Institut für Kontinuumsmechanik

Leibniz Universität Hannover

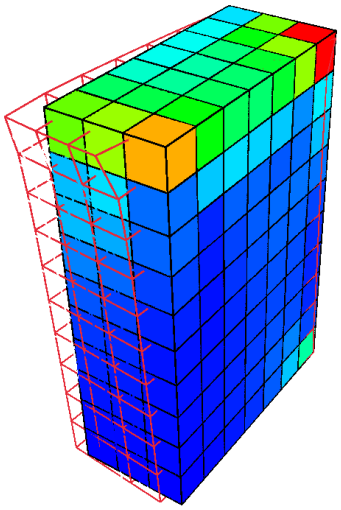
Leibniz Universität Hannover

Fakultät für Maschinenbau

Institut für Kontinuumsmechanik

Studentische Arbeit zum Thema

Programmierung einer Finiten Elemente Software für Unsicherheitsberechnungen mit Benutzeroberfläche in C++



Tim Rother

Betreuer:	Dr. Hendrik Geisler
1. Prüfer:	Dr. Dustin R. Jantos
2. Prüfer:	-

14. Juli 2025

Angefertigt im Studiengang Maschinenbau am Institut für Kontinuumsmechanik der Leibniz Universität Hannover.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die am 14. Juli 2025 eingereichte studentische Arbeit zum Thema „Programmierung einer Finiten Elemente Software für Unsicherheitsberechnungen mit Benutzeroberfläche in C++" unter Betreuung von Dr. Hendrik Geisler selbstständig erarbeitet, verfasst und Zitate kenntlich gemacht habe, dass ich keine anderen als die angegebenen Hilfsmittel verwendet habe, und dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Hannover, 14. Juli 2025 _____

Tim Rother

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Dateiauszugsverzeichnis

Algorithmenverzeichnis

1 Einleitung

In der numerischen Mechanik haben sich FEM-Softwarelösungen wie Abaqus, ANSYS oder Code-Aster etabliert. Diese Programme sind umfangreich, effizient und bieten eine Vielzahl an Funktionen für verschiedenste Anwendungsbereiche. Ihre Implementierungen sind häufig schwergewichtig und erfordern teilweise aufwendige Modellierungen, tiefgreifende Einarbeitung und komplexe Schnittstellenanbindung.
Für einfache Voruntersuchungen und Erprobungen von Ansätzen für die Unsicherheitsquantifizierung ist dieser Aufwand oft eine Hürde. Deshalb soll im Rahmen dieser Arbeit ein leichtgewichtiges FEM-Programm entwickelt werden, das diese Voruntersuchungen und Tests ohne großen Modellierungs- und Einarbeitungsaufwand ermöglicht und damit eine Zeitersparnis bietet. Die zentralen Features sollen das Erproben verschiedener Wahrscheinlichkeitsdichtefunktionen für die Unsicherheitsquantifizierung und simpler, nicht-linearer Materialmodelle sein. Dabei sollen dynamische Adaptierbarkeit, Wiederverwendbarkeit und eine möglichst selbsterklärende Anwendung gewährleistet werden.
Ziel der Arbeit ist die Bereitstellung eines entsprechenden Werkzeugs, das eine potentielle, experimentelle Vorstufe für besagte Modelle vor der finalen Implementierung mit gängigen Softwarelösungen darstellen soll.

2 Stand der Technik

2.1 Finite Elemente Methode

Die Finite Elemente Methode (FEM) ist ein numerisches Verfahren zur näherungsweisen Lösung partieller Differentialgleichungen. Anwendungsgebiete sind die Kontinuumsmechanik, Strömungsmechanik und Wärmeübertragung. Bei der FEM wird das betrachtete Kontinuum bzw. Simulationsgebiet in eine endliche Anzahl von Teilgebieten, den sogenannten Finiten Elementen unterteilt. Diese Finiten Elemente ermöglichen die Approximation der Lösung über einen Galerkin-Ansatz. Dabei wird die Lösung der partiellen Differentgleichung als gewichtete Summe von Knotenwerten approximiert. Ein umfangreicher Überblick über die Anwendung des Verfahrens innerhalb der Kontinuumsmechanik wurde geboten in [1].

2.1.1 Impulsbilanz

Die zentrale partielle Differentialgleichung für die Untersuchung der Deformation von Körpern innerhalb der Kontinuumsmechanik ist die Impulsbilanz

$$\nabla \cdot \underline{\sigma} + \underline{b} = \underline{0}.$$

Dabei beschreibt der Vektor $\underline{b}$ die auf den Körper einwirkenden Volumenkräfte und der Tensor $\underline{\sigma}$ den sich im betrachteten Kontinuum ausbildenen Spannungszustand. Um diese Differentialgleichung mittels der FEM numerisch lösen zu können, ist es notwendig, sie in die schwache bzw. variationelle Form zu überführen. Dazu wird sie mit einer Testfunktion, hier der virtuellen Arbeit $\delta\underline{u}$ multipliziert und über das Volumen des Kontinuums integriert

$$\int_\Omega (\nabla \cdot \underline{\sigma} + \underline{b}) \cdot \delta\underline{\vec{u}}\, dV = 0.$$

Durch Anwendung des Gaußschen Integralsatzes auf den Spannungsterm $\nabla \cdot \sigma$ ergeben sich zusätzlich zu den Volumenkräften Beiträge durch Oberflächenkräfte, die über die Oberfläche $\partial\Omega$

auf das Kontinuum einwirken. Das Produkt $\nabla\delta\underline{u}$ ist der Gradient der virtuellen Verschiebung und entspricht der virtuellen Dehnung $\delta\underline{\varepsilon}$. Mit diesen Umformungen kann das Prinzip der virtuellen Arbeit als

$$\int_\Omega \underline{\sigma} \cdot \delta\underline{\varepsilon}\, dV = \underbrace{\int_\Omega \underline{f} \cdot \delta\underline{u}\, dV}_{\text{Volumenkräfte}} + \underbrace{\int_{\partial\Omega} \underline{t} \cdot \delta\underline{u}\, dA}_{\text{Oberflächenkräfte}}$$

formuliert werden. Die Überführung der Impulsbilanz in ihre schwache Form bzw. das Prinzip der virtuellen Arbeit wurde beschrieben in [1, 2].

2.1.2 Diskretisierung

Zur numerischen Approximation des Prinzips der virtuellen Arbeit wird das betrachtete Kontinuum in Teilbereiche unterteilt, die Finiten Elemente Ω_e. An den Knotenpunkten des finiten Elements wird eine Diskretisierung auf Basis des Galerkin-Verfahrens vorgenommen. Das Galerkin-Verfahren nähert die Lösung einer Differentialgleichung als gewichtete Summe unbekannter Knotenwerte. Die Knotenwerte entsprechen dabei dem Wert der Differentialgleichung an den entsprechenden Knotenpunkten. Die Interpolation erfolgt über sogenannte Ansatzfunktionen. Die Anwendung des Galerkin-Verfahrens bei der Diskretisierung und die Wahl der Ansatzfunktionen wurden im Detail erklärt in [1, 2]. Mit dem Galerkin-Ansatz ergibt sich die approximierte Verschiebung als

$$u(\underline{x}) \approx \sum_i N_i(\underline{x}) \cdot \hat{u}_i \qquad \forall \underline{x} \in \Omega_e.$$

Dabei müssen die Ansatzfunktionen für den Knoten, dessen Wert sie gewichten, den Wert 1 und für alle anderen Knoten im Element den Wert 0 annehmen. Somit nimmt die gewichtete Summe für den Knotenort den jeweiligen Knotenwert und für die Bereiche zwischen den Knotenorten eine Summe der Knotenwerte unter Berücksichtigung des Abstands an. Damit ergibt sich die Vorschrift

$$\text{mit} \quad N_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \qquad \forall\, N_i,\, x_j \in \Omega_e.$$

Damit lässt sich der Galerkin-Ansatz der Verschiebung in 2D als der Tensorausdruck

$$\underline{u}\,(\underline{x}) \approx \begin{bmatrix} \sum_i N_i(\underline{x}) \cdot \underline{\hat{u}}_{xi} \\ \sum_i N_i(\underline{x}) \cdot \underline{\hat{u}}_{yi} \end{bmatrix} = \underbrace{\begin{bmatrix} N_1(\underline{x}) & 0 & \cdots & N_n(\underline{x}) & 0 \\ 0 & N_1(\underline{x}) & \cdots & 0 & N_n(\underline{x}) \end{bmatrix}}_{\underline{N}(\underline{x})} \cdot \underbrace{\begin{bmatrix} \hat{u}_{x1} \\ \hat{u}_{y1} \\ \vdots \\ \hat{u}_{xn} \\ \hat{u}_{yn} \end{bmatrix}}_{\underline{\hat{u}}} \qquad \forall \underline{x} \in \Omega_e$$

schreiben. Der Ansatz lässt sich analog und mit den gleichen Ansatzfunktionen für weitere Größen wie die virtuelle Verschiebung aufstellen als

$$\underline{u}(\underline{x}) \approx \underline{N}(\underline{x}) \cdot \underline{\hat{u}} \qquad \delta\underline{u}(\underline{x}) \approx \underline{N}(\underline{x}) \cdot \delta\underline{\hat{\boldsymbol{u}}} \qquad \forall \underline{x} \in \Omega_e.$$

Der Galerkin-Ansatz für die Dehnung wird über den symmetrischen Gradienten der Verschiebung aufgestellt. Dabei wird das Produkt aus symmetrischem Gradienten ∇_{sym} und dem Tensor der Ansatzfunktionen $\underline{N}(\underline{x})$ als Gradientenmatrix $\underline{B}$ definiert. Diese ergibt sich aus dem Ausdruck

$$\underline{\varepsilon}(\underline{x}) = \nabla_{sym}\underline{u}(\underline{x}) \approx \underbrace{\nabla_{sym}\underline{N}(\underline{x})}_{\underline{B}(\underline{x})}\underline{\hat{u}}.$$

Die B-Matrix $\underline{B}$ ist die Dehnungs-Verschiebungsbeziehung für das diskretisierte Problem. Damit ist die Gradientenmatrix für ein exemplarisches, zweidimensionales Element mit n Knoten gegeben als

$$\underline{\varepsilon}(\underline{x}) = \begin{bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\gamma_{xy} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \cdots & \frac{\partial N_n}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & \cdots & 0 & \frac{\partial N_n}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \cdots & \frac{\partial N_n}{\partial y} & \frac{\partial N_n}{\partial x} \end{bmatrix}}_{\underline{B}(\underline{x})} \cdot \underbrace{\begin{bmatrix} \hat{u}_{x1} \\ \hat{u}_{y1} \\ \vdots \\ \hat{u}_{xn} \\ \hat{u}_{yn} \end{bmatrix}}_{\underline{\hat{u}}_e} \qquad \forall \underline{x} \in \Omega_e.$$

Die Galerkin-Ansätze für Dehnung und virtuelle Dehnung sind damit

$$\underline{\varepsilon} \approx \underline{B} \cdot \underline{\hat{u}} \qquad \delta\underline{\varepsilon} \approx \underline{B} \cdot \delta\underline{\hat{u}}.$$

Über das Einsetzen des Hooke'schen Gesetzes $\underline{\sigma} = \underline{\mathbb{E}} \cdot \underline{\varepsilon}$ und der Galerkin-Ansätze wird die diskretisierte Form des Prinzips der virtuellen Arbeit als

$$\underbrace{\int_\Omega \underline{B}^T \cdot \underline{\mathbb{E}} \cdot \underline{B}\, dV}_{\underline{K}} \cdot \underline{\hat{u}} = \underbrace{\int_\Omega \underline{f} \cdot \underline{N}\, dV + \int_{\partial\Omega} \underline{t} \cdot \underline{N}\, dA}_{\underline{f}}$$

formuliert.

2.1.3 Isoparametrische Elemente

Um das Aufstellen der Ansatzfunktionen und die Volumenintegration für verschieden geformte Elemente zu vereinheitlichen, werden sogenannte isoparametrische Elemente verwendet. Dabei handelt es sich um Referenzelemente, auf denen die Ansatzfunktionen und die numerische Integration definiert sind. Die Definitionen erfolgen in einem lokalen Koordinatensystem mit der Basis $\underline{r}$. Das Konzept und die Anwendung der isoparametrischen Elemente wurden in [1, 2] beschrieben. Über eine geometrische Transformation der lokalen Koordinaten ins globale Koordinatensystem lassen sich die physischen Koordinaten $\underline{x}$ in Abhängigkeit der lokalen Koordinaten $\underline{r}$ darstellen. Diese Abbildung erfolgt mithilfe der für das isoparametrische Element definierten Ansatzfunktionen $\underline{N}(\underline{r})$ als

$$\underline{x}(\underline{r}) = \underline{N}(\underline{r}) \cdot \underline{x}.$$

Die Transformation erlaubt die Übertragung aller Integrations- und Ableitungsvorgänge auf die Referenzgeometrie. Der Zusammenhang zwischen lokalen und globalen Ableitungen wird durch die Jacobi-Matrix beschrieben. Diese ergibt sich zu

$$\underline{J}(\underline{r}) = \frac{\partial \underline{x}}{\partial \underline{r}} = \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \cdots \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

Die Jacobi-Matrix lässt sich für ein Element des diskretisierten Systems aufstellen als

$$\underline{J}(\underline{r}) = \frac{\partial \underline{x}}{\partial \underline{r}} \approx \frac{\partial \underline{N}(\underline{r}) \cdot \hat{\underline{x}}}{\partial \underline{r}} = \begin{bmatrix} \sum_{i=1}^{n} \frac{\partial N_i}{\partial r} x_i & \sum_{i=1}^{n} \frac{\partial N_i}{\partial r} y_i & \cdots \\ \sum_{i=1}^{n} \frac{\partial N_i}{\partial s} x_i & \sum_{i=1}^{n} \frac{\partial N_i}{\partial s} y_i & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}.$$

Über die Inverse der Jacobi-Matrix lassen sich die Ableitungen der Ansatzfunktionen nach den lokalen Koordinaten in das globale Koordinatensystem transformieren mit

$$\frac{\partial \underline{N}}{\partial \underline{x}} = \frac{\partial \underline{N}}{\partial \underline{r}} \cdot \frac{\partial \underline{r}}{\partial \underline{x}} = \frac{\partial \underline{N}}{\partial \underline{r}} \cdot \underline{J}^{-1}.$$

Basierend auf dem isoparametrischen Ansatz kann die Elementsteifigkeitsmatrix eines Elements im physikalischen Raum bestimmt werden als

$$\underline{K}_e = \int_{\Omega_e} \underline{B}^T \cdot \mathbb{E} \cdot \underline{B} \, dV.$$

Das Volumenelement im globalen Koordinatensystem ergibt sich aus dem lokalen Volumenelement dv durch Multiplikation mit der Determinante der Jacobi-Matrix zu

$$dV = \det(\underline{J}(\underline{r})) \cdot dv.$$

Damit kann die K-Matrix bestimmt werden als

$$\underline{K}_e = \int_{\Omega} \underline{B}^T(\underline{r}) \cdot \mathbb{E} \cdot \underline{B}(\underline{r}) \cdot \det \underline{J}(\underline{r}) \, dv.$$

Im Rahmen der numerischen Integration wird das Integral über eine gewichtete Summe approximiert. Dabei wird der Ausdruck an den Quadraturpunkten des isoparametrischen Elements ausgewertet und über den entsprechenden Wichtungsfaktor summiert. Die Summe der Gewichte muss dabei dem Flächeninhalt bzw. Volumen des isoparametrischen Elements entsprechen. Die

Elementsteifigkeitsmatrix lässt sich als der Tensor

$$\underline{K}_e \approx \sum_{i=1}^{n_{\mathrm{QP}}} \underline{B}^T(\underline{r}_i) \cdot \mathbb{E} \cdot \underline{B}(\underline{r}_i) \cdot w_i \cdot \det(\underline{J}(\underline{r}_i))$$

schreiben.

2.1.4 Assemblierung

Um das globale Finite Elemente Problem zu lösen, müssen die Elementsteifigkeitsmatrizen in eine globale Steifigkeitsmatrix assembliert werden. Das bedeutet, dass die Einträge über die Transformation der lokalen in die globalen Freiheitsgrade an der richtigen Stelle in die globale K-Matrix summiert werden. Damit ist der ij-te Eintrag der Systemsteifigkeitsmatrix die Summe der Wechselwirkungen des globalen i-ten und j-ten Freiheitsgrades, die aus den Elementsteifigkeitsmatrizen hervorgehen. Die Vorgehensweise zur Assemblierung der globalen Steifigkeitsmatrix ist detailliert dargestellt worden in [1, 2].
Die Assemblierung soll im Folgenden anhand eines Beispiels verdeutlicht werden. Das betrachtete zweidimensionale System besteht aus zwei Viereckselementen und sechs Freiheitsgraden. Die Zuordnung der lokalen auf die globalen Freiheitsgrade ist gegeben durch

$$\underline{u}_{\mathrm{system}} = [u_1, u_2, u_3, u_4, u_5, u_6]^T \qquad \underline{u}^{(1)} = [u_1, u_2, u_5, u_4]^T \qquad \underline{u}^{(2)} = [u_2, u_3, u_6, u_5]^T.$$

Die Elementsteifigkeitsmatrizen der beiden Elemente sind bekannt und hier exemplarisch mit $2x2$ Submatrizen als Einträgen dargestellt mit

$$\underline{K}^{(i)} = \begin{bmatrix} K_{11}^{(i)} & K_{12}^{(i)} & K_{13}^{(i)} & K_{14}^{(i)} \\ K_{21}^{(i)} & K_{22}^{(i)} & K_{23}^{(i)} & K_{24}^{(i)} \\ K_{31}^{(i)} & K_{32}^{(i)} & K_{33}^{(i)} & K_{34}^{(i)} \\ K_{41}^{(i)} & K_{42}^{(i)} & K_{43}^{(i)} & K_{44}^{(i)} \end{bmatrix} \quad | \quad \underline{K}_{jk}^{(i)} \in \mathbb{R}^{2\times 2}.$$

Die Assemblierung geschieht nun über Tansformation der Matrizen auf die globalen Freiheitsgrade und die Summation über die Elemente. Dabei kann die Transformation auch über Transformations- bzw. Sortiermatrizen realisiert werden. Damit wird die globale Steifigkeitsmatrix als

$$K_{\mathrm{gesamt}} = \underbrace{\begin{bmatrix} K_{11}^{(1)} & K_{12}^{(1)} & 0 & K_{14}^{(1)} & K_{13}^{(1)} & 0 \\ K_{21}^{(1)} & K_{22}^{(1)} & 0 & K_{24}^{(1)} & K_{23}^{(2)} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ K_{41}^{(1)} & K_{42}^{(1)} & 0 & K_{44}^{(1)} & K_{43}^{(1)} & 0 \\ K_{31}^{(1)} & K_{32}^{(1)} & 0 & K_{34}^{(1)} & K_{33}^{(1)} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\underline{T}_1^T \underline{K}^{(1)} \underline{T}_1} + \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & K_{11}^{(2)} & K_{12}^{(2)} & 0 & K_{14}^{(2)} & K_{13}^{(2)} \\ 0 & K_{21}^{(2)} & K_{22}^{(2)} & 0 & K_{24}^{(2)} & K_{23}^{(2)} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & K_{41}^{(2)} & K_{42}^{(2)} & 0 & K_{44}^{(2)} & K_{43}^{(2)} \\ 0 & K_{31}^{(2)} & K_{32}^{(2)} & 0 & K_{34}^{(2)} & K_{33}^{(2)} \end{bmatrix}}_{\underline{T}_2^T \underline{K}^{(2)} \underline{T}_2}$$

formuliert.

2.1.5 Randbedingungen

Durch die assemblierte K-Matrix des Gesamtsystems wird dessen Steifigkeit unabhängig von äußeren Einflüssen beschrieben. Die Randbedingungen werden bei der Aufstellung des globa-

len Gleichungssystems $\underline{K} \cdot \underline{\hat{u}} = \underline{f}$ berücksichtigt. Das Vorgehen zum Einbeziehen verschiedener Randbedingungen in das Gleichungssystem wurde im Detail aufgezeigt in [1].
Mit gegebenem Knotenkraftvektor und Freiheitsgradfixierungen durch z.B. Lagerung kann das Gleichungssystem mit dem gegebenen Knotenkraftvektor erstellt und um die fixierten Freiheitsgrade reduziert werden. Da eine Verschiebung an diesen Freiheitsgraden ausgeschlossen werden kann, werden die jeweiligen Zeilen und Spalten vollständig aus dem Gleichungssystem gestrichen, bevor es gelöst wird.
Sind zudem auch Verschiebungen für Freiheitsgrade bekannt, die ungleich Null sind, muss das Gleichungssystem vor seiner Lösung umgestellt werden. Die Freiheitsgrade mit bekannten Verschiebungen ungleich Null werden in der K-Matrix in die ersten Zeilen sortiert. Damit ergibt sich ein Gleichungssystem der Form

$$\begin{bmatrix} \underline{K}_{uu} & \underline{K}_{u\bar{u}} \\ \underline{K}_{\bar{u}u} & \underline{K}_{\bar{u}\bar{u}} \end{bmatrix} \cdot \begin{bmatrix} \underline{u} \\ \underline{\bar{u}} \end{bmatrix} = \begin{bmatrix} \underline{\bar{f}} \\ \underline{f} \end{bmatrix}.$$

$\underline{u}$ und $\underline{f}$ sind die unbekannten Verschiebungen und Lasten und $\underline{\bar{u}}$ und $\underline{\bar{f}}$ die bekannten Verschiebungen und Lasten.
Durch die Umstellung ergibt sich das lösbare Gleichungssystem

$$\underline{K}_{uu}\underline{u} = \underline{\bar{f}} - \underline{K}_{u\bar{u}}\underline{\bar{u}}.$$

Die unbekannten Kräfte können anschließend über

$$\underline{f} = \underline{K}_{\bar{u}u}\underline{u} + \underline{K}_{\bar{u}\bar{u}}\underline{\bar{u}}$$

bestimmt werden.

2.1.6 Lösung globales Gleichungssystem

Für die effiziente Lösung des aufgestellten Gleichungssystems wird ein LGS Solver ausgewählt. Bei der Auswahl eines Solvers spielen unter anderem die folgenden Aspekte eine Rolle.

- Linearität des Gleichungssystems
- Anzahl der Freiheitsgrade
- Besetzung der Matrizen

Im Rahmen dieser Arbeit ist für die lineare FEM aufgrund der Linearität der Gleichungssysteme, der dünn besetzten K-Matrizen und der verhältnismäßig geringen Anzahl an Freiheitsgraden der Cholesky-Solver verwendet worden. Die Verwendung verschiedener Solver in der FEM und eine Übersicht über deren Anwendungsgebiete und -einschränkungen wurde aufgeführt in [1].

2.1.7 Unsicherheitsquantifizierung

Ziel der Unsicherheitsquantifizierung ist die Berechnung der linearen FEM für ein Material mit zufallsverteilten Materialkennwerten. Da der Elastizitätstensor von diesen abhängig ist, wird er stellvertretend über die zufallsverteilte Variable ξ variiert. Die Verteilung von ξ wird über eine Wahrscheinlichkeitsdichtefunktion beschrieben. Damit ergibt sich die Verteilung des Tensors als

$$\underline{\mathbb{E}}(\xi) = \underline{\mathbb{E}}_0 \cdot (1 + \xi) \;\mid\; \underline{\mathbb{E}}_0 = \underline{\mathbb{E}}(E, \nu).$$

Die Steifigkeitsmatrix des Systems in Abhängigkeit von ξ ist damit

$$\underline{K}(\xi) = \int_\Omega \underline{B}^T \cdot \underline{\mathbb{E}}_0 \cdot (1+\xi) \cdot \underline{B} d\Omega = (1+\xi) \cdot \underbrace{\int_\Omega \underline{B}^T \cdot \underline{\mathbb{E}}_0 \cdot \underline{B} d\Omega}_{\underline{K}_0}.$$

Das verteilte Gleichungssystem ergibt sich als

$$(1+\xi) \cdot \underline{K}_0 \cdot \underline{u} = \underline{f} \;\mid\; \underline{u} = \underline{u}(\xi).$$

Der von der zufallsverteilten Variable abhängige Verschiebungsvektor $\underline{u}(\xi)$ wird über eine Taylorreihendarstellung genähert mit

$$\underline{u}(\xi) = \sum_{n=0}^{\infty} \frac{\underline{u}^{(n)}(\xi_0)}{n!} (\xi - \xi_0)^n.$$

Die Nutzung von Taylorreihen im Bereich der Unsicherheitsanalyse wurde im Detail diskutiert in [3, 4]. Für den weiteren Verlauf wird die Annahme getroffen, dass sich die Verschiebung $\underline{u}(\xi)$ über ξ für kleine Abweichungen um den Arbeitspunkt linear verhält. Folglich kann die Verschiebung als Taylorreihe mit zwei Reihengliedern beschrieben werden. Als Stützstelle wird $\xi_0 = 0$ gewählt. Somit vereinfacht sich der gewählte Ansatz zu

$$\underline{u}(\xi) = \underline{u}^{(0)}(\xi_0) + \xi \cdot \underline{u}^{(1)}(\xi_0).$$

Eingesetzt ins Geichungssystem ergibt sich

$$(1+\xi) \cdot \underline{K}_0 \cdot (\underline{u}^{(0)} + \xi \cdot \underline{u}^{(1)}) = \underline{f}.$$

Nach dem Ausmultiplizieren der linken Seite ist

$$\underline{K}_0 \cdot \underline{u}^{(0)} + \xi \cdot (\underline{K}_0 \cdot \underline{u}^{(0)} + \underline{K}_0 \cdot \underline{u}^{(1)}) + \xi^2 \cdot \underline{K}_0 \cdot \underline{u}^{(1)} = \underline{f}.$$

Wird der Term partiell nach ξ abgeleitet und die Stützstelle ξ_0 eingesetzt, ergibt sich der Zusammenhang

$$\underline{K}_0 \cdot \underline{u}^{(0)} + \underline{K}_0 \cdot \underline{u}^{(1)} = 0 \Leftrightarrow \underline{u}^{(1)} = -\underline{u}^{(0)}.$$

$\underline{u}(\xi)$ kann damit bestimmt werden als

$$\underline{u}(\xi) = \underline{u}^{(0)} \cdot (1 - \xi).$$

Analog dazu werden Spannung und Dehnung in Abhängigkeit der verteilten Variable ermittelt als

$$\underline{\varepsilon}(\xi) = (1-\xi) \int_{\Omega_e} \underline{B} \cdot \underline{u}_0 dV = (1-\xi) \cdot \underline{\varepsilon}_0$$

$$\underline{\sigma}(\xi) = (1-\xi^2) \int_{\Omega_e} \underline{\mathbb{E}}_0 \cdot \underline{B} \cdot \underline{u}_0 dV = (1-\xi^2) \cdot \underline{\sigma}_0.$$

Da die über ξ variierten Verschiebungen, Spannungen und Dehnungen von denen an der Stützstelle

$\xi_0 = 0$ abhängig sind, kann die Berechnung dieser Größen im Anschluss an die lineare FEM erfolgen. Das geschieht über das Multiplizieren der ermittelten Größen mit den Koeffizienten $(1 - \xi)$ und $(1 - \xi^2)$.

2.1.8 Nicht-lineares Materialverhalten

Für die lineare FEM wird das linear-elastische Materialmodell nach Hooke verwendet mit $\underline{\sigma} = \underline{\underline{\mathbb{E}}} \cdot \underline{\varepsilon}$. Das Modell nimmt dabei sowohl die Proportionalität von Spannung und Dehnung als auch reversible, konservative Deformationen an. Bei der Belastung einiger Materialien kommt es allerdings zu Nicht-Linearitäten und Energiedissipation. Beispiele hierfür sind Plastizität, Kriechvorgänge und Schädigung. Für die numerische Beschreibung dieser Effekte müssen alternative Materialmodelle gewählt werden. Die Simulation solcher Modelle und mehrere Anwendungsbeispiele wurden beschrieben in [1].
Um die FEM mit beliebigen Spannungs-Dehnungsbeziehungen ausführen zu können, wird das diskretisierte Prinzip der virtuellen Arbeit angepasst mit

$$\underline{\underline{K}} \cdot \underline{u} = \int_\Omega \underline{\underline{B}}^T \cdot \underline{\underline{\mathbb{E}}} \cdot \underline{\underline{B}} \cdot \underline{u} \, d\Omega = \underline{f}.$$

$\underline{\underline{B}} \cdot \underline{u}$ wird als Dehnung $\underline{\varepsilon}$ und $\underline{\underline{\mathbb{E}}} \cdot \underline{\varepsilon}$ als Spannung $\underline{\sigma}$ zusammengefasst. Damit ergibt sich

$$\int_\Omega \underline{\underline{B}}^T \, \underline{\underline{\mathbb{E}}} \, \underline{\underline{B}} \cdot \underline{u} \, d\Omega = \int_\Omega \underline{\underline{B}}^T \underline{\underline{\mathbb{E}}} \cdot \underline{\varepsilon} \, d\Omega = \int_\Omega \underline{\underline{B}}^T \cdot \underline{\sigma} d\Omega = \underline{f}.$$

Das Residuum $\underline{R}$ wird formuliert als

$$\underline{R} = \int_\Omega \underline{\underline{B}}^T \cdot \underline{\sigma} \, d\Omega - \underline{f} \stackrel{!}{=} \underline{0}.$$

In das Residuum können die Spannungsdefinitionen von Materialmodellen eingesetzt werden. Einige Materialeffekte sind zudem historienabhängig. Deshalb werden Zeitschrittsimulationen mit inneren Variablen durchgeführt. Die inneren Variablen sind Parameter, die den Materialzustand charakterisieren und für jeden Zeitschritt in Abhängigkeit ihrer eigenen Historie und des Spannungszustands neu berechnet werden.
Ein Beispiel für ein solches Modell ist das viskoelastische Maxwell-Element, bestehend aus einer linearen Feder (Elastizitätsmodul E) und einem Dämpfer (Viskosität η) in Reihe. Dieses Modell wurde zusammen mit anderen viskoelastischen Materialmodellen beleuchtet und diskutiert in [5]. Die Gesamtdehnung wird als Summe der linear-elastischen Dehnung $\underline{\varepsilon}^{\mathrm{e}}$ und einer viskosen Dehnung $\underline{\varepsilon}^{\mathrm{v}}$ beschrieben. Die viskose Dehnung ist dabei die innere Variable

$$\underline{\varepsilon} = \underline{\varepsilon}^{\mathrm{e}} + \underline{\varepsilon}^{\mathrm{v}}.$$

Die zeitliche Entwicklung der viskosen Dehnung $\underline{\varepsilon}^{\mathrm{v}}$ ergibt sich mit dem Viskositätskoeffizienten η als

$$\dot{\underline{\varepsilon}}^{\mathrm{v}} = \frac{1}{\eta} \cdot \underline{\sigma}.$$

Für den ersten Zeitschritt der Simulation wird der Wert der viskosen Dehnung $\underline{\varepsilon}^{\mathrm{v}}$ auf $\underline{0}$ gesetzt. Für alle weiteren Zeitschritte wird die innere Variable mithilfe der Evolutionsgleichung

aktualisiert über

$$\underline{\varepsilon}^{\mathrm{v}}_{n+1} = \underline{\varepsilon}^{\mathrm{v}}_{n} + \Delta t \cdot \underline{\dot{\varepsilon}}^{\mathrm{v}}.$$

Der Zeitschritt, für den die Ableitung der inneren Variablen $\underline{\dot{\varepsilon}}^{\mathrm{v}}$ ausgewertet wird, hängt dabei vom verwendeten Simulationsverfahren ab. Das vollexplizite Euler-Verfahren wertet die Ableitung für den vorherigen Zeitschritt aus. Die aktualisierte Variable ergibt sich als

$$\underline{\varepsilon}^{\mathrm{v}}_{n+1} = \underline{\varepsilon}^{\mathrm{v}}_{n} + \Delta t \cdot \underline{\dot{\varepsilon}}^{\mathrm{v}}_{n}.$$

Das implizite Euler-Verfahren nutzt die Auswertung der Ableitung für den aktuellen Zeitschritt. Damit ergibt sich die innere Variable für den aktuellen Zeitschritt mit

$$\underline{\varepsilon}^{\mathrm{v}}_{n+1} = \underline{\varepsilon}^{\mathrm{v}}_{n} + \Delta t \cdot \underline{\dot{\varepsilon}}^{\mathrm{v}}_{n+1}.$$

Nach der Weiterführung der inneren Variablen wird die Spannungsdefinition des Materialmodells in das Residuum eingesetzt und das aufgestellte Gleichungssystem über einen Solver gelöst. Dabei werden aufgrund der Nicht-Linearitäten iterative Solver verwendet.

2.1.9 Newton Raphson

Ein klassisches Lösungsverfahren, das sich für die Nullstellenbestimmung des Residuums anbietet, ist das Newton-Raphson-Verfahren. Es ist iterativ und nähert den Arbeitspunkt $\underline{u}_n$ über die wiederholte Lösung der linearisierten Gleichung um den Arbeitspunkt an die Nullstelle an. Die Anwendung dieses Verfahrens innerhalb der FEM wurde erläutert in [1].
Das Verfahren kann über die Taylorreihenentwicklung erster Ordnung des Residuums $\underline{R}(\underline{u})$ um den aktuellen Arbeitspunkt $\underline{u}_n$ hergeleitet werden. Diese lässt sich aufstellen als

$$\underline{R}(\underline{u}_{n+1}) \approx \underline{R}(\underline{u}_n) + \underline{\underline{T}}(\underline{u}_n) \cdot (\underline{u}_{n+1} - \underline{u}_n) \mid \underline{\underline{T}} = \frac{\partial \underline{R}}{\partial \underline{u}}.$$

Der Arbeitspunkt im nächsten Iterationsschritt wird über die Nullstelle des Residuums ermittelt. Mit $\underline{R}(\underline{u}_{n+1}) = \underline{0}$ ergibt sich

$$\underline{0} \approx \underline{R}(\underline{u}_n) + \underline{\underline{T}}(\underline{u}_n) \cdot (\underline{u}_{n+1} - \underline{u}_n) \Leftrightarrow \underline{\underline{T}}(\underline{u}_n) \cdot (\underline{u}_{n+1} - \underline{u}_n) = -\underline{R}(\underline{u}_n).$$

Wird der Ausdruck nach dem neuen Iterationswert $\underline{u}_{n+1}$ umgestellt, ergibt sich

$$\Rightarrow \underline{u}_{n+1} = \underline{u}_n - \underline{\underline{T}}(\underline{u}_n)^{-1} \cdot \underline{R}(\underline{u}_n) = \underline{u}_n + \underbrace{-\underline{\underline{T}}(\underline{u}_n)^{-1} \cdot \underline{R}(\underline{u}_n)}_{\Delta \underline{u}_n}.$$

Für das Abbruchkriterium der Iteration können verschiedene Vergleichswerte verwendet werden. Fällt der gewählte Vergleichswert unter die Toleranz, wird die Schleife beendet. Mögliche Werte für die Verwendung im Abbruchkriterium sind der Betrag oder eine Norm des Residuums, die Schrittweite etc.
Der Newton-Raphson-Solver wird innerhalb der nicht-linearen FEM eingesetzt, um die Nullstelle des Residuums zu bestimmen. Dieses wird aufgestellt als

$$\underline{R} = \int_{\Omega} \underline{\underline{B}}^{T} \cdot \underline{\sigma} \, d\Omega - \underline{f}$$

3 Zielsetzung

3.1 Anforderungen

In dem folgenden Kapitel werden die Anforderungen an das im Rahmen dieser Arbeit erstellte Werkzeug formuliert. Aus dem Titel der Arbeit geht die primäre Anwendung hervor. Ziel ist die Durchführung von Finite Elemente Berechnungen im Rahmen der Unsicherheitsquantifizierung. Dabei soll das Programm eine experimentelle Vorstufe zur Implementierung in gängigen Softwarelösungen sein. Aus diesem Grund empfiehlt es sich, dass das Pogramm ein Netzformat verwendet, das auch innerhalb von finalen Lösungen zum Einsatz kommt. Des Weiteren soll ein minimaler Modellierungs- und Einarbeitungsaufwand geboten werden. Optionale Ziele sind das Implementieren nicht-linearer FEM-Modelle. Zudem wird eine möglichst effiziente Arbeitsweise und Ressourcenverwaltung angestrebt.

Primäre Anforderungen

- Lösen linearer FEM-Probleme
- Unsicherheitsquantifizierung
- Lösen nicht-linearer FEM-Probleme für simple Materialmodelle
- geringer Modellierungs- und Einarbeitungsaufwand
- Verwendung von gängigem Netzformat
- Effiziente Arbeitsweise, Ressourcen- und Speicherverwaltung

3.2 Konzept

Um den Anforderungen aus 3.1 gerecht zu werden, wird ein einfaches Konzept für den Aufbau des Werkzeugs erstellt. Dieses Konzept sieht ein eigenständiges, ausführbares Programm mit einer rudimentären Benutzeroberfläche vor. Die Benutzeroberfläche bietet dabei Steuerungsoptionen und Ergebnisvisualisierung. Das Programm lädt beim Import dynamisch alle modellspezifischen Informationen. Das bedeutet, dass jegliche Information, die für die konkrete Berechnung des Modells relevant ist (isoparametrische Elemente, Wahrscheinlichkeitsdichten, nicht-lineare Materialmodelle) über externe Textdateien und nicht über die ausführbare Datei selbst bereitgestellt wird. Damit sind Inhalte dynamisch implementierbar und veränderbar (erneutes Laden der veränderten Datei). Die Dateien, die die Modelldefinition bereitstellen, verwenden ein gut lesbares und selbserklärendes Dateiformat. Damit kann die Modellierung und Anspassung der FEM-Probleme ausschließlich über das Erstellen und Bearbeiten von Dateisätzen erfolgen. Das Netz wird aus einem mit der Software Abaqus kompatiblen Format geladen. Für eine effiziente Arbeitsweise wird ein Speicher implementiert, der Berechnungsergebnisse verwaltet. Beim Import wird festgestellt, ob eine Neuberechnung des Problems erforderlich ist (bei Änderung der Dateisätze). Je nachdem werden die Berechnungsergebnisse entweder von Neuem kalkuliert oder aus dem Speicher geladen.

3.2.1 Benutzeroberfläche

Die Benutzeroberfläche dient dem dynamischen Laden und Entladen von Modellen. Zudem werden Bearbeitungsmöglichkeiten für die Wahrscheinlichkeitsdichtefunktionen (pdf) der Unsicherheitsquantifizierung geboten. Die Bearbeitung über den modellbeschreibenden Dateisatz

ist ebenfalls möglich, allerdings bietet die Oberfläche konkrete Funktionen zum schnellen Erproben der Verteilung. Dazu zählen die Möglichkeit vorgefertigte Funktionsvorlagen zu importieren und die definierten Parametersätze über Schieber zu variieren. Die Vorlagen sind dabei wiederum über Textdateien definiert und können somit angepasst und erweitert werden.

3.2.2 Modellvisualisierung

Die Modellvisualisierung erfolgt im Programm über ein zwei- oder dreidimensionales Rendering des verformten und unverformten Netzes. Die Benutzeroberfläche bietet dabei mehrere Einstelloptionen für das Rendering. Zum Beispiel die Auswahl der auf dem Netz visualisierten Größe, dem Netz, auf dem diese Größe visualisiert wird (verformt oder unverformt), und einen Abspieldialog für die Animationen der nichtlinearen Zeitschrittsimulationen.

4 Implementierung

Im folgenden Kapitel wird die Umsetzung des Konzepts im Detail gezeigt. Dabei wird auf die konkrete Auswahl der verwendeten Datei- und Speicherformate, Bibliotheken, implementierte Algorithmen und Ressourcenverwaltung eingegangen. Des Weiteren wird die dateigetriebene Modellierung des Werkzeugs gezeigt.

4.1 Dateiformate

Das Programm verwendet für die Ressourcenverwaltung drei verschiedene Dateiformate. Jedes einzelne Format findet dabei Anwendung für eine bestimmte Anforderung.

Verwendete Formate

- 1. INP
- 2. JSON
- 3. Binäre Dateien

Im Folgenden werden die Eigenschaften der einzelnen Dokumenttypen und die resultierenden Einsatzgebiete aufgezählt und begründet.

4.1.1 INP

Das INP-Dateiformat ist ein textbasiertes Eingabeformat, das Finite-Elemente-Modelle definiert. Es wird vor allem von der FEM-Software Abaqus verwendet und enthält alle relevanten Modellinformationen wie Knotenkoordinaten, Elementknoten, Materialeigenschaften, Randbedingungen und Lasten [6]. Das entwickelte Werkzeug verwendet das Format, um Netzgeometrie und Vernetzung zu laden. Die Verwendung des INP-Formats hat den Vorteil, dass die Netze mit Abaqus erzeugt und bearbeitet werden können. Zudem sind die im Rahmen der Voruntersuchungen verwendeteten Netze mit Abaqus kompatibel und können für finale Lösungen ohne Abänderungsaufwand wiederverwendet werden.

4.1.2 JSON

JSON (JavaScript Object Notation) ist ein leichtgewichtiges, textbasiertes Datenformat. Es basiert auf einer einfachen Schlüssel-Wert-Paar Struktur. Damit ist es gut lesbar und leicht verständlich. Auch Computer können das Format effizient verarbeiten und verwalten [7]. Über die Benenung der Schlüssel lässt sich direkt die Funktion des zugehörigen Wertes beschreiben. Deshalb ist es in vielen Fällen nahezu selbsterklärend. Aufgrund dieser Eigenschaften wird das Format für die Dateien verwendet, aus denen Informationssätze dynamisch geladen werden. Die Modellierung ist damit unkompliziert und selbsterklärend.

4.1.3 Binäre Dateien

Als binäre Dateien werden hier Dokumente bezeichnet, die Bytefolgen oder -sequenzen enthalten. Die Dokumente sind dabei nicht textbasiert und damit für den Nutzer nicht mehr direkt lesbar. Zudem wird zur Interpretierung die konkrete Information benötigt, in welche Struktur die Bytefolge übersetzt werden soll. Die Dokumente bieten sich damit als Speicherdateien an, in denen Inhalte aufbewahrt werden, die für die Programmausführung relevant sind, aber nicht vom

Nutzer bearbeitet oder gelesen werden. Anwendung findet das Dateiformat beim Abspeichern der Simulationsergebnisse.

4.2 Verwendete Bibliotheken

Für die Umsetzung des Konzepts werden mehrere konkrete Funktionalitäten benötigt, die entweder aus vorhandenen Bibliotheken importiert oder eigenimplementiert werden müssen.

Benötigte Funktionen

- Lesen (und Schreiben) der verwendeten Dateiformate
- symbolische Ausdrücke mit Skalar- und Matrixsubstitution
- lineare Tensoralgebra
- lineare Tensoralgebra für Tensoren mit symbolischen Einträgen
- Fenstererstellung und Rendering
- Benutzeroberfläche

4.2.1 Dateimanagement

Für die Verwaltung der gewählten Dateiformate werden mehrere Funktionalitäten benötigt. Dazu zählt die Übersetzung des INP-Formats in programminterne Speicherstrukturen. Die Rückübersetzung ist nicht erforderlich. JSON-Dateien hingegen müssen gelesen und erzeugt werden können. Die Anpassungsoption der Wahrscheinlichkeitsdichtefunktionen aus der Oberfläche macht die Rückübersetzung erforderlich. Binäre Dateien werden ebenfalls gelesen und geschrieben. Zudem soll die Übersetzung von verschiedenen Objekten in die Bytesequenzspeicher möglichst simpel gestaltet werden. Dazu wird eine Logik benötigt, die eine eigenständige und erweiterbare Übersetzung bzw. Serialisierung von Objekten vornimmt.

Benötigte Funktionen

- Einlesen der INP-Files
- Einlesen und Abspeichern der JSON-Files (Bearbeitung über Benutzeroberfläche)
- Einlesen und Abspeichern der binären Dateien
- Serialisierung von Objekten für die binären Dateien

Für das Laden der INP-Dateien wurde keine direkt nutzbare Option gefunden. Die Bibliothek Assimp [8], die häufig für den Import von Netzen verwendet wird, unterstützt das INP-Format derzeit nicht. Deshalb nutzt das Programm einen rudimentären Lademechanismus, der eigens für diese Anwendung geschrieben worden ist. Zum Verwalten der JSON-Dateien wird die etablierte, in C++ geschriebene Bibliothek nlohmann/json [9] verwendet. Beim Verwalten der Dateien mit dieser Bibliothek wird die JSON-Datei in ein bibliotheksinternes JSON-Objekt überführt. Dieses organisiert die Daten in einer abstrakten, hierarchischen Baumstruktur und speichert die Werte jeweils in Form des erkannten Datentyps. Mit Angabe des Schlüssels und des Zieltyps lässt sich der zugehörige Wert extrahieren. Für die Verwaltung der binären Datei in der gewünschten

Art und Weise konnte ebenfalls keine Bibliotheksfunktion gefunden werden. Deshalb nutzt das Prgramm zur Übersetzung von Objekten in Bytesequenzen und dem Schreiben und Lesen der Bytesequenzspeicher eine eigens entwickelte Serialisierungslogik. Diese ermöglicht über Entpackung und rekursiven Aufruf die automatische Aufschlüsselung von Objekten in Attribute mit statischer Speicherstruktur. Die Entpackung nutzt dabei unter anderem die in Boost enthaltene Bibliothek PFR [10], die Objekte automatisch in ihre öffentlichen Attribute zerlegen kann, ohne dass eine explizite Serialisierungsdeklaration oder manuelle Zugriffe erforderlich sind. Damit können PODs (Plain Old Datastructures) direkt in alle enthaltenen, auch nicht statischen Attribute aufgeschlüsselt werden. Die Bytesequenzentpackung nutzt dieses Verhalten. Zudem werden häufig genutzte dynamische Strukturen (Map, Vektor, etc.) rekursiv in Entpackungsaufrufe der Einträge und Größe umgeleitet. Die Definition der Übersetzung von nichttrivialen dynamischen oder privaten Attributen wird über eine Eigenanbindung ermöglicht. Damit kann jede beliebige Struktur ohne großen Serialisierungsaufwand in statische Größen zerlegt werden, die über einen blockweisen Speichertransfer in die Bytefolge kopiert werden. ASCII-Zeichenfolgen bleiben dabei (umgekehrt) lesbar. Damit auch diese nicht mehr erkannt werden können, nutzt die Serialisierung eine einfache XOR-Sicherheitsverschlüsselung. Dabei wird die Bytefolge mittels der exklusiven-Oder-Konjunktion (XOR) mit einem Byte bzw. ASCII-Zeichenfolge-Schlüssel verknüpft, dadurch bleibt die Operation reversibel. Zum Lesen der Bytedateien wird damit ein Schlüssel benötigt. Für die interne Codierung und Decodierung wird der Schlüssel selbstständig vom Programm bereitgestellt. In Tabelle 1 ist eine Übersicht über die Umsetzung der im Rahmen der Dateiverwaltung benötigten Funktionen gezeigt.

Tabelle 1: Umsetzung der Dateiverwaltungs-Funktionalitäten

Funktion	**Umsetzung / verwendete Bibliothek**
Verwaltung INP-Files	Eigenimplementierung
Verwaltung JSON-Files	nlohmann/json
Verwaltung der binären Dateien	Eigenimplementierung
Serialisierung für die binären Dateien	Eigenimplementierung, verwendet Boost/PFR

4.2.2 arithmetische Funktionen

Für die FEM-Berechnungen werden Lösungsalgorithmen und arithmetische Funktionen für die Tensoralgebra mit numerischen und symbolischen Einträgen benötigt. Dafür werden die beiden Bibliotheken Eigen und SymEngine verwendet. Eigen [11] ist eine C++-Bibliothek, die effiziente Funktionen für lineare Algebra, einschließlich Tensoroperationen bietet. Die Bibliothek wird für die rein numerische Tensoralgebra verwendet. SymEngine [12] ist eine in C++ geschriebene Bibliothek für symbolische Mathematik, die für das effiziente Arbeiten mit komplexen symbolischen Ausdrücken ausgelegt ist. Sie stellt Objekte für Vektoren und Matrizen mit symbolischen Einträgen bereit, Tensoren höherer Stufe können dabei nicht genutzt werden. Die SymEngine wird im Rahmen dieser Anwendung für die symbolische Algebra von Skalaren und Tensoren mit symbolischen Einträgen genutzt. Für die Substitution von Skalaren in symbolische Ausdrücke wird sie ebenfalls verwendet. Sie ist allerdings nicht in der Lage Matrizen in symbolische Ausdrücke zu substituieren und formt symbolische Ausdrücke intern nach skalaren Rechenregeln um. Aus diesem Grund wird für die Matrixsubstitution in symbolische Ausdrücke eine Eigenimplementierung verwendet. Diese wird in einen Lexer und in einen Parser unterteilt. Der Parser funktioniert dabei über den rekursiven Aufruf des Lexers und arbeitet mit SymEngine-Matrizen. Die Lösung der Gleichungssysteme und Residuen für die FEM geschieht mittels der Solver aus

Eigen und einem eigenimplementierten Newton-Raphson-Solver, der ebenfalls mit SymEngine Matrizen arbeitet. In Tabelle 2 sind die Lösungen für die verschiedenen benötigten arithmetischen Funktionen aufgeführt.

Tabelle 2: Umsetzung der algebraischen Funktionalitäten

Funktion	**Umsetzung / verwendete Bibliothek**
Tensoralgebra	Eigen
Algebra Tensoren symbolische Ausdrücke	SymEngine
symbolische Skalarsubsitution	SymEngine
symbolische Matrixsubstitution	Eigenimplementierung über Lexer/Parser
Cholesky-Solver	Eigen
Newton-Raphson Solver	Eigenimplementierung mit SymEngine Matrizen

4.2.3 Fenster und Rendering

Das folgende Unterkapitel beschreibt, welche Lösungen für das Erstellen des Fensters, das Rendering und die Benutzeroberfläche verwendet worden sind. Für die Erstellung des Fensters und das Rendering der Ergbnisse wird die Bibliothek raylib genutzt. raylib [13] ist eine benutzerfreundliche C-Bibliothek für die Grafikprogrammierung. Sie bietet Schnittstellen für Fensterverwaltung und Rendering. Die Benutzeroberfläche wird mithilfe der etablierten Bibliothek ImGui erzeugt. ImGui (Immediate Mode GUI) [14] wird zur Erstellung von plattformunabhängigen, grafischen Benutzeroberflächen benutzt. Sie ist darauf ausgelegt, schnelle und flexible Oberflächen zu erzeugen. Für die Implementierung der ImGui-Oberfläche in das Raylibfenster wird rlImGui verwendet. rlImGui [15] ist eine Bibliothek, die die Integration von ImGui in raylib ermöglicht. Für die Oberflächenelemente Plots und Dateidialog kommen ImPlot [16] und ImFileBrowser [17] zum Einsatz. Tabelle 3 zeigt eine Übersicht über die beschriebenen Lösungen für die benötigten graphischen Funktionen.

Tabelle 3: Umsetzung der algebraischen Funktionalitäten

Funktion	**Umsetzung / verwendete Bibliothek**
Fenstererstellung/Rendering	raylib
Erzeugen GUI	ImGui
Integration der GUI	rlImGui
Plots	ImPlots
Dateiauswahldialog	ImFileBrowser

4.3 Modelldefinition

Die vollständige Beschreibung eines FEM-Problems wird über einen Dateisatz realisiert. Dieser enthält Dokumente, die Netzgeometrie, Materialverhalten und Randbedingungen beschreiben. Die Definition des isoparametrischen Elements ist kein Teil der Modelldefinition. Es wird einmalig in das Ressourcen-Verzeichnis des Programms implementiert. Damit steht es für Berechnungen zur Verfügung. Die Implementierungen mehrerer Standardelemente (Viereckselement `CPS4R`, Würfelelement `C3D8R`) sind bereits vorhanden. Das Netz wird aus einer INP-Datei geladen, Randbedingungen, Material und isoparametrisches Element jeweils aus einer JSON-Datei.

Die modellbeschreibenden Dateien werden für den Import in einem Verzeichnis mit der Endung `*.model` bereitgestellt. Das Prefix ist dabei der Name, unter dem das Modell ins Programm geladen wird. Der Import wird über den programminternen Dateidialog gestartet, der standardmäßig im Verzeichnis `./Import/` geöffnet wird. Über die Auswahl eines Modellverzeichnisses wird der Importvorgang des Modells angestoßen. Der Dialog ermöglicht auch eine Navigation zu anderen Ordnern. Damit können Modelle aus jedem beliebigen zugänglichen Verzeichnis geladen werden. Das vom Netz benötigte isoparametrische Element wird aus dem Ordner `./Recc/Cells/` importiert. Bei der erstmaligen Berechnung wird eine binäre Datei erzeugt und ins Modellverzeichnis geschrieben, diese speichert die Berechnungsergebnisse. Zudem enthält das Dokument die letzten Bearbeitungszeiten des modellbeschreibenden Dateisatzes. Für jeden weiteren Import des Modells wird geprüft, ob die letzten Änderungsdaten des Dateisatzes von den hinterlegten abweichen. Ist das der Fall, wird das Modell neu berechnet und die binäre Datei aktualisiert. Entsprechen die letzten Änderungszeiten den hinterlegten, werden die Berechnungsergebnisse aus dem Bytesequenzspeicher entnommen. Die manuelle Bearbeitung der Speicherdatei führt zu undefiniertem Verhalten. Das Löschen dieser binären Datei ist allerdings jederzeit und gefahrlos möglich. Es führt lediglich zur Neuberechnung des Systems und erneuter Erzeugung der Speicherdatei. Nach der Anpassung oder Implementierung von isoparametrischen Elementen empfiehlt es sich, alle binären Dateien zu löschen, um Konflikte zwischen der alten und der neuen Indizierung der Elemente auszuschließen. Dafür steht in der Benutzeroberfläche eine Schaltfläche zur Verfügung. Algorithmus 1 zeigt die für den Import benötigte Dateistruktur exemplarisch. Dabei liegt das Importmodell im Ordner `./Import`. Es enthält alle für die Modelldefinition relevanten Dateien und den vom Programm erzeugten und verwalteten Bytesequenzspeicher. Das isoparamtrische Element, das vom importierten Netz genutzt wird, ist in `./Recc/Cells/` implementiert.

Algorithmus 1: Ladestruktur Modellimport

```
Arbeitsverzeichnis des Programms/
|__ build/
|__ Recc/
|   |__ Cells/
|   |   |__ CPS4R.ISOPARAM                    // JSON-Datei
|   |   |__ CPS3.ISOPARAM                     // JSON-Datei
|   |   |__ für Modell relevantes Element     // JSON-Datei
|   |   |__ ...
|   |__ ...
|
|__ Import/
|   |__ ModelName.model/
|   |   |__ .Mesh             // INP-Datei        über Abaqus oder manuell anlegen und bearbeiten
|   |   |__ .Material         // JSON-Datei       manuell anlegen und bearbeiten
|   |   |__ .Constraints      // JSON-Datei       manuell anlegen und bearbeiten
|   |   |__ .RESULTCACHE      // Bytecode-Datei   wird vom Programm erzeugt und verwaltet
|   |__ ...
|__ ...
```

4.3.1 Definition des isoparametrischen Elements

Die Definition des isoparametrischen Elements geschieht über die JSON-Datei `PREFIX.ISOPARAM`, wobei `PREFIX` durch die von Abacus vorgegebene Bezeichnung des isoparametrischen Elements im INP-Dokument ersetzt wird. Das zweidimensionale, quadratische Element mit vier Knoten wird als `CPS4R` bezeichnet. Die Datei, die dieses isoparametrische Element definiert, heißt `CPS4R.ISOPARAM`. Stimmt das Prefix nicht mit der Bezeichnung aus dem INP-Dokument überein,

kommt es zu einem Importabbruch und einer Fehlermeldung, die beschreibt, dass das Dokument `CPS4R.ISOPARAM` nicht gefunden werden konnte. Bei Existenz der Datei werden alle zur Verwendung des isoparametrischen Elements relevanten Information aus der JSON-Datei geladen, darunter die Dimension des Elements, Knotenanzahl, Ansatzfunktionen und mehr. Die Größen müssen dabei als Werte für gültige Schlüssel übergeben werden. Ganzzahlige Werte, Fließkommazahlen und boolsche Werte werden ohne, Zeichenfolgen mit Anführungszeichen angeführt. In Dateiauszug 1 ist das Dokument abgebildet, das die Definition für das Element `CPS3R` anstellt. Die Felder `NDIMENSIONS` und `NNODES` definieren die Dimension und Knotenanzahl des Elements. Die Ansatzfunktionen, Quadraturpunkte und Gewichte werden als Liste übergeben. Dabei wird auf die lokalen Koordinaten `r`, `s` und `t` zurückgegriffen.

Dateiauszug 1: Auszug aus CPS4R.ISOPARAM

```
1 {
2     "NDIMENSIONS":      2,
3     "NNODES":           3,
4     "SHAPEFUNCTIONS":   ["1-r-s", "r", "s"],
5     "QUADRATUREPOINTS": [[{"r": "0.1666666"}, {"s": "0.1666666"}],
6                          [{"r": "0.6666666"}, {"s": "0.1666666"}],
7                          [{"r": "0.1666666"}, {"s": "0.6666666"}]],
8     "WEIGHTS":          [0.16666, 0.16666, 0.16666]
9 }
```

Für das zweidimensionale Rendering werden keine weiteren Informationen benötigt. Für das dreidimensionale Rendering hingegen schon. Der Zeichenaufruf mit einer Grafikschnittstelle erfordert die Deklarierung, welche Dreiecke von der Grafikkarte gezeichnet werden sollen. Diese werden für dreidimensionale Elemente ebenfalls über die JSON-Dateien definiert. Der Dateiauszug 2 zeigt die Angabe der für das Rendering notwendigen Informationen über die `*.ISOPARAM`-Datei. Bei dem abgebildeten Quelltext handelt es sich um einen Auszug aus der Datei, die das Würfelelement `C3D8R` implementiert. Die Definition der zu zeichnenden Dreicke erfolgt als Angabe mit den lokal gültigen Knotenindizes. Das Rendering der Vorderfläche des Würfelelements, aufgespannt durch die Knoten 0, 1, 2 und 3, geschieht über das Zeichnen der zwei Dreiecke $0, 1, 2$ und $0, 2, 3$. Diese Indizes werden als Array über das Feld `FACEINDICES` bereitgestellt. Zur Reduzierung des Rechenaufwands beim Rendering wird das sogenannte Back-Face Culling eingesetzt. Dabei werden Flächen, deren berechnete Normalenvektoren von der Kamera weg zeigen, als rückseitig interpretiert und vom Rendering ausgeschlossen. Nur diejenigen Flächen, die als vorderseitig erkannt werden, werden auch gerendert. Der Normalenvektor wird über die gegebenen Knotenkoordinaten und die Indexreihenfolge bestimmt. Aus diesem Grund müssen die Indizes im Uhrzeigersinn (aus Sicht der Kamera) angegeben werden. Ansonsten werden die Flächen als Rückseiten erkannt und nicht gezeichnet. Zusätzlich wird zur besseren, visuellen Unterscheidbarkeit der eingefärbten Elemente eine Umrandung gezeichnet. Die Umrandung geschieht durch das Hervorheben der Kantenlinien des Elements in einer einheitlichen Farbe. Diese werden dabei über das Array `WIREFRAMEINDICES` definiert. Jede einzelne Kante wird über das Indexpaar der beiden Knoten angegeben, die sie verbindet. Im Rendering werden die übergebenen Kanten als zylindrische Verbindungen gezeichnet. Damit ergibt sich das Drahtgitter des Elements. Dieses grenzt das eingefärbte Element optisch von den anderen, häufig ähnlich eingefärbten Elementen ab. Die Angabereihenfolge der Indizes bei der Kantendefinition ist irrelevant.

Dateiauszug 2: Auszug aus C3D8R.ISOPARAM

```
{
    "NDIMENSIONS":      3,
    "NNODES":           8,
    // ...
    "FACEINDICES":      [
                          0,1,2, 0,2,3, 1,6,2, 1,5,6, 3,6,7, 3,2,6,
                          0,7,4, 0,3,7, 0,5,1, 0,4,5, 4,6,5, 4,7,6
                        ],
    "WIREFRAMEINDICES": [[0,1],[1,2],[2,3],[3,0], [4,5],[5,6],[6,7],[7,4], [4,5],[5,6],[6,7],[7,4]]
}
```

4.3.2 Definition der Netzgeometrie

Die Definition der Netzgeometrie und Vernetzung geschieht über das INP-Dokument `.Mesh`. Dieses kann über die Benutzeroberfläche von Abaqus erzeugt und verwaltet werden. Netze mit mehreren isoparametrischen Elementen können derzeit nicht berechnet werden. Belastungen und Materialmodelle werden ebenfalls nicht aus der INP-Datei, sondern aus den JSON-Dateien der Modelldefinition geladen. Deshalb empfiehlt sich die Erstellung eines einfachen Netzes ohne Randbedingungen und Materialmodell.

4.3.3 Definition der Randbedingungen

Die Definition der Randbedingungen geschieht über das JSON-Dokument `.Constraints`. Für die Definition der Randbedingungen werden beaufschlagte Kräfte und fixierte Freiheitsgrade angegeben. Die Angabe der fixierten Freiheitsgrade erfolgt dabei über das `Constraints` Array, das Schlüssel-Wert-Paare enthält. Die Schlüssel sind dabei die Indizes der Nodes, deren Freiheitsgrade fixiert werden, und die Werte Listen mit Raumrichtungsindizes der fixierten Knotenfreiheitsgrade. Dabei gibt der Index die Raumrichtung der Fixierung an. 0 ist der Index für den x-Freiheitsgrad, 1 für den y-Freiheitsgrad und 2 für den z-Freiheitsgrad. Die Angabe der beaufschlagten Kräfte erfolgt über das `Loads` Array. Dieses enthält ebenfalls Schlüssel-Wert-Paare mit dem Knotenindex als Schlüssel und der Beschreibung des auf den Knoten aufgebrachten Kraftprofils als Wert. Die Angabe des Kraftprofils ist dabei eine Zuordnung von Raumrichtungen und Kraftmagnituden. Im Dateiauszug 3 ist eine exemplarische `.Constraints` Datei gezeigt. Die Datei beschreibt die Fest-/Loslagerung eines Systems an den Knoten 1 und 11 und die Belastung an den Knoten 56 und 66.

Dateiauszug 3: exemplarische .Constraints Datei

```
{
    "Constraints" : [
        {"1" : [0,1]},
        {"11" : [0]}
    ],
    "Loads" : [
        {"56" : [{"0": -1000}, {"1": -1000}]},
        {"66" : [{"1": -1000}]}
    ]
}
```

4.3.4 Definition des Materials

Die Definition des Materialmodells und Steuerung der durchgeführten Simulation geschieht über das JSON-Dokument `.Material`. Dabei sind für verschiedene Analyseoptionen verschiedene Parameterübergaben erforderlich. Im Folgenden werden die benötigten Parameterangaben für jeden Anwendungsfall einzeln betrachtet.

zentrale Anwendungsfälle

- 1. lineare FEM
- 2. Unsicherheitsquantifizierung (für lineare FEM)
- 3. nicht-lineare FEM

Innerhalb des Materialdokuments gibt es vier übergeordnete Sektionen, die Felder `isLinear`, `stdParams`, `nonLinearApporach` und `pdf`. Der Schlüssel `isLinear` erwartet einen boolschen Wert. Wird für den Parameter `false` übergeben, wird die lineare FEM ausgeführt. Dabei entscheidet das Vorhandensein der Sektion `pdf`, ob im Anschluss an die lineare FEM eine Unsicherheitsquantifizierung vorgenommen wird. Wird der Parameter `isLinear` mit `true` übergeben, wird die nicht-lineare FEM ausgeführt. Alle drei Anwendungsfälle greifen auf drei Standardparameter zurück. Dabei handelt es sich um das Elastizitätsmodul E, die Querkontraktionszahl ν und die Materialdicke t. Die Dicke wird für den dreidimensionalen Fall unabhängig vom übergebenen Wert mit 1 substituiert. In Dateiauszug 4 ist die Übergabe der Standardparameter und der Linearität des Materialmodells anhand einer exemplarischen Materialdatei gezeigt. Die Standardparameter werden dabei für die Schlüssel `E`, `v` und `t` übergeben, das Elastizitätmodul in der Einheit Megapascal, t in der Einheit, in der die Knotenkoordinaten angegeben sind und ν und als dimensionslose Größe.

Dateiauszug 4: Standardparameter in der .Material Datei

```
1 {
2     "stdParams": { "E": 20000, "t": 0.1, "v": 0.3 },
3     "isLinear": false
4 }
```

Soll nach der linearen FEM eine Unsicherheitsquantifizierung vorgenommen werden, wird die JSON-Datei um die `pdf`-Sektion ergänzt. In dieser Sektion werden Parameter, Gültigkeitsbereich, Dichtefunktion und weitere Vorgaben für die Unsicherheitsquantifizierung übergeben. In Dateiauszug 5 ist diese Parameterübergabe für ein exemplarisches Material gezeigt. Die Dichtefunktion wird dabei als Zeichenfolge für den Schlüssel `function` übergeben. Dabei ist sie ausschließlich von der zufallsverteilten Variablen `xi` und den ebenfalls über die Datei deklarierten Parametern abhängig. Der Gültigkeitsbereich der Funktion wird über eine Ober- und Untergrenze abgesteckt. Werden Grenzwerte mit 0 übergeben, ermittelt das Programm sie iterativ. Diese Iteration kann enbenfalls über die Datei spezifiziert werden. Dazu dient die `pdfPreprocessing`-Sektion. Der `segmentation`-Wert beschreibt die Schrittweite, in der die Funktion abgetastet wird und der `tolerance`-Wert den Schwellwert für den Abtastvorgang. Fällt der Funktionswert beim Abtasten unter diesen Schwellwert, wird an der entsprechenden Stelle der Rand des Gültikeitsbereiches definiert. Die iterative Grenzwertermittlung setzt voraus, dass die Funktion für $\xi = 0$ größer 0 ist, also $pdf(\xi = 0) > 0$. Ist dies nicht der Fall, muss der Gültigkeitsbereich abgesteckt werden. Über die `params`-Sektion wird ein Satz an Funktionsparametern definiert. Diese können für die Funktionsdefinition verwendet werden. Innerhalb der Benutzeroberfläche lassen sich die übergebenen

Parameter mit Schiebern variieren. Die Anzahl an Testwerten, die im Rahmen der Monte-Carlo Methode ermittelt werden, wird über den `samples`-Wert in der `pdfSampling`-Sektion eingestellt.

Dateiauszug 5: pdf-Definition in der .Material Datei

```
{
    "pdf": {
        "borders": {"xi_max": 0, "xi_min": 0},
        "function": "exp(-0.5*((xi-mu)/sigma)**2)/(sqrt(2*pi)*sigma)",
        "params": {
            "mu": 0.0,
            "sigma": 0.5
        },
        "pdfPreprocessing": {"segmentation": 0.1, "tolerance": 0.1},
        "pdfSampling": {"samples": 100000}
    },
}
```

Die Simulation wird mit einem nicht-linearen Materialmodell durchgeführt, wenn der Wert für den Schlüssel `isLinear` `true` ist. In diesem Fall wird die Sektion `nonLinearApproach` eingelesen. Diese definiert über `innerVariable` zunächst den Namen der verwendeten inneren Variablen und deren Tensorstufe. Das Programm, das im Rahmen dieser Arbeit entwickelt worden ist, kann nur mit Skalaren, Vektoren oder Matrizen arbeiten. Zudem ist die Übergabe der Größe der inneren Variablen erforderlich. Dabei muss für jede Tensorstufe, die bei der Definition der inneren Variablen angegeben worden ist, ein Eintrag in der Größenangabe stehen. Mit der benannten inneren Variablen werden die Spannungsfunktion mit `sigma` und die Evolutionsgleichung der inneren Variablen mit `evolutionEquation` deklariert. Dabei ist es wichtig, dass für die Substitution der inneren Variablen mit den Werten aus dem aktuellen oder letzten Simulationsframe die Erweiterungen `_n` und `n_plus_1` an den Namen der inneren Variablen angehängt werden. Bei der Erstellung der Gleichungen kann auf mehrere vom Werkzeug für die Substitution zur Verfügung gestellte Größen, Parserfunktionen und selbst definierte Parameter zurückgegriffen werden. Eine Übersicht über die Größen, die das Werkzeug zur Substitution und damit zur Verwendung im Spannungsterm und der Evolutiongleichung bereitstellt, bietet Tabelle 4.

Tabelle 4: Beschreibung der verwendeten Variablen im Simulationsmodell

Name / Feld	**Beschreibung**
sigma	Spannungsansatz (substituierte, vom Nutzer definierte Größe)
epsilon_n	Dehnung im letzten Frame
epsilon_n_plus_1	Dehnung im aktuellen Frame
uCell	Spannung des Elements im aktuellen Frame
ElastTensor	Elastizitätstensor des Materials
t	Dicke des Materials (nur relevant für 2D)
B	B-Matrix
jDet	Determinante der Jacobi-Matrix
w	Gewicht des Quadraturpunkts
S	Deviator-Matrix entsprechend der vorliegenden Dimension
I	Einheitsmatrix mit Größe entsprechend der Zeilenanzahl von B
deltaT	Zeitschritt der Simulation
innereVariable_n	Innere Variable im letzten Frame
innereVariable_n_plus_1	Innere Variable im aktuellen Frame

Der Parser bietet momentan nur die Funktion `Identity(size)` an, die eine Einheitsmatrix der angegebenen Größe erzeugt. Eigene Parameter können unter `params` übergeben werden. Dateiauszug 6 zeigt die Definition eines nicht-linearen, viskoelastischen Materialmodells über die Materialdatei. Dabei wird der Fließgeschwindigkeitsparameter `eta` und ein exemplarischer, beispielhafter Matrixparameter definiert. Aufgrund der rudimentären, eigenentwickelten Ausdrucksanalyse, können für die Übergabe nur die Operatoren `+`, `*`, `^` und definierte Funktionsnamen verwendet werden. Allerdings ist das Negieren von Argumenten über das Vorzeichen `-` möglich. Damit können Operationen, wie die Division oder Subtraktion, über die implementierten Operatoren und Argumentnegierungen angestellt werden. Die Subtraktion muss geschrieben werden als `argument + -argument` und die Division als `argument*argument^-1`. Zudem ist die Verwendung der Exponenten `t`, `T`, `-t` und `-T` für die Transponierte und die Inverse der Tranponierten möglich.

Dateiauszug 6: Definition des nicht-linearen Materialverhaltens in der .Material Datei

```
{
    "isLinear": true,
    "nonLinearApproach": {
        "innerVariable": ["epsilon_v", 1],              // [Name, Tensorstufe]
        "innerVariableSize": [6],                       // Skalar: [], Vektor: [i], Matrix: [i,j]

        "sigma": "ElastTensor*(epsilon_n_plus_1+-epsilon_v_n_plus_1)",
        "evolutionEquation": "(I+eta^-1*S*ElastTensor*deltaT)^-1*
                              (epsilon_v_n+eta^-1*S*ElastTensor*deltaT*epsilon_n_plus_1)",
        "params": {
            "Identity2D": {"rows": 3, "cols": 3,
                "data": [["1","0","0"], ["0","1","0"], ["0","0","1"]]},
            "eta": {"rows": 1, "cols": 1, "data": [["10000"]]}
        }
    }
}
```

4.4 Laden und Abspeichern der Informationen

Die Inhalte des Modells werden über die beschriebenen Lademechanismen eingelesen und in interne Datenstrukturen geschrieben. Das isoparametrische Element wird mit nlohmanm/json aus der `*.ISOPARAM` Dateien geladen und in die Struktur `CellPrefab` übersetzt. Dabei gibt es für jedes angegebene Feld der JSON-Datei ein zugehöriges Objektattribut. Die Dimension und Knotenanzahl beispielsweise werden als nicht vorzeichenbehaftete 8-Bit-Ganzzahlen in die Attribute `CellPrefab::nDimensions` und `CellPrefab::nNodes` des konstruierten Objekts extrahiert. Für Vektor- und Arrayfelder, wie die Ansatzfunktionen, wird dabei zunächst Speicherplatz in entsprechender Menge im Attribut allokiert oder reserviert. Dies spart wiederholte Neuallokierungen und Kopiervorgänge in dynamischen Attributen wie Vektoren oder Maps. Nach der Allokierung werden die Inhalte der Arrayfelder direkt in die reservierten Register konstruiert. Bei der Konstruktion wird zusätzlich das Prefix des Textdateinamens im Objekt gespeichert. Wird das isoparametrische Element beispielsweise aus der Datei `./Recc/Cells/CPS4R.ISOPARAM` geladen, wird die Zeichenfolge `CPS4R` im Objekt hinterlegt. Dieses Prefix dient der späteren Identifikation. Die Möglichkeit der indirekten Referenzierung besteht über das Attribut `CellPrefab::pID`, kurz für Prefab Identification und dient der Abspeicherung einer vergebenen Identifikationsnummer. Diese kann als indirekte Referenz oder indirekter Verweis in einem dynamsichen Speicher verwendet werden. Bei der Konstruktion wird zudem mit mehreren Annahmeprüfungen (Assertions) gearbeitet, die die Konsistenz des übergebenen isoparametrischen Elements prüfen. Zum

Beispiel, ob so viele Gewichte angegeben sind wie Quadraturpunkte. Zudem werden im gleichen Aufruf die Ableitungen nach den lokalen Raumrichtungen gebildet. Der Algorithmus 2 zeigt den exemplarischen Aufbau der Objektstruktur. Dabei wird im Konstruktor die Extraktion der Dimension und Knotenanzahl gezeigt, die Herausstellung des Elementlabels aus dem Speicherpfad und die Übersetzung der als Zeichenfolge vorliegenden Ansatzfunktionen in symbolische Ausdrücke.

Algorithmus 2: Exemplarische Definition CellPrefab-Struktur

```
class CellPrefab{

    CellPrefab(const std::string& path) : CellPrefab(){

        nlohmann::json isoParamData = ...;
        nDimensions = isoParamData["NDIMENSIONS"].get<uint8_t>();
        nNodes = isoParamData["NNODES"].get<uint8_t>()
        label = fs::path(path).stem().string()

        shapeFunctions.reserve(nNodes);
        for(const auto& [nodeNum, shapeFunction] : ...)){
            shapeFunctions.emplace_back(SymEngine::parse(shapeFunction));
        }
    }

    // ...
    std::string label = NULLSTR;
    uint8_t pID, nDimensions, nNodes;
    std::vector<Expression> shapeFunctions = {};
};
```

Um die isoparametrischen Elemente nicht für jeden Gebrauch neu laden zu müssen und universell für alle Elemente einsetzen zu können, werden sie in einen dynamischen Speicher konstruiert. Der Speicher ist eine einfache Map, die die ID als Schlüssel und das isoparametrische Element als Wert verwendet. Die Speicherstruktur ist global gültig und wird über zwei zentrale Funktionen verwaltet, der Registrierungs- und Abfragelogik. Die Registrierungsfunktion wird als Konstruktionsanfrage verwendet. Sie überprüft bei Aufruf anhand eines Labelabgleichs, ob ein identisches Element bereits vorhanden ist und gibt für diesen Fall die ID des bestehenden Objekts zurück. Ist es noch nicht vorhanden, konstruiert sie das Objekt in den Speicher und weist ihm eine eindeutige ID zu. Diese ID wird in das `CellPrefab` geschrieben und zurückgegeben. Nach der Zuweisung können isoparametrische Elemente über ihre `pID` indirekt referenziert werden. Die Abfragefunktion gibt bei Aufruf einer konkreten ID eine konstante Referenz auf das zugehörige Objekt zurück. Über diese Referenz lässt sich lesend auf Attribute des Objekts zugreifen. Der Aufbau des Speichersystems ist exemplarisch in Algorithmus 3 gezeigt. Aufgeführt wird die außerhalb eines Funktionskontexts, in einer zentralen Headerdatei deklarierte Map `g_cellPrefabCache`, die über eine Registrier- und eine Abfragefunktion verwaltet wird.

Algorithmus 3: Struktur des CellPrefabCaches

```
typedef uint8_t prefabIndex;

NodeIndex cacheCellPrefab(const std::string& prefabLabel);
const CellPrefab& getCachedCellPrefab(const prefabIndex& index);

extern std::map<prefabIndex, CellPrefab> g_cellPrefabCache;
```

Zur Speicherung von Knoten und Elementen werden zusätzliche Datenstrukturen verwendet. Die Speicherstruktur für die Knoten verwaltet Dimension und Koordinaten. Die Koordinaten werden in einem Vektorattribut gespeichert. Aufgrund benötigter Flexibilität und des quasistatischen Verhaltens einmalig allokierter Vektoren wird eine solche und kein statisches Attribut verwendet. Um das quasistatische Verhalten zu gewährleisten und die Speicherverwaltung effizient zu gestalten, wird beim Konstruktoraufruf entsprechend der übergebenen Dimension Speicher innerhalb des Vektors reserviert. Anschließend werden die Knotenkoordinaten direkt in die Liste konstruiert. Über den Aufruf des Objekts mit Index der Raumrichtung in eckigen Klammern kann auf die jeweilige Koordinate zugegriffen werden. Die Ausführung der Datenstruktur erfolgt dabei über ein template, das die Verwendung mit verschiedenen Koordinatentypen ermöglicht. Das Werkzeug benutzt primär Knoten mit Fließkommazahl-Koordinaten. Der Algorithmus 4 veranschaulicht den Aufbau der Datenstruktur, die für die Abspeicherung von Knoten genutzt wird.

Algorithmus 4: Exemplarische Definition dynNodeXd-Struktur

```
template<typename T>
struct dynNodeXd {

    dynNodeXd(const size_t& dimension, const std::vector<T>& Coordinates);

    const T& operator[](size_t koordIndex) const {
        return m_Coordinates[koordIndex];
    }

    size_t m_dimension = 0;
    std::vector<T> m_Coordinates = {};
};
```

Die Datenstrukturen zum Abspeichern der Elemente enthalten die global gültigen Indizes der Knoten, auf die sie sich beziehen. Dabei wird das Attribut so verwaltet wie die Koordinaten des Knotenobjekts. Die Speicherreservierung erfolgt dabei abhängig von der übergebenen Anzahl an Knoten. Über den lokal gültigen Knotenindex kann der zugehörige globale Index abgefragt werden. Zusätzlich referenziert das Element das zugehörige isoparametrische Element indirekt über die abgespeicherte ID. Der Algorithmus 5 verdeutlicht den Aufbau der Datenstruktur für die Abspeicherung von Elementen.

Algorithmus 5: Exemplarische Definition Cell-Struktur

```
class Cell{

    Cell(const prefabIndex& prefIndex, const std::vector<NodeIndex>& nodeIndizes);

    const NodeIndex& operator[](size_t index) const;

    prefabIndex m_prefabIndex = 0;
    std::vector<NodeIndex> m_nodeIndizes = {};
};
```

Über diese Speicherstrukturen kann das Netz vollständig abgebildet werden. Dazu wird ein Netzobjekt verwendet, das die Knoten und Elemente mit Indizes in zwei Attributen abspeichert. Dabei handelt es sich um zwei Maps, die die jeweiligen Indizes als Wert und Knoten bzw. Elementobjekte als Schlüssel beinhalten. Die beschriebenen Typen der Attribute werden im Weiteren als `NodeSet` und `CellSet` bezeichnet. Der Algorithmus 6 veranschaulicht die Bereitstellung von Knoten und Elementen innerhalb des Netzobjekts.

Algorithmus 6: Attribute der Netz-Struktur

```
typedef std::unordered_map<NodeIndex, dynNodeXd<float>> NodeSet;
typedef std::unordered_map<CellIndex, Cell> CellSet;

class IsoMesh{

    NodeSet m_nodes = {};
    CellSet m_Cells = {};
}
```

4.5 Lineare FEM

Die lineare FEM nutzt die geladenen Ressourcen und erstellt zunächst die Steifigkeitsmatrix des Systems. Dazu werden die Elemente des Netzes in einer Schleife durchlaufen und die symbolischen Jacobi- und B-Matrizen erstellt. Um den Rechenaufwand für die nächsten Schritte zu verringern, werden die symbolischen Einträge der beiden Matrizen vor Weiterverwendung ausmultipliziert. Anschließend werden die Koeffizienten auf sechs Nachkommastellen gerundet. Da symbolische Ausdrücke intern als Zeichenfolgen oder abstrakte Baumstrukturen (SymEngine) verwaltet werden, hängt die Rechenzeit vieler Operationen direkt von der Länge der Folgen bzw. der Komplexität der Baumstrukturen ab. Eine Vereinfachung der Ausdrücke führt daher zu einer Verminderung des Speicheraufwands und der Rechenzeit. Nach der Vereinfachung werden die B-Matrizen und Jacobi-Determinanten unter Angabe des Elementindizes als Schlüssel in zwei temporäre, netzobjektinterne Speicherstrukturen geschrieben. Das hat den Vorteil, dass die Matrizen bei späteren Berechnungen, wie die Umrechung von Dehnung und Verschiebung und die Residuenerstellung für nicht-lineare Probleme, aus dem Speicher referenziert werden können und nicht neu aufgestellt werden müssen. Die Speicher werden vorab anhand der Elementanzahl reserviert. Eine andere Lösung ist das nicht symbolische Aufstellen und Speichern der B-Matrizen für die Quadraturpunkte. Diese Lösung wäre etwas schneller, da die Substitution in die symbolischen Ausdrücke später nicht gemacht werden müsste, würde aber einen deutlich erhöhten Speicherbedarf bedeuten. Zudem würde die schnellere Berechnung vermutlich erst bei großen Systemen einen signifikanten Unterschied machen. Da es sich bei diesem Pogramm um ein Testwerkzeug handelt und die damit erwarteten Systeme eher klein sind, ist der Overhead, der durch die Substitutionen entsteht, vernachlässigbar klein. Deshalb wurde hier die flexiblere, speichereffizientere Variante gewählt. Der Algorithmus 7 verdeutlicht anschaulich die beschriebene Verwaltung der ermittelten symbolischen Tensoren.

Algorithmus 7: Verwaltung von B-Matrizen und Jacobi-Determinanten

```
for(const auto& [cellIndex, cell] : m_Cells){

    ASSERT(cell.getPrefabIndex() == currentPrefabIndex, ...);
    Jacobi = ...; jDet = Jacobi.det(); BMatrix = ...;
    simplify(jDet, BMatrix);

    m_cachedJDets.emplace(cellIndex,jDet);
    m_cachedBMats.emplace(cellIndex, BMatrix);
}
```

Für die Erstellung der Elementsteifigkeitsmatrix gibt es von diesem Ausgangspunkt zwei Möglichkeiten. Die symbolische Erstellung der K-Matrix oder die rein numerische. Bei der symbolischen Erstellung wird die K-Matrix unter Verwendung der symbolischen B-Matrix und

Jacobi-Determinante aufgestellt. Anschließend erfolgt im Rahmen der Gaußintegration die Substitution der Quadraturpunkte und Summierung der Teilmatrizen. Die rein numerische Erstellung funktioniert über die vorangestellte Substitution der Quadraturpunkte und die Erstellung der Teilmatrizen über rein numerische Größen. Für Elemente mit wenigen Quadraturpunkten und kleinen B-Matrizen wie dem Dreieckselement `CPS3` unterscheiden sich die Berechnungszeiten kaum. Spürbar wird der Unterschied bei Elementen mit mehr Quadraturpunkten und größeren B-Matrizen wie dem Würfelelement `C3D8R`. Durch die zahlreicheren symbolischen Operationen mit deutlich längeren symbolischen Termen wird die erste Option für besagte Elemente schnell ineffizient. Aus diesem Grund wurde die zweite Option gewählt. In Algorithmus 8 wird die Erstellung der Elementsteifigkeitsmatrix exemplarisch gezeigt. Zu sehen ist die Substitution der Quadraturpunkte in B-Matrix und Jacobi-Determinante, die über die rein numerischen Substitutionsergebnisse die Elementsteifigkeitsmatrix summieren.

Algorithmus 8: Exemplarische Summierung der Elementsteifigkeitsmatrix

```
for(uint8_t nodeNum = 0; nodeNum < currentCellPrefab.nNodes; nodeNum++){

    if(!subMatrix(BMatrix, substitutedBMatrix, currentCellPrefab.quadraturePoints[nodeNum])){...}
    // ...

    kCell += subsitutedBMatrix.transpose() * CMatrix * subsitutedBMatrix * substitutedJDet
        currentCellPrefab.weights[nodeNum] * (nDimensions == 2 ? m_material.t : 1);
}
```

Anschließend wird die Elementsteifigkeitsmatrix in die Gesamtsteifigkeitsmatrix assembliert. Dabei werden nur Einträge betrachtet, die in der Diagonalen oder darüber liegen. So wird nur die obere Dreiecksmatrix inklusive Diagonale der Systemsteifigkeitsmatrix befüllt. Anschließend werden die redundanten Einträge der unteren Dreiecksmatrix aus der oberen übertragen. Nach Aufstellen des rein numerischen Gleichungssystems wird dieses um die fixierten Freiheitsgrade gekürzt und mittels des Cholesky-Solvers aus der Bibliothek Eigen gelöst. Anhand der ermittelten Verschiebung werden Spannungen, Dehnungen, Elementvolumen und die Van-Mises-Vergleichsspannung elementweise ermittelt und in einen netzobjektinternen Speicher geschrieben. Der Container `CellData` erfüllt diesen Zweck. Er ist eine Datenstruktur, die alle elementspezifischen Ergebnisse abspeichert. Zudem verwaltet er Abfragen, bei der er unter Angabe von Größe und Raumrichtungsindex den entsprechenden Wert zurückgibt. Die beschriebene Struktur wird exemplarisch in Algorithmus 9 gezeigt. Die Nutzung der `CellData`-Struktur in einer Map wird dabei als `DataSet` definiert.

Algorithmus 9: Exemplarische Definition CellData-/DataSet-Struktur

```
struct CellData{

    // ...
    float getData(const MeshData& data, int globKoord) const;

    prefabIndex m_prefIdx = 0;
    Eigen::MatrixXf cellDisplacement, strain, stress, innerVariable;
    float cellVolume, vanMisesStress;
};
// Speicherstruktur im Netzobjekt
typedef std::unordered_map<CellIndex, CellData> DataSet;
```

4.6 Unsicherheitsquantifizierung

Die Unsicherheitsquantifizierung wird im Anschluss an und mit den Ergebnissen der linearen FEM berechnet. Das geschieht über das Multiplizieren von Verschiebung, Dehnung und Spannung mit den Faktoren $(1-\xi)$ und $(1-\xi^2)$. Als konkrete Werte für ξ werden dabei die positive und negative Standardabweichung eingesetzt $\xi_{1,2} = \pm\sigma$. Um die Ergebnisse der variierten Größen abzuspeichern, werden zwei `DataSets` verwendet. Zudem werden die Knoten-Koordinaten des verformten Netzes für jedes eingesetzte ξ in einem `NodeSet` gespeichert. Die Standardabweichung wird dabei anhand der im Material-Dokument übergebenen bzw. in der Benutzeroberfläche eingestellten Wahrscheinlichkeitsdichte ermittelt. Dazu wird der Gültigkeitsbereich wie beschrieben eingegrenzt, über Rejection-Sampling die vorgegebene Anzahl an Samples erstellt und aus diesen die Standardabweichung ermittelt. Das Rejection-Sampling bestimmt dabei einen zufälligen Wert ξ im Gültigkeitsbereich und den Funktionswert $pdf(\xi)$. Zudem wird ein zufälliger Wert q zwischen Null und dem Funktionsmaximum, der höchsten vorkommenden Wahrscheinlichkeitsdichte ermittelt. Liegt der Wert q oberhalb des Funktionswerts der Dichtefunktion $pdf(\xi)$, wird er abgelehnt. So wird die Dichtefunktion durch eine Probenreihe abgebildet, die nach der Dichtefunktion verteilt ist. Der Algorithmus 10 zeigt exemplarisch die Funktionsweise des beschriebenen Rejection-Samplings.

Algorithmus 10: Rejection-Sampling

```
1 void rejectionSampling(const Expression& pdensity, std::vector<float>& samples, unsigned int
  ↪ nSamples, const float& xi_min, const float& xi_max, const float& tolerance, const float&
  ↪ segmentation){
2
3     // ...
4
5     //
6     unsigned int generatedSamples = 0;
7     float xi_random, p_xi, q_random;
8
9     while(generatedSamples < nSamples){
10
11        xi_random = randFloat(leftBorder, rightBorder); p_xi = eval(*pdensity->subs({{xi,
          ↪ toExpression(xi_random)}}));
12        q_random = randFloat(0,maxP_Xi);
13
14        if(q_random < p_xi){ samples.emplace_back(xi_random); generatedSamples++; }
15    }
16 }
```

4.7 Strukturelle Analyse

Zur Berechnung der nicht-linearen Materialmodelle anhand von als Zeichenfolgen übergebenen Gleichungen wird eine strukturelle Analyse benötigt. Diese soll in der Lage sein, den Ausdruck als Abstract Syntax Tree (AST) zu repräsentieren, bekannte Symbole an entsprechender Stelle zu substituieren und die Lösung des substituierten Ausdrucks zu evaluieren. Da die SymEngine Ausdrücke nicht für Matrixsubstitutionen ausgelegt sind und die Ausdrücke intern nach den Gesetzen der Skalarrechnung umgeformt werden, wird hierfür eine Eigenimplementierung genutzt. Diese basiert auf der Verarbeitung der Ausdrücke als Zeichenfolge. Die Logik setzt sich aus einer lexikalischen, semantischen und syntaktischen Analyse zusammen. Im Rahmen der lexikalischen Analyse werden Argumente identifiziert, die semantische Analyse stellt die Rechenoperation heraus, über die diese Argumente verknüpft sind und die syntaktische Analyse

repräsentiert die Gesamtstruktur hierarchisch. Die Substitution und Evaluierung ist dabei an die syntaktische Analyse gekoppelt.

Aufgaben der strukturellen Analyse

- 1. Extrahieren der verknüpfenden Operation
- 2. Zerlegung in Argumente
- 3. syntaktische/hierarchische Interpretierung
- 4. Substitution und Evaluierung

Die Aufgaben werden auf zwei Logiken aufgeteilt, die hier als Lexer und Parser bezeichnet werden. Diese Ausdrücke stammen ursprünglich aus dem Compiler-Front-End und bezeichnen in ihrer ursprünglichen Bedeutung nur die lexikalische und syntaktische Analyse. Unter dem Lexen wird in der klassichen Compilerarchitektur das Zerlegen von Zeichenfolgen in Tokens und unter dem Parsen die syntaktische Überprüfung der Tokenfolge verstanden. Da die Aufgabenteilung beim angewandten Problem ähnlich (in Argumente zerlegen und diese verknüpfen), wenn auch nicht identisch ist, wird die Nomenklatur trotz leicht abweichender Definitionen beibehalten.

Aufteilung der strukturellen Analyse

- **Lexer**
  - Extrahieren verknüpfender Operation
  - Zerlegung in Argumente
- **Parser**
  - Syntaktische/hierarchische Interpretierung
  - Substitution und Evaluierung

4.7.1 Lexer

Der Lexer ist die Analyseeinheit, die einen gegebenen Ausdruck in die Argumente der stattfindenden makroskopischen Rechenoperation aufteilt. Dazu wird zunächst der primäre Operator ermittelt. Der Ausdruck wird dazu in einer Schleife durchlaufen und jedes Zeichen einzeln betrachtet. Handelt es sich dabei um eine öffnende oder schließende Klammer, wird das durch einen ganzzahligen Wert abgebildete Hierarchielevel um Eins erhöht oder vermindert. Wenn das Hierachielevel für ein betrachtetes Zeichen größer Null ist, springt der Lexer direkt zum nächsten Zeichen. Ist die Hierarchie Null und das Zeichen ein gültiger Operator, wird diesem eine Priorität zugeordnet und in einer Variablen gespeichert. Die Darstellung der validen Operatoren wird dabei durch eine Zeichenfolge definiert. Die Position des Operators in der Zeichenfolge ist seine Priorität. Die verwendete Zeichenfolge ist `+-^`. Je weiter vorne der Operator steht und je geringer seine Priorität damit ist, desto eher neigt er dazu im Vergleich zu den anderen Operatoren die makrokopische Operation vorzugeben. Ist die Priorität des aktuellen Operators niedriger als die bereits abgespeicherte, wird sie mit der des betrachteten Operators überschrieben. Fällt die gespeicherte Priorität auf Null, wird die Schleife direkt abgebrochen, da es keinen Operator gibt, der eine makroskopischere Operation als die gefundene vorgibt. Nach Abbruch der Schleife wird der über die abgespeicherte Prioriät identifizierte primäre Operator zurückgegeben. Wird kein

Operator gefunden, wird überprüft, ob ein Funktionsname auftaucht. Als Funktionsname gilt dabei ein Ausdruck, der direkt vor einem Klammerpaar steht und keine Operatoren und Formatierungszeichen enthält. Taucht ein solcher auf, wird dieser als Operator zurückgegeben. Die Identifikation des makroskopischen Operators wird exemplarisch im Algorithmus 11 dargestellt.

Algorithmus 11: exemplarische Ermittlung der makroskopischen Operation

```
1 std::string getDominantOperator(const std::string& expr){
2
3     // ...
4     for(size_t i = 0; i < expr.size(); i++) {
5
6         if(expr[i] == '('){ hierarchie++; continue; }
7         else if(expr[i] == ')'){ hierarchie--; continue; }
8
9         if (hierarchie != 0) { continue; }
10
11        size_t pos = operators.find(expr[i]);
12        if (pos < dominantOperator){ dominantOperator = pos; }
13        if(dominantOperator == 0){ break; }
14    }
15
16    if(dominantOperator < operators.size()){ return operators[dominantOperator]; }
17    else if(functionNameFound){ funtionname = ...; return functionName; }
18    else{ return ""; }
19 }
```

Anschließend wird der Ausdruck nach dem makroskopischen Operator in Argumente aufgeteilt. Das geschieht über das Teilen der Zeichenfolge nach dem Operator. Ist der Operator ein Funktionsname, sind die Argumente der nach Kommata geteilte Ausdruck, der im Klammerpaar hinter dem Funktionsnamen steht.

4.7.2 Parser

Der Parser ist die Analyseeinheit, die den Lexer für eine syntaktische Analyse und die Substitution verwendet. Zudem ist der Parser rekursiv programmiert, das bedeutet, er gibt eine Verknüpfung von Eigenaufrufen für die einzelnen vom Lexer erhaltenen Argumente zurück. Die Verknüpfung findet dabei über die makroskopische Operation statt. Wird vom Lexer kein Funktionsname oder Operator gefunden, gibt der Parser eine Konstante oder den Substitutionswert des Symbols zurück. Kann er den verbliebenen Ausdruck nicht in eine Konstante konvertieren und keinen Substitutionswert für den gegebenen Ausdruck finden, gibt er den Ausdruck als skalares Symbol zurück. Die Übergabe der Substitutionswerte erfolgt dabei über eine Substitutionstabelle. Zudem kann der Parser Argumente über das Vorzeichen `-` negieren. Als Exponenten können zusätzlich zu Ganz- und Fließkommazahlen auch `t`, `T`, `-t` und `-T` für die Transponierte oder Transponierte der Inverse angegeben werden. Aufgrund der Lexerarchitektur gibt es jedoch mehrere Einschränkungen für den Eingabeausdruck. Dieser darf keine Leerzeichen und als Operatoren nur `+`, `-` und `^` enthalten. Das bedeutet, dass Subtraktion und Division über die vorgegebene Operationsauswahl und negierte Argumente dargestellt werden müssen. Bei Erweiterung der Operatorenauswahl können nur Operatoren implementiert werden, die über ein Zeichen verfügen. Das Verhalten für konkrete Funktionsnamen wird ebenfalls vom Parser deklariert. Damit ist die Funktionsauswahl beliebig erweiterbar. Algorithus 12 zeigt die Funktionsweise des Parsers am Beispiel der Addition. Dabei werden die Rückgaben der rekursiven Parseraufrufe für die Argumente der Addition aufsummiert und zurückgegeben. Die Funktionsweise der Parserfunktionen wird am Beispiel der Einheitsmatrixerstellung `Identity` verdeutlicht,

die für die Funktionsargumente vor Weiterverwendung ebenfalls den Parser rekursiv aufruft und das Ergebnis zu einem Skalar konvertiert.

Algorithmus 12: exemplarische Parserlogik

```
SymEngine::DenseMatrix evalSymbolicMatrixExpr(std::string expr, const
  ↪ std::unordered_map<std::string, SymEngine::DenseMatrix>& symbolTable){

    // ...
    const std::string dominantOperator = getDominantOperator(expr);
    std::vector<std::string> args = lexExpression(expr, dominantOperator);

    if(dominantOperator == "+"){

        SymEngine::DenseMatrix result = evalSymbolicMatrixExpr(args[0], symbolTable);
        for(size_t argIdx = 0; argIdx < args.size(); argIdx++){

            if(argIdx == 0){ continue; }
            result += evalSymbolicMatrixExpr(args[argIdx],symbolTable);
        }

        return result;
    }
    // ...
    else if(dominantOperator == "Identity"){

        size_t matrixSize = convert(evalSymbolicMatrixExpr(args[0], symbolTable));
        SymEngine::DenseMatrix result(matrixSize, matrixSize); SymEngine::eye(result);
        return result;
    }
    else if(symbolTable.contains(expr)) { return symbolTable.at(expr); }
    else if(dominantOperator != ""){ // hier neue Funktionen deklarieren }
    else { return SymEngine::DenseMatrix(1,1, {toExpression(expr)}); }
}
```

4.8 Newton-Raphson-Solver

Der Newton-Raphson-Solver, der zur Nullstellensuche des Residuums verwendet wird, ist eine Eigenimplementierung. Dieser nimmt ein Residuum und einen Vektor mit symbolischen Einträgen entgegen. Der Vektor enthält die Symbole nach denen das Residuum abgeleitet wird und für die die Substitution des Ergebnisvektors ins Residuum und die Jacobi-Matrix erfolgt. Die Jacobi-Matrix wird aufgrund einer nichtvorhandenen Sparse-Matrix-Funktionalität der SymEngine durch eine Liste an Triplets mit Nicht-Null-Einträgen dargestellt. Dabei enthält jedes Triplet die Eintragsindizes und den symbolischen Ausdruck. Nach Aufstellen der Jacobi-Matrix als Tripletliste mit symbolischen Einträgen wird die Iteration gestartet. Dabei werden die Lösungsvektoren über den Symbolvektor in Residuum und Jacobi-Matrix substituiert und der Ergebnisvektor angepasst. Nach dem Abgleich der Norm und Toleranz wird entweder der nächste Iterationsschritt gestartet oder die Schleife abgebrochen und der aktuelle Lösungsvektor zurückgegeben.

4.9 Nicht-lineare FEM

Für die Berechnungen der nicht-linearen FEM-Probleme werden Zeitschrittsimulationen durchgeführt. Dazu werden vorab der Simulationszeitraum, die Schrittweite und die Schrittanzahl

deklariert. In einer Schleife, die über alle Zeitschritte iteriert wird dann jeweils die innere Variable aktualisiert, das Residuum aufgestellt und die Verschiebung bestimmt. Um die Simulationsergebnisse abzuspeichern und innerhalb der Simulation auf die Werte aus dem voherigen Schritt zugreifen zu können, werden mehrere Speicher benötigt. Innerhalb der Attribut-Map `m_simulationFrames` werden elementspezifische Informationen für jeden Simulationsschritt unter einer Simulationsschritt ID abgespeichert. Der Wert ist dabei eine Datenstruktur, die Verschiebung, ein `DataSet` mit elementspezifischen Informationen und das `NodeSet` des verformten Netzes speichert. Die Werte der inneren Variablen an den Quadraturpunkten werden in einem Vektor aufbewahrt. Dieser Aufbau ermöglicht den Zugriff auf die Werte der inneren Variablen aus dem letzten Schritt und weiterer Größen wie der Verschiebung, Dehnung und Spannung aus allen vorhergegangenen Schritten. Die beschriebene Speicherstruktur ist in Algorithmus 13 aufgezeigt.

Algorithmus 13: Speicherstruktur für Zeitschrittsimulation

```
1  struct SimulationFrame{
2
3      // ...
4
5      DataSet cellDataSet;
6      Eigen::MatrixXf displacement;
7      NodeSet deformedNodes;
8  };
9
10 class FEMModel{
11     std::vector<SimulationFrame> m_simulationFrames = {};
12 }
```

Während der Simulation wird über die Simulationsschritte iteriert. Dabei werden für jeden Schritt alle Elemente durchlaufen und die Zuordnung der lokalen Freiheitsgrade auf die symbolischen, globalen Verschiebungseinträge ermittelt. Der globale Verschiebungseintrag wird dabei über das Symbol `ui` mit dem globalen Freiheitsgrad `i` in den Vektor der lokalen Verschiebungen eingetragen. Anstelle fixierter Freiheitsgrade wird Null in den Vektor geschrieben. Der Vektor ist dabei Bestandteil einer Substitutionstabelle, die für die Tensor-Substitution genutzt wird. Diese enthält zudem weitere System-, Schritt- und Elementspezifische Größen, die entsprechend ihres Gültigkeitsbereichs immer wieder aktualisiert werden und zur Substitution zur Verfügung stehen. Mittels dieser Größen, dem symbolischen Verschiebungsvektor des Elements und den gespeicherten Werten der inneren Variablen aus dem letzten Schritt, wird die innere Variable aktualisiert und ebenfalls in die Substitutionstabelle und den Speicher für die Werte der inneren Variablen geschrieben. Dabei bleiben die aktualisierten Werte von den symbolischen Einträgen des globalen Verschiebungsvektors abhängig. Anschließend wird mit dem aus der Materialdatei übergegebenen Spannungsansatz das Elementresiduum summiert. Dieses wird im Anschluss in das Systemresiduum assembliert. Algorithmus 14 zeigt die Ablaufstruktur der Elementresiduenerstellung.

Algorithmus 14: exemplarische Elementresiduenerstellung

```
for(size_t stepIdx = 0; stepIdx < m_simulationSteps; stepIdx++){

    bool firstFrame = stepIdx == 0;
    for(const auto& [cellIdx, cell] : m_isoMesh.getCells()){

        symbolTable["uCell"] = ...;

        for(size_t quadPoint = 0; quadPoint < r_pref.nQuadraturePoints; quadPoint++){

            // ...
            symbolTable[innerVariablePreviousFrame] = innerVariableContainer[storePosition];
            symbolTable[innerVariableCurrentFrame] = firstFrame ?
            ↪  symbolTable[innerVariablePreviousFrame] :
                evalSymbolicMatrixExpr(mat.evolutionEquation, symbolTable);

            symbolTable["sigma"] = evalSymbolicMatrixExpr(mat.stressApproach, symbolTable);
            cellResidual += evalSymbolicMatrixExpr("BˆT*sigma*jDet*w*t",symbolTable);
        }
    }
}
```

Nach der Assemblierung wird das Residuum gekürzt. Analog dazu wird der globale Verschiebungsvektor mit den symbolischen Einträgen `ui` aufgestellt und ebenfalls gekürzt. Mit diesen beiden Größen ermittelt der Newton-Raphson-Solver die Nullstelle des Residuums. Im Anschluss werden die Zielgrößen ermittelt und in den Zeitschrittergebnisspeicher geschrieben. Im Nachgang werden die Werte der inneren Variablen, die noch von den symbolischen Einträgen des globalen Verschiebungsvektors abhängig sind, mit der Lösung für den Verschiebungsvektor substituiert. Damit kann der nächste Schleifendurchlauf begonnen werden.

4.10 Rendering

Die Ergebnisvisualierung des Werkzeugs erfolgt über ein Rendering mit der Bibliothek raylib. Sowohl für zwei- als auch dreidimensionale Systeme werden dabei zunächst die Knoten aller zu zeichnenden Netze durchlaufen und das Zentrum und die Ausdehnung des Punkteclusters ermittelt. Für das zweidimensionale System werden im Anschluss ein Skalierungsfaktor und ein Offset bestimmt, über den das gesamte Punktecluster in den zur Verfügung stehenden Fensterbereich transformiert werden kann. Für dreidimensionale Systeme wird ein global gültiges Netz angelegt und für die erwartete Elementvorlage aufgesetzt. Über das Ersetzen der Vertex-Koordinaten lassen sich die verschiedenen Elemente dann über das gleiche Netz rendern. Dabei wird das Zeichnen der ausgefüllten Elemente nur für ein ausgewähltes Netz durchgeführt, alle anderen Netze werden als Drahtgitter gezeichnet. In Abbildung 1 ist die Visualisierung eines 3D FEM-Problems gezeigt.

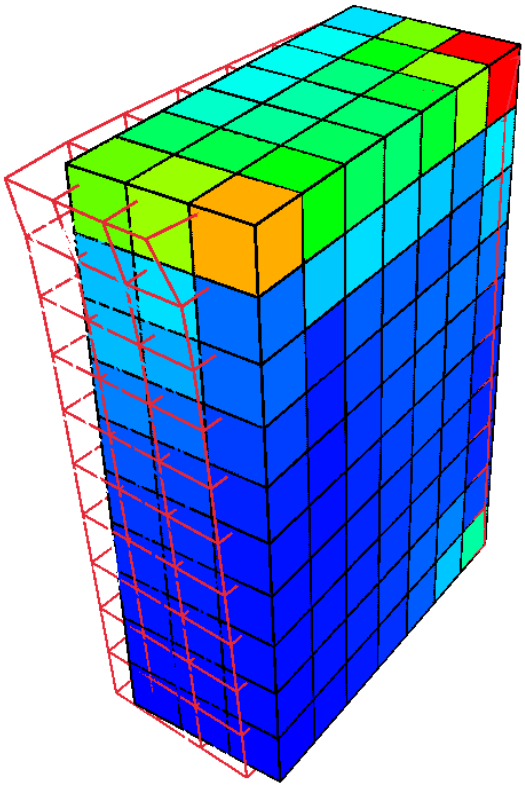


Abbildung 1: 3D-Rendering

Bei der Visualisierung der Zeitschrittsimulation wird als verformtes Netz jeweils das zum aktuellen Zeitschritt gehörende Netz gezeichnet. Der aktuelle Zeitschritt wird dabei über einen Zähler repräsentiert, der hochgezählt wird, sobald eine Schrittweite verstrichen ist. Mit diesem Zähler als Schlüssel bzw. Zeitschritt-Index wird auf das verformte Netz aus dem Zeitschrittergebnisspeicher zugegriffen. Die zweidimensionale Anzeige ist statisch und kann nicht manipuliert werden, in der dreidimensionalen Anzeige sind allerdings freie Kamerafahrten möglich. Dazu sind drei verschiedene Kameraführungen implementiert worden.

Kameraführungen

- Normal-Planar
- Orbital
- First-Person

Die Normal-Planar-Führung ermöglicht das Verschieben der Kamera innerhalb der Ebene, die senkrecht zur aktuellen Blickrichtung steht. Dadurch lässt sich die Kamera parallel zur betrachteten Ebene verschieben, ohne die Blickrichtung zu verändern. Die Orbital-Führung erlaubt es, die Kamera auf einer Kugelbahn um das betrachtete Objekt zu bewegen. Dabei bleibt der Fokus immer im Zentrum des betrachteten Modells. Die First-Person-Führung bietet eine freie Bewegung und Blickrichtungsänderung im Raum. Zudem kann in allen Kameraführungen gescrollt werden. Die Tabelle 5 zeigt die Tastenkombinationen mit denen Aktivierung, Navigation verschiedener Kameraführungen und weitere Anzeigefunktionen durchgeführt werden können.

Tabelle 5: Kamerasteuerung und Bewegungsfunktionen

Tasten	Aktion
l hold	Aktivierung Normal-Planar Kamera
r hold	Aktivierung Orbital Kamera
l+r hold	Aktivierung First-Person Kamera
w/a/s/d/shift/space	Bewegung in First-Person Kamera
scroll	Bewegung in Blickrichtung
c	toggle Cursor

4.11 Benutzeroberfläche

Die Benutzeroberfläche bietet den Modellimportdialog, die Bearbeitung der Wahrscheinlichkeitsdichtefunktionen und Dialoge zur Konfiguration des Renderings an. Abbildung 2 zeigt die Benutzeroberfläche. Dabei ist bereits ein System importiert, gelöst und unsicherheitsquantifiziert worden.

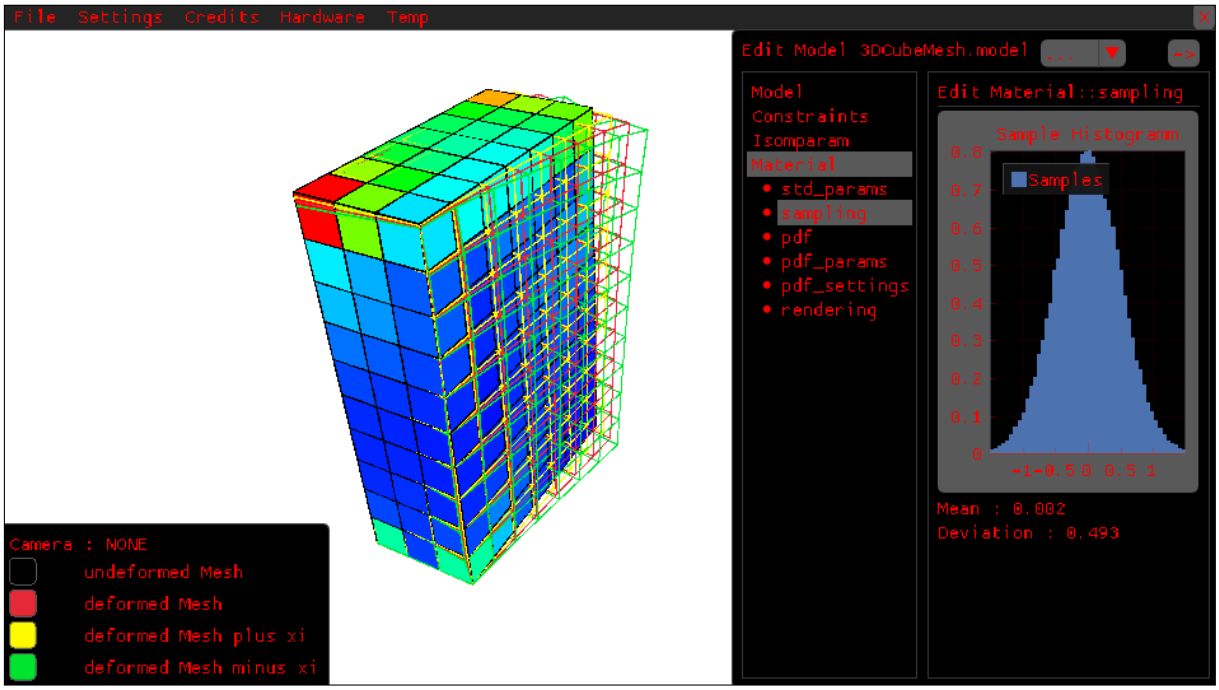


Abbildung 2: Übersicht über die vollständige Oberfläche

Die Oberfläche besteht aus drei Bereichen. Über die Menüleiste werden unter anderem Modellimports, Hardwareübersicht und die zentrale Löschung aller Bytesequenzspeicher ermöglicht. In der linken unteren Ecke ist eine Legende abgebildet, die den Farben der gezeichneten Drahtgitter ein Netz zuweist. Zudem lassen sich die Farben über Farbdialoge direkt anpassen. Auf der rechten Seite befindet sich eine ausklappbare, vertikale Registeransicht mit verschiedenen Tabs. Über dieses lassen sich das geladene Modell verwalten und die Wahrscheinlichkeitsdichtefunktionen bearbeiten. Dabei können Grenzen abgesteckt, Parameter definiert und variiert, Gleichungen importiert und ein Histogramm des Samplings angezeigt werden. In Tabelle 6 sind diese Optionen und die Tabs, unter denen sie angeboten werden aufgeführt.

Tabelle 6: Tab Übersicht Pdf Konfiguration

Tab	Möglichkeiten
Material/std_params	Anpassung Standardparameter (E,v,t)
Material/sampling	Anzeige des Sampling Histogramms
Material/pdf	Eingabe und Import der Pdf-Funktion
Material/pdf_params	Variation und Bearbeitung des Parametersatzes
Material/pdf_settings	Einstellwerte für Abtastvorgang
Material/rendering	Split Screen für 2D-Renderings

In Abbildung 3 ist der Dialog gezeigt, über den der selbst gewählte Parametersatz variiert und bearbeitet werden kann. Die importierte Dichtefunktion im Beispiel ist die Normalverteilung, für die μ und σ variiert werden können. Zudem wird die Möglichkeit geboten, Parameter händisch zu erstellen oder zu löschen.

Abbildung 3: vertikale Registeransicht

5 Schlussteil und Ausblick

Aufgrund des oft hohen Modellierungs- und Einarbeitungsaufwand, der mit der Nutzung etablierter Softwarelösungen einhergeht, sind Voruntersuchungen im Bereich der Unsicherheitsquantifizierung und simpler nicht-linearer Materialmodelle oft zeitintensiv. Um diesen Aufwand zu reduzieren, wurde im Rahmen dieser Arbeit ein FEM-Werkzeug entwickelt, das gezielt für Voruntersuchungen in diesen Bereichen konzipiert ist.
Das Werkzeug erfüllt diese Anforderungen und stellt eine effiziente Möglichkeit dar, Wahrscheinlichkeitsdichtefunktionen für die Unsicherheitsquantifizierung und simple nicht-lineare Materialmodelle zu erproben. Durch die dateigestütze Modellbeschreibung wird ein geringer Einarbeitungsaufwand und selbsterklärende Modellierung gewährleistet. Zudem sind Funktionalitäten schnell und dynamisch implementier- und erweiterbar.
Zur weiteren Verbesserung der Effizienz und Reduzierung der Rechenzeit gibt es mehrere Optionen. Potential bietet die Parallelisierung innerhalb der Solver und algebraischen Tensoroperationen durch die Verwendung von GPU basierten Lösungen wie CUDA oder Computeshadern und Multithreading. Zusätzlich kann das Rendering mit Frustum-Culling betrieben werden, sodass nur noch sichtbare, nicht verdeckte Flächen gerendert werden. Um Benutzeroberfläche und Berechnung gleichzeitig betreiben zu können bietet sich die asynchrone Verwaltung von Lade- und Berechnungsmechanismen an.

Literatur

[1] M. Wagner. *Lineare und nichtlineare FEM*. Springer Fachmedien Wiesbaden GmbH, 2022.

[2] Dustin Roman Jantos. „Innovative Ansätze zur Topologie- und Materialoptimierung basierend auf thermodynamischen Prinzipien“. Diss. Ruhr-Universität Bochum, 2019.

[3] Hendrik Geisler. „Uncertainty quantification for inelastic materials and structures: Time-separated stochastic mechanics“. English. Diss. Leibniz University Hannover, 2025. ISBN: 9783941302525.

[4] Wing Kam Liu, Ted Belytschko und Anil Mani. „Random Field Finite Element“. In: *International Journal for Numerical Methods in Engineering* 23.10 (1986), S. 1831–1845. DOI: 10.1002/nme.1620231004. URL: https://doi.org/10.1002/nme.1620231004.

[5] Mario Kunzemann. „Simulationsgestützte Charakterisierung viskoelastischer Materialien“. Masterarbeit. Johannes Kepler Universität Linz, 2022. URL: https://epub.jku.at/download/pdf/8045548.pdf.

[6] Dassault Systèmes. *Abaqus 2024 Documentation*. Zugriff am 5. Juli 2025. 2024. URL: https://help.3ds.com/2024/english/dssimulia_established/SIMACAEEXCRefMap/simaexc-c-inpfileintro.htm.

[7] *JSON Introduction*. Zugriff am 05. Juli 2025. 2025. URL: https://www.w3schools.com/js/js_json_intro.asp (besucht am 05. 07. 2025).

[8] Assimp Contributors. *Assimp issue #2882: Support for .inp format*. https://github.com/assimp/assimp/issues/2882. Zugriff am 7. Juli 2025. 2023. URL: https://github.com/assimp/assimp/issues/2882.

[9] Niels Lohmann. *JSON for Modern C++*. Zugriff am 05. Juli 2025. 2025. URL: https://github.com/nlohmann/json (besucht am 05. 07. 2025).

[10] Antony Borodin und Boost C++ Libraries. *Boost.PFR: Precise Function Reflection*. https://www.boost.org/doc/libs/release/doc/html/boost_pfr.html. Zugriff am 7. Juli 2025. 2023. URL: https://www.boost.org/doc/libs/release/doc/html/boost_pfr.html.

[11] Eigen Developers. *Eigen: C++ template library for linear algebra*. Zugriff am 7. Juli 2025. 2023. URL: https://eigen.tuxfamily.org/.

[12] SymEngine Contributors. *SymEngine: Fast symbolic manipulation library*. Zugriff am 7. Juli 2025. 2023. URL: https://symengine.org/.

[13] Ray San Pedro. *raylib: A simple and easy-to-use library to enjoy videogame programming*. Zugriff am 7. Juli 2025. 2023. URL: https://www.raylib.com/.

[14] Omar Cornut. *Dear ImGui: Bloat-free Immediate Mode Graphical User interface for C++*. Zugriff am 7. Juli 2025. 2023. URL: https://github.com/ocornut/imgui.

[15] Chandler Prall. *rlImGui: ImGui binding for raylib*. Zugriff am 7. Juli 2025. 2022. URL: https://github.com/chandlerprall/rlImGui.

[16] Evan Pezent. *ImPlot: Immediate Mode Plotting for Dear ImGui*. https://github.com/epezent/implot. Online; accessed 2025-07-07. 2021.

[17] Andrey Kalinin. *ImFileBrowser: Simple file dialog for Dear ImGui*. https://github.com/AirGuanZ/imgui-filebrowser. Online; accessed 2025-07-07. 2019.