()
                print("The entered SG_ID is not existing in the DB. Do you want to generate a new SG_ID? [y/n]")
```

```
49
50          # y -> Generating not existing SG in DB
51                  if input() == "y":
52                      print("Please add a description for the new SG_ID!")
53                      choice3 = input("Description for new SG_ID: ")
54                      c.execute("INSERT INTO SG(SG_description) VALUES(?)", (choice3,))
55                      c.execute("SELECT SG_ID FROM SG WHERE SG_ID = (SELECT max(SG_ID) FROM SG)")
56                      lastSGID = c.fetchone()[0]
57                      c.execute("INSERT INTO SC(SG_ID, SC_description) VALUES(?,?)", (lastSGID, choice2))
58                      c.execute("SELECT SC_ID FROM SC WHERE SC_ID = (SELECT max(SC_ID) FROM SC)")
59                      lastSCID = c.fetchone()[0]
60                      conn.commit()
61                      print("SG_ID" + " " + str(lastSGID) + " " + "is generated. It will be used for the calculation."
62                            + "\n" + "The generated SC_ID: " + str(lastSCID))
63
64          # n -> back to main menu
65                  else:
66                      print("Please select[1-3]: ")
67
68          # Existing SG will be added to calculation
69              elif len(exists) >= 0:
70                  c.execute("INSERT INTO SC(SG_ID, SC_description) VALUES(?, ?)", (choice, choice2))
71                  c.execute("SELECT SC_ID FROM SC WHERE SC_ID = (SELECT max(SC_ID) FROM SC)")
72                  lastSCID = c.fetchone()[0]
73                  conn.commit()
74                  print("The ID " + choice + " is existing in the DB! It will be used for the calculation."
75                        + "\n" + "The generated SC_ID: " + str(lastSCID))
76
77          # Exit
78          elif choice == "3":
79              int_choice = 3
80              print("Exit")
81              loop = False
82          else:
83              input("Wrong selected option. Enter any key to try again!")
84
85
86      if __name__ == '__main__':
87          print_menu()
```

Anhang 6: Python-Code für den Einstellparameter-Import

params_import_to_db.py

```python
import hashlib
import os
import re
import shutil
import sqlite3


def get_db_connection():
    conn = sqlite3.connect(r'.\CalcDB.db', timeout=10)
    return conn, conn.cursor()


def fetchall(sql, *fields):
    con, cur = get_db_connection()
    cur.execute(sql, *fields)
    result = cur.fetchall()
    con.close()
    return result


def commit(sql, *fields):
    con, cur = get_db_connection()
    cur.execute(sql, *fields)
    con.commit()
    con.close()


def commitmany(sql, *fields):
    con, cur = get_db_connection()
    cur.executemany(sql, *fields)
    con.commit()
    con.close()


def convert_string_to_id(params_comi_dir):
    module_id_name_list = fetchall('''SELECT Module_ID, ModuleName FROM MODULE''')

    with open(params_comi_dir) as params_comi_file:
        config_string_module_list = re.findall(r'^\s*\$\s*STR\s+CONFIG\s+(\b.*)\s*$', params_comi_file.read(),
                                               flags=(re.IGNORECASE | re.MULTILINE))
    config_string = ''.join(config_string_module_list).replace('&CONFIG&', '')

    module_dict = {key: value for value, key in module_id_name_list}
    for module_name in sorted(set(re.findall(r'([^<-]+)-', config_string)), key=len, reverse=True):
        config_string = config_string.replace(str(module_name), str(module_dict[module_name]))

    return config_string
```

```python
def read_params_comi(comi):
    param_dict = dict()

    with open(comi, 'r') as comi_file:
        type_value_list = re.findall(r'^\s*\$\s*CON\s+#(\w+)\s+(\S+)', comi_file.read(),
                                     flags=(re.IGNORECASE | re.MULTILINE))

    for param_name, param_value in type_value_list:
        try:
            param_dict[param_name.lower()] = float(param_value)
        except ValueError:
            param_dict[param_name.lower()] = param_value

    return param_dict


def extract_geo_id(comi):
    with open(comi, 'r') as comi_file:
        geo_id = re.findall(r'^\s*\$\s*STR\s+GEOID\s+[ST]G_(\d+)\b', comi_file.read(),
                            flags=(re.IGNORECASE | re.MULTILINE))
    # TODO: handle state if there is no geo_id
    return geo_id[0]


def calculate_checksum_from_file(filename):
    try:
        with open(filename, 'rb') as fb:
            file_content = fb.read()
    except FileNotFoundError:
        return 0
    hasher = hashlib.md5()
    hasher.update(file_content)
    return hasher.hexdigest().upper()


# TODO: Remove folder, create folder from data type and file type as preparation for TG/TC
file_types = dict(RotSolid=[r'{}\SG_{:05d}_{}_rot.txt', r'{}\{}.txt', r'.\db_data\SG\rotsolid'],
                  Cutplane=[r'{}\SG_{:05d}_{}.txt', r'{}\{}.txt', r'.\db_data\SG\cutplane'],
                  OPC=[r'{}\SG_{:05d}_{}.opc', r'{}\{}.opc', r".\db_data\SG\opc"],
                  Blending=[r'{}\SG_{:05d}_{}_blending.txt', r'{}\{}.txt', r'.\db_data\SG\blending'],
                  meand_seggeo=[r'{}\SG_{:05d}_{}.txt', r'{}\{}.txt', r'.\db_data\SG\meand_seggeo'],
                  block_seggeo=[r'{}\SG_{:05d}_{}.txt', r'{}\{}.txt', r'.\db_data\SG\block_seggeo'],
                  Textfile=[r'{}\SC_{:05d}_{}.txt', r'{}\{}.txt', r'.\db_data\SC\textfile'],
                  )


def prepare_import_data(import_folder_dir, ident, data_type='SG'):
    import_data = dict(Config={}, Params={}, Files={}, Type=data_type, Comi={}, Folder=import_folder_dir)
    name_params_comi = '{}\\{}_{:05d}_params.comi'.format(
        import_folder_dir, import_data['Type'], ident)
    letter_0, letter_1 = import_data['Type'][0:2]

    if letter_1 == 'G':
        config_id_string = convert_string_to_id(name_params_comi)
    elif letter_1 == 'C':
        geo_ident = extract_geo_id(name_params_comi)
        conf = fetchall('SELECT Configuration FROM {0}G WHERE {0}G_ID = ?'.format(letter_0), [geo_ident])
        if len(conf) == 0:
            # TODO: Handle state if there is no Configuration in database
            return
        config_id_string = conf[0][0]
        import_data['Config'].update(dict(GeoIdent=geo_ident))
    else:
        return

    import_data['Config'].update(dict(Ident=ident, Configuration=config_id_string, Description=None))
    import_data['Comi'] = read_params_comi(name_params_comi)
    return import_data
```

```python
def extract_import_data(import_data):
    def add_missing_parameter(name):
        if not param_optional:
            missing.append(name)
            print('Not found: {}'.format(name))

    missing = []
    additional = []
    config_dict, params_dict, file_dict = import_data['Config'], import_data['Params'], import_data['Files']
    import_folder_dir, params_comi_dict = import_data['Folder'], import_data['Comi']
    ident = config_dict['Ident']
    config_id_list = re.findall(r'[0-9]+', config_dict['Configuration'])
    config_id_list.insert(0, str(fetchall('SELECT Module_ID FROM Module WHERE ModuleName = "Main"')[0][0]))
    sql_params = 'SELECT * FROM Module_{}_Params WHERE Module_ID = ?'.format(import_data['Type'])

    occurrence_counter_dict = dict()
    for module_id in config_id_list:
        module_exists = fetchall('SELECT MultipleUse FROM Module WHERE Module_ID = ?', (module_id,))
        if not module_exists:
            print(module_id + ' is not existing in the database.')
            continue
        module_multiple_use = module_exists[0][0]
        occurrence_counter_dict[module_id] = occurrence_counter_dict.get(module_id, 0) + 1

        module_params = fetchall(sql_params, (module_id,))
        for _, param_name, param_type, param_optional, _ in module_params:
            param_name = param_name.lower()
            if module_multiple_use == 1:
                if param_type == 'Value':
                    param_name = '{}{}'.format(param_name, occurrence_counter_dict[module_id])
                else:
                    param_name = '{}_{}'.format(param_name, occurrence_counter_dict[module_id])
            if param_type == 'Value':
                if param_name in params_comi_dict:
                    params_dict[param_name] = params_comi_dict[param_name]
                else:
                    add_missing_parameter(param_name)
                continue
            if param_type in file_types:
                expected_filename = file_types[param_type][0].format(import_folder_dir, ident, param_name)
                if os.path.isfile(expected_filename):
                    file_dict[param_name] = [param_type, expected_filename]
                else:
                    add_missing_parameter(param_name)
                continue

            print('Unknown parameter type.')
    additional.extend(sorted(set(params_comi_dict) - set(params_dict)))
    return import_data, missing, additional
```

```python
def store_files(import_data: dict):
    config_dict = import_data['Config']
    sql_select = 'SELECT FileCheckSum FROM {}_Files WHERE FileCheckSum = ?'.format(import_data['Type'])
    sql_insert = 'INSERT OR REPLACE INTO {}_Files VALUES(?,?,?,?)'.format(import_data['Type'])

    for param_name, param_data in import_data['Files'].items():
        file_type, text_file = param_data
        textfile_checksum = calculate_checksum_from_file(text_file)
        textfile_size = os.path.getsize(text_file)
        checksums_in_db_list = fetchall(sql_select, (textfile_checksum,))
        file_in_db_folder = file_types[file_type][1].format(file_types[file_type][2], textfile_checksum)
        if len(checksums_in_db_list) > 0:
            db_checksum = calculate_checksum_from_file(file_in_db_folder)
            if textfile_checksum != db_checksum:
                shutil.copy(text_file, file_in_db_folder)
        else:
            shutil.copy(text_file, file_in_db_folder)

        commit(sql_insert, (config_dict['Ident'], param_name, textfile_checksum, textfile_size))


def store_import_data(store_data: dict):
    config_dict = store_data['Config']
    letter_0, letter_1 = store_data['Type'][0:2]

    if letter_1 == 'G':
        commit('INSERT OR REPLACE INTO {0}G({0}G_ID, Configuration, {0}G_description) VALUES(?,?,?)'.format(letter_0),
               [config_dict['Ident'], config_dict['Configuration'], config_dict['Description']])
    elif letter_1 == 'C':
        commit('INSERT OR REPLACE INTO {0}C({0}C_ID, {0}G_ID, {0}C_description) VALUES(?,?,?)'.format(letter_0),
               [config_dict['Ident'], config_dict['GeoIdent'], config_dict['Description']])
    else:
        return
    commitmany('INSERT OR REPLACE INTO {}_Params VALUES(?,?,?)'.format(store_data['Type']),
               ((config_dict['Ident'], k, v) for k, v in store_data['Params'].items()))
    store_files(store_data)


if __name__ == '__main__':
    # TODO: Handle missing parameter and files
    # TODO: Handle additional parameters in comi file
    # TODO: test with TC and TG data when DB is ready

    data = prepare_import_data(r'.\example\to_import\SG\6', 66666, 'SG')
    data, _, _ = extract_import_data(data)
    store_import_data(data)
    data = prepare_import_data(r'.\example\to_import\SC\6', 66666, 'SC')
    data, _, _ = extract_import_data(data)
    store_import_data(data)
```

Anhang 7: Python-Code für den Einstellparameter-Export

geo_params_export_from_db.py

```python
import sqlite3
import re
import os.path
import hashlib

sg_id = 66666
export_file_dir = r"C:\Users\320000863\Desktop\example\to_export\SG"


def get_db_connection():
    conn = sqlite3.connect(r"C:\Users\320000863\Desktop\CalcDB.db", timeout=10)
    return conn, conn.cursor()


def check_if_sg_exist():
    conn, c = get_db_connection()
    c.execute('SELECT SG_ID FROM SG WHERE SG_ID = ?', (sg_id,))
    sg_exist = c.fetchall()
    sg_exist = len(sg_exist)
    conn.close()
    return sg_exist


def check_if_config_exist():
    conn, c = get_db_connection()
    c.execute('SELECT Configuration FROM SG WHERE SG_ID = ?', (sg_id,))
    configuration = c.fetchone()[0]
    conn.close()
    return configuration


def get_config_string(configuration):
    conn, c = get_db_connection()
    params_comi_dir = '{}\\SG_{}_params.comi'.format(export_file_dir, sg_id)
    params_comi = open(params_comi_dir, "w")
    c.execute('SELECT Module_ID, ModuleName FROM MODULE')
    module_list = c.fetchall()
    for module_id, module_name in module_list:
        re_pat = '{:2d}-'.format(module_id)
        re_sub = '{}-'.format(module_name)
        configuration = re.sub(re_pat, re_sub, configuration)
    for module_id, module_name in module_list:
        re_pat2 = '{}-'.format(module_id)
        re_sub2 = '{}-'.format(module_name)
        configuration = re.sub(re_pat2, re_sub2, configuration)
    print(configuration)
    params_comi.write(configuration)
    conn.close()


def get_params_of_sg_id():
    conn, c = get_db_connection()
    paramdict = dict()
    c.execute('SELECT * FROM SG_Params WHERE SG_ID = ?', (sg_id,))
    paramlist_of_id = c.fetchall()
    for _, paramname, paramvalue in paramlist_of_id:
        try:
            paramdict[paramname] = float(paramvalue)
        except ValueError:
            paramdict[paramname.lower()] = paramvalue

    c.execute('SELECT * FROM SG_Files WHERE SG_ID = ?', (sg_id,))
    paramfiles_of_id = c.fetchall()
    for _, paramname, paramchecksum, _ in paramfiles_of_id:
        paramdict[paramname.lower()] = paramchecksum
    conn.close()
    return paramdict
```

```
69
70  def get_module_id_of_sg_id_list():
71      conn, c = get_db_connection()
72      c.execute('SELECT Configuration FROM SG WHERE SG_ID = ?', (sg_id,))
73      configuration = c.fetchone()[0]
74      module_id_pat = r'(\d+)'
75      module_id_list = re.findall(module_id_pat, configuration)
76      conn.close()
77      return module_id_list
78
79
80  def calculate_checksum_from_file(filename):
81      with open(filename, 'rb') as fb:
82          file_content = fb.read()
83      hasher = hashlib.md5()
84      hasher.update(file_content)
85      return hasher.hexdigest().upper()
86
87
88  def export_geo_parameter(paramname, paramtype, filetype, filename):
89      db_dir = r"C:\Users\320000863\Desktop\db_data\SG"
90      print('\t{} = {}'.format(paramname.upper(), paramdict[paramname]))
91      file_dir = '{}\\{}'.format(db_dir, paramtype)
92      file_in_db = filetype.format(file_dir, paramdict[paramname])
93      if os.path.isfile(file_in_db):
94          read_file = open(file_in_db, 'r')
95          content = read_file.read()
96          checksum_file = calculate_checksum_from_file(file_in_db)
97          export_type_file_dir = filename.format(export_file_dir, sg_id, paramname)
98          if checksum_file == paramdict[paramname]:
99              export_rot_file = open(export_type_file_dir, "w")
100             export_rot_file.write(content)
101             print("File exported.")
102         else:
103             export_file = open(export_file_dir, "w")
104             export_file.write(content)
105             print("Notification: Exported file has deviation! Please check the file content.")
106
```

```
107
108 def check_if_params_complete(paramdict):
109     conn, c = get_db_connection()
110     params_comi_dir = '{}\\SG_{}_params.comi'.format(export_file_dir, sg_id)
111     params_comi = open(params_comi_dir, "a")
112     module_id_list = get_module_id_of_sg_id_list()
113     occurrence_counter_dict = dict()
114     if module_id_list is not None:
115         c.execute('SELECT Module_ID FROM Module WHERE ModuleName = "Main"')
116         main_exists = c.fetchall()[0][0]
117         module_id_list.insert(0, str(main_exists))
118         for module_id in module_id_list:
119             c.execute('SELECT MultipleUse FROM Module WHERE Module_ID = ?', (module_id,))
120             module_exists = c.fetchall()
121             if not module_exists:
122                 print(module_id + " is not existing in the database anymore.")
123                 continue
124             module_multiple_use = module_exists[0][0]
125             if module_id in occurrence_counter_dict:
126                 occurrence_counter_dict[module_id] += 1
127             else:
128                 occurrence_counter_dict[module_id] = 1
129
130             c.execute('''SELECT * FROM Module_SG_Params
131                          WHERE Module_ID = ?''', (module_id,))
132             module_params = c.fetchall()
133
134             for _, param_name, param_type, param_optional, _ in module_params:
135                 param_name = param_name.lower()
136                 if module_multiple_use == 1:
137                     if param_type == 'Value':
138                         param_name = '{}{}'.format(param_name, occurrence_counter_dict[module_id])
139                     else:
140                         param_name = '{}_{}'.format(param_name, occurrence_counter_dict[module_id])
141
142                 if param_name in paramdict:
143                     if param_type == 'Value':
144                         print('\t{} = {}'.format(param_name.upper(), paramdict[param_name]))
145                         params_comi.write('\n{} = {}'.format(param_name.upper(), paramdict[param_name]))
146
147                     if param_type == 'RotSolid':
148                         export_geo_parameter(param_name, 'rotsolid', '{}\\{}.txt', '{}\\SG_{}_{}_rot.txt')
149
150                     if param_type == 'Cutplane':
151                         export_geo_parameter(param_name, 'cutplane', '{}\\{}.txt', '{}\\SG_{}_{}.txt')
152
153                     if param_type == 'OPC':
154                         export_geo_parameter(param_name, 'opc', '{}\\{}.opc', '{}\\SG_{}_{}.opc')
155
156                     if param_type == 'Blending':
157                         export_geo_parameter(param_name, 'blending', '{}\\{}.txt', '{}\\SG_{}_{}.txt')
158
159                     if param_type == 'meand_seggeo':
160                         export_geo_parameter(param_name, 'meand_seggeo', '{}\\{}.txt', '{}\\SG_{}_{}.txt')
161
162                     if param_type == 'block_seggeo':
163                         export_geo_parameter(param_name, 'block_seggeo', '{}\\{}.txt', '{}\\SG_{}_{}.txt')
164                 else:
165                     if not param_optional:
166                         print('Not found: {}'.format(param_name))
167     conn.close()
```

```
169
170 ▶ if __name__ == '__main__':
171       if check_if_sg_exist() == 1:
172           if check_if_config_exist() is not None:
173               configuration = check_if_config_exist()
174               print(configuration)
175               get_config_string(configuration)
176               paramdict = get_params_of_sg_id()
177               check_if_params_complete(paramdict)
178           elif check_if_config_exist() is None:
179               print("SG_ID " + str(sg_id) + " does not contain a configuration!")
180       elif check_if_sg_exist() == 0:
181           print("SG_ID " + str(sg_id) + " does not exist!")
182       else:
183           print("SG_ID " + str(sg_id) + " fails unique constraint!")
```

Anhang 8: Python-Code für den Ergebnis-Import

calc_results_import_to_db.py

```python
import sqlite3
import re
import hashlib
import shutil
from os.path import isfile
import os
import glob

sc_id = 66666
results_files_dir = r"C:\Users\320000863\Desktop\example\results"


def get_db_connection():
    conn = sqlite3.connect(r"C:\Users\320000863\Desktop\CalcDB.db", timeout=10)
    return conn, conn.cursor()


def get_paramname_from_filename(pattern, directory, extension, dictionary):
    os.chdir(directory)
    for file_name in glob.glob(extension):
        files = re.findall(pattern, file_name)

        for param_name in files:
            dictionary[param_name.lower()] = 0


def get_paramname_from_txt_file():
    results_dict = dict()

    with open(txt_file_dir, 'r') as txt_file:
        txt_file_content = txt_file.read()
    results_value_list = re.findall(r'#(\w+)\s+(\S+)', txt_file_content,
                                    flags=(re.IGNORECASE | re.MULTILINE))
    for param_name, param_value in results_value_list:
        try:
            results_dict[param_name.lower()] = float(param_value)
        except ValueError:
            results_dict[param_name.lower()] = param_value

    get_paramname_from_filename(r'^\s*\SC_\d+_(fs_image).bmp', results_files_dir, "*.bmp", results_dict)
    return results_dict


def calculate_checksum_from_file(filename):
    with open(filename, 'rb') as fb:
        file_content = fb.read()
    hasher = hashlib.md5()
    hasher.update(file_content)
    return hasher.hexdigest().upper()
```

```
def results_paramtypes(paramname, typename, filetype, import_textfile_dir):
    conn, c = get_db_connection()
    expected_import_file = typename.format(results_files_dir, 'SC', sc_id, paramname)
    # check if file exist
    if isfile(expected_import_file):
        # check if param_name is in filename
        if str.find(expected_import_file, paramname):
            import_file_checksum = calculate_checksum_from_file(expected_import_file)
            import_file_size = os.path.getsize(expected_import_file)
            # get checksums from db
            c.execute('SELECT FileCheckSum FROM Results_Picture WHERE FileCheckSum = "{}"'.format(
                import_file_checksum))
            conn.commit()
            checksums_in_db_list = c.fetchall()
            file_in_db_folder = filetype.format(import_textfile_dir,
                                                import_file_checksum)
            if len(checksums_in_db_list) > 0:
                # checksum which are already in db
                db_checksum = calculate_checksum_from_file(file_in_db_folder)
                if import_file_checksum != db_checksum:
                    print('WARNING ' + 'for ' + import_file_checksum +
                          ': File content has changed! File will be replaced!')
                    shutil.copy(expected_import_file, file_in_db_folder)
            else:
                # checksum of import file is not in db yet
                shutil.copy(expected_import_file, file_in_db_folder)

            c.execute('SELECT SC_ID, Results_ParamName FROM Results_Picture')
            sg_files_db = c.fetchall()
            sg_params_list = [sc_id, paramname]
            if tuple(sg_params_list) in sg_files_db:
                c.execute('''UPDATE Results_Picture SET FileCheckSum = ?, Filesize = ?
                          WHERE SC_ID = ? AND Results_ParamName = ?''',
                          (import_file_checksum, import_file_size, sc_id, paramname))
                conn.commit()
                print("\t" + paramname + " = " + import_file_checksum + " (updated)")
            else:
                c.execute('''INSERT INTO Results_Picture(SC_ID, Results_ParamName, FileCheckSum, FileSize)
                          VALUES(?,?,?,?)''',
                          (sc_id, paramname, import_file_checksum, import_file_size))
                conn.commit()
                print("\t" + paramname + " = " + import_file_checksum)
    else:
        print("ERROR: " + expected_import_file + " does not exist!")
    conn.close()


def import_results(results_dict):
    conn, c = get_db_connection()
    c.execute('SELECT * FROM Results_Params')
    params_exist = c.fetchall()
    for param_name, param_type, _ in params_exist:
        param_name = param_name.lower()
        if param_name in results_dict:
            if param_type == 'Value':
                print('\t{} = []'.format(param_name, results_dict[param_name]))
                c.execute('SELECT SC_ID, Results_ParamName FROM Results_Value')
                results_params_db = c.fetchall()
                results_params_list = [sc_id, param_name]
                if tuple(results_params_list) in results_params_db:
                    c.execute('UPDATE Results_Value SET ParamValue = ? WHERE SC_ID = ? AND Results_ParamName = ?',
                              (results_dict[param_name], sc_id, param_name))
                    conn.commit()
                else:
                    c.execute('INSERT INTO Results_Value(SC_ID, Results_ParamName, ParamValue) VALUES(?,?,?)',
                              (sc_id, param_name, results_dict[param_name]))
                    conn.commit()

            if param_type == 'Picture':
                import_picture_file_dir = r'C:\Users\320000863\Desktop\db_data\results'
                results_paramtypes(param_name,
                                   '{}\\{}_{:05d}_{}.bmp',
                                   '{}\\{}.bmp',
                                   import_picture_file_dir)
        else:
            print('Not found: {}'.format(param_name))


if __name__ == '__main__':
    txt_file_dir = '{}\\SC_{}_result_values.txt'.format(results_files_dir, sc_id)
    results_dict = get_paramname_from_txt_file()
    import_results(results_dict)
```