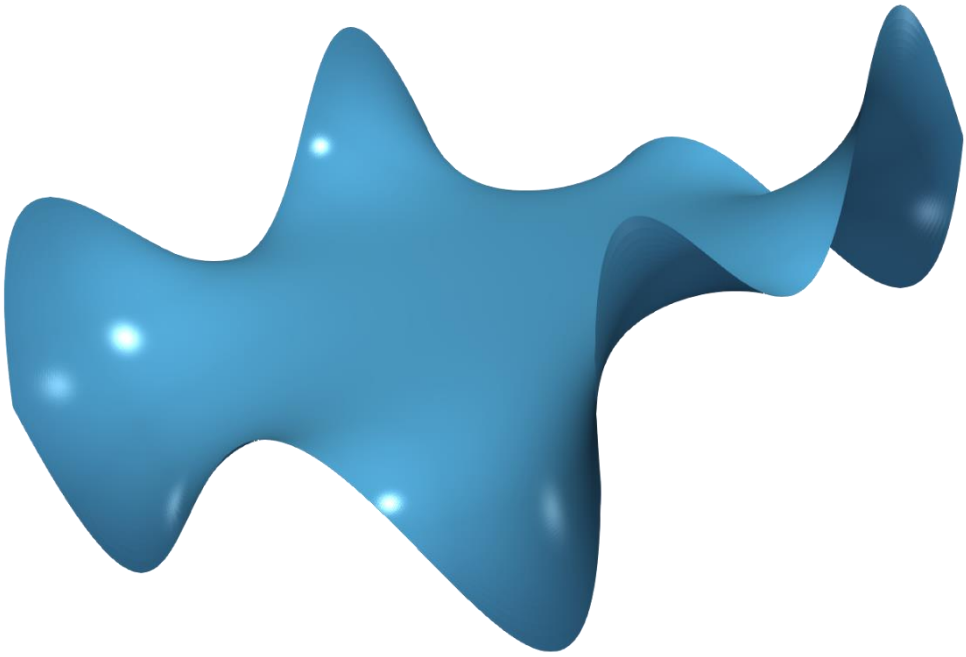
minsurf

Analyse- und Entwurfsdokument



Lukas Heyn, Laurens Lueg, Donat Weniger
Matrikelnummern: 367833, 367426, 371614
Mail: [lukas.heyn|laurens.lueg|donat.weniger]@rwth-aachen.de

Inhaltsverzeichnis

1 Vorwort

1.1 Aufgabenstellung und Struktur des Dokuments

Dieses Dokument beschreibt Analyse und Entwurf des Programmes *minsurf*, welches zur Berechnung von Minimalflächen über ein Einheitsquadrat konzipiert ist. In Kapitel 2 dieses Dokuments wird die Analyse des Programms beschrieben, insbesondere die Benutzeranforderungen aus der Spezifikation und die Anwendungsfallanalyse, inklusive entsprechender UML-Diagramme. Die Begriffsanalyse dient zur Beschreibung der mathematischen Hintergründe und Lösungsmethoden des Problems, sowie die sich daraus ergebenden Klassenkandidaten.

Kapitel 3 behandelt den Entwurf. Die Klassenstruktur wird mithilfe von UML-Klassendiagrammen beschrieben. Weiterhin wird die Dynamik des Programms durch ein Sequenzdiagramm dargestellt.

Kapitel 4 beschreibt alle für den Benutzer essenziellen Informationen, insbesondere zur Installation und Benutzung des Programms. Eine detaillierte Beschreibung des Codes sowie Informationen zu Tests und Parallelisierung befinden sich in Kapitel 5. Schließlich folgt in Kapitel 6 das Literaturverzeichnis und in Kapitel 7 der gesamte Quellcode.

1.2 Projektmanagement

Die Aufgaben während der Entwicklung wurden nicht zwingend auf die Gruppenmitglieder aufgeteilt, viele Probleme wurden stattdessen in Teamarbeit zusammen gelöst. Die folgende Auflistung beschreibt also die grobe Aufgabenteilung, die sich während der Entwicklung entsprechend der Fähigkeiten und Vorlieben der Teammitglieder ergeben hat.

- Lukas Heyn: Programmierung
- Donat Weniger: Numerik, Dokumentation
- Laurens Lueg: Visualisierung, Dokumentation

1.3 Lob und Kritik

An dieser Stelle würden wir gerne unseren Betreuern Klaus Leppkes, Johannes Lotz, Sandra Wienke und Julian Miller für ihre sehr hilfreiche Beratung und Expertise danken. Insbesondere danken wir auch Uwe Naumann, der uns kulanterweise das Vorziehen dieses Projekts genehmigt hat.

2 Analyse

2.1 Anforderungsanalyse

2.1.1 Benutzeranforderungen

Im Auftragsschreiben wurde die Entwicklung einer Software zur Berechnung von Minimalflächen über das Einheitsquadrat für vom Nutzer vorgegebene Randwerte angefordert. Diese Anforderungen sind durch die Anforderungsspezifikation verbindlich und präzise beschrieben. Auftraggeber und Entwickler haben die Spezifikation unterzeichnet. In den folgenden Kapiteln sind die Anforderungen gemäß der Spezifikation sowie die Anwendungsfälle beschrieben.

2.1.1.1 Zweck und Ziele des Produkts

Das Produkt soll minimale Flächen über das Einheitsquadrat berechnen. Die Randwerte sind frei wählbar.

2.1.1.2 Nutzer des Produkts

Der Nutzer des Produkts benötigt keine Kenntnisse über die zugrundeliegenden mathematischen Fragestellungen.

2.1.1.3 Dokumentation

Das Programm ist durch ein Analyse- und Entwurfsdokument dokumentiert. Zusätzlich befasst sich ein Abschnitt mit *openMP*. Insbesondere werden

- Laufzeitvergleiche durchgeführt,
- die Skalierung des Programms festgestellt,
- die Teile des Codes hervorgehoben, für die die meiste Zeit aufgewendet werden muss.

2.1.2 Anwendungsfallanalyse

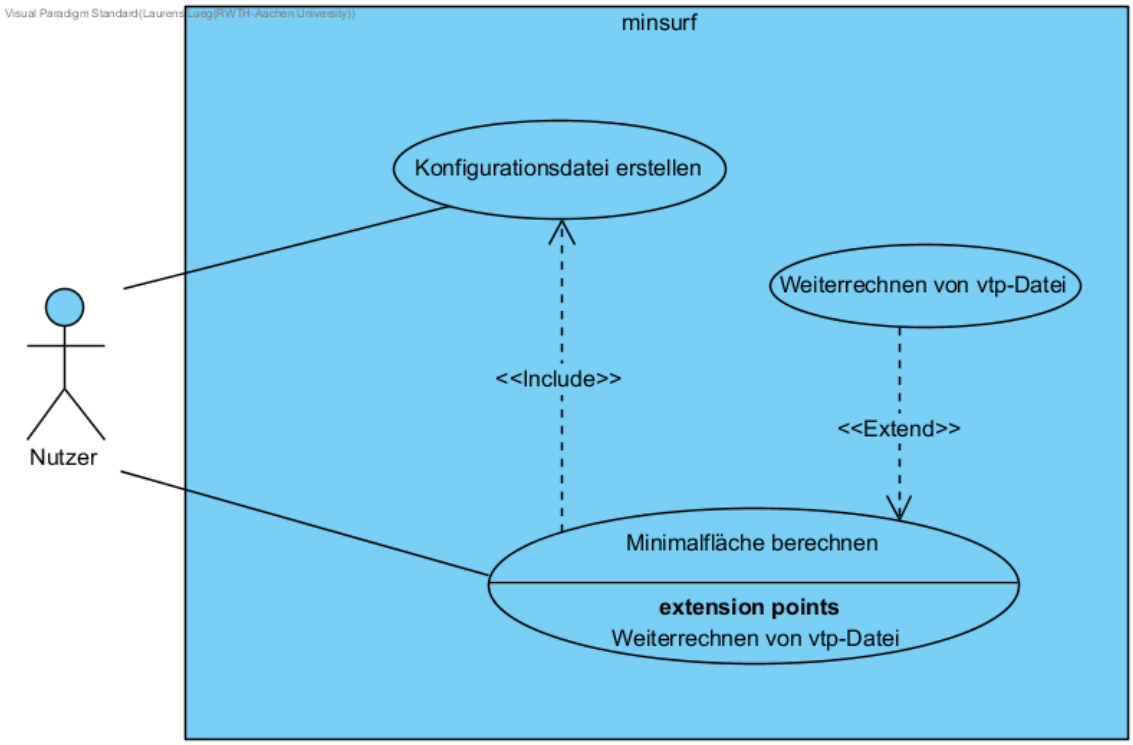


Abbildung 1: Anwendungsfalldiagramm für „minsurf“

2.1.2.1 Beschreibung der Anwendungsfälle:

1. Berechnung der Minimalfläche anhand von Randbedingungen
    - *Ziel:* Ausgabe einer in *paraview* visualisierbaren Datei, welche die entsprechende Minimalfläche darstellt.
    - *Einordnung:* Hauptfunktion
    - *Vorbedingung:* Das System entspricht den Anforderungen. Eine vollständige Konfigurationsdatei liegt vor.
    - *Nachbedingung:* Das Programm hat eine Datei zur Visualisierung der korrekten Minimalfläche ausgegeben. Das Programm ist beendet
    - *Nachbedingung im Fehlerfall:* Eine Fehlermeldung wird ausgegeben/ der Nutzer kann per Eingabe das weiter Vorgehen festlegen.
    - *Hauptakteur:* Nutzer
    - *Nebenakteure:* ---
    - *Auslöser:* Der Nutzer möchte die Minimalfläche über das Einheitsquadrat bei den in der Konfigurationsdatei bestimmten Randbedingungen errechnen und darstellen.
    - *Standardablauf:*
        (1) UC „Konfigurationsdatei erstellen“
        (2) Der Nutzer führt das Programm über die Kommandozeile aus.
        (3) Das Programm generiert die entsprechende Datei und wird beendet.

- *Verzweigungen:*

  (3a) Die eingegebenen Randbedingungen oder Parameter sind syntaktisch unzulässig
  - Eine entsprechende Fehlermeldung wird durch das System ausgegeben.
  - Das Programm wird beendet.
  - Gehe zu (1).

  (3b) Die eingegebenen Randbedingungen oder Parameter liegen außerhalb der Anforderungen, sodass Genauigkeitsvorgaben nicht garantiert werden können

  - Das System gibt eine Warnung aus.
  - Der Nutzer kann per Kommandozeilendialog entscheiden, ob das Programm beendet oder fortgeführt werden soll.

  (3c) Die Anzahl der Iteration ist ein Vielfaches des Sicherheitsoutput-Integers

  - Das System generiert eine Outputdatei.

  (3d) Das Lösungsverfahren konvergiert nur langsam.

  - Das System gibt eine Warnung aus.
  - Der Nutzer kann entscheiden, ob der Vorgang fortgesetzt oder abgebrochen werden soll.

  (3e) Die maximale Iterationsanzahl wird überschritten.

  - Eine Ausgabedatei wird erstellt.
  - Das Programm wird beendet.

  (3f) Das Lösungsverfahren konvergiert nicht.

  - Eine Meldung wird ausgegeben.
  - Das Programm wird beendet.

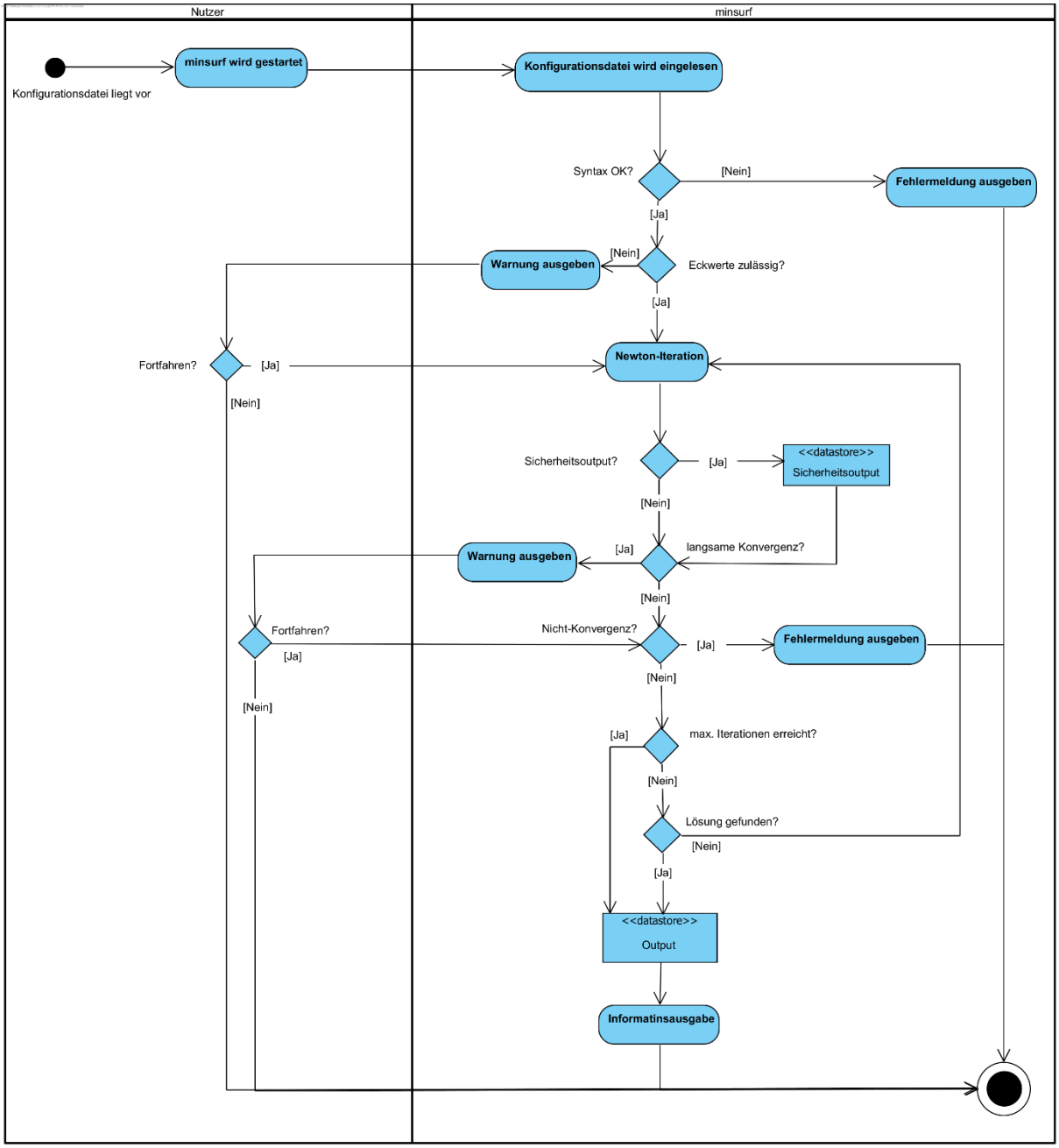


Abbildung 2: Aktivitätsdiagramm UC1

2. Konfigurationsdatei erstellen
    - *Ziel:* Eine an die Wünsche des Nutzers angepasste, korrekte Konfigurationsdatei wird erstellt.
    - *Einordnung:* Nebenfunktion
    - Vorbedingung: Der Nutzer befindet sich im Programmverzeichnis und verfügt über einen Texteditor.
    - *Nachbedingung:* Der Nutzer befindet sich im Programmverzeichnis. Die Konfigurationsdatei wurde erstellt.
    - *Nachbedingung im Fehlerfall:* --
    - *Hauptakteure:* Nutzer
    - *Nebenakteure:* --
    - *Auslöser:* Der Nutzer will die Parameter zur Berechnung der Minimalfläche anpassen.
    - *Standardablauf:*
        (1) Der Nutzer öffnet die Konfigurationsdatei mit einem Texteditor.
        (2) Der Nutzer fügt anhand der Dokumentation die gewünschten numerischen Parameter und Randwerte ein. Der Nutzer schließt und speichert die Datei.
    - *Verzweigungen:*

      (1a) Der Nutzer benutzt die mitgelieferte Beispiel-Konfigurationsdatei.

3. Weiterrechnen von *vtp*-Datei
    - *Ziel:* Ausgabe einer in *paraview* visualisierbaren Datei, welche die entsprechende Minimalfläche darstellt.
    - *Einordnung:* Hauptfunktion
    - *Vorbedingung:* Das System entspricht den Anforderungen. Eine Konfigurationsdatei sowie eine von *minsurf* (UC 1) generierte *vtp*-Datei liegt vor.
    - *Nachbedingung:* Das Programm hat eine Datei zur Visualisierung der korrekten Minimalfläche ausgegeben. Das Programm ist beendet
    - *Nachbedingung im Fehlerfall:* Eine Fehlermeldung wird ausgegeben/ der Nutzer kann per Eingabe das weiter Vorgehen festlegen.
    - *Hauptakteure:* Nutzer
    - *Nebenakteure:* ---
    - *Auslöser:* Der Nutzer möchte die Minimalfläche über das Einheitsquadrat ausgehend von einer vorherigen Ausgabe von *minsurf* (evtl. von Sicherheits-Output) berechnen und visualisieren.
    - *Standardablauf:*
        (1) UC „Konfigurationsdatei bearbeiten“
        (2) Der Nutzer führt das Programm über die Kommandozeile aus.
        (3) Das Programm generiert die entsprechende Datei und wird beendet.
    - *Verzweigungen:*
      (3a) Die eingegebenen, den Löser betreffenden Parameter sind syntaktisch unzulässig.
        - Eine entsprechende Fehlermeldung wird durch das System ausgegeben.
        - Das Programm wird beendet.

- Gehe zu (1)

(3b) Die mitgegebene Datei hat ein unzulässiges Format.

- Das System gibt eine Warnung aus.
- Das Programm wird beendet.

(3c) Die Anzahl der Iteration ist ein Vielfaches des Sicherheitsoutput-Integers

- Das System generiert eine Outputdatei.

(3d) Das Lösungsverfahren konvergiert nur langsam.

- Das System gibt eine Warnung aus.
- Der Nutzer kann entscheiden, ob der Vorgang fortgesetzt oder abgebrochen werden soll.

(3e) Die maximale Iterationsanzahl wird überschritten.

- Eine Ausgabedatei wird erstellt.
- Das Programm wird beendet.

(3f) Das Lösungsverfahren konvergiert nicht.

- Eine Meldung wird ausgegeben.
- Das Programm wird beendet.

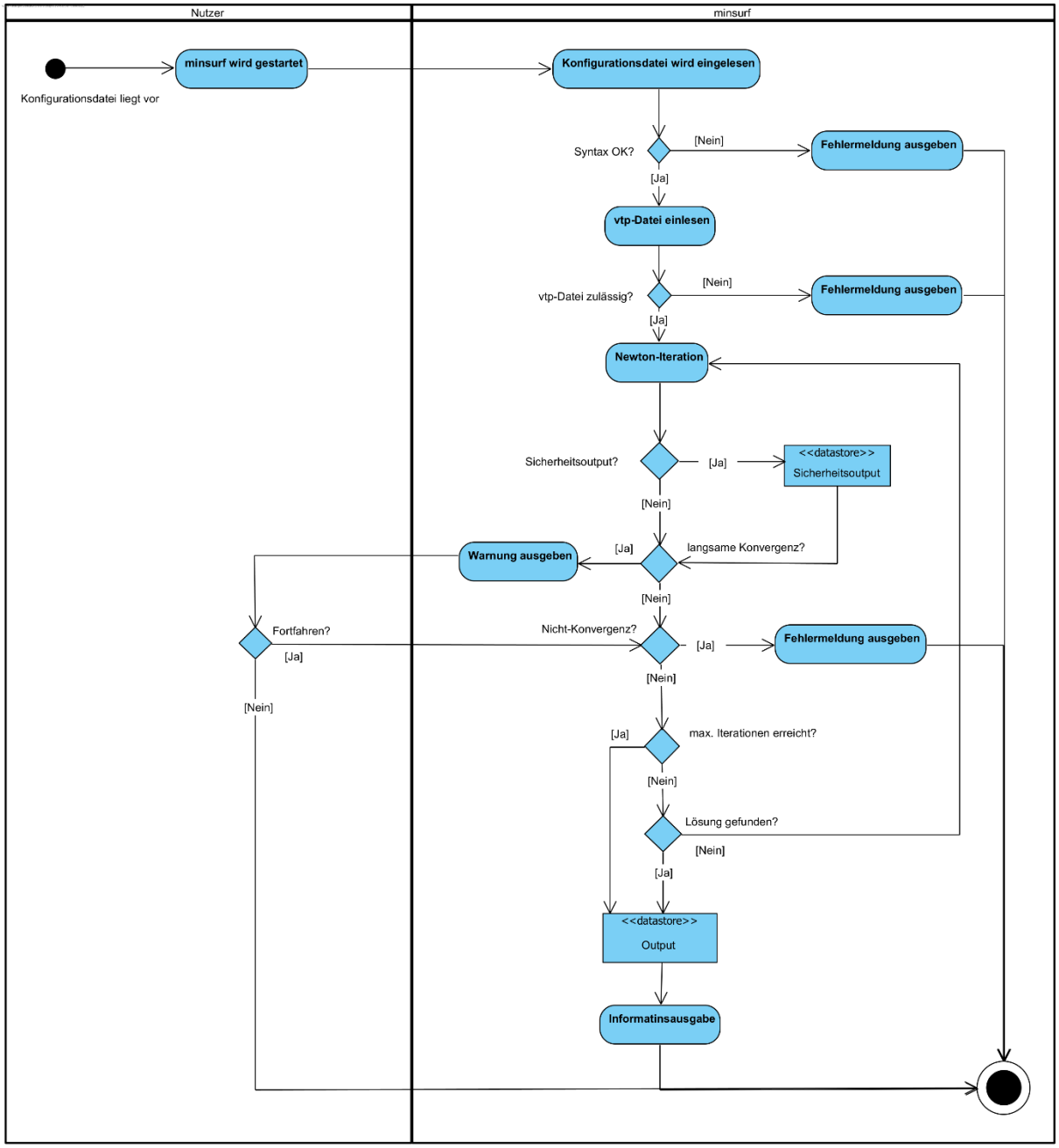


Abbildung 3:Aktivitätsdiagramm UC3

2.1.2.2 Systemanforderungen

2.1.2.2.1 Funktionale Anforderungen

- **FA010**: Das Programm errechnet auf einem Gitter über das Einheitsquadrat die minimale Fläche. Dazu trägt der Nutzer Funktionen für die Randwerte und den Wert für die Gittergröße in eine Konfigurationsdatei ein. Das Programm berechnet daraufhin die Minimalfläche und gibt eine Datei aus. Die Datei stellt die Lösung des Problems grafisch dar. Zur Anzeige dieser ist *paraview* erforderlich.
- **FA020:** Das Programm ist eine Konsolenanwendung. Nach dem Ausführen von *minsurf* wird die Ausgabedatei an den in der Konfigurationsdatei vorgeschriebenen Platz gespeichert.
- **FA030:** Der Nutzer trägt für jede Seite des Einheitsquadrates eine eigene Randwertfunktion ein. Für die Randwerte werden immer die Funktionswerte im Definitionsbereich (0,1] der Funktionen genutzt. Ihm stehen dabei Verkettungen der folgenden Funktionen und Operatoren zur Verfügung:
  - Standartoperatoren: Plus (*+*), Minus (-), Mal (*), Geteilt (*/*), Hoch (^)
  - Trigonometrische Funktionen: Sinus (*sin*), Cosinus (*cos*), Tangens (*tan*), Sinus hyperbolicus (*sinh*), Cosinus hyperbolicus (*cosh*), Tangens hyperbolicus (*tanh*)
  - Inverse trigonometrische Funktionen: Arcus Sinus (*arcsin*), Arcus Cosinus (*arccos*), Arcus Tangens (*arctan*), Arcus Sinus hyperbolicus (*arcsinh*), Arcus Cosinus hyperbolicus (*arccosh*), Arcus Tangens hyperbolicus (*arctanh*)
  - Exponentialfunktion: Exp (*exp*), natürlicher Logarithmus (*ln*)
  - Andere Funktionen: Minimumfunktion (*min*), Maximumfunktion (*max*), Absolutbetrag (*abs*), Logarithmus zur Basis 10 (*lg*)
- **FA040:** Der Benutzer passt numerische Werte für die Berechnung über die Konfigurationsdatei an. Dazu gehören:
  - Das Abbruchkriterium der skalierten 2-Fehlernorm und der maximalen Anzahl an Newton – Schritten,
  - Eine untere Schranke für den Dämpfer des gedämpften Newton – Verfahrens,
  - Die Stützstellenanzahl pro Zeile.
- **FA050:** Der Nutzer wählt in der Konfigurationsdatei eines der implementierten Verfahren aus:
  - Klassisches Newton – Verfahren
  - Triviales gedämpftes Newton – Verfahren
  - Newton – Verfahren mit Backtracking
- **FA060:** Bei nicht – Konvergenz des gewählten Newton – Verfahrens bricht das Programm mit einer beschreibenden Fehlermeldung ab. Nicht – Konvergenz wird durch sich kaum noch ändernde Residuen

$$\frac{\|R_{k+1} - R_k\|_2}{\|R_{k+1}\|_2} < \delta$$

über einen Zeitraum von $\tilde{k}$ Schritten festgestellt. Die Standartwerte $\delta = 10^{-5}$ und $\tilde{k} = 10$ können in der Konfigurationsdatei angepasst werden. R_i bezeichnet das Residuum im i – ten Schritt.

2.1.2.2.2 Nichtfunktionale Anforderungen

2.1.2.2.2.1 Anforderungen an die Benutzerschnittstelle

NFA010: Alle Eingaben des Benutzers erfolgen über eine Konfigurationsdatei. Diese ist kommentiert und dokumentiert, um die erforderlichen Eingaben zu erklären.

2.1.2.2.2.2 Anforderungen an das Leistungsverhalten

NFA110: Bei einer Gittergröße von 100x100 Punkten und den unten folgenden, beispielhaften Randwerten, dauert die Berechnung der Minimalfläche nicht länger als eine Minute auf dem Cluster des IT – Centers. Dabei beträgt der skalierte Fehler in der 2-Norm höchstens 10^{-7}. Die Berechnung wird in *double precision* ausgeführt. Die Randwerte lauten wie folgt:

- Norden: $N(x) = 4x^4 - 6x^3 + 3x^2 - x$
- Osten: $\mathrm{O(x)} = \sin\left(3{,}1415\sqrt{x}\right)$
- Süden: $S(x) = x^2$
- Westen: $\mathrm{W(x)} = \tan^{-1}(0{,}9x) - 1{,}733x^3 + 1$

Zusammen mit dem Programm wird eine komplette Konfigurationsdatei mit diesen Funktionen und numerischen Werten ausgeliefert.

2.1.2.2.2.3 Anforderung an die Entwicklungs- und Zielplattform

- **NFA210**: Das Programm ist für das Cluster des RWTH IT Centers entwickelt. Die Funktionstüchtigkeit wird bei ordnungsgemäßer Ausführung nur dort garantiert.
- **NFA220**: Das Programm ist in C++ geschrieben.
- **NFA230**: Das Erstellen der Ausgabedatei erfordert *VTK 8.1*. Um *VTK 8.1* auf dem Cluster verfügbar zu machen, wird eine Anleitung in der Dokumentation bereitgestellt.

2.1.2.2.2.4 Sonstige nichtfunktionale Anforderungen

- **NFA310**: Das Programm ist für Hochleistungsrechner mittels *openMP* parallelisiert.
- **NFA320**: Bei fehlerhaften oder schlechten Eingaben werden Warnungen ausgegeben. Diese weisen den Nutzer auf eine mögliche Fehlerquelle oder eine potentiell nicht gegebene bzw. schlechte Konvergenz hin.

2.2 Begriffsanalyse

Der wesentliche Anwendungsfall ist, eine Minimalfläche über das Einheitsquadrat bei gegebenen Randbedingungen zu berechnen. In diesem Kapitel wird dieses Problem zunächst mathematisch definiert, außerdem werden die verschiedenen numerischen Verfahren zur Lösung des Problems erklärt. Diese theoretische Einführung ist notwendig, um passende Klassenkandidaten zu identifizieren und die grundlegende Struktur des Programmes zu planen.

2.2.1 Mathematische Formulierung des Problems

Das Minimalflächenproblem lässt sich als partielle Differentialgleichung (PDE) zweiter Ordnung

$$R(u_x, u_y, u_{xx}, u_{yy}) = 0$$

formulieren. Gesucht ist $u(x, y)$: $\Omega \to \mathbb{R}$ mit

$$\begin{cases} (1 + u_x^2)u_{yy} - 2u_x u_y u_{xy} + (1 + u_y^2)u_{xx} = 0 \; in \; \Omega \\ u = B \; auf \; \partial\Omega \end{cases}$$

mit $\Omega = [0,1] \times [0,1] \subseteq \mathbb{R}^2$, $\partial\Omega$ als Rand von Ω und gegebenen Randwerten B [5]. Für die Lösung dieser Gleichung am Computer sind numerische Verfahren praktikabel. Das bedeutet zunächst, dass das Lösungsgebiet Ω mit einem äquidistanten Gitter $\overline{\Omega}_h$ mit $s \times s$ Gitterpunkten diskretisiert werden muss. $h = \frac{1}{s-1}$ ist dabei die Gitterweite. Dazu wird eine Klasse, hier `DISCRETIZATION`, bereitgestellt. Es bezeichnet $\Omega_h = \{(ih, jh) | 1 \leq i, j \leq s-1\}$ das diskrete Gebiet ohne die Randwerte. Dieses hat demnach pro Zeile $m = s - 2$ und insgesamt $n = m^2$ Gitterpunkte.

Die partiellen Ableitungen werden mittels finiter Differenzen diskretisiert, woraus sich für jeden Gitterpunkt (i, j) eine nichtlineare Gleichung ergibt. Dies wird in einer Klasse `RESIDUAL` realisiert.

$$R_{i,j}(u_{1,1}, u_{1,2}, \dots, u_{2,1}, \dots, u_{s-1,s-1}) = 0$$

Diese Residuumsfunktion ist ein nichtlineares Gleichungssystem, wobei R als Vektor der Größe n zu verstehen ist (das Gleiche gilt für u). Zur Lösung dieses Systems wird das Newton-Verfahren genutzt.

2.2.2 Numerik

2.2.2.1 Newton-Verfahren

Im Newton-Verfahren wird in jedem Schritt ein lineares Gleichungssystem (LGS) gelöst, um sich der Lösung des gegebenen nichtlinearen Gleichungssystems iterativ zu näheren und so eine Approximation zu finden. Die allgemeine Form ist:

$$JR(Z^{(i)})\, X^{(i)} = -R(Z^{(i)})$$
$$Z^{(i+1)} = Z^{(i)} + X^{(i)}$$

Die Systemmatrix des LGS ist die Jacobi-Matrix $JR \in \mathbb{R}^{n \times n}$ der Residuumsfunktion $R \in \mathbb{R}^n$. Es gilt somit $JR_{i,j} = \frac{\partial R_i}{\partial u_j}$ für $i, j = 1, \dots, n$. Zum Erstellen der Systemmatrix dient eine Klasse `SYSTEM_MATRIX`. Zur Berechnung der Ableitungen ist ein Klasse `DERIVATIVE` vorgesehen. Die rechte Seite ist das negative Residuum, also $-R$. Sowohl JR, als auch $-R$ werden bei $Z^{(i)} \in \mathbb{R}^n$ ausgewertet. Dieser Vektor wird in jedem Schritt durch die Lösung $X^{(i)} \in \mathbb{R}^n$ des LGS aktualisiert. Der Startvektor $Z^{(0)}$ muss passend gewählt werden, ansonsten besteht die Gefahr, dass das Newton-Verfahren nicht konvergiert. Das Verfahren bricht ab, falls eine maximale Anzahl an Newton-Schritten i_{max} erreicht ist oder die skalierte

Fehlernorm des Residuums $\frac{\|R(Z^{(i)})\|_2}{n} < \varepsilon$ wird. ε und i_{max} werden dabei vom Nutzer vorgegeben. Eine klassische Newton-Verfahren-Implementierung sollte in einer Klasse `NEWTON_CLASSIC` vorhanden sein.

2.2.2.2 Wahl des Startvektors

Um gute Startvektoren $Z^{(0)}$ zu finden, ist es hilfreich, die theoretischen Hintergründe des Problems zu kennen. Die vorliegende PDE ist elliptisch. Damit ist garantiert, dass Minima und Maxima von $u(x, y)$ auf dem Rand $\partial\Omega$ vorkommen. Wenn nun $max \in \mathbb{R}$ der Maximalwert und $min \in \mathbb{R}$ der Minimalwert der Randwertfunktion B ist, dann gilt für alle berechneten Werte $u_{i,j}$ im Gebiet Ω_h:

$$min \leq u_{i,j} \leq max$$

Es bietet sich folglich an, das arithmetische Mittel über die diskreten Randwerte zu berechnen und jeden Eintrag $Z_i^{(0)}$, $i = 1, \dots, n$, auf diesen Wert zu setzen. Diese einfache Berechnung kann in einer Methode in `DISCRETIZATION` untergebracht werden.

Eine aufwändigere, aber exaktere Approximation für den Startvektor ist die Lösung der Laplace-Gleichung in zwei Dimensionen mit identischen Randwerten wie für das Minimalflächenproblem:

$$u_{xx} + u_{yy} = 0$$

Die Struktur der beiden PDEs ist gleich, sie sind elliptisch. Um die Laplace-Gleichung zu lösen, wird ähnlich wie zur Lösung der Minimalflächengleichung vorgegangen. Es wird auf einem Gitter derselben Größe diskretisiert, als Differenzenstempel wird jedoch ein Stern-Stempel genutzt. Das entstehende Gleichungssystem ist bei dieser linearen partiellen Differentialgleichung linear, sodass kein Newton-Verfahren angewandt werden muss, sondern nur ein einziges Mal gelöst wird. Dafür sollte ein eigene Klasse `LAPLACE` zur Verfügung stehen.

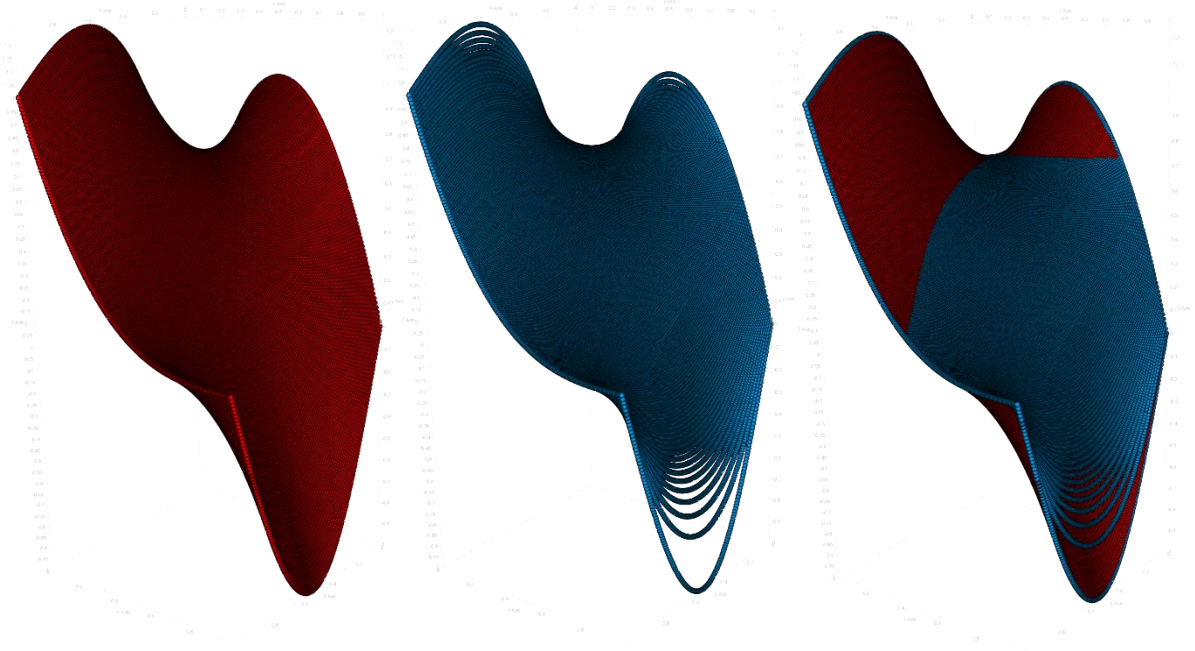


Abbildung 4: Laplace (rot) und Minimalflächenproblem (blau), Visualisierung mittels paraview

In Abbildung 3 erkennt man die Ähnlichkeit der Lösungen und damit die Sinnhaftigkeit der vorigen Berechnung der Laplace-Lösung. Links die Lösung der Laplace-Gleichung, in der Mitte die Lösung des Minimalflächenproblems und rechts beide übereinandergelegt.

2.2.2.3 Lösung des Gleichungssystems

Es fällt auf, dass durch die Diskretisierung des Residuums mit einem 3×3 Differenzenstempel der Wert von R in jedem Punkt $u_{i,j}$ von nur höchstens acht anderen Punkten (in anderen Worten: Variablen der Residuumsfunktion, vgl. Kapitel 2.2.1) abhängt. Somit hat jede Zeile in JR höchstens neun Einträge ungleich null, die Matrix hat also einen Besetztheitsgrad $\beta < \frac{9}{n}$ und ist somit dünnbesetzt. Für eine Diskretisierung mit 100×100 Gitterpunkten ergibt sich damit $\beta < 0{,}1\%$. Es ist deshalb ratsam, spezielle Datentypen für dünnbesetzte Matrizen zu verwenden, um effiziente Speichernutzung zu garantieren. Um außerdem eine effiziente Lösung des LGS umzusetzen, bieten sich parallelisierbare, iterative Algorithmen, die auch auf dünnbesetzten, nicht positiv definiten Matrizen arbeiten, an.

Iterative Lösungsverfahren lösen das LGS nicht exakt, sondern approximieren die Lösung, bis ein gewisser Zwangsterm η mit

$$\eta^{(i)} < \frac{\left\| JR\left(Z^{(i)}\right) X^{(i)} + R\left(Z^{(i)}\right) \right\|_2}{\left\| R(Z^{(i)}) \right\|_2}$$

erreicht ist. Der Zwangsterm wird wie in [2] beschrieben bestimmt. $\eta^{(i)}$ wird in jedem Newton-Schritt i adaptiv gewählt, sodass ein guter Kompromiss zwischen Genauigkeit und Laufzeit erzielt wird. Adaptive Zwangsterme gewähren superlineare Konvergenz des Newton-Verfahrens, während feste Werte für η nur lineare Konvergenz erreichen. Als Startwert wird $\eta^{(0)} = \frac{1}{10}$ gesetzt. In den Newton-Schritten $i = 1, 2, \ldots, i_{max}$ berechnet man zunächst

$$\eta^{(i)} = \frac{\left| \left\| R\left(Z^{(i)}\right) \right\|_2 - \left\| R\left(Z^{(i-1)}\right) + JR\left(Z^{(i-1)}\right) X^{(i-1)} \right\|_2 \right|}{\left\| R(Z^{(i-1)}) \right\|_2}$$

und kontrolliert anschließend den Zwangsterm in zwei Schritten:

- Falls $\eta^{(i)} < \eta^{(i-1)^{\frac{1+\sqrt{5}}{2}}}$ und $\eta^{(i-1)^{\frac{1+\sqrt{5}}{2}}} > \frac{1}{10}$, dann setze $\eta^{(i)} = \eta^{(i-1)^{\frac{1+\sqrt{5}}{2}}}$.
- Dann, falls $\eta^{(i)} > \eta_{max}$, setze $\eta^{(i)} = \eta_{max}$.

Es wird der Wert $\eta_{max} = \frac{9}{10}$ benutzt.

2.2.2.4 Globalisierung des Newton-Verfahrens

Das klassische Newton Verfahren (siehe Kapitel 2.2.2.1) konvergiert nicht immer, gerade für schlecht gewählte Anfangsvektoren $Z^{(0)}$ kann Divergenz auftreten [1]. Es ist deshalb ratsam, die Aktualisierung des Lösungsvektors zu dämpfen, um den Einzugsbereich des Newton-Verfahren zu erhöhen. Dies nennt man Globalisierung; der Konvergenzradius wird vergrößert. Das Newton-Verfahren ändert sich dann zu

$$JR\left(Z^{(i)}\right) X^{(i)} = -R\left(Z^{(i)}\right)$$
$$Z^{(i+1)} = Z^{(i)} + \lambda\, X^{(i)}$$

mit $\lambda \in \mathbb{R}$. Der Dämpfungsfaktor λ wird in jedem Newton-Schritt i neu berechnet. Dafür gibt es verschiedene Methoden.

2.2.2.4.1 Trivial gedämpftes Newton-Verfahren

Eine einfach zu berechnende Dämpfung ergibt sich aus einer trivial Verfahrensvorschrift. Wenn die Fehlernorm des aktualisierten Residuums im Schritt i größer ist als die Fehlernorm im Schritt $i-1$, dann reduziere die Aktualisierung und überprüfe die Bedingung erneut. Es wird ein Vorgehen gewählt, wie es in [3] beschrieben ist. Mathematisch bedeutet dies: $k = 0$. Solange

$$\left\|R\left(Z^{(i)}\right)\right\|_2 < \left\|R\left(Z^{(i)} + \lambda_k^{(i)} X^{(i)}\right)\right\|_2$$

ist, setze $\lambda_{k+1}^{(i)} = \frac{1}{2}\lambda_k^{(i)}$ und $k = k+1$. Falls $\lambda_k^{(i)} \leq \lambda_{min}$, dann setze $\lambda_k^{(i)} = \lambda_{min}$.

Anschließend wird der gedämpfte Newton-Schritt

$$Z^{(i+1)} = Z^{(i)} + \lambda_k^{(i)} X^{(i)}$$

ausgeführt. Hierbei ist $\lambda_0^{(0)} = 1$ vorgegeben. In allen weiteren Newton-Schritten $i = 1, 2, \ldots, i_{max}$ wird für $\lambda_0^{(i)}$ der Wert des Dämpfungsfaktors im vorangegangenen Schritt genommen, also $\lambda_0^{(i)} = \lambda_k^{(i-1)}$. Sollte im Schritt i außerdem der vorige Dämpfungswert sofort akzeptiert werden, also der Newton-Schritt mit $\lambda_0^{(i)} = \lambda_0^{(i-1)}$ ausgeführt werden, dann setze nach der Berechnung $\lambda_0^{(i)} = 2\lambda_0^{(i)}$, falls $\lambda_{k=0}^{(i)} < 1$ ist. Der Parameter λ_{min} ist dabei vom Nutzer wählbar. Für diese Implementierung sollte eine eigene Klasse `NEWTON_DAMPED_TRIVIAL` zur Verfügung stehen.

2.2.2.4.2 Dämpfung mittels Backtracking

Eine etwas aufwändigere Methode zur Globalisierung ist die Dämpfung mittels eines durch „backtracking“ berechneten Parameters $\theta_k^{(i)}$. Das Vorgehen ist in [2] beschrieben: In jedem Newton-Schritt i, setze $k = 0$. Solange

$$\left\|R\left(Z^{(i)} + X_k^{(i)}\right)\right\|_2 > \left\|\left(1 - t\left(1 - \eta^{(i)}\right)\right) R\left(Z^{(i)}\right)\right\|_2$$

gilt, bestimme $\theta_k^{(i)}$, aktualisiere $X_{k+1}^{(i)} = \theta_k^{(i)} X_k^{(i)}$, setze $\eta^{(i)} = 1 - \theta_k^{(i)}\left(1 - \eta^{(i)}\right)$ und inkrementiere $k = k+1$. Es bezeichnet $\eta^{(i)}$ den Zwangsterm des iterativen Verfahrens, welches das lineare Gleichungssystem löst (siehe Kapitel 2.2.2.3) und $X_0^{(i)}$ die Lösung des LGS (siehe Kapitel 2.2.2.1). Wenn die obige Bedingung erfüllt ist, setze $Z^{(i+1)} = Z^{(i)} + X_k^{(i)}$.

Die gesamte Dämpfung in $Z^{(i+1)} = Z^{(i)} + \lambda^{(i)} X^{(i)}$ ergibt sich damit aus

$$\lambda^{(i)} = \prod_{l=0}^{k} \theta_l^{(i)}$$

Die Berechnung von $\theta_k^{(i)}$ erfolgt mittels *backtracking*. Die Idee dahinter ist es, ein Interpolationspolynom $\mathcal{P}(v)$ zu minimieren. Dieses ist quadratisch und besitzt folgende Eigenschaften:

- $\mathcal{P}(0) = \frac{1}{2}\left\|R\left(Z^{(i)}\right)\right\|_2^2$
- $\mathcal{P}(1) = \frac{1}{2}\left\|R\left(Z^{(i)} + X_k^{(i)}\right)\right\|_2^2$
- $\frac{d\mathcal{P}(0)}{dv} = \frac{d}{dv}\frac{1}{2}\left\|R\left(Z^{(i)} + v\, X_k^{(i)}\right)\right\|_2^2\Big|_{v=0}$

Das Interpolationspolynom $\mathcal{P}(v)$ errechnet sich damit zu

$$\mathcal{P}(v) = \left(\mathcal{P}(1) - \mathcal{P}(0) - \frac{d\mathcal{P}(0)}{dv}\right)v^2 + \left(\frac{d\mathcal{P}(0)}{dv}\right)v + \mathcal{P}(0)$$

und das eindeutige Minimum von $P(v)$ liegt bei

$$v^* = -\frac{\frac{d\mathcal{P}(0)}{dv}}{2\left(\mathcal{P}(1) - \mathcal{P}(0) - \frac{d\mathcal{P}(0)}{dv}\right)}$$

Für Eindeutigkeit und Existenz siehe [4]. In jedem Teilschritt $k = 0,1, \ldots$ kann also $\theta_k^{(i)} = v^*$ direkt berechnet werden. Sollte nun $\theta_k^{(i)} < \theta_{min}$, dann wird $\theta_k^{(i)} = \theta_{min}$ beziehungsweise falls $\theta_k^{(i)} > \theta_{max}$ ist, so wird $\theta_k^{(i)} = \theta_{max}$ gesetzt. Es werden die numerischen Werte gewählt, die in [4] empfohlen werden:

$\theta_{min} = \frac{1}{10}$, $\theta_{max} = \frac{1}{2}$, $t = 10^{-4}$.

Der Parameter t ist hierbei ein Maß dafür, wie groß Differenz der Fehlernormen werden muss, damit ein Schritt akzeptiert wird.

Hierfür sollte ebenso eine Klasse `NEWTON_BACKTRACKING` vorhanden sein.

2.2.3 Konfigurationsdatei

Um den zweiten in Kapitel 2.1.2 beschriebenen Anwendungsfall „Konfigurationsdatei erstellen“ in das Programm einzubinden, muss ein Übersetzter implementiert werden, der die in der Konfigurationsdatei angegebenen Parameter in vom Hauptprogramm nutzbare Ausdrücke umwandelt. Diese Klasse wird als `INTERPRETER` bezeichnet.

2.2.4 Klassenkandidaten

Durch die bisherigen Überlegungen ergeben sich somit die folgenden Klassenkandidaten:

- `DISCRETIZATION`
- `RESIDUAL`
- `DERIVATIVE`
- `SYSTEM_MATRIX`
- `NEWTON_CLASSIC`
- `LAPLACE`
- `NEWTON_BACKTRACKING`
- `NEWTON_DAMPED_TRIVIAL`
- `INTERPRETER`

Diese Begriffsanalyse soll nur zum allgemeinen Verständnis des Problems, der Lösungsmethoden und der rudimentären Implementierung dienen. Für eine detailliertere Beschreibung aller Klassen, notwendiger Vererbungshierachien und Methoden, siehe Kapitel 3 und 5.

2.2.5 Bibliothek

Die *eigen3.3.4* Bibliothek stellt sowohl einen dünnbesetzen Matrixdatentyp als auch den iterativen Löser *BiCGSTAB* zur Verfügung, eine Variante des *Konjugierte Gradienten* – Verfahrens, welche auch auf nicht zwingend positiv definiten Systemmatrizen arbeitet. Dieser Löser ist bereits parallelisiert und für dünnbesetze Matrizen optimiert. Für die Lösung eines LGS $Ax = b$ kann zudem die Toleranz $\eta = \frac{\|Ax-b\|_2}{\|b\|_2}$ über eine bereits implementierte Funktion kontrolliert werden (siehe [7]).

Die Ausgabe der Datei zur Visualisierung der Minimalfläche wird mit der frei verfügbaren Software *Visualization Tool Kit* (VTK) umgesetzt. Es wird die aktuelle Version 8.1.1 benutzt. Die Ausgabedatei hat das Format *.vtp* (*vtkpolydata*), welches mit *Paraview* geöffnet und angezeigt werden kann.

3 Entwurf

3.1 Statik

Die Statik des Programmentwurfs wird durch ein UML Klassendiagramm beschrieben. Die folgenden Klassendiagramme wurde in *Visual Studio* erstellt. Um die Übersichtlichkeit zu erhöhen, wird zunächst nur eine Übersicht der Klassenhierachie aufgeführt, gefolgt von ausführlichen Diagrammen der einzelnen Klassen. Private Datenelemente (hier: Felder) und Methoden sind hierbei mit einem Schloss-Symbol gekennzeichnet, beschützte (*protected*) mit einem Stern-Symbol. Elemente ohne solche Kennzeichnungen sind öffentlich.

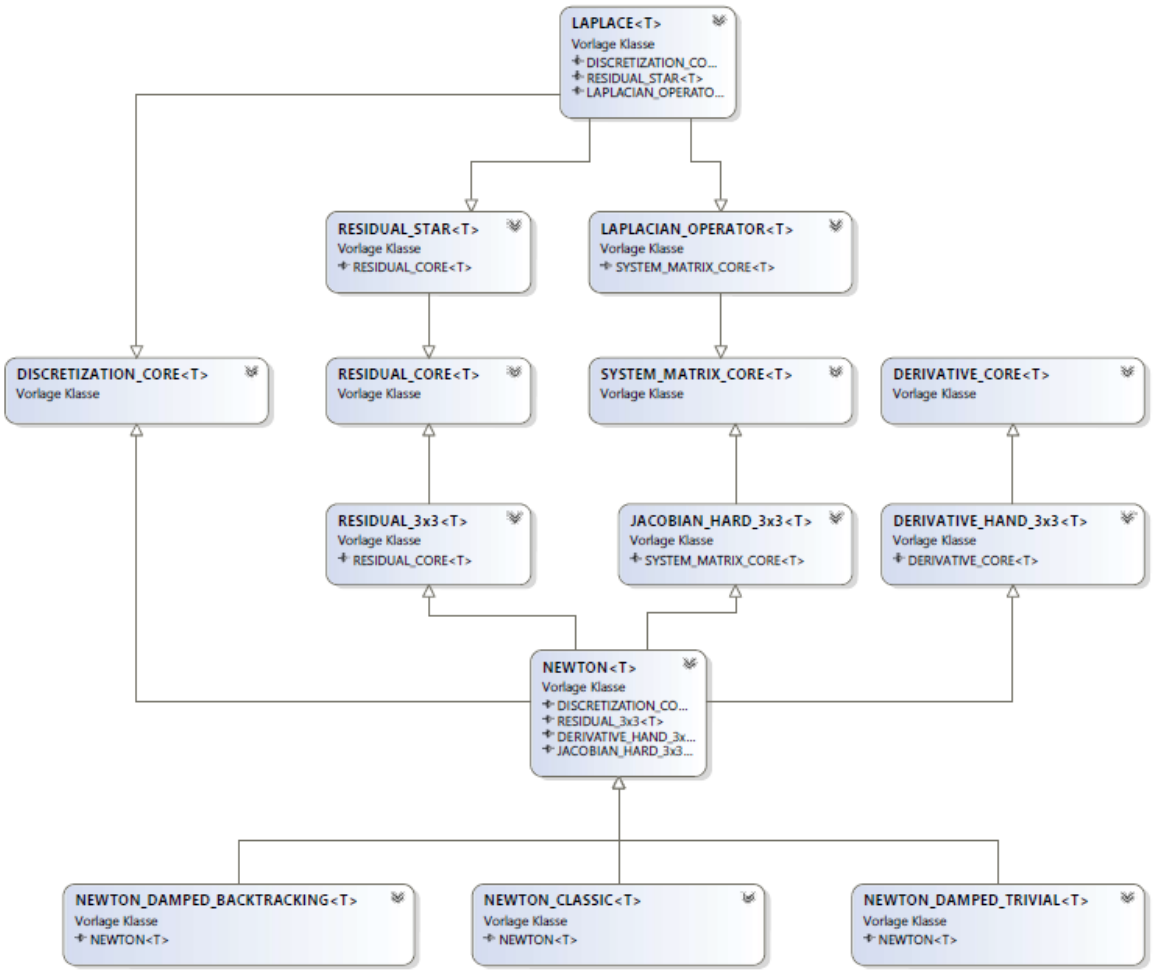


Abbildung 5: Übersicht Klassendiagramm

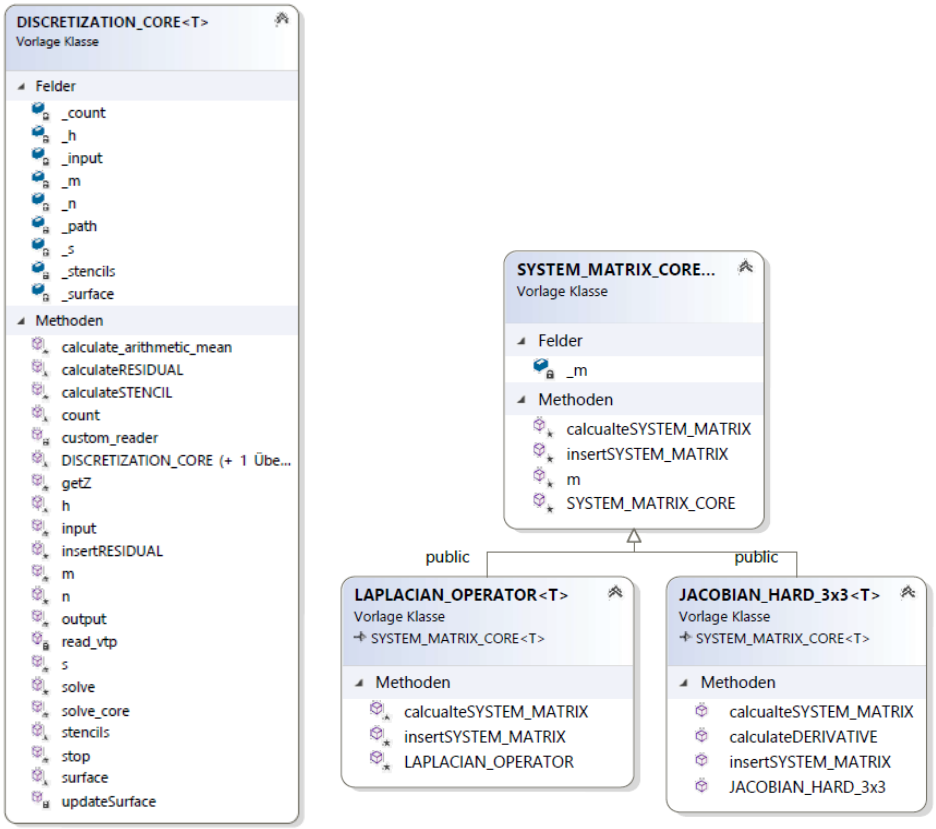


Abbildung 6: Klassendiagramme DISCRETIZATION, SYSTEM_MATRIX

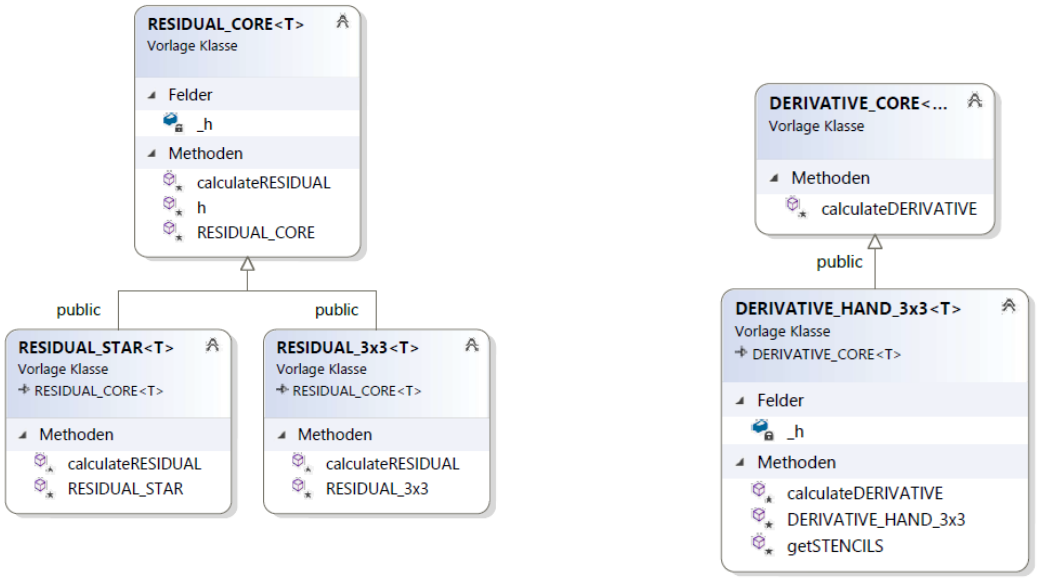


Abbildung 7: Klassendiagramme RESIDUAL, DERIVATIVE

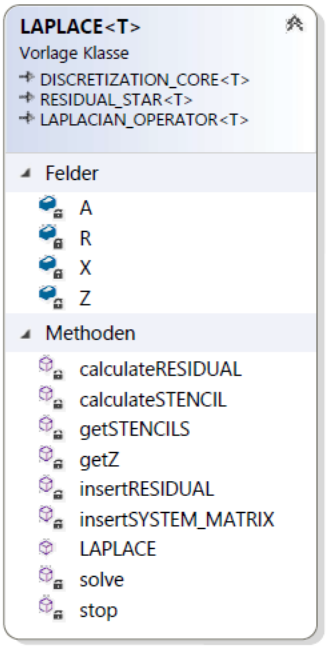


Abbildung 8: Klassendiagramm LAPLACE

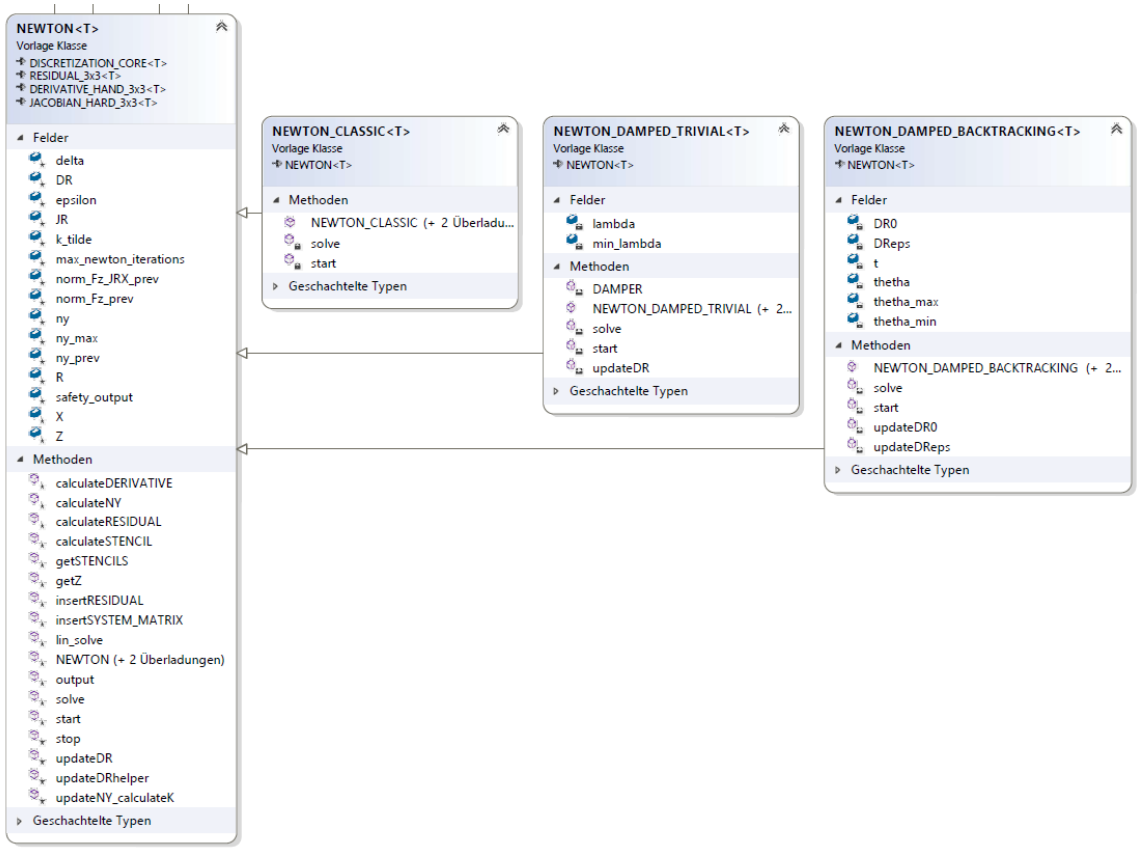


Abbildung 9: Klassendiagramme NEWTON

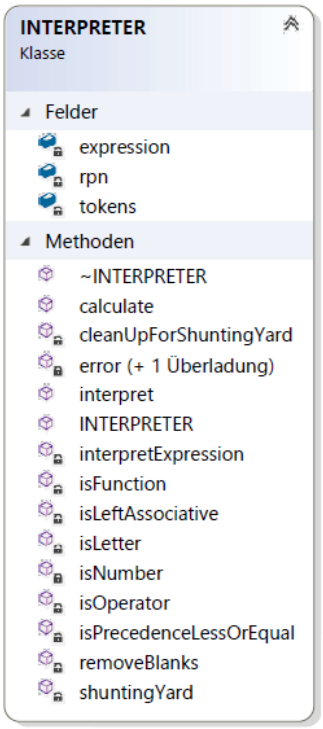


Abbildung 10: Klassendiagramm INTERPRETER

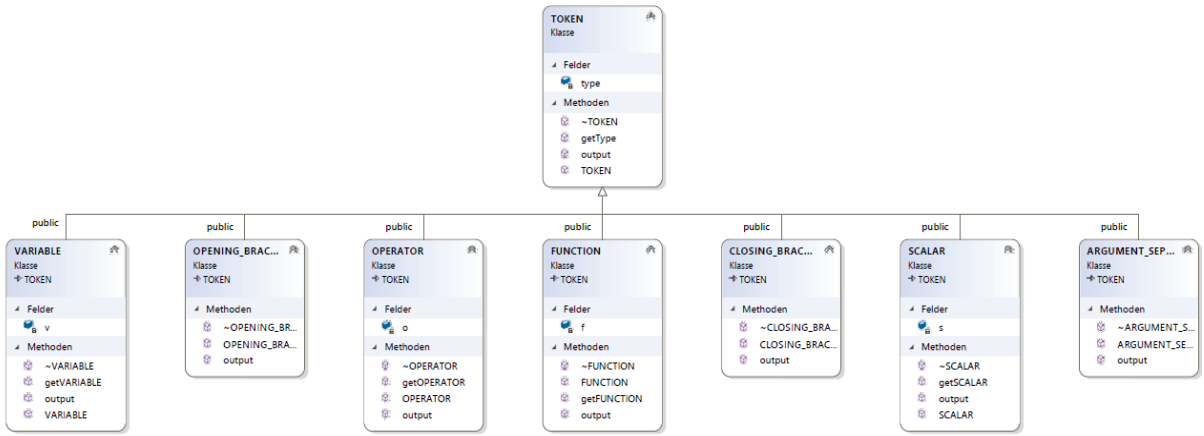


Abbildung 11: Klassendiagramm TOKEN

3.2 Dynamik

Da die gesamte Funktionalität des Programmes praktisch im Konstruktor der Löser-Klassen untergebracht ist, welche von zahlreichen anderen Klassen erben (vgl. Kapitel 5.1.5), werden nicht viele Informationen zwischen unterschiedlichen Objekten ausgetauscht. Es folgt die Darstellung der Programmdynamik für den Fall, dass der Nutzer als Löser die Variante mit Laplace-Anfangsvektor und anschließendem klassischen Newton-Verfahren gewählt hat. Der Ablauf der Bearbeitung der Konfigurationsdatei ist mit Sequenzdiagramm nicht sinnvoll, da es sich nicht um einen Programmablauf handelt.

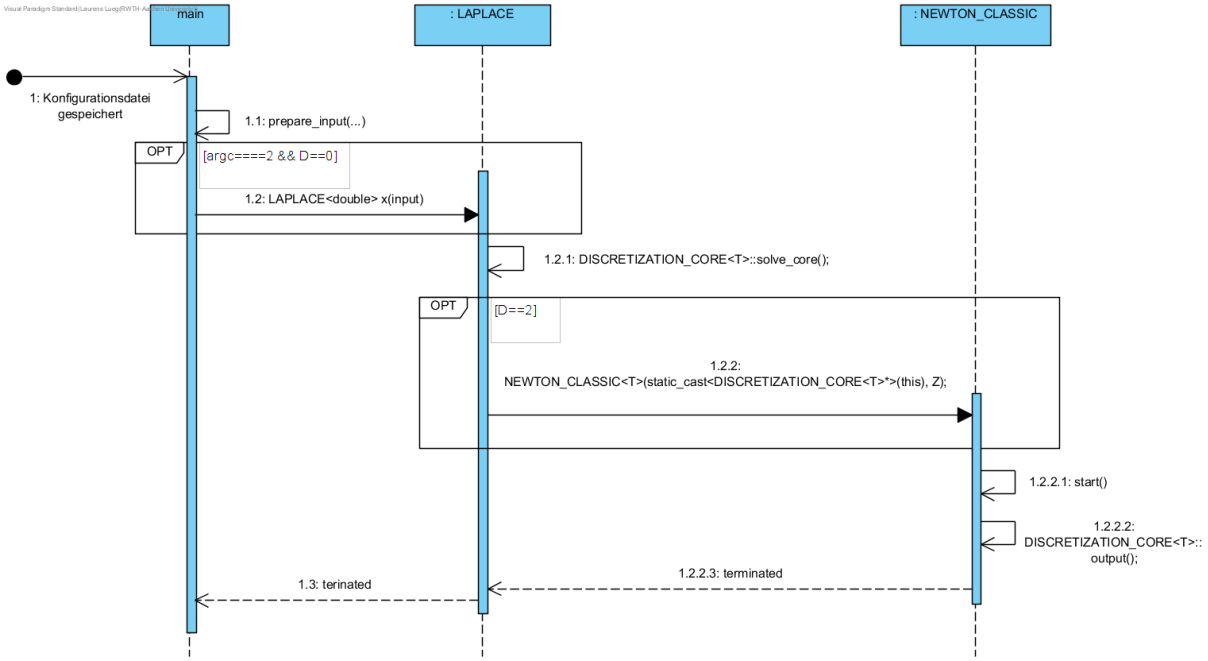


Abbildung 12: Sequenzdiagramm für "Minimalfläche berechnen" mit Löser 02

4 Benutzerdokumentation

4.1 Installation

Die Software wird als *minsurf.zip* Archiv ausgeliefert. Es enthält neben dem Quellcode des Programms auch die nötigen Bibliotheken *Eigen* und *VTK 8.1.* Zur Installation des Programms und der Bereitstellung der Bibliotheken sind folgende Schritte notwendig:

1) Entpacken Sie *minsurf.zip* in neuen Ordner am gewünschten Ort mittels

```
$ unzip minsurf.zip
```

2) Navigieren Sie in den Ordner *minsurf.*
3) Entpacken Sie *VTK* mittels

```
$ tar -xvf VTK-8.1.1.tar.gz
```

Dies kann eine Weile dauern.

4) Erstellen Sie einen neuen Ordner und navigieren Sie in diesen mittels

```
$ mkdir VTK-build
$ cd ./VTK-build
```

5) Builden Sie *VTK*
   a. Führen Sie über die Kommandozeile aus:

```
$ module load cmake
$ cmake . -DCMAKE_C_COMPILER=/usr/bin/gcc -DCMAKE_CXX_COMPILER=/usr/bin/g++ ../VTK-8.1.1
```

Dies kann eine Weile dauern.

   b. Führen Sie über die Kommandozeile aus:

```
$ make -j<#Cores>
```

Dies kann eine Weile dauern. *VTK 8.1.1* ist jetzt in */minsurf/VTK-build/* erstellt worden.

6) Führen sie in */minsurf/* über die Kommandozeile aus:

```
$ cmake . -DCMAKE_C_COMPILER=/usr/bin/gcc -DCMAKE_CXX_COMPILER=/usr/bin/g++
```

7) Führen sie in */minsurf/* über die Kommandozeile aus:

```
$ make
```

8) Die Installation ist nun abgeschlossen. Sie rufen das Programm über die generierte ausführbare Datei *minsurf* auf. Die Konfigurationsdatei kann zum Beispiel über

```
$ vim config.txt
```

bearbeitet werden (siehe Kap. 4.2).

9) Zum Ausführen von *minsurf*, rufen Sie über die Kommandozeile im Ordner /minsurf/

```
$ ./minsurf config.txt
```

auf. Zum Weiterrechnen von einer vorhandenen *.vtp* Datei (zum Beispiel *output.vtp*), rufen Sie über die Kommandozeile

```
$ ./minsurf config.txt output.vtp
```

auf. Dank der Parallelisierung kann *minsurf* auch mit mehreren Kernen ausgeführt werden. Hierfür rufen Sie über die Kommendozeile

```
$ OMP_PROC_BIND=spread OMP_NUM_THREADS=<#Cores> ./minsurf config.txt
```

bzw.

```
$ OMP_PROC_BIND=spread OMP_NUM_THREADS=<#Cores> ./minsurf config.txt output.vtp
```

auf. `<#Cores>` ist hierbei die gewünschte Anzahl der Kerne, die genutzt werden sollen. Für optimale Performance sollte die maximale Anzahl der physischen Kerne eines NUMA-Knotens gewäht werden.

4.2 Konfigurationsdatei

Numerische Parameter und die Randwert-Funktionen werden über eine Konfigurationsdatei *config.txt* festgelegt. Diese Konfigurationsdatei ist folgendermaßen aufgebaut:

config.txt

```
1       //0 = LAPLACE, 2 = CLASSIC, 3 = DAMPED (TRIVIAL), 4 = DAMPED (BACKTRACKING):
2
3       //DATA TYPE (float, double, long double):
4
5       //GRID-POINTS PER ROW (MIN 4 POINTS PER ROW):
6
7       //OUTPUT-PATH (RELATIVE):
8
9       //FUNCTIONS:
10      //NORTH:
11
12      //EAST:
13
14      //SOUTH:
15
16      //WEST:
17
18      //MAX NEWTON ITERATIONS (ONLY 2, 3, 4):
19
20      //NEWTON ITERATIONS PER OUTPUT (ONLY 2, 3, 4):
21
22      //ERROR (ONLY 2, 3, 4):
23
24      //CRITERIA FOR NON-CONVERGENCE (ONLY 2, 3, 4):
25      //delta (ONLY 2, 3, 4):
26
27      //k_tilde (ONLY 2, 3, 4):
28
29      //MIN LAMBDA (ONLY 3):
30
```

Der Nutzer schreibt in die freien Zeilen die passenden, in den Kommentaren vermerkten Werte. Es gibt also die folgenden Möglichkeiten der Konfiguration des Programms:

Zeile	Eingabe	Beschreibung
2	0	Es wird die zweidimensionale Laplace Gleichung gelöst (siehe Kapitel 2.2.2.2)
	2	Es wird das Minimalflächenproblem mit dem klassischen Newton-Verfahren gelöst (siehe Kapitel 2.2.2.1). Als Startvektor wird das arithmetische Mittel benutzt (siehe Kapitel 2.2.2.2).
	3	Es wird das Minimalflächenproblem mit dem trivial gedämpften Newton-Verfahren gelöst (siehe Kapitel 2.2.2.4.1). Als Startvektor wird das arithmetische Mittel benutzt.
	4	Es wird das Minimalflächenproblem mit dem *backtracking* Newton-Verfahren gelöst (siehe Kapitel 2.2.2.4.2). Als Startvektor wird das arithmetische Mittel benutzt.
	02	Es wird das Minimalflächenproblem mit dem klassischen Newton-Verfahren gelöst. Als Startvektor wird Laplace benutzt (siehe Kapitel 2.2.2.2).
	03	Es wird das Minimalflächenproblem mit dem trivial gedämpften Newton-Verfahren gelöst. Als Startvektor wird Laplace benutzt.
	04	Es wird das Minimalflächenproblem mit dem *backtracking* Newton-Verfahren gelöst. Als Startvektor wird Laplace benutzt.
4	float	Es wird in einfacher Genauigkeit gerechnet.
	double	Es wird in doppelter Genauigkeit gerechnet.
	long double	Es wird in vierfacher Genauigkeit gerechnet.
6	s	$s \in \mathbb{N}$, $s \geq 4$ bezeichnet die Stützstellen pro Zeile. Insgesamt ergeben sich damit s^2 Stützstellen.
8	`_path`	Der Nutzer legt den Namen `_path` der Outputdatei fest und kann zusätzlich einen relativen Pfad vom ausführenden Verzeichnis angeben.
11	$N(x)$	$N(x)$ ist die Randwertfunktion für die nördliche Seite des Einheitsquadrates.
13	$E(x)$	$E(x)$ ist die Randwertfunktion für die östliche Seite des Einheitsquadrates.
15	$S(x)$	$S(x)$ ist die Randwertfunktion für die südliche Seite des Einheitsquadrates.
17	$W(x)$	$W(x)$ ist die Randwertfunktion für die westliche Seite des Einheitsquadrates.
19	i_{max}	Das Newton-Verfahren stoppt nach $i_{max} \in \mathbb{N}$ Iterationen.
21	i_{out}	Immer, nachdem $i_{out} \in \mathbb{N}$ Iterationen ausgeführt wurden, wird eine Outputdatei erstellt, von der später wieder eingelesen werden kann.

Zeile	Eingabe	Beschreibung
23	ε	Das Newton-Verfahren stoppt, sobald die Fehlernorm kleiner als ε ist (siehe Kapitel 2.2.2.1).
26	δ	$\delta \in \mathbb{R}$ ist die Schranke für das Kriterium für schlechte Konvergenz (siehe Kapitel 4.4).
28	$\tilde{k}$	Sollte das Kriterium für schlechte Konvergenz über einen Zeitraum von $\tilde{k} \in \mathbb{N}$ Schritten erfüllt sein, so bricht das Newton-Verfahren ab.
30	λ_{min}	$\lambda_{min} \in \mathbb{R}$ ist eine untere Schranke für den Dämpfungsfaktor bei dem trivial gedämpften Newton-Verfahren (siehe Kapitel 2.2.2.4.1).

Für die Randwertfunktionen $N(x)$, $E(x)$, $S(x)$, $W(x)$ stehen dem Nutzer die Standardoperatoren und einige Funktionen zu Verfügung:

Operatoren:

Eingabe	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
^	Potenzierung

Funktionen:

Eingabe	Bedeutung	Funktion
sin()	Sinus	$\sin(\dots)$
cos()	Cosinus	$\cos(\dots)$
tan()	Tangens	$\tan(\dots)$
sinh()	Sinus hyperbolicus	$\sinh(\dots)$
cosh()	Cosinus hyperbolicus	$\cosh(\dots)$
tanh()	Tangens hyperbolicus	$\tanh(\dots)$

Eingabe	Bedeutung	Funktion
arcsin()	Arcus Sinus	$\sin^{-1}(\dots)$
arccos()	Arcus Cosinus	$\cos^{-1}(\dots)$
arctan()	Arcus Tangens	$\tan^{-1}(\dots)$
arsinh()	Arcus Sinus hyperbolicus	$\sinh^{-1}(\dots)$
arcosh()	Arcus Cosinus hyperbolucus	$\cosh^{-1}(\dots)$
artanh()	Arcus Tangens hyperbolicus	$\tanh^{-1}(\dots)$
exp()	Exponentialfunktion	$e^{(\dots)}$
ln()	Natürlicher Logarithmus	$\ln(\dots)$
min(,)	Minimumfunktion	$\min\{(\dots),(\dots)\}$
max(,)	Maximumfunktion	$\max\{(\dots),(\dots)\}$
abs()	Absolutbetrag	$\lvert(\dots)\rvert$
log10()	Logarithmus zur Basis 10	$\lg(\dots)$
log2()	Logarithmus zur Basis 2	$\mathrm{ld}(\dots)$

Die Anordnung der Randwertfunktionen wird durch die folgende Tabelle veranschaulicht, die die Diskretisierung $\overline{\Omega}_h$ darstellen soll.

<table>
<tr><td>$W(1)$</td><td>$N(h)$</td><td>…</td><td>$N\big((s-2)h\big)$</td><td>$N(1)$</td></tr>
<tr><td>$W\big((s-2)h\big)$</td><td colspan="3" rowspan="3">Ω_h</td><td>$O(h)$</td></tr>
<tr><td>…</td><td>…</td></tr>
<tr><td>$W(h)$</td><td>$O\big((s-2)h\big)$</td></tr>
<tr><td>$S(1)$</td><td>$S\big((s-2)h\big)$</td><td>…</td><td>$S(h)$</td><td>$O(1)$</td></tr>
</table>

Die diskreten Werte im Bereich $(0,1]$ der einzelnen Funktionen wird dementsprechen als Randwerte benutzt.

4.3 Beispielsitzung

Der Nutzer sei der Anleitung in Kapitel 4.1 gefolgt. Die ausführbare Datei *minsurf* ist also in einem Ordner zusammen mit der *config_example.txt* vorhanden. Der Nutzer passt zunächst die Konfigurationsdatei an:

config_example.txt

```
1       //0 = LAPLACE, 2 = CLASSIC, 3 = DAMPED (TRIVIAL), 4 = DAMPED (BACKTRACKING):
2       02
3       //DATA TYPE (float, double, long double):
4       double
5       //GRID-POINTS PER ROW (MIN 4 POINTS PER ROW):
6       100
7       //OUTPUT-PATH (RELATIVE):
8       output.vtp
9       //FUNCTIONS:
10      //NORTH:
11      4x^4-6x^3+3x^2-x
12      //EAST:
13      sin(3.1415x^(1/2))
14      //SOUTH:
15      x^2
16      //WEST:
17      arctan(0.9x)-1.733x^3+1
18      //MAX NEWTON ITERATIONS (ONLY 2, 3, 4):
19      2000
20      //NEWTON ITERATIONS PER OUTPUT (ONLY 2, 3, 4):
21      500
22      //ERROR (ONLY 2, 3, 4):
23      10e-7
24      //CRITERIA FOR NON-CONVERGENCE (ONLY 2, 3, 4):
25      //delta (ONLY 2, 3, 4):
26      10e-5
27      //k_tilde (ONLY 2, 3, 4):
28      10
29      //MIN LAMBDA (ONLY 3):
30      10e-4
```

Es wird nun das Minimalflächenproblem mittels klassischem Newton-Verfahren gelöst (vergleiche Kapitel 2.2.2.1) und als Startvektor wird vorher die Laplace-Gleichung in zwei Dimensionen ausgewertet (siehe Kapitel 2.2.2.2). Die Berechnung wird in *double precision* ausgeführt und die Diskretisierung besteht aus 100×100 Gitterpunkten. Der Ausgabepfad ist *output.vtp*. Die Randwertfunktionen werden wie in Kapitel 2.1.2.2.2.2 gewählt. Das Verfahren bricht ab, falls 2000 Newton-Iterationen ausgeführt wurden oder die skalierte Residuumsnorm, wie in Kapitel 2.2.2.1, kleiner als 10^{-7} wird. Weiterhin gilt das Konvergenzkriterium mit $\delta = 10^{-5}$ und $\tilde{k} = 10$ (siehe Kapitel 4.4). Der Nutzer möchte alle 500 Newton-Iterationen eine Output-Datei erhalten.

Im nächsten Schritt wird *minsurf* mittels

```
$ OMP_PROC_BIND=spread OMP_NUM_THREADS=16 minsurf config_example.txt
```

ausgeführt.

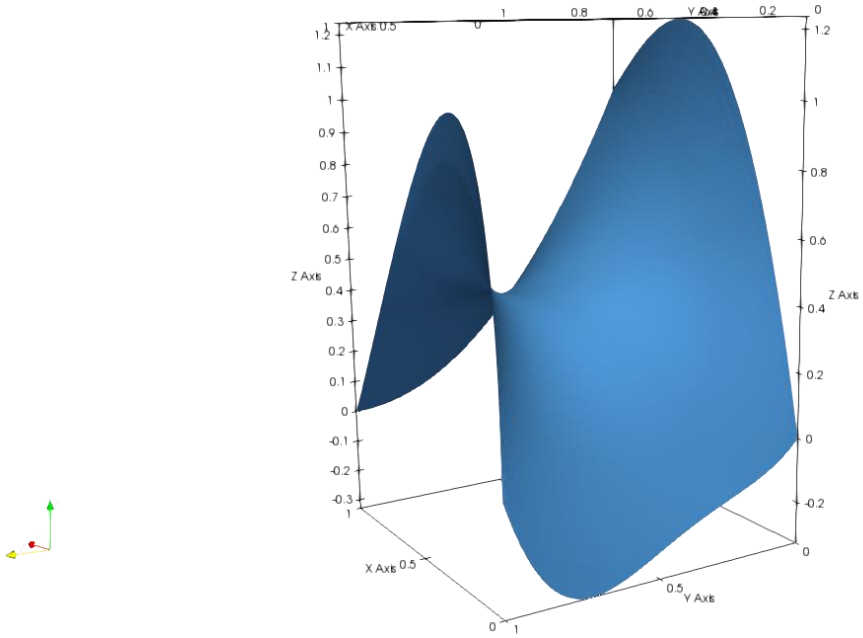


Abbildung 13: Visualisierung von output.vtp mit Paraview

Es wird die Datei `output.vtp` ausgegeben, welche mittels *paraview* visualisiert werden kann.

4.4 Fehlersituationen

Die Haupt-Fehlerquelle ist die Bearbeitung der Konfigurationsdatei durch den Nutzer. Hierbei kann man zwischen verschiedenen Arten von Fehlern unterscheiden.

- Syntaktische Fehler bei der Eingabe: Wann immer der Nutzer Eingaben macht, die nicht die in der Dokumentation geforderte Form haben, oder Felder der Konfigurationsdatei frei lässt, wird das Programm abgebrochen und auf den entsprechenden Fehler hingewiesen. Beispiel:
  - Stützstellenanzahl „2k“
  - Lösungsstrategie „58“
- Logischerweise unzulässige Eingaben: Eingaben, die syntaktisch korrekt sind, aber nicht umgesetzt werden können. Auch hier wird das Programm abgebrochen (siehe Kapitel 7.1 Zeile 18ff). Beispiel:
  - Stützstellenanzahl „-5“
  - Randwertfunktion „3*x*y“.
- Eingabe von Randwertfunktionen, die an den Ecken stark unterschiedliche Funktionswerte aufweisen: Es wird eine Warnung ausgegeben, der Nutzer entscheidet, ob die Berechnung trotzdem ausgeführt werden soll.
- Eingabe von Randwertfunktionen, die im Intervall $(0,1]$ nicht definiert sind. Es wird dann eine Fehlermeldung ausgegeben, dass die Randwertfunktion keine Zahl ist. Beispiel:
  - $W(x) = \ln(-x)$
  - $S(x) = \frac{x}{0}$

Weiterhin wird über nicht – Konvergenz oder eine langsame Konvergenz informiert. Dies wird durch sich kaum noch ändernde Residuen

$$\frac{\|R_{k+1} - R_k\|_2}{\|R_{k+1}\|_2} < \delta$$

über einen Zeitraum von $\tilde{k} \in \mathbb{N}$ Schritten festgestellt. R_i bezeichnet das Residuum im i – ten Newton-Schritt. Es wird dann eine Warnung ausgegeben und man kann entscheiden, ob das Programm abbricht oder weiterrechnet. Sollte weitergerechnet werden, wird keine Warnung wegen langsamer Konvergenz mehr ausgegeben. Bei den Standartwerten $\delta = 10^{-5}$ und $\tilde{k} = 10$ wird erst sehr spät abgebrochen, dem Programm wird also Zeit gegeben, die Lösung zu finden.

Ein Abbruch der Berechnung erfolg, falls der *BiCGSTAB* – Löser bei der Berechnung versagt. Es wird eine entprechende Fehlermeldung ausgegeben.

Zusätzlich wird bei einer Ausführung des Programmes geprüft, ob die potentiell mitgegebene vtp – Datei das passende Dateiformat hat.

5 Entwicklerdokumentation

5.1 Codestruktur

Der Code besteht aus den im Klassendiagramm in Kapitel 3.3.1 gezeigten Klassen sowie einem Hauptprogramm. Aus dem Hauptprogramm wird nur auf eine spezifizierte Löser-Klasse zugegriffen. Jede der Löser-Klassen (außer `LAPLACE<T>`) erbt von der Klasse `NEWTON<T>`, welche das Grundgerüst der Implementierung zur Verfügung stellt. Diese Klasse `NEWTON<T>` erbt von `DISCRETIZATION_CORE<T>`, `RESIDUAL_3x3<T>`, `DERIVATIVE_HAND_3x3<T>` und `JACOBIAN_HARD_3x3<T>`. In den folgenden Kapiteln sollen Eigenschaften und Funktionalitäten zu jeder im Klassendiagramm gezeigten Klasse beschrieben werden.

5.1.1 Klasse `DISCRETIZATION_CORE<T>`

Diese typgenerische Klasse stellt grundlegende Funktionalitäten sowie virtuelle Methoden zur Spezifizierung in den Lösern zu Verfügung (siehe Kapitel 5.1.5, 7.1).

Hierbei ist `_s` (s) die Anzahl an Gitterpunkten pro Zeile mit Randwerten, `_m` ($m = s - 2$) gibt die Anzahl der Gitterpunkte ohne die Randwerte pro Zeile an, `_n` ($n = m^2$) ist die Gesamtzahl der Gitterpunkte ohne Rand und `_h` ($h = \frac{1}{s-1}$) ist der Abstand zwischen den äquidistanten Gitterpunkten. In `_surface` wird die Lösung und in `_stencils` die Differenzenstempel für jeden Gitterpunkt gespeichert. In `_input` sind die Ausdrücke der Konfigurationsdatei gespeichert. Die Iterationen der Lösungsmethode werden in `count` gezählt. Zu diesen Datenelementen gibt es entsprechende getter/setter-Methoden (siehe 7.1 Zeile 192ff). `_path` legt den Namen der Ausgabedatei fest.

An den Konstruktor von `DISCRETIZATION_CORE` wird im Normalfall die Information aus der Konfigurationsdatei als `vector<string>& _input` gegeben. Diese wird dann durch Benutzung der Klasse `INTERPRETER` (siehe Kapitel 5.1.6) und der seperaten Funktion `test_on_input_value(…)` (siehe 7.1 Zeile 29ff.) ausgewertet, um die Gitterparameter und Randwerte Werte für die Berechnung in `_surface` festzulegen. Außerdem wird im Konstruktor `count` zu 1 initialisiert und die Eingaben aus `_input` auf Zulässigkeit geprüft (7.1, Zeile 90ff).

Die zentrale Methode von `DISCRETIZATION_CORE` ist `solve_core()` (7.1, Zeile 221ff), in der die grundlegende Struktur des Lösungsverfahrens bereitgestellt wird. In jedem Schritt wird `_surface` durch `updateSurface()` aktualisiert und `_stencils` durch `calculateSTENCIL()` und `calculateRESIDUAL()` berechnet. Die Methode `solve()` wird aufgerufen, solange die `stop()` – Methode einen positiven Rückgabewert gibt. `solve()` dient zur Lösung des linearen Gleichungssystems in jedem Schritt, `stop()` zu Überprüfung der Abbruchkriterien. Diese Funktionen werden in den Löser-Klassen für den entsprechenden Löser spezifiziert.

`calculate_arithmetic_mean()` berechnet das arithmetische Mittel der in `INTERPRETER` errechneten Randwerte, um einen adäquaten Startpunkt für das Newton-Verfahren zu erhalten (vgl. Kap. 2.2.2.2.). `output()` dient zur Ausgabe einer vtk-Datei, welche die Lösungsfläche darstellt. Dazu wird auch die Funktion `custom_reader()` benötigt. Der Konstruktor ist überladen, um auch das Einlesen der Randwerte aus einer *vtp* – Datei zu ermöglichen. Dazu wird die Methode `read_vtp(…)` (7.1 Zeile 326ff) eingesetzt.

5.1.2 Klasse RESIDUAL_CORE<T>

RESIDUAL_CORE ist eine typgenerische Basisklasse, die als einziges Datenelement den Gitterparameter _h enthält, der auch im Konstruktor mitgegeben wird. Zugriff auf dieses Element erfolgt mit der getter-Methode h(). Die virtuelle Methode calculateResidual(…) = 0 wird in den von RESIDUAL_CORE abgeleiteten Klassen, je nach Art des verwendeten Stencils, spezialisiert (siehe Kapitel 7.3)

5.1.2.1 Klasse RESIDUAL_3x3<T>

RESIDUAL_3x3<T> implementiert in calculateResidual(…) die Berechnung des Residuums für das Minimalflächenproblem mit einem 3×3 Differenzenstempel (Siehe Kapitel 3.2.2.1). Diese Klasse wird von allen Löser-Klassen, außer LAPLACE, geerbt (7.3, Zeile 18ff).

5.1.2.2 Klasse RESIDUAL_STAR<T>

RESIDUAL_STAR<T> implementiert in calculateResidual(…) die Berechnung des Residuums für die Laplace-Gleichung mit einem Stern-Differenzenstempel (siehe Kapitel 3.2.2.2). Diese Klasse wird nur in LAPLACE genutzt (7.3, Zeile 41ff).

5.1.3 Klasse SYSTEM_MATRIX_CORE<T>

SYSTEM_MATRIX_CORE ist eine templatisierte Basisklasse, die als einziges Datenelement den Gitterparameter _m enthält, der auch im Konstruktor mitgegeben wird. Zugriff auf dieses Element erfolgt mit der get-Methode m(). insertSYSTEM_MATRIX(…) wird in den Lösern spezialisiert und wird zum Einfügen von Elementen in die Systemmatrix genutzt. calculateSYSTEM_MATRIX(…) ist Vorlage zur Berechnung der Systemmatrix des Residuums und wird in den Klassen zur Berechnung der Ableitungen spezialisiert (siehe Kapitel 7.4).

5.1.3.1 Klasse JACOBIAN_HARD_3x3<T>

Die Methode calculateSYSTEM_MATRIX(…) wird spezialisiert, um die Jacobimatrix des Residuums des Minimalflächenproblems zu berechnen. Dafür werden die weiterhin nicht spezialisierte Funktion insertSYSTEM_MATRIX(…) und die neue virtuelle Funktion calculateDERIVATIVE(…) benötigt. Diese Klasse wird von allen Löser-Klassen, außer LAPLACE, geerbt (7.4, Zeile 64ff).

5.1.3.2 Klasse LAPLACIAN_OPERATOR<T>

Die Methode calculateSYSTEM_MATRIX(…) wird spezifiziert, um die Systemmatrix der Laplace-Gleichung zu berechnen. Dafür wird die weiterhin nicht spezialisierte Funktion insertSYSTEM_MATRIX(…) benötigt. Diese Klasse wird nur im LAPLACE genutzt (7.4, Zeile 21ff).

5.1.4 Klasse DERIVATIVE_CORE<T>

Hier wird eine nur die rein virtuelle Methode calculateDERIVATIVE(…) deklariert.

5.1.4.1 Klasse DERIVATIVE_HAND_3x3<T>

Als einziges Datenelement enthält diese Klasse die Gitterbreite _h. calculateDERIVATIVE(…) wird hier spezialisiert. Da die Ableitungen bei einem 3×3 – Stempel für die entsprechenden Elemente immer gleichbleiben, wurden die Formeln „hart“ programmiert (7.2, Zeile 44ff). Dazu braucht man allerdings die hier virtuelle Funktion getSTENCILS(…), die im Löser dann von DISCRETIZATION_CORE (siehe Kapitel 3.2.1) geerbt wird. Diese Klasse wird von allen Löser-Klassen, außer LAPLACE, geerbt.

5.1.5 Die Löser-Klassen

In Kapitel 2.2 wurde bereits angedeutet, dass die Klassen zur Lösung des nichtlinearen Gleichungssystems mittels Newton-Verfahren implementiert werden. Die grundlegende Struktur des Verfahrens ist bereits in `DISCRETIZATION_CORE` umgesetzt (vgl. Kap. 5.1.1).

Die verschiedenen Versionen des Newton-Verfahrens unterscheiden sich im Wesentlichen, bis auf `LAPLACE`, nur in der Berechnung des Dämpfers (vgl. Kap. 2.2.2.4). Es bietet sich deshalb an, für die Komponenten, die in den verschiedenen Newton-Verfahren identisch sind, eine Ober-Klasse zu erstellen, von der dann geerbt wird. Diese Klasse heißt `NEWTON`.

In `NEWTON_CLASSIC` ist dann das klassische, nicht gedämpfte Newton – Verfahren umgesetzt. In `NEWTON_DAMPED_TRIVIAL` ist ein einfacher Dämpfer implementiert (siehe Kapitel 2.2.2.4.1), während in `NEWTON_DAMPED_BACKTRACKING` der Dämpfer mittels Backtracking berechnet wird (siehe Kapitel 2.2.2.4.2). In `LAPLACE` wird die Lösung der Laplace-Gleichung berechnet, welche anschließend als Anfangswert für eines der Newton-Verfahren genutzt werden kann.

5.1.5.1 Klasse `NEWTON<T>`

`NEWTON` enthält die grundlegenden Datenelemente des Newton-Verfahrens: `R` (Residuum), `Z` (Lösungsvektor), `X` (Aktualisierungsterm), `DR` (neues Residuum) und `JR` (Jacobimatrix von `R`). Diese werden in `start()` initialisiert.

Die vom Nutzer gewählten Parameter des Newton Verfahrens `epsilon` (Genauigkeit), `delta` (Mindestverbesserung der Lösung pro Schritt), `k_tilde` (maximale Anzahl von konsekutiven Schritten mit Verbessrung kleiner `delta` bis Warnung), `max_newton_iterations` und `safety_output` (Anzahl von Schritten, nach denen während Berechnung eine Ausgabedatei erstellt wird) werden ebenfalls in `start()` eingelesen.

Die Lösung des linearen Gleichungssystems mittels des iterativen Algorithmus `Eigen::BiCGSTAB` geschieht in `lin_solve()`. Dieser basiert auf dem Konjugierte – Gradienten Verfahren und ist für Matrizen stabilisiert, die nicht zwingend symmetrisch positiv definit sind. Hierfür werden auch die den Zwangsterm des linearen Lösers betreffende Werte `ny`, `ny_prev` und `ny_max` gebraucht. `ny` wird in `calculateNY()` wie in Kap. 2.2.2.3 beschrieben berechnet.

Die Funktionen `updateDR()` und `updateDRhelper()` befassen sich mit der Berechnung des Residuums der aktualisierten Lösung. In `updateNY_calculateK()` werden die zur Berechnung von `ny` benötigten Werte aktualisiert und die Konvergenzgeschwindigkeit überprüft.

Weiterhin beinhaltet `NEWTON` die `stop()` – Methode. Diese gibt ein `bool` zurück, der auf `false` gesetzt wird, falls $\frac{\|R(Z^{(i)})\|_2}{n} < \varepsilon$ oder $i \geq i_{max}$ (ε entspricht `epsilon`, i_{max} `max_newton_iter` und n ist die Stützstellenzahl pro Zeile). Die `output()` Methode gibt Informationen zum Iterationsschritt und zur Fehlernorm aus, außerdem wird `output()` aus `DISCRETIZATION_CORE` aufgerufen.

Der Konstruktor von Newton ist überladen, um den Aufruf mittels Eingabedatei, vtk-Datei oder durch `LAPLACE` zu ermöglichen. Es wird der entsprechende Konstruktor von `DISCRETIZATION_CORE` und die Konstruktoren der anderen Basisklassen mit den entsprechenden Gitterparametern aus `DISCRETIZATION_CORE` aufgerufen. `ny` (η) wird zu 0.1 initialisiert. Danach wird lediglich `start()` aufgerufen.

5.1.5.2 Klasse *NEWTON_CLASSIC<T>*

Der Konstruktor wird in der `main` aufgerufen. Wie in `NEWTON` gibt es drei Überladungen, entsprechend der gewünschten Startwertberechung. Der passende `NEWTON`-Konstruktor wird jeweils aufgerufen. Danach wird lediglich die Methode `start()`ausgeführt. Mit Aufruf des Konstruktors in der `main` werden somit alle Funktionalitäten des Programms ausgeführt.

In `start()` wird die Funktion `solve_core()` aus `DISCRETIZATION_CORE`, welche dann die spezialisierte `solve()` Methode aufruft. Falls `solve_core()` erfolgreich durchgeführt wurde, wird eine finale Ausgabe durch `output()` aus `NEWTON` gemacht.

In `solve()` wird das klassische Newton – Verfahren aus Kapitel 2.2.2.1 implementiert. Hierzu werden die benötigten Funktionen und Datenelemente aus `NEWTON`, `JACOBIAN_HARD_3x3` und `DISCRETIZATION_CORE` verwendet. Außerdem wird alle in `safety_output` festgelegten Schritte `output()` aus `NEWTON<T>` aufgerufen.

Die übrigen Methoden des Lösers sind getter- und setter- Methoden für `Z`, `R` und `JR` sowie Berechnungsmethoden für den Stempel, das Residuum (aus `RESIDUAL_CORE`) und für die partiellen Ableitungen (aus `DERIVATIVE_CORE`). Der gesamte Quellcode befindet sich in Kapitel 7.5.

5.1.5.3 Klasse *NEWTON_DAMPED_TRIVIAL<T>*

Diese Klasse enthält zusätzlich die Datenelemente `lamda` (λ) und `min_lamda`. `lambda` ist der Dämpfungsfaktor, der in jedem Schritt berechnet wird. `lambda_min` (λ_{min}) ist die untere Schranke für λ und wird in `start()` aus der Eingabedatei eingelesen. Zur Berechnung des Dämpfungsfaktors wird in der spezialisierten `solve()` – Routine auch die neue Funktion `DAMPER()` benötigt. In `DAMPER()` wird λ berechnet und der Lösungsvektor Z danach entsprechend aktualisiert (siehe Kapitel 2.2.2.4.1).

5.1.5.4 Klasse *NEWTON_DAMPED_BACKTRACKING<T>*

Die zusätzlichen Datenelemente `DReps` und `DR0` werden zur Berechnung des Dämpfungsfaktors `thetha` (θ) benötigt (siehe Kapitel 2.2.2.4.2). `thetha_min` (θ_{min}) und `thetha_max` (θ_{max}) sind Minimum und Maximum für die Dämpfung in jedem Teilschritt. `t` steuert die benötigte Reduktion, damit die Dämpfung angenommen wird. Die `solve()` Methode wird dahingehend spezifiziert, dass mit Hilfe der Funktionen `updateDReps()` und `updateDR0()` der Dämpfungsfaktor mittels Backtracking gemäß Kapitel 2.2.2.4.2 berechnet wird.

5.1.5.5 Klasse *LAPLACE<T>*

Diese Klasse dient dazu, einen besseren Starvektor für das Newton-Verfahren zu berechnen, indem die Laplace-Gleichung in zwei Dimensionen gelöst wird (seine Kapitel 2.2.2.2).

Zur Berechnung dieser muss ein lineares Gleichungssystem ausgerechnet werden. Dies geschieht mittels der bekannten `solve_core()` Methode aus `DISCRETIZATION_CORE`. Dazu müssen die Methoden `solve()`, `calculateRESIDUAL(…)` und `calculateSTENCIL(…)` spezifiziert werden. In `solve()` wird nun die Laplace-Gleichung mit der Systemmatrix aus `LAPLACIAN_OPERATOR<T>` gelöst. Die Genauigkeit des iterativen Lösers wird konstant gehalten, da nur ein einziges Mal gelöst wird. Das Ergebnis wird in der Ergebnismatrix `_surface` gespeichert. `calculateRESIDUAL(…)` wird von `RESIDUAL_STAR` abgeleitet, `calculateSTENCIL(…)` entsprechend dem Sternstempel definiert. Anschließend wird der Konstruktor eines anderen Lösers, abhängig von der Konfigurationsdatei, mit dem berechneten Startvektor aufgerufen.

5.1.6 Klasse INTERPRETER

Diese Klasse dient dazu, Teile der Konfigurationsdatei auszuwerten. Das bedeutet, die Randwertfunktionen zu übersetzen und an den entsprechenden Gitterpunkten auszurechnen. Dies erfordert zum Teil sehr umfangreiche Methoden, da es offensichtlich sehr viele Kombinationsmöglichkeiten von Operatoren, Variablen und dergleichen gibt. Da es sich bei dieser Klasse um einen eher technischen Teil des Programms handelt, soll im Folgenden nur die grundlegende Funktionsweise erklärt werden, ohne auf jede einzelne Methode genau einzugehen. Für Hintergrundinformationen zu der theoretischen Grundlage des `INTERPRETER` siehe [6].

`INTERPRETER` besitzt drei Datenelemente. In `expression` ist eine Funktion aus der Konfigurationsdatei als string gespeichert, ohne Kommentare und Leerzeichen. `tokens` ist ein Vektor aus verschiedenen Typen (z. B. Variable, Operator oder Skalar, siehe Kap. 5.1.7) der Eingabe, die Umwandlung der `expression` zu `tokens` geschieht in der Funktion `interpretExpression()`.In ihr werden Hilfsfunktionen `isOperator(…), isNumber(…), isLetter(…)` und `isFunction(…)` verwendet um die Token den Typen zuzuweisen. Die Funktion `shuntingYard()`[6] wandelt `tokens` dann in ein Resultat `rpn` (Reverse Polish Notation) um. Damit lässt sich der mathematische Ausdruck in einem einfachen Baum speichern. Schließlich wird in `calculate(…)` die jeweilige Funktion an einer bestimmten Stelle ausgerechnet. Weiterhin gibt es eine Funktion `cleanUpForShuntingYard()` in der gewisse Kombinationen von Ausdrücken überprüft werden. Dadurch lässt sich z.B. $2x$ statt $2 * x$ oder $\sin 2$ statt $\sin(2)$ schreiben. Die restlichen Funktionen werden zur Fehleridentifikation verwendet. Die Klassifizierung der Ausdrücke übernimmt die Klasse `TOKEN` (siehe Kap. 5.1.7).

5.1.7 Klasse TOKEN

`TOKEN` ist die Basisklasse, über die die verschiedenen Ausdrücke der Eingabedatei klassifiziert werden. Das einzige Datenelement ist `type`, welches abhängig vom Ausdruck einen Wert zwischen -10 und 10 annimmt. Dazu verfügt die Klasse noch über eine getter-Methode sowie eine virtuelle `output(…)` Funktion. Bei den Ausdrücken wird zwischen Operatoren, Skalaren, Variablen, Funktionen, separierenden Zeichen und Klammern unterschieden. Für jeden dieser Ausdrücke gibt es eine Unterklasse, die von `TOKEN` erbt. In diesen Unterklassen wird dann die `output(…)` Funktion spezifiziert und ggf. ein neues Datenelement für die jeweilige Klasse definiert (z. B. ein double in `SCALAR`).

5.2 Detaillierte Dokumentation des Codes

In der *main.cpp* wird, je nach Konfiguration, eine der Newton-Klassen bzw. Laplace erstellt. Dabei wird zunächst der Inhalt der Konfigurationsdatei gereinigt und dann als Argument an den jeweiligen Konstruktor gegeben.

5.2.1 Aufstellen der Systemmatrix

In der Klasse `JACOBIAN_HARD_3x3<T>` (siehe Kapitel 5.1.3.1) wird die virtuelle Methode `calculateSYSTEM_MATRIX(…)` spezialisiert (7.4, Zeile 80ff). Als Argument nimmt diese Funktion die aktuelle Fläche `surface`, welche nach den einzelnen Variablen abgeleitet werden muss, um die Systemmatrix des LGS zu erhalten. Dazu wird die Ableitung jedes Elements mittels `calculateDERIVATIVE(…)` berechnet und anschließend in die dünnbesetzte Matrix mittels `insertSYSTEM_MATRIX(…)` eingefügt.

Der spezielle Aufbau von `calculateSYSTEM_MATRX(…)` erlaubt eine äußert effiziente parallele Implementierung. Jedes Element in `surface` muss nach höchstens neun Variablen abgeleitet werden (siehe Kapitel 2.2.2.3). Es wird also zwischen folgenden Arten von Punkten unterschieden:

- Eckpunkte (vier Ableitungen nötig)
- Randpunkte (ohne Eckpunkte, sechs Ableitungen nötig)
- Innere Punkte (neun Ableitungen nötig)

Jede Art Punkt wird gesondert behandelt. Es gibt insgesamt immer vier Eckpunkte und dementsprechen $4 * 4 = 16$ Werte, welche in die Systemmatrix eingetragen werden müssen. Dies erfolgt seriell (siehe 7.4, Zeilen 81-84, 96-99, 147-154, 172,179).

Die Randpunkte werden in parallelisierten for – Schleifen eingefügt (siehe 7.4, Zeilen 86-94, 101-110, 133-145, 156-170).

Der größte Anteil an Punkten, die Inneren, werden in einer zweiten for – Schleife, welche sich in einer der parallelisierten for – Schleifen für die Randpunkte befindet, eingefügt (siehe 7.4, Zeilen 112-131).

5.2.2 Implementierung der Newton-Klassen

Die Berechnung des Zwangsterms `ny` für das iterative Lösungsverfahren ist bei allen Verfahren identisch (siehe Kapitel 2.2.2.3). Im Konstruktor jedes Lösers wird der erste Zwangsterm `ny_0` festgelegt, welcher im ersten Newton-Schritt benutzt wird. Am Ende jeder Newton-Iteration, nachdem der Lösungsvektor Z aktualisiert ist, werden die für die Berechnung notwendigen Normen gespeichert. `norm_Fz_prev` $\left(\left\|R\left(Z^{(i-1)}\right)\right\|_2\right)$ und `norm_Fz_JRX_prev` $\left(\left\|R\left(Z^{(i-1)}\right) + JR\left(Z^{(i-1)}\right) X^{(i-1)}\right\|_2\right)$ werden also im Newton-Schritt $i-1$ für den Zwangsterm des Schrittes i berechnet. Weiterhin wird der Zwangsterm in `ny_prev` abgespeichert (siehe Kapitel 7.5 Zeile 136ff).

Für alle Iterationen nach dem ersten Schritt kann der Zwangsterm damit berechnet werden. Anschließend wird der Absolutbetrag bestimmt und es findet die Kontrolle von `ny` statt (7.5, Zeile 128ff).

5.2.3 Implementierung des trivial gedämpften Newton-Verfahrens

Hier wird zunächst überprüft, ob man sich in der ersten Newton-Iteration befindet. Sollte dies der Fall sein, wird direkt die Funktion `DAMPER()` aufgerufen, die einen passenden Dämpfungsfaktor `lambda` bestimmt und den Ergebnisvektor `z` direkt aktualisiert. In allen anderen Schritten wird überprüft, ob

`lambda` wieder erhöht werden kann (siehe Kapitel 2.2.2.4.1). Falls ja, wird `z` berechnet und `lambda` erhöht, falls nicht wird wieder die Funktion `DAMPER()` aufgerufen (7.5 Zeile 257ff).

5.2.4 Implementierung des *backtracking* Newton-Verfahrens

Der Kern dieses Verfahrens besteht aus der Berechnung des Dämpfungsparameters `thetha`. Dieser wird direkt berechnet, indem direkt das Minimum der Interpolationsfunktion der Fehlernorm bestimmt wird (siehe Kapitel 2.2.2.4.2). Dafür werden die Werte `p_0` $\left(\mathcal{P}(0)\right)$, `p_1` $\left(\mathcal{P}(1)\right)$ und `p_prime_0` $\left(\frac{d\mathcal{P}(0)}{dv}\right)$ benötigt. `p_0` ist für jeden Newton-Schritt konstant und wird direkt bestimmt (7.5, Zeile 341). In einer Schleife, welche die Bedingung für einen akzeptablen Schritt prüft, werden `p_1` und `p_prime_0` ausgerechnet (7.5, Zeile 345f). Zum Ermitteln von `p_prime_0` wird der Differenzenquotient benutzt. Dafür sind zwei weitere Vektoren `DReps` und `DR0` nötig, welche zunächst passend bestimmt werden (7.5, Zeile 361ff). Jetzt sind alle Unbekannten der Formel bestimmt und `thetha` kann berechnet werden. Nun wird `thetha` noch mit `thetha_min` und `thetha_max` verglichen und gegebenenfalls angepasst (7.5, Zeile 348ff). Im letzten Schritt wird der Aktualisierungsvektor `X` gedämpft, das Residuum wird aktualisiert und die Bedingung der Schleife wird überprüft. Bei Abbruch wird der Ergebnisvektor `z` aktualisiert (7.5, Zeile 342ff).

5.3 Software Tests

Das Programm *minsurf* wurde mit einigen Eingaben getestet. Eingaben, die nicht in der Form der Tabelle in Kapitel 4.2 sind, werden durch bezeichnende Fehlermeldungen abgefangen. Die folgenden Tests wurden durchgeführt.

Zeile	**Gültige Eingabe**			**Ungültige Eingabe**			
2	0	2	04	-1	1	05	
4	float	double		int	flota		
6	4	800	2147483647	-8	3	100.7	2147483648
8	output.vtp			ß	*		
11, 13, 15, 17	2x-x^2	y/2+y^(1/2)		x+y	sin(x]		
19	1	2000	2147483647	-1	0	22.7	2147483648
21	1	100	2147483647	-5	0	40.2	2147483648
23	0.000005	10e-7	1.79769e+308	-5	0	1.7977e+308	
26	0.000005	10e-7	1.79769e+308	-1	0	1.7977e+308	
28	1	10	2147483647	-2	0	2147483648	
30	0.000005	10e-7	1.79769e+308	-1	0	1.7977e+308	

5.4 Parallelisierung

minsurf wurde mithilfe der Programmierschnittstelle *openMP* für die parallele Nutzung mehrerer Kerne entwickelt. Dies hatte besondere Auswirkung auf die Wahl des linearen Lösers bzw. der verwendeten Bibliotheken (vgl. Kapitel 2.2.5). Im folgenden Kapitel werden für Laufzeittests stets die numerischen Werte aus der Beispielsitzung in Kapitel 4.3 genutzt. Lediglich die Anzahl der Gitterpunkte ändert sich.

5.4.1 Analyse des Codes

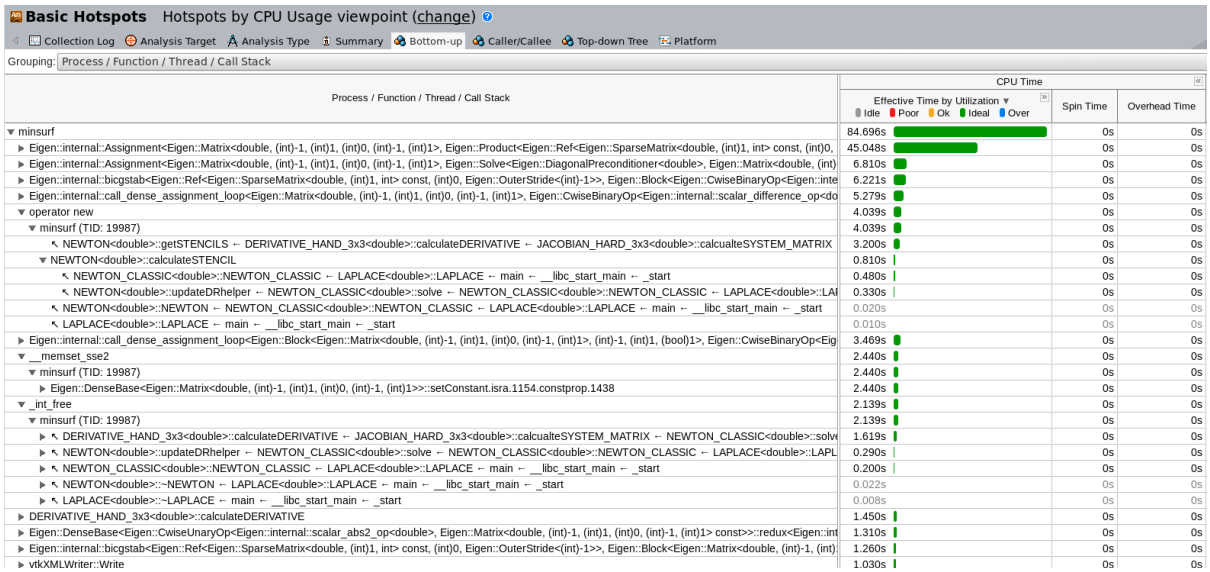


Abbildung 14: Hotspot-Test mittels IntelVTune, optimierte single-core Variante, 600x600

Zur Identifikation von Hotspots, also denjenigen Regionen im Code, in denen die meiste Zeit verbracht wird, kommt *Intel VTune* zum Einsatz. Es wird hierfür die Anzahl der Stützstellen auf $s = 600$ erhöht. Die Hotspot-Analyse wurde mit einer seriell optimierten Variante des Codes erstellt.

Das Programm befindet sich zum Großteil der Laufzeit in *eigen* – Methoden (Zeilen 2, 3, 4, 5, 14, 17). Weiterhin wird einige Zeit für das Berechnen der Ableitungen und das Einfügen in die dünnbesetze Systemmatrix aufgewendet (Zeilen 8, 20, 25).

Diese und die kommende Hotspot-Analyse wurden auf dem Front-End des RWTH-Aachen IT-Center Cluster durchgeführt (login.hpc.itc.rwth-aachen.de). Die Compilerflags sind:

`-std=c++11 -O3 -fopenmp`.

`Die Version des GNU-Compilers ist 4.8.5-16.`

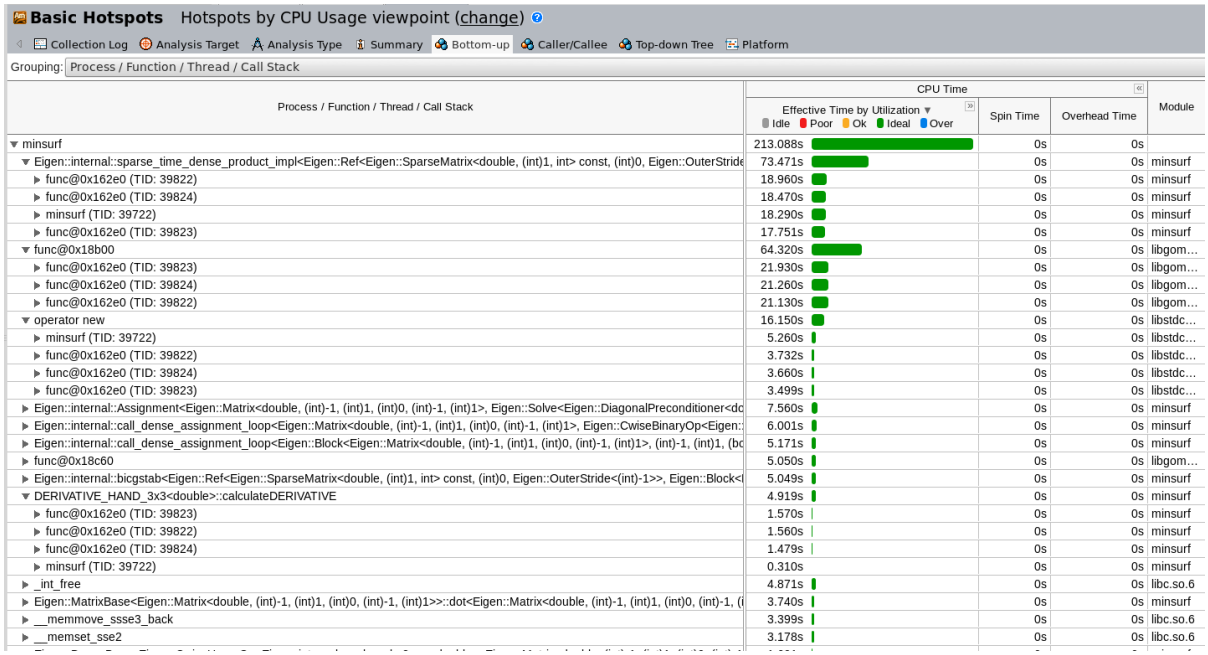
5.4.2 Parallelisierung im Code

Die Parallelisierung von *BiCGSTAB* mittels *openMP* ist für dünnbesetzte Matrizen, die im *row-major* – Format abgespeichert sind, in *eigen3.3.4* bereits implementiert (siehe [7]). Dieses Format ist gefordert, um die Matrix-Vektor-Multiplikationen, die hinter dem spezialisierten Konjugierte-Gradienten Verfahren stehen, möglichst effizient auszuführen. Da die vorliegende Systemmatrix stets symmetrisch ist, ist die Speichermethode im Hinblick auf Platzbedarf irrelevant, das *row-major* Format kann also gezielt und unproblematisch benutzt werden (7.5, Zeile 18).

In `JACOBIAN_HARD_3x3<T>` wird die Funktion `calculateSYSTEM_MATRIX(…)` parallelisiert. In dieser Funktion werden wiederholt `calculateDERIVATIVE(…)` und `insertSYSTEM_MATRIX(…)` aufgerufen, was nun auf die Kerne verteilt ist (siehe Kapitel 5.2.1).

5.4.3 Theoretische Analyse und Laufzeittests

Eine weitere Hotspot-Analyse mit vier Kernen wird durchgeführt, um zu überprüfen, wie die benötigte Rechenzeit für die Funktionen bei paralleler Ausführung entsprechend aufgeteilt wird.



Process / Function / Thread / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Module
minsurf	213.088s	0s	0s	
Eigen::internal::sparse_time_dense_product_impl<Eigen::Ref<Eigen::SparseMatrix<double, (int)1, int> const, (int)0, Eigen::OuterStride	73.471s	0s	0s	minsurf
func@0x162e0 (TID: 39822)	18.960s	0s	0s	minsurf
func@0x162e0 (TID: 39824)	18.470s	0s	0s	minsurf
minsurf (TID: 39722)	18.290s	0s	0s	minsurf
func@0x162e0 (TID: 39823)	17.751s	0s	0s	minsurf
func@0x18b00	64.320s	0s	0s	libgom…
func@0x162e0 (TID: 39823)	21.930s	0s	0s	libgom…
func@0x162e0 (TID: 39824)	21.260s	0s	0s	libgom…
func@0x162e0 (TID: 39822)	21.130s	0s	0s	libgom…
operator new	16.150s	0s	0s	libstdc…
minsurf (TID: 39722)	5.260s	0s	0s	libstdc…
func@0x162e0 (TID: 39822)	3.732s	0s	0s	libstdc…
func@0x162e0 (TID: 39824)	3.660s	0s	0s	libstdc…
func@0x162e0 (TID: 39823)	3.499s	0s	0s	libstdc…
Eigen::internal::Assignment<Eigen::Matrix<double, (int)-1, (int)1, (int)0, (int)-1, (int)1>, Eigen::Solve<Eigen::DiagonalPreconditioner<dc	7.560s	0s	0s	minsurf
Eigen::internal::call_dense_assignment_loop<Eigen::Matrix<double, (int)-1, (int)1, (int)0, (int)-1, (int)1>, Eigen::CwiseBinaryOp<Eigen::	6.001s	0s	0s	minsurf
Eigen::internal::call_dense_assignment_loop<Eigen::Block<Eigen::Matrix<double, (int)-1, (int)1, (int)0, (int)-1, (int)1>, (int)-1, (int)1, (bc	5.171s	0s	0s	minsurf
func@0x18c60	5.050s	0s	0s	libgom…
Eigen::internal::bicgstab<Eigen::Ref<Eigen::SparseMatrix<double, (int)1, int> const, (int)0, Eigen::OuterStride<(int)-1>>, Eigen::Block<	5.049s	0s	0s	minsurf
DERIVATIVE_HAND_3x3<double>::calculateDERIVATIVE	4.919s	0s	0s	minsurf
func@0x162e0 (TID: 39823)	1.570s	0s	0s	minsurf
func@0x162e0 (TID: 39822)	1.560s	0s	0s	minsurf
func@0x162e0 (TID: 39824)	1.479s	0s	0s	minsurf
minsurf (TID: 39722)	0.310s	0s	0s	minsurf
_int_free	4.871s	0s	0s	libc.so.6
Eigen::MatrixBase<Eigen::Matrix<double, (int)-1, (int)1, (int)0, (int)-1, (int)1>>::dot<Eigen::Matrix<double, (int)-1, (int)1, (int)0, (int)-1, (	3.740s	0s	0s	minsurf
__memmove_ssse3_back	3.399s	0s	0s	libc.so.6
__memset_sse2	3.178s	0s	0s	libc.so.6

Abbildung 15: Hotspot-Test, vier Kerne, 600x600

Wie man sieht, ist der *eigen* – Löser für das vorliegende Problem nicht optimal parallelisiert. Der große serielle Block (siehe Kapitel 5.4.1, Zeile 2) wird zwar parallel ausgeführt, jedoch bleibt der serielle Teil in Zeilen 16-18 erhalten. Das Füllen der Systemmatrix ist jetzt allerdings gut auf die Kerne verteilt (Zeile 21). Der Grund für die schlechte parallele Ausführung des Lösers wird in Kapitel 5.4.3 erklärt.

Unter der Annahme, dass von der gesamten seriellen Laufzeit $T_{real}(1) = 84{,}696$ Sekunden der erwähnte Block von $T_{1,par} = 45{,}048$ s und das Füllen der Systemmatrix mit $T_{2,par} = (3{,}200 + 1{,}619 + 1{,}450)$ s parallelisiert werden können, ergibt sich der parallele Anteil von *minsurf* zu

$$p = \frac{\sum_i T_{i,par}}{T_{real}(1)} = \frac{(45{,}048 + 3{,}200 + 1{,}619 + 1{,}450)\ \text{s}}{84{,}696\ \text{s}} \approx 60{,}6\%$$

und der sequentielle Anzeil zu $s = 1 - p = 39{,}4\%$. Nach dem Amdahlschen Gesetz beläuft sich die parallele Laufzeit mit N Kernen auf

$$\mathrm{T(N)} = \left(s + \frac{p}{N}\right) T_{real}(1) = \left(0{,}394 + \frac{0{,}606}{N}\right) \cdot 84{,}696\ \text{s}$$

Bei vier Kernen ergibt das $\mathrm{T}(4) = 46{,}202$ s. Die reale Laufzeit beläuft sich jedoch auf $T_{real}(4) = 56{,}931$ s.

Weiterhin beträgt der theoretische *Speedup*

$$S_p(N) = \frac{T(1)}{T(N)} = \frac{1}{0{,}394 + \frac{1 - 0{,}394}{N}}$$

Für vier Kerne ergibt sich damit $S_p(4) = 1{,}833$, praktisch ist jedoch $S_{p,real}(4) = \frac{84{,}696}{56{,}931} = 1{,}488$ zu beobachten. Die Differenz der theoretischen und praktischen Werte kommt durch den zusätzlichen overhead von *openMP* zustande, welcher komplett vernachlässigt wurde (siehe Zeile 7).

Um die Entwicklung der Laufzeit über mehrere Kerne zu verfolgen, wurden Laufzeittests durchgeführt. Diese (und alle folgenden Laufzeittests) wurden auf dem backend durchgeführt (CLX-MPI). Die Compilerflags sind:

```
-std=c++11 -O3 -fopenmp
```

(GNU-C++-Compiler, Version 4.8.5-16). Das folgende Batchskript wurde genutzt.

script.sh

```
#!/usr/local_rwth/bin/zsh
#BSUB -J OMPminsurf
#BSUB -o core24
#BSUB -W 0:10
#BSUB -M 8192
#BSUB -S 8192
#BSUB -n 24
#BSUB -a openmp
#BSUB -x
#BSUB -P lect0024
cd /home/ex535503/projects/minsurf
export OMP_PROC_BIND=spread
minsurf config.txt
```

Die Anzahl der Kerne bei `#BSUB -n` wird dementsprechend angepasst. Auch hier werden die numerischen Werte aus der Beispielsitzung 4.3 benutzt, die Anzahl der Stützstellen wird wieder auf 600 erhöht.

Für den Besetztheitsgrad ergibt sich damit $\beta \approx \frac{9}{600*600} = 0{,}0025\%$.

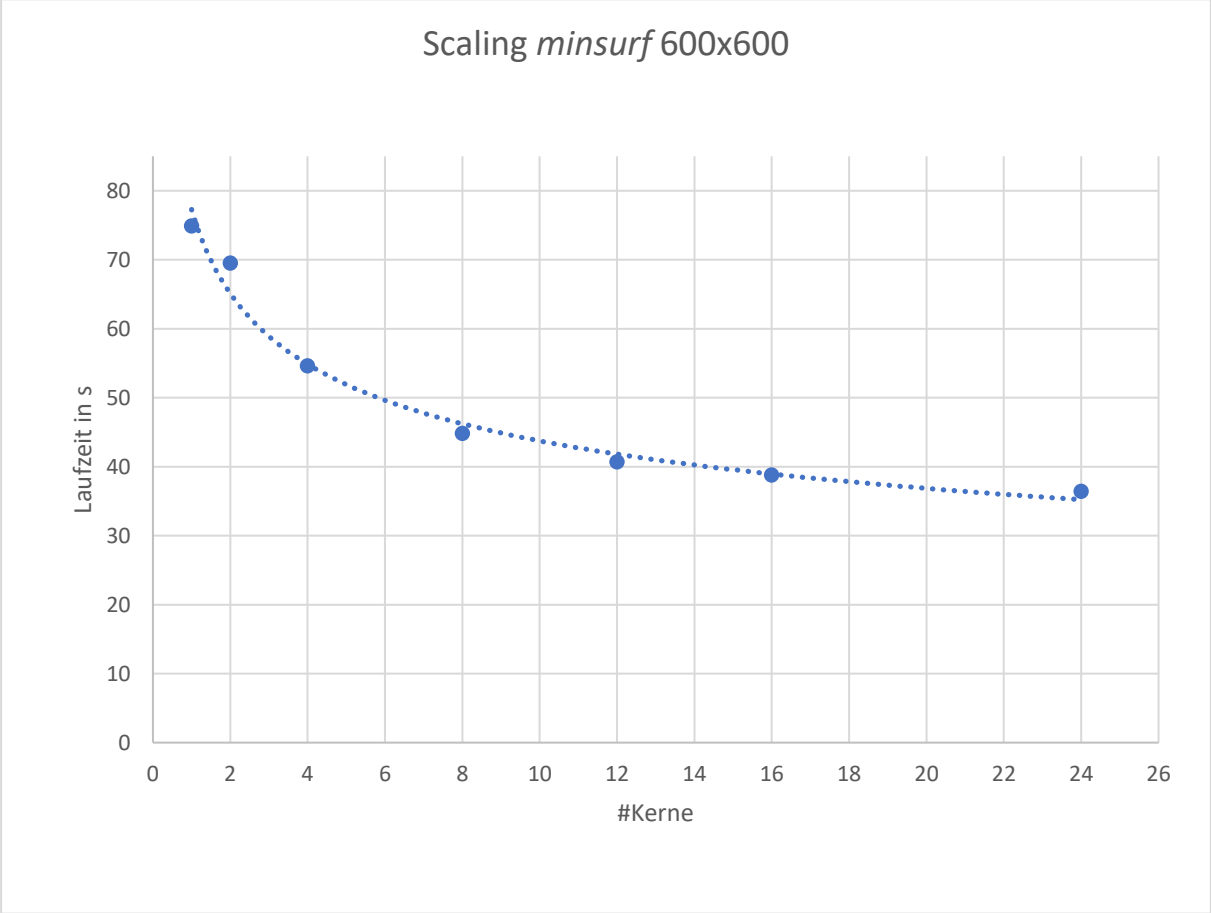


Abbildung 16: Laufzeit minsurf

Es ist zu erkennen, dass die Laufzeit für eine erhöhte Anzahl von Kernen gegen einen bestimmten Grenzwert konvergiert. Nach Amdahl beträgt diese Laufzeit

$$T(\infty) = s \cdot T_{backend}(1) = 0{,}394 \cdot 74{,}9\ \text{s} \approx 29{,}5\ \text{s}.$$

Der gemessene Wert für $T_{backend}(24) = 36{,}4$ s liegt schon recht nah an dem theoretischen Minimum, trotzdem skaliert *minsurf* langsam weiter.

Die Laufzeittests bestätigen in Verbindung mit der Theorie also grob die Erkenntnisse aus Kapitel 5.4.1 bezüglich des parallelen Anteils des Programms. An dieser Stelle noch zwei Diagramme zu Speedup $\left(S_p(N) = \frac{T(1)}{T(N)}\right)$ und Effizienz $\left(E_N = \frac{S_p(N)}{N}\right)$ für die verschiedenen Kernzahlen.

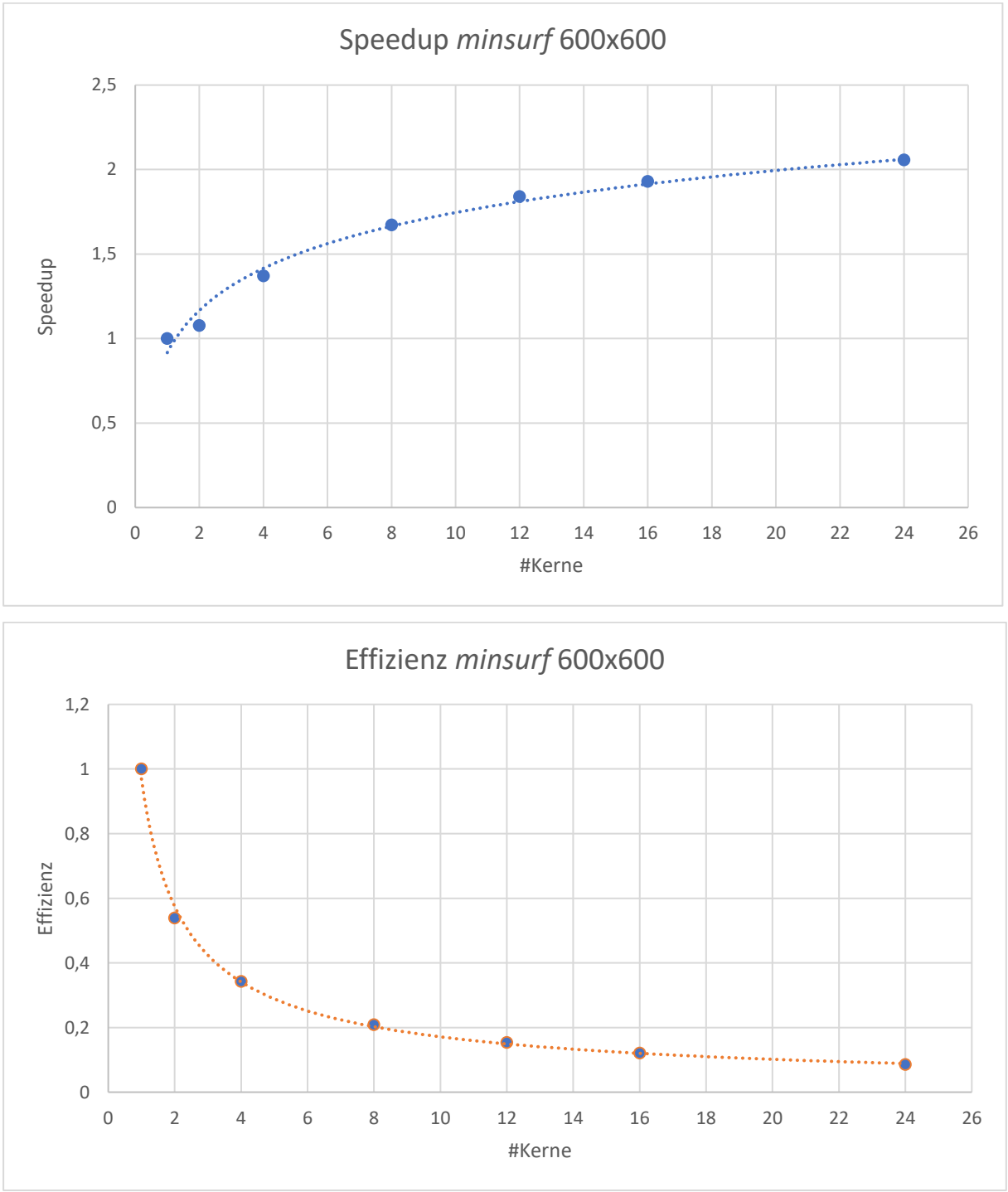


Abbildung 17: Speedup und Effizienz minsurf

Es bleibt zu klären, was der Grund für die mäßige Skalierung in *eigen* ist, und woraus sich damit der hohe serielle Anteil ergibt. Aus der Hotspot-Analyse geht hervor, dass der Algorithmus zur Lösung des LGS einen Großteil der Zeit beansprucht. Dessen parallele Eigenschaften sollten deshalb gesondert analysiert werden. Dazu wird ein Testprogramm entworfen, welches eine dünnbesetzte Matrix $A \in \mathbb{R}^{n \times n}$ und eine rechte Seite $b \in \mathbb{R}^n$ erstellt und das LGS $Ax = b$ mithilfe von *BiCGSTAB* von *eigen* auswertet. Mittels dieses Programms kann getestet werden, wie effizient die in *eigen* implementierte Parallelisierung für die vorliegende dünnbesetze Struktur einer Matrix ist, ohne eine Verfälschung durch das restliche Programm zu riskieren. Um aussagekräftige Ergebnisse zu erzielen, muss auf einige Dinge geachtet werden:

- Die Struktur der Matrix A muss der Struktur der originalen Systemmatrix JR ähneln. JR ist eine Bandmatrix mit einer vollbesetzen Hauptdiagonalen und zwei vollbesetzen Nebendiagonalen mit jeweils bis zu drei Elementen pro Zeile je Diagonale. A wird als Diagonalbandmatrix gewählt, besitzt also nur eine vollbesetze Diagonale mit jeweils bis zu neun Elementen pro Zeile (die ersten und letzten Zeilen haben weniger als neun Elemente, damit die Matrix symmetrisch ist).
- Die Besetztheitsgrade der Matrizen müssen nahezu übereinstimmen. Aufgrund der Wahl von A ist dies erfüllt; es gilt $\beta(A) \approx \frac{9}{n} \approx \beta(JR) \approx \frac{9}{n}$
- Das Vorgehen beim Lösen muss identisch sein. Anstatt einmal genau zu Lösen, approximiert *minsurf* mehrmals und mit zunehmender Genauigkeit lineare Gleichungssysteme. Das Testprogramm approximiert insgesamt 100-mal das LGS, um dem originalem Verhalten nahe zu kommen; beim ersten Mal mit einer *BiCGSTAB* – Iteration, beim zweiten Mal mit zwei usw. Nach jeder Approximation werden die rechte Seite und der Ergebnisvektor etwas angepasst, damit die Optimierung nicht einschreitet.

Die Ausführungen für $n = 300000$ und damit $\beta(A) \approx \frac{9}{300000} = 0{,}0027\%$ ergibt folgende Laufzeiten.

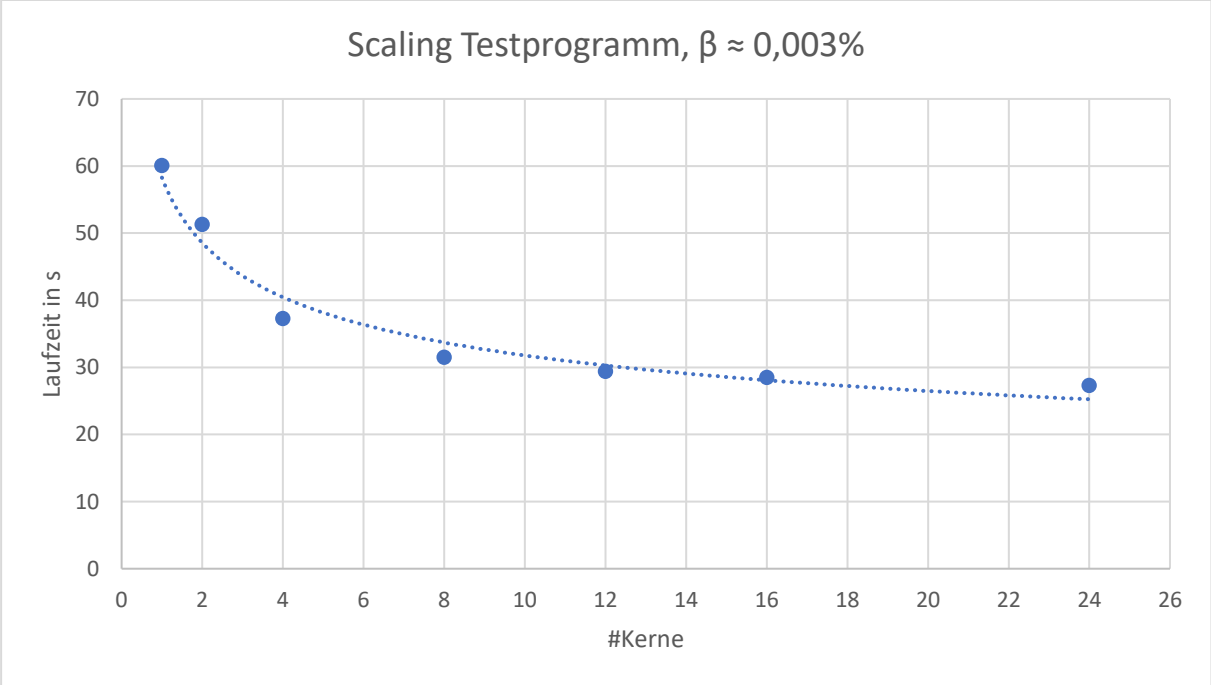


Abbildung 18: Laufzeit Testprogramm, Besetztheitsgrad der Matrix ähnlich zu minsurf

Das *scaling* ähnelt dem des Hauptprogramms sehr. Mittels der Laufzeiten lassen sich Effizienz und Speedup darstellen.

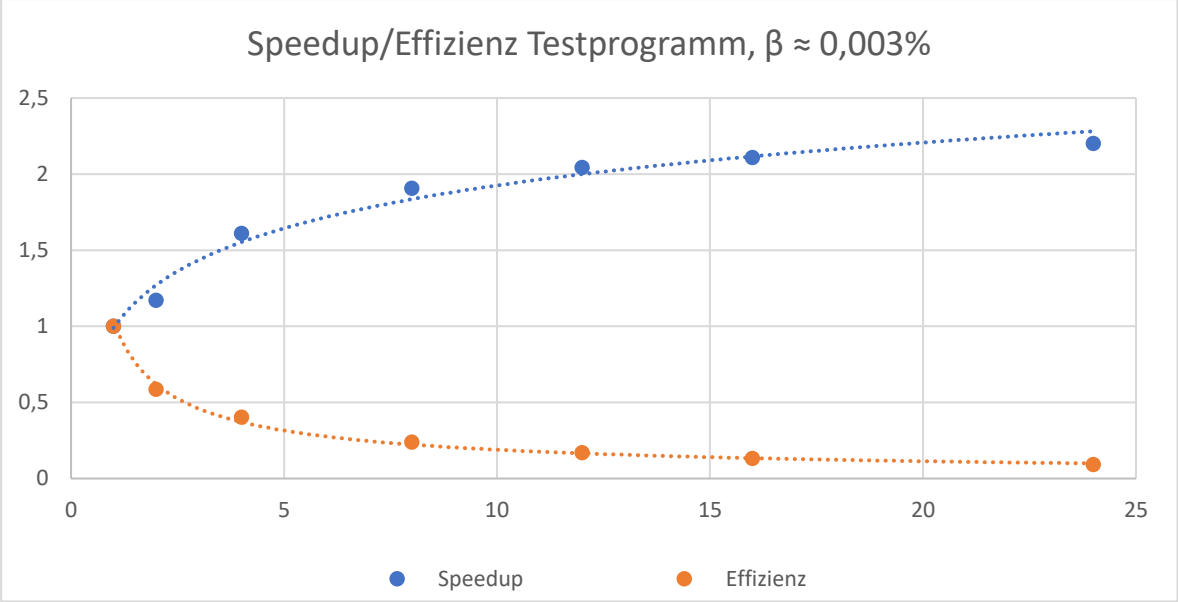


Abbildung 19: Speedup und Effizienz Testprogramm, Besetztheitsgrad der Matrix ähnlich zu minsurf

Der limitierende Faktor von *minsurf* ist also der *eigen* – Löser in Verbindung mit der vorliegenden Struktur der Matrix bzw. der extremen Dünnbesetztheit.

Im nächsten Schritt soll eine Verringerung des Besetzheitsgrades von A erreicht werden. Dies gelingt durch niedrigere Dimensionen der Matrix und mehr Elementen pro Zeile. Es ergibt sich mit 200 Elementen pro Zeile und Dimensionen von $n = 20000$ ein Besetztheitsgrad $\beta(A) \approx \frac{200}{20000} = 1\%$. Der Rest des Testprogrammes bleibt gleich. Nun kann das folgende *scaling* kann beobachtet werden.

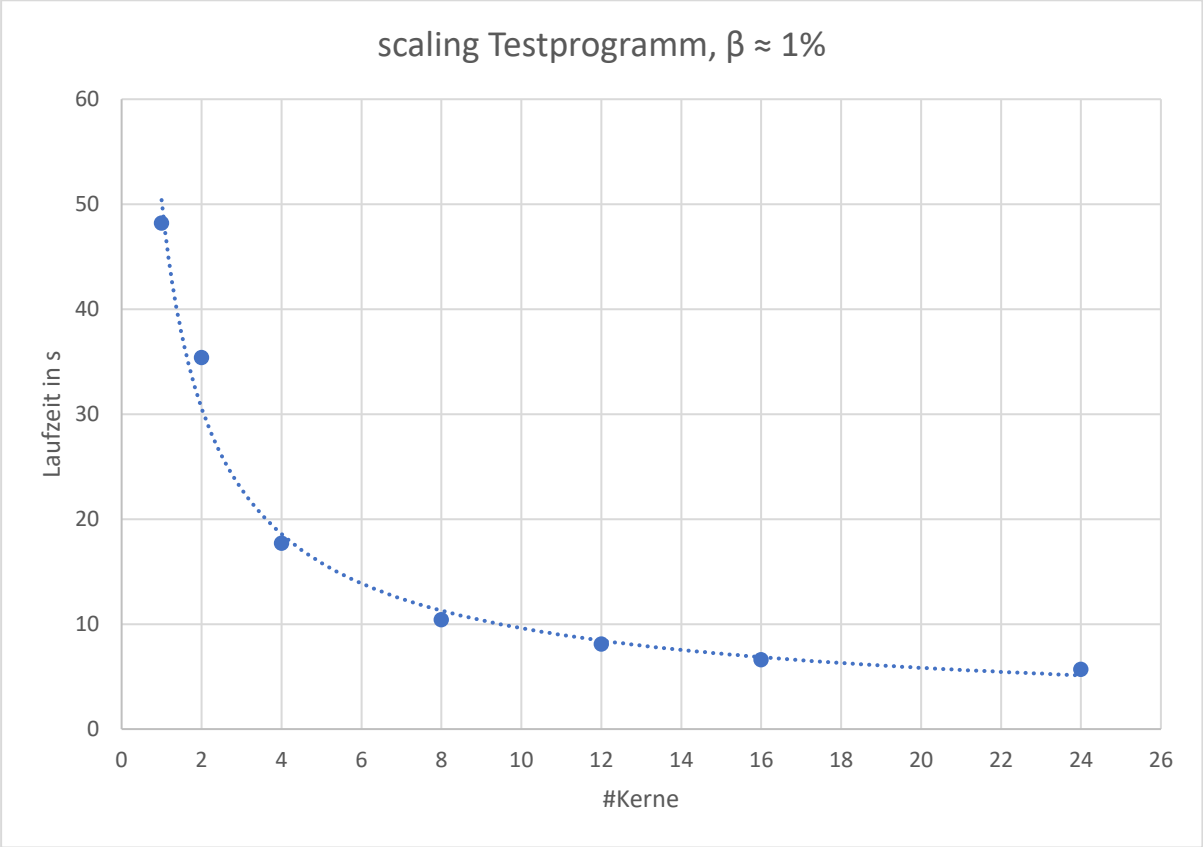
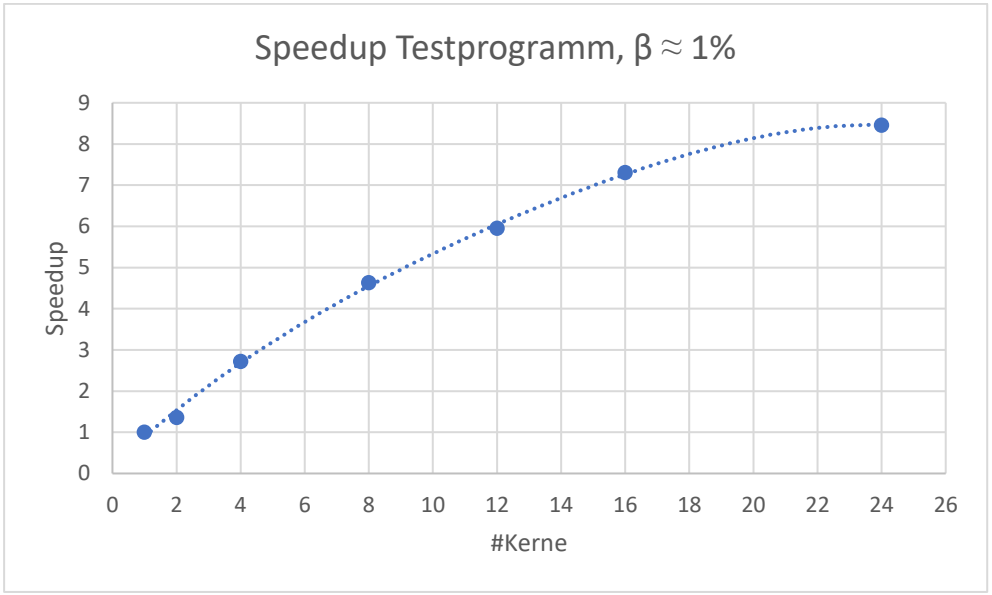


Abbildung 20: Laufzeit Testprogramm, Matrix dichter besetzt als in minsurf

Für Speedup und Effizienz ergeben sich weiterhin die folgenden Diagramme.



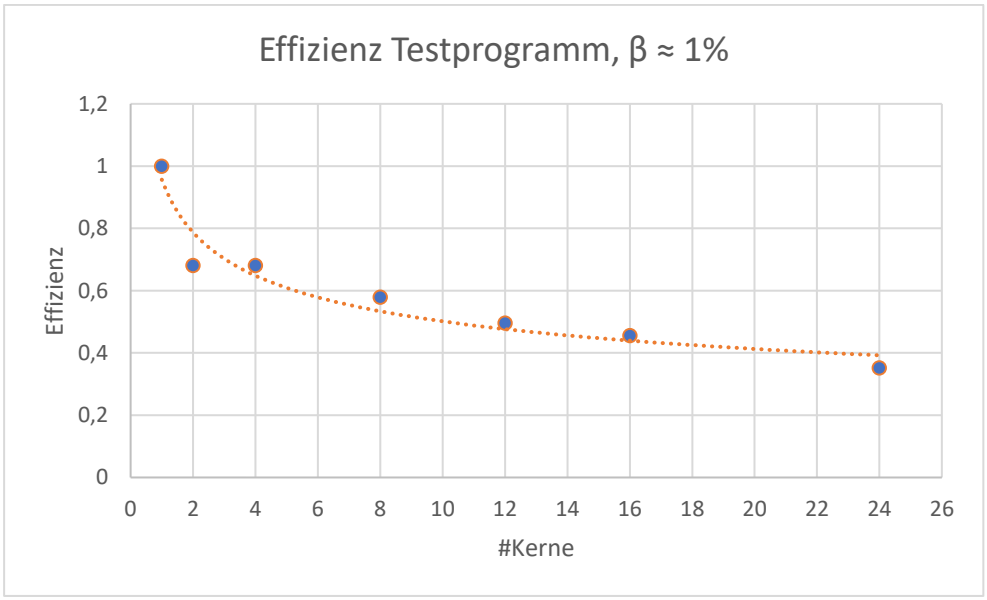


Abbildung 21: Speedup und Effizienz Testprogramm, Matrix dichter besetzt als in minsurf

Dieser Verlauf, welcher für dichter besetze Matrizen einen wesentlich bessere Parallelisierung zeigt, bestätigt die Aussage, dass die hohe Dünnbesetztheit verantwortlich für die langsame Skalierung ist. Offenbar überwiegt ist der interne Overhead und der sequentielle Anteil in dem *eigen* – Löser bei einer so gering besetzten Systemmatrix. Die Anforderungen an das Leistungsverhalten sind jedoch solide erfüllt, weshalb der *eigen* – Löser *BiCGSTAB* benutzt wird.

Es sei außerdem angemerkt, dass auch der Intel Compiler (ICC) des Clusters des RWTH Aachen ITCenters keine anderen Ergebnisse liefert.

6 Literaturverzeichnis

[1] Dahmen, W.; Reusken, A. (2008): *Numerik für Ingenieure und Naturwissenschaftler*, Kap. 5.6.2, 2. Auflage, Springer-Verlag Berlin Heidelberg.

[2] Pawlowski, R.; Shadid, J.; Simonis, J.; Walker, H.: *Globalization Techniques for Newton-Krylov methods and applications to the fully-coupled solution of the Navier-Stokes equations* (2005), Kap. 2. URL http://users.wpi.edu/~walker/Papers/globalization_techniques_report.pdf (letzter Abruf: 28.06.2018, 11:14 Uhr).

[3] Vorlesungsskript: *Anwendung der Differentialrechnung mehrerer Variabler. TUHH Hamburg.* URL https://www.math.uni-hamburg.de/teaching/export/tuhh/cm/a3/0708/vorl08_a3.pdf (letzter Abruf: 28.06.2018, 11:15 Uhr).

[4] J. E. Dennis, Jr., Robert B. Schnabel: *Numerical Methods for Unconstrained Optimization and Nonlinear Equations* (1996), Kap. 6. URL: https://epubs.siam.org/doi/pdf/10.1137/1.9781611971200.ch6 (letzter Abruf: 28.06.2018).

[5] Naumann, U.; Lotz, J.; Lux, F. Vorlesungsskript: *Programmier-Praktikum Physik/CES*. RWTH Aachen 2012/2013.

[6] Dijkstra, E.W.: *An ALGOL 60 Translator for the X1 (1961).* In: Mathematisch Centrum, Nr. 10. URL: http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF S21ff. (letzter Abruf 01.07.18).

[7] eigen – Dokumentation: URL: https://eigen.tuxfamily.org/dox/TopicMultiThreading.html (letzter Abruf: 02.07.2018)

7 Quellcode

7.1 DISCRETIZATION_CORE.h

```
1   #pragma once
2
3   #define VTK
4
5   #ifdef VTK
6   #include <vtkVersion.h>
7   #include <vtkSmartPointer.h>
8   #include <vtkPolyData.h>
9   #include <vtkPoints.h>
10  #include <vtkCellArray.h>
11  #include <vtkXMLPolyDataWriter.h>
12  #include <vtkXMLPolyDataReader.h>
13  #include <vtkPointData.h>
14  #include <vtkPolyDataNormals.h>
15  #include <vtkDoubleArray.h>
16  #include <vtkCellData.h>
17  #endif
18
19  #include <string>
20  #include <vector>
21  #include <iostream>
22  #include <fstream>
23  #include <omp.h>
24  #include <typeinfo>
25
26  #include "INTERPRETER.h"
27
28  template<class D>
29  D test_on_input_value(const char * what, unsigned int pos_in_input, const
30  std::vector<std::string>& input) {
31    D result;
32    try {
33      if (input.size() <= pos_in_input) {
34        std::stringstream tmp;
35        tmp << what << " NOT ENTERED!";
36        throw tmp.str();
37      }
38      if (std::is_same<D, int>::value) {
39        result = std::stoi(input[pos_in_input]);
40      }
41      else if (std::is_same<D, float>::value) {
42        result = std::stof(input[pos_in_input]);
43      }
44      else if (std::is_same<D, double>::value) {
45        result = std::stod(input[pos_in_input]);
46      }
47      else if (std::is_same<D, long double>::value) {
48        result = std::stold(input[pos_in_input]);
49      }
50      else {
51        std::stringstream tmp;
52        tmp << "DATA TYPE '" << typeid(D).name() << "' IS NOT SUPPORTED!" << std::endl;
53        throw tmp.str();
54      }
55      if (!(result > 0)) {
```

```
56          std::stringstream tmp;
57          tmp << what << " HAS TO BE LARGER THAN 0, IS: '" << result << "'!";
58          throw tmp.str();
59        }
60      }
61      catch (std::invalid_argument e) {
62        std::stringstream tmp;
63        tmp << "ERROR AT CONVERTING TO '" << typeid(D).name() << "' FOR " << what << " '"
64    << e.what() << "'!";
65        throw tmp.str();
66      }
67      catch (std::out_of_range e) {
68        std::stringstream tmp;
69        tmp << "ERROR AT CONVERTING TO '" << typeid(D).name() << "' FOR " << what << " '"
70    << e.what() << "'!";
71        throw tmp.str();
72      }
73      return result;
74    }
75
76    template <class T>
77    class  DISCRETIZATION_CORE {
78      int _s;
79      unsigned int _n;
80      unsigned int _m;
81      T _h;
82      std::vector<std::vector<T>> _surface;
83      std::vector<std::vector<T>> _stencils;
84      std::string _path;
85      std::vector<std::string> _input;
86      unsigned int _count;
87
88    protected:
89      DISCRETIZATION_CORE(std::vector<std::string>& input) : _input(input), _count(1) {
90        _s = test_on_input_value<int>("GRID POINTS PER ROW", 2, _input);
91        _h = 1. / (_s - 1);
92        _m = _s - 2;
93        _n = _m * _m;
94
95        //OUTPUT-PATH:
96        if (_input.size() <= 3) {
97          std::stringstream tmp;
98          tmp << "OUTPUT-PATH NOT ENTERED!";
99          throw tmp.str();
100       }
101       _path = _input[3];
102       //NORTH:
103       std::cout << "NORTH-";
104       if (_input.size() <= 4) {
105         std::stringstream tmp;
106         tmp << "NORTH FUNCTION NOT ENTERED!";
107         throw tmp.str();
108       }
109       INTERPRETER north = INTERPRETER(_input[4]);
110       //EAST:
111       std::cout << "EAST-";
112       if (_input.size() <= 5) {
113         std::stringstream tmp;
114         tmp << "EAST FUNCTION NOT ENTERED!";
```

```
115         throw tmp.str();
116       }
117       INTERPRETER east = INTERPRETER(_input[5]);
118       //SOUTH:
119       std::cout << "SOUTH-";
120       if (_input.size() <= 6) {
121         std::stringstream tmp;
122         tmp << "SOUTH FUNCTION NOT ENTERED!";
123         throw tmp.str();
124       }
125       INTERPRETER south = INTERPRETER(_input[6]);
126       //WEST:
127       std::cout << "WEST-";
128       if (_input.size() <= 7) {
129         std::stringstream tmp;
130         tmp << "WEST FUNCTION NOT ENTERED!";
131         throw tmp.str();
132       }
133       INTERPRETER west = INTERPRETER(_input[7]);
134
135       _surface.resize(_s);
136       for (unsigned int i = 0; i < _s; i++) {
137         _surface[i].resize(_s);
138       }
139       _stencils.resize(_n);
140       for (unsigned int i = 0; i < (_s /*- 1*/); i++) {
141         _surface[0][i] = north.calculate({ i * 1.0 / (_s - 1) });
142         _surface[i][_s - 1] = east.calculate({ i * 1.0 / (_s - 1) });
143         _surface[_s - 1][_s - 1 - i] = south.calculate({ i * 1.0 / (_s - 1) });
144         _surface[_s - 1 - i][0] = west.calculate({ i * 1.0 / (_s - 1) });
145       }
146
147       double north_west = std::abs(north.calculate({ 0 }) - west.calculate({ 1 }));
148       double north_east = std::abs(north.calculate({ 1 }) - east.calculate({ 0 }));
149       double south_east = std::abs(south.calculate({ 0 }) - east.calculate({ 1 }));
150       double south_west = std::abs(south.calculate({ 1 }) - west.calculate({ 0 }));
151
152       if (north_west > 0.1) { std::cout << "BOUNDARY VALUES IN THE NORTH-WEST CORNER DO
153   NOT MATCH! TOTAL DIFFERENCE: " << north_west << std::endl; }
154       if (north_east > 0.1) { std::cout << "BOUNDARY VALUES IN THE NORTH-EAST CORNER DO
155   NOT MATCH! TOTAL DIFFERENCE: " << north_east << std::endl; }
156       if (south_east > 0.1) { std::cout << "BOUNDARY VALUES IN THE SOUTH-EAST CORNER DO
157   NOT MATCH! TOTAL DIFFERENCE: " << south_east << std::endl; }
158       if (south_west > 0.1) { std::cout << "BOUNDARY VALUES IN THE SOUTH-WEST CORNER DO
159   NOT MATCH! TOTAL DIFFERENCE: " << south_west << std::endl; }
160
161       if (north_west > 0.1 || north_east > 0.1 || south_east > 0.1 || south_west > 0.1)
162   {
163         std::string tmp;
164         std::cout << "CONTINUE? (y/n) ";
165         std::cin >> tmp;
166         while (tmp != "y" && tmp != "n") {
167           std::cout << "CONTINUE? (y/n) ";
168           std::cin >> tmp;
169         }
170         if (tmp == "n") {
171           throw std::string("BREAK!");
172         }
173       }
```

```
174     }
175
176     DISCRETIZATION_CORE(std::vector<std::string>& input, char * vtk_filename) :
177   _input(input), _count(1) {
178       //OUTPUT-PATH:
179       if (_input.size() <= 3) {
180         std::stringstream tmp;
181         tmp << "OUTPUT-PATH NOT ENTERED!";
182         throw tmp.str();
183       }
184       _path = _input[3];
185       read_vtp(vtk_filename);
186       _h = 1. / (_s - 1);
187       _m = _s - 2;
188       _n = _m * _m;
189       _stencils.resize(_n);
190     }
191
192     //GETTER-METHODS:
193     unsigned int s() { return _s; }
194     unsigned int n() { return _n; }
195     unsigned int m() { return _m; }
196     T h() { return _h; }
197     std::vector<std::vector<T>>& surface() { return _surface; }
198     std::vector<std::vector<T>>& stencils() { return _stencils; }
199     std::vector<std::string>& input() { return _input; }
200     unsigned int& count() { return _count; }
201     virtual T getZ(unsigned int i) = 0;
202     virtual bool stop() = 0;
203
204     //SETTER-METHODS:
205     virtual void insertRESIDUAL(unsigned int i, T value) = 0;
206
207     //CALCULATION-METHODS:
208     virtual std::vector<T> calculateSTENCIL(unsigned int i, unsigned int j) = 0;
209     virtual T calculateRESIDUAL(const std::vector<T>& stencil) = 0;
210     virtual void solve() = 0;
211     void solve_core() {
212       do {
213         updateSurface();
214
215   #pragma omp parallel for
216         for (int i = 0; i < _m; i++) {
217           for (unsigned int j = 0; j < _m; j++) {
218             unsigned int pos = i * _m + j;
219             std::vector<T> stencil = calculateSTENCIL(i, j);
220             _stencils[pos] = stencil;
221             insertRESIDUAL(pos, calculateRESIDUAL(stencil));
222           }
223         }
224
225         solve();
226       } while (stop());
227
228       updateSurface();
229     }
230     //Calculates the arithmetic mean over the boundary values. The result are the values
231   of the start vector for the newton iteration:
232     T calculate_arithmetic_mean() {
```

```
233       T result = 0;
234   #pragma omp parallel for reduction(+:result)
235       for (int i = 0; i < (_s - 1); i++) {
236         result += _surface[0][i] + _surface[i][0] + _surface[_s - 1][i] + _surface[i][_s
237   - 1];
238       }
239       result /= 4 * (_s - 1);
240       return result;
241     }
242
243     void output() {
244   #ifdef VTK
245       vtkSmartPointer<vtkPolyData> polydata =
246   vtkSmartPointer<vtkPolyData>::Take(custom_reader());
247
248       vtkSmartPointer<vtkXMLPolyDataWriter> writer =
249   vtkSmartPointer<vtkXMLPolyDataWriter>::New();
250   #if VTK_MAJOR_VERSION <= 5
251       writer->SetInput(polydata);
252   #else
253       writer->SetInputData(polydata);
254   #endif
255       writer->SetFileName(_path.c_str());
256       writer->Write();
257   #else
258       std::ofstream out(_path);
259       for (unsigned int i = 0; i < _surface.size(); i++) {
260         for (unsigned int j = 0; j < _surface[i].size(); j++) {
261           out << _surface[i][j] << " ";
262         }
263         out << std::endl;
264       }
265   #endif // VTK
266     }
267
268   private:
269     void updateSurface() {
270   #pragma omp parallel for
271       for (int i = 1; i < _s - 1; i++) {
272         for (unsigned int j = 1; j < _s - 1; j++) {
273           unsigned int pos = (i - 1) * _m + (j - 1);
274           _surface[i][j] = getZ(pos);
275         }
276       }
277     }
278
279   #ifdef VTK
280     vtkPolyData * custom_reader() {
281       vtkIdType number_of_points, number_of_triangles, N;
282       N = _s;
283       number_of_points = N * N;
284       number_of_triangles = 2 * (N - 1)*(N - 1);
285       vtkSmartPointer<vtkPoints> points
286         = vtkSmartPointer<vtkPoints>::New();
287       points->SetNumberOfPoints(number_of_points);
288
289       //write points
290       for (vtkIdType i = 0; i < N; i++) {
291         for (vtkIdType j = 0; j < N; j++) {
```

```
292           double x, y, z;
293           x = i * _h;
294           y = j * _h;
295           z = _surface[i][j];
296           points->SetPoint(i*N + j, x, y, z);
297         }
298       }
299
300       //insert triangle-cells
301       vtkSmartPointer<vtkCellArray> polys
302         = vtkSmartPointer<vtkCellArray>::New();
303       for (vtkIdType i = 0; i < N - 1; i++)
304       {
305         for (vtkIdType j = 0; j < N - 1; j++)
306         {
307           polys->InsertNextCell(3);
308           polys->InsertCellPoint(i*N + j);
309           polys->InsertCellPoint(i*N + j + 1);
310           polys->InsertCellPoint((i + 1)*N + j + 1);
311
312           polys->InsertNextCell(3);
313           polys->InsertCellPoint(i*N + j);
314           polys->InsertCellPoint((i + 1)*N + j);
315           polys->InsertCellPoint((i + 1)*N + j + 1);
316         }
317       }
318
319       vtkPolyData * polydata = vtkPolyData::New();
320       polydata->SetPoints(points);
321       polydata->SetPolys(polys);
322       return polydata;
323     }
324   #endif // VTK
325
326     void read_vtp(char * filename) {
327   #ifdef VTK
328   const char* name = filename;
329       size_t l = strlen(name);
330       if (name[l - 4] != '.' || name[l - 3] != 'v' || name[l - 2] != 't' || name[l - 1]
331   != 'p') {
332         std::stringstream tmp;
333         tmp << "WRONG FILE FORMAT - USE MINSURF OUTPUT FILE!";
334         throw tmp.str();
335       }
336       // Read the file
337       double timestamp = omp_get_wtime();
338
339       vtkSmartPointer<vtkXMLPolyDataReader> reader =
340         vtkSmartPointer<vtkXMLPolyDataReader>::New();
341       reader->SetFileName(filename);
342       reader->Update();
343
344       std::cout << "READING '" << filename << "' ." << std::flush;
345
346       // Extract the polydata
347       vtkSmartPointer<vtkPolyData> polydata =
348         reader->GetOutput();
349                       std::cout << "." << std::flush;
350       // Get the number of points in the polydata
```

```
351       unsigned int idNumPointsInFile = polydata->GetNumberOfPoints();
352       _s = std::sqrt(idNumPointsInFile);
353       _surface.resize(_s);
354
355       for (unsigned int i = 0; i < _s; i++) {
356         _surface[i].resize(_s);
357         for (unsigned int j = 0; j < _s; j++) {
358           double p[3];
359           polydata->GetPoint(i*_s + j, p);
360           _surface[i][j] = p[2];
361         }
362       }
363       std::cout << ". TOOK (" << omp_get_wtime() - timestamp << "s)" << std::endl;
364   #else
365       throw std:string("NOT IMPLEMENTED!");
366       return;
367   #endif // VTK
368     }
369   };
```

7.2 DERIVATIVE_CORE.h

```
1   #pragma once
2
3   #include <vector>
4   #include <string>
5
6   template <class T>
7   class DERIVATIVE_CORE {
8   protected:
9     //CALCULATION-METHODS:
10    virtual T calculateDERIVATIVE(unsigned int x, unsigned int derivative_to) = 0;
11  };
12
13  template <class T>
14  class DERIVATIVE_DIFFERENCEQUOTIENT : public DERIVATIVE_CORE<T> {
15    T _epsilon;
16  protected:
17    DERIVATIVE_DIFFERENCEQUOTIENT(T epsilon) : _epsilon(epsilon) {}
18
19    //GETTER-METHODS:
20    virtual std::vector<T> getSTENCILS(unsigned int i) = 0;
21    virtual T getRESIDUAL(unsigned int i) = 0;
22
23    //CALCULATION-METHODS:
24    virtual T calculateRESIDUAL(const std::vector<T> & stencil) = 0;
25    virtual T calculateDERIVATIVE(unsigned int x, unsigned int derivative_to) {
26      T R = getRESIDUAL(x);
27      std::vector<T> stencil = getSTENCILS(x);
28      stencil[derivative_to] += _epsilon;
29      T Re = calculateRESIDUAL(stencil);
30      return (Re - R) / _epsilon;
31    }
32  };
33
34  template <class T>
35  class DERIVATIVE_HAND_3x3 : public DERIVATIVE_CORE<T> {
36    T _h;
37  protected:
38    DERIVATIVE_HAND_3x3(T h) : _h(h) {}
```

```
39
40    //GETTER-METHODS:
41    virtual std::vector<T> getSTENCILS(unsigned int i) = 0;
42
43    //CALCULATION-METHODS:
44    virtual T calculateDERIVATIVE(unsigned int x, unsigned int derivative_to) {
45      std::vector<T> stencil = getSTENCILS(x);
46      T a = stencil[0];
47      T b = stencil[1];
48      T c = stencil[2];
49      T d = stencil[3];
50      T e = stencil[4];
51      T f = stencil[5];
52      T g = stencil[6];
53      T i = stencil[7];
54      T j = stencil[8];
55      if (derivative_to == 0 || derivative_to == 8) {
56        return ((b - i) * (f - d)) / (8 * _h * _h * _h * _h);
57      }
58      else if (derivative_to == 1) {
59        return (1 / (8 * _h * _h * _h * _h)) * (-a * d + a * f + 4 * b * d - 8 * b * e +
60  4 * b * f + c * d - c * f + 2 * d * d - 4 * d * f + d * g - 4 * d * i - d * j + 8 * e
61  * i + 2 * f * f - f * g - 4 * f * i + f * j + 8 * _h * _h);
62      }
63      else if (derivative_to == 2 || derivative_to == 6) {
64        return ((b - i) * (d - f)) / (8 * _h * _h * _h * _h);
65      }
66      else if (derivative_to == 3) {
67        return (1 / (8 * _h * _h * _h * _h)) * (-a * b + a * i + 2 * b * b + b * c + 4 *
68  b * d - 4 * b * f + b * g - 4 * b * i - b * j - c * i - 8 * d * e + 4 * d * i + 8 * e
69  * f - 4 * f * i - g * i + 2 * i * i + i * j + 8 * _h * _h);
70      }
71      else if (derivative_to == 4) {
72        return -(b * b - 2 * b* i + d * d - 2 * d * f + f * f + i * i + 8 * _h * _h) /
73  (2 * _h * _h * _h * _h);
74      }
75      else if (derivative_to == 5) {
76        return (1 / (8 * _h * _h * _h * _h)) * (a * b - a * i + 2 * b * b - b * c - 4 *
77  b * d + 4 * b * f - b * g - 4 * b * i + b * j + c * i + 8 * d * e - 4 * d * i - 8 * e
78  * f + 4 * f * i + g * i + 2 * i * i - i * j + 8 * _h * _h);
79      }
80      else if (derivative_to == 7) {
81        return (1 / (8 * _h * _h * _h * _h)) * (a * d - a * f - 4 * b * d + 8 * b * e -
82  4 * b * f - c * d + c * f + 2 * d * d - 4 * d * f - d * g + 4 * d * i + d * j - 8 * e
83  * i + 2 * f * f + f * g + 4 * f * i - f * j + 8 * _h * _h);
84      }
85      else {
86        throw std::string("NOT REACHABLE!");
87        return -1;
88      }
89    }
90  };
```

7.3 RESIDUAL_CORE.h

```
1   #pragma once
2
3   #include <vector>
4
5   template <class T>
```

```
6   class RESIDUAL_CORE {
7     T _h;
8   protected:
9     RESIDUAL_CORE(T h) : _h(h) {}
10
11    //GETTER-METHODS:
12    T h() { return _h; }
13
14    //CALCULATION-METHODS:
15    virtual T calculateRESIDUAL(const std::vector<T>& stencil) = 0;
16  };
17
18  template <class T>
19  class RESIDUAL_3x3 : public RESIDUAL_CORE<T> {
20    using RESIDUAL_CORE<T>::h;
21    /*
22    0  1  2
23    3  4  5
24    6  7  8
25    */
26  protected:
27    RESIDUAL_3x3<T>(T h) : RESIDUAL_CORE<T>(h) {}
28
29    //CALCULATION-METHODS:
30    virtual T calculateRESIDUAL(const std::vector<T> & Z) {
31      T tmp = (Z[5] - Z[3]) / (2. * h());
32      T I = (1. + tmp * tmp)*((Z[7] - 2. * Z[4] + Z[1]) / (h() * h()));
33      T II = -1. / (8. * h() * h() * h() * h())*(Z[5] - Z[3])*(Z[1] - Z[7])*(Z[2] - Z[0]
34  - Z[8] + Z[6]);
35      T tmp2 = (Z[1] - Z[7]) / (2. * h());
36      T III = (1. + tmp2 * tmp2)*((Z[3] - 2. * Z[4] + Z[5]) / (h() * h()));
37      return I + II + III;
38    }
39  };
40
41  template <class T>
42  class RESIDUAL_STAR : public RESIDUAL_CORE<T> {
43    using RESIDUAL_CORE<T>::h;
44    /*
45    x  0  x
46    1  2  3
47    x  4  x
48    */
49  protected:
50    RESIDUAL_STAR<T>(T h) : RESIDUAL_CORE<T>(h) {}
51
52    //CALCULATION-METHODS:
53    virtual T calculateRESIDUAL(const std::vector<T> & Z) {
54      T I = (Z[0] + Z[1] - 4 * Z[2] + Z[3] + Z[4]) / (h() * h());
55      return I;
56    }
57  };
```

7.4 SYSTEM_MATRIX_CORE.h

```
1   #pragma once
2
3   #include <vector>
4
5   template<class T>
```

```
6   class SYSTEM_MATRIX_CORE {
7     unsigned int _m;
8   protected:
9     SYSTEM_MATRIX_CORE(unsigned int m) : _m(m) {}
10
11    //GETTER-METHODS:
12    unsigned int m() { return _m; }
13
14    //SETTER-METHODS:
15    virtual void insertSYSTEM_MATRIX(unsigned int i, unsigned int j, T value) = 0;
16
17    //CALCULATION-METHODS:
18    virtual void calcualteSYSTEM_MATRIX(std::vector<std::vector<T>>& surface) = 0;
19  };
20
21  template<class T>
22  class LAPLACIAN_OPERATOR : public SYSTEM_MATRIX_CORE<T> {
23    using SYSTEM_MATRIX_CORE<T>::m;
24    /*
25    x  0  x
26    1  2  3
27    x  4  x
28    */
29  protected:
30    LAPLACIAN_OPERATOR(unsigned int m) : SYSTEM_MATRIX_CORE<T>(m) {}
31
32    //SETTER-METHODS:
33    virtual void insertSYSTEM_MATRIX(unsigned int i, unsigned int j, T value) = 0;
34
35    //CALCULATION-METHODS:
36    void calcualteSYSTEM_MATRIX(std::vector<std::vector<T>>& surface) {
37  #pragma omp parallel for
38      for (unsigned int i = 0; i < m() * m(); i++) {
39        insertSYSTEM_MATRIX(i, i, -4);
40      }
41  #pragma omp parallel for
42      for (unsigned int i = 0; i < m() * m() - 1; i++) {
43        if (!((i + 1) % m() == 0)) {
44          insertSYSTEM_MATRIX(i, i + 1, 1);
45        }
46      }
47  #pragma omp parallel for
48      for (unsigned int i = 1; i < m() * m(); i++) {
49        if (!(i % m() == 0)) {
50          insertSYSTEM_MATRIX(i, i - 1, 1);
51        }
52      }
53  #pragma omp parallel for
54      for (unsigned int i = 0; i < m() * m() - m(); i++) {
55        insertSYSTEM_MATRIX(i, i + m(), 1);
56      }
57  #pragma omp parallel for
58      for (unsigned int i = m(); i < m() * m(); i++) {
59        insertSYSTEM_MATRIX(i, i - m(), 1);
60      }
61    }
62  };
63
64  template <class T>
65  class JACOBIAN_HARD_3x3 : public SYSTEM_MATRIX_CORE<T> {
```

```
66      using SYSTEM_MATRIX_CORE<T>::m;
67      /*
68      0  1  2
69      3  4  5
70      6  7  8
71      */
72    public:
73      JACOBIAN_HARD_3x3(unsigned int m) : SYSTEM_MATRIX_CORE<T>(m) {}
74
75      //SETTER-METHODS:
76      virtual void insertSYSTEM_MATRIX(unsigned int i, unsigned int j, T value) = 0;
77
78      //CALCULATION-METHODS:
79      virtual T calculateDERIVATIVE(unsigned int x, unsigned int derivative_to) = 0;
80      void calcualteSYSTEM_MATRIX(std::vector<std::vector<T>>& surface) {
81        insertSYSTEM_MATRIX(0, 0, calculateDERIVATIVE(0, 4));
82        insertSYSTEM_MATRIX(0, 1, calculateDERIVATIVE(0, 5));
83        insertSYSTEM_MATRIX(0, m(), calculateDERIVATIVE(0, 7));
84        insertSYSTEM_MATRIX(0, m() + 1, calculateDERIVATIVE(0, 8));
85
86    #pragma omp parallel for
87        for (int i = 1; i < m() - 1; i++) {
88          insertSYSTEM_MATRIX(i, i - 1, calculateDERIVATIVE(i, 3));
89          insertSYSTEM_MATRIX(i, i, calculateDERIVATIVE(i, 4));
90          insertSYSTEM_MATRIX(i, i + 1, calculateDERIVATIVE(i, 5));
91          insertSYSTEM_MATRIX(i, m() + i - 1, calculateDERIVATIVE(i, 6));
92          insertSYSTEM_MATRIX(i, m() + i, calculateDERIVATIVE(i, 7));
93          insertSYSTEM_MATRIX(i, m() + i + 1, calculateDERIVATIVE(i, 8));
94        }
95
96        insertSYSTEM_MATRIX(m() - 1, m() - 2, calculateDERIVATIVE(m() - 1, 3));
97        insertSYSTEM_MATRIX(m() - 1, m() - 1, calculateDERIVATIVE(m() - 1, 4));
98        insertSYSTEM_MATRIX(m() - 1, 2 * m() - 2, calculateDERIVATIVE(m() - 1, 6));
99        insertSYSTEM_MATRIX(m() - 1, 2 * m() - 1, calculateDERIVATIVE(m() - 1, 7));
100
101   #pragma omp parallel for
102       for (int j = 1; j < m() - 1; j++) {
103         insertSYSTEM_MATRIX(m() * j, (j - 1) * m(), calculateDERIVATIVE(m() * j, 1));
104         insertSYSTEM_MATRIX(m() * j, (j - 1) * m() + 1, calculateDERIVATIVE(m() * j,
105   2));
106         insertSYSTEM_MATRIX(m() * j, j * m(), calculateDERIVATIVE(m() * j, 4));
107         insertSYSTEM_MATRIX(m() * j, j * m() + 1, calculateDERIVATIVE(m() * j, 5));
108         insertSYSTEM_MATRIX(m() * j, (j + 1) * m(), calculateDERIVATIVE(m() * j, 7));
109         insertSYSTEM_MATRIX(m() * j, (j + 1) * m() + 1, calculateDERIVATIVE(m() * j,
110   8));
111
112         for (unsigned int i = 1; i < m() - 1; i++) {
113           insertSYSTEM_MATRIX(m() * j + i, (j - 1) * m() + (i - 1),
114   calculateDERIVATIVE(m() * j + i, 0));
115           insertSYSTEM_MATRIX(m() * j + i, (j - 1) * m() + i, calculateDERIVATIVE(m() *
116   j + i, 1));
117           insertSYSTEM_MATRIX(m() * j + i, (j - 1) * m() + (i + 1),
118   calculateDERIVATIVE(m() * j + i, 2));
119           insertSYSTEM_MATRIX(m() * j + i, j * m() + (i - 1), calculateDERIVATIVE(m() *
120   j + i, 3));
121           insertSYSTEM_MATRIX(m() * j + i, j * m() + i, calculateDERIVATIVE(m() * j + i,
122   4));
123           insertSYSTEM_MATRIX(m() * j + i, j * m() + (i + 1), calculateDERIVATIVE(m() *
124   j + i, 5));
```

```
125           insertSYSTEM_MATRIX(m() * j + i, (j + 1) * m() + (i - 1),
126   calculateDERIVATIVE(m() * j + i, 6));
127           insertSYSTEM_MATRIX(m() * j + i, (j + 1) * m() + i, calculateDERIVATIVE(m() *
128   j + i, 7));
129           insertSYSTEM_MATRIX(m() * j + i, (j + 1) * m() + (i + 1),
130   calculateDERIVATIVE(m() * j + i, 8));
131         }
132
133         insertSYSTEM_MATRIX(m() * j + m() - 1, (j - 1) * m() + (m() - 2),
134   calculateDERIVATIVE(m() * j + m() - 1, 0));
135         insertSYSTEM_MATRIX(m() * j + m() - 1, (j - 1) * m() + (m() - 1),
136   calculateDERIVATIVE(m() * j + m() - 1, 1));
137         insertSYSTEM_MATRIX(m() * j + m() - 1, j * m() + (m() - 2),
138   calculateDERIVATIVE(m() * j + m() - 1, 3));
139         insertSYSTEM_MATRIX(m() * j + m() - 1, j * m() + (m() - 1),
140   calculateDERIVATIVE(m() * j + m() - 1, 4));
141         insertSYSTEM_MATRIX(m() * j + m() - 1, (j + 1) * m() + (m() - 2),
142   calculateDERIVATIVE(m() * j + m() - 1, 6));
143         insertSYSTEM_MATRIX(m() * j + m() - 1, (j + 1) * m() + (m() - 1),
144   calculateDERIVATIVE(m() * j + m() - 1, 7));
145       }
146
147       insertSYSTEM_MATRIX(m() * (m() - 1), m() * (m() - 2), calculateDERIVATIVE(m() *
148   (m() - 1), 1));
149       insertSYSTEM_MATRIX(m() * (m() - 1), m() * (m() - 2) + 1, calculateDERIVATIVE(m()
150   * (m() - 1), 2));
151       insertSYSTEM_MATRIX(m() * (m() - 1), m() * (m() - 1), calculateDERIVATIVE(m() *
152   (m() - 1), 4));
153       insertSYSTEM_MATRIX(m() * (m() - 1), m() * (m() - 1) + 1, calculateDERIVATIVE(m()
154   * (m() - 1), 5));
155
156   #pragma omp parallel for
157       for (int i = 1; i < m() - 1; i++) {
158         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 2) + (i - 1),
159   calculateDERIVATIVE(m() * (m() - 1) + i, 0));
160         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 2) + i,
161   calculateDERIVATIVE(m() * (m() - 1) + i, 1));
162         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 2) + (i + 1),
163   calculateDERIVATIVE(m() * (m() - 1) + i, 2));
164         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 1) + (i - 1),
165   calculateDERIVATIVE(m() * (m() - 1) + i, 3));
166         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 1) + i,
167   calculateDERIVATIVE(m() * (m() - 1) + i, 4));
168         insertSYSTEM_MATRIX(m() * (m() - 1) + i, m() * (m() - 1) + (i + 1),
169   calculateDERIVATIVE(m() * (m() - 1) + i, 5));
170       }
171
172       insertSYSTEM_MATRIX(m() * m() - 1, m() * (m() - 1) - 2, calculateDERIVATIVE(m() *
173   m() - 1, 0));
174       insertSYSTEM_MATRIX(m() * m() - 1, m() * (m() - 1) - 1, calculateDERIVATIVE(m() *
175   m() - 1, 1));
176       insertSYSTEM_MATRIX(m() * m() - 1, m() * m() - 2, calculateDERIVATIVE(m() * m() -
177   1, 3));
178       insertSYSTEM_MATRIX(m() * m() - 1, m() * m() - 1, calculateDERIVATIVE(m() * m() -
179   1, 4));
180     }
181   };
```

7.5 DISCRETIZATIONS.h

```
1   #pragma once
2
```

```
3   #include "Eigen/SparseCore"
4
5   #include "Eigen/IterativeLinearSolvers"
6
7   #include "DERIVATIVE_CORE.h"
8   #include "DISCRETIZATION_CORE.h"
9   #include "SYSTEM_MATRIX_CORE.h"
10  #include "RESIDUAL_CORE.h"
11
12  template <class T>
13  class NEWTON : public DISCRETIZATION_CORE<T>, public RESIDUAL_3x3<T>, public
14  DERIVATIVE_HAND_3x3<T>, public JACOBIAN_HARD_3x3<T> {
15  protected:
16    using DISCRETIZATION_CORE<T>::h; using DISCRETIZATION_CORE<T>::m; using
17  DISCRETIZATION_CORE<T>::n; using DISCRETIZATION_CORE<T>::s;
18    typedef Eigen::SparseMatrix<T, Eigen::RowMajor> MT;
19    typedef Eigen::Matrix<T, Eigen::Dynamic, 1> VT;
20    MT JR;
21    VT R, Z, X, DR;
22    T norm_Fz_prev/*||F(z[k-1])||*/, norm_Fz_JRX_prev/*||F(z[k-1]) - F'(z[k-1]) * X[k-
23  1]||*/;
24    T ny, ny_prev, ny_max;
25    T epsilon, delta;
26    int max_newton_iterations, k_tilde, safety_output;
27
28    NEWTON(std::vector<std::string>& input) : DISCRETIZATION_CORE<T>(input),
29  RESIDUAL_3x3<T>(h()), DERIVATIVE_HAND_3x3<T>(h()), JACOBIAN_HARD_3x3<T>(m()), ny(0.1),
30  norm_Fz_JRX_prev(0.0), norm_Fz_prev(0.0), ny_prev(0.0), ny_max(0.9) {
31      Z = Eigen::Array<T, Eigen::Dynamic, 1>::Constant(n(),
32  DISCRETIZATION_CORE<T>::calculate_arithmetic_mean());
33      start();
34    }
35    NEWTON(std::vector<std::string>& input, char * vtk_filename) :
36  DISCRETIZATION_CORE<T>(input, vtk_filename), RESIDUAL_3x3<T>(h()),
37  DERIVATIVE_HAND_3x3<T>(h()), JACOBIAN_HARD_3x3<T>(m()), ny(0.1),
38  norm_Fz_JRX_prev(0.0), norm_Fz_prev(0.0), ny_prev(0.0), ny_max(0.9) {
39      Z.resize(n());
40
41  #pragma omp parallel for
42      for (unsigned int i = 0; i < m(); i++) {
43        for (unsigned int j = 0; j < m(); j++) {
44          Z(m()*i + j) = DISCRETIZATION_CORE<T>::surface()[i][j];
45        }
46      }
47      start();
48    }
49    NEWTON(DISCRETIZATION_CORE<T>* dis, VT Z) : DISCRETIZATION_CORE<T>(*dis),
50  RESIDUAL_3x3<T>(h()), DERIVATIVE_HAND_3x3<T>(h()), JACOBIAN_HARD_3x3<T>(m()), Z(Z),
51  ny(0.1), norm_Fz_JRX_prev(0.0), norm_Fz_prev(0.0), ny_prev(0.0), ny_max(0.9) {
52      start();
53    }
54
55    void start() {
56      max_newton_iterations = test_on_input_value<int>("MAX NEWTON ITERATIONS", 8,
57  DISCRETIZATION_CORE<T>::input()); // TEST ON MAX NEWTON ITERATIONS
58      safety_output = test_on_input_value<int>("NEWTON ITERATIONS PER OUTPUT", 9,
59  DISCRETIZATION_CORE<T>::input()); // TEST ON NEWTON ITERATIONS PER OUTPUT
60      epsilon = test_on_input_value<T>("EPSILON", 10, DISCRETIZATION_CORE<T>::input());
61  // TEST ON EPSILON
```

```
62        delta = test_on_input_value<T>("DELTA", 11, DISCRETIZATION_CORE<T>::input()); //
63    TEST ON DELTA
64        k_tilde = test_on_input_value<int>("K_TILDE", 12,
65    DISCRETIZATION_CORE<T>::input()); // TEST ON K_TILDE
66        JR.resize(n(), n());
67        JR.reserve(Eigen::Matrix<T, Eigen::Dynamic, 1>::Constant(n(), 9));
68        R.resize(n());
69        X.resize(n());
70        DR.resize(n());
71      }
72
73      //GETTER-METHODS:
74      virtual std::vector<T> getSTENCILS(unsigned int i) { return
75    DISCRETIZATION_CORE<T>::stencils()[i]; }
76      virtual T getZ(unsigned int i) { return Z(i); }
77      virtual bool stop() { return (DISCRETIZATION_CORE<T>::count()++ <=
78    max_newton_iterations && (DR.norm() / n()) > epsilon); }
79
80      //SETTER-METHODS:
81      virtual void insertRESIDUAL(unsigned int i, T value) { R(i) = value; }
82      virtual void insertSYSTEM_MATRIX(unsigned int i, unsigned int j, T value) {
83    JR.coeffRef(i, j) = value; }
84
85      //CALCULATION-METHODS:
86      virtual std::vector<T> calculateSTENCIL(unsigned int i, unsigned int j) {
87        return { DISCRETIZATION_CORE<T>::surface()[i][j],
88    DISCRETIZATION_CORE<T>::surface()[i][j + 1], DISCRETIZATION_CORE<T>::surface()[i][j +
89    2],
90          DISCRETIZATION_CORE<T>::surface()[i + 1][j], DISCRETIZATION_CORE<T>::surface()[i
91    + 1][j + 1], DISCRETIZATION_CORE<T>::surface()[i + 1][j + 2],
92          DISCRETIZATION_CORE<T>::surface()[i + 2][j], DISCRETIZATION_CORE<T>::surface()[i
93    + 2][j + 1], DISCRETIZATION_CORE<T>::surface()[i + 2][j + 2] };
94      }
95      virtual T calculateRESIDUAL(const std::vector<T> & stencil) { return
96    RESIDUAL_3x3<T>::calculateRESIDUAL(stencil); }
97      virtual T calculateDERIVATIVE(unsigned int x, unsigned int derivative_to) { return
98    DERIVATIVE_HAND_3x3<T>::calculateDERIVATIVE(x, derivative_to); }
99      virtual void solve() = 0;
100     void updateDRhelper() {
101   #pragma omp parallel for
102       for (int i = 0; i < m(); i++) {
103         for (unsigned int j = 0; j < m(); j++) {
104           unsigned int pos = i * m() + j;
105           std::vector<T> stencil = calculateSTENCIL(i, j);
106           DISCRETIZATION_CORE<T>::stencils()[pos] = stencil;
107           DR(pos) = calculateRESIDUAL(stencil);
108         }
109       }
110     }
111     void updateDR() {
112   #pragma omp parallel for
113       for (int i = 1; i < s() - 1; i++) {
114         for (unsigned int j = 1; j < s() - 1; j++) {
115           unsigned int pos = (i - 1) * m() + (j - 1);
116           DISCRETIZATION_CORE<T>::surface()[i][j] = Z(pos) + X(pos);
117         }
118       }
119       updateDRhelper();
120     }
121     void output() {
```

```
122       std::cout << "NEWTON ITERATION: " << DISCRETIZATION_CORE<T>::count() << ", " <<
123   "SCALED ERROR: '" << (DR.norm() / n()) << "', OUTPUTTING ." << std::flush;
124       double timestamp = omp_get_wtime();
125       DISCRETIZATION_CORE<T>::output();
126       std::cout << ".. TOOK (" << omp_get_wtime() - timestamp << "s)" << std::endl;
127     }
128     void calculateNY() {
129       //ny: Forcing term for linear system solver
130       ny = (R.norm() - norm_Fz_JRX_prev) / norm_Fz_prev;
131       ny < 0 ? ny = ny * (-1.) : 0;
132       T temp = pow(ny_prev, (1. + sqrt(5.)) / 2.);
133       (ny < temp) && (temp > 0.1) ? ny = temp : 0;
134       ny > ny_max ? ny = ny_max : 0;
135     }
136     void updateNY_calculateK() {
137       norm_Fz_prev = R.norm();
138       norm_Fz_JRX_prev = (R + JR * X).norm();
139       ny_prev = ny;
140       static bool ignore_warnings = false;
141       if (!ignore_warnings) {
142         static int k = 0;
143         if ((DR - R).norm() / DR.norm() < delta) {
144           k = k + 1;
145         }
146         else {
147           k = 0;
148         }
149         if (!(k < k_tilde)) {
150           std::string tmp;
151           std::cout << "SLOW CONVERGENCE!, CONTINUE? (y/n) ";
152           std::cin >> tmp;
153           while (tmp != "y" && tmp != "n") {
154             std::cout << "SLOW CONVERGENCE!, CONTINUE? (y/n) ";
155             std::cin >> tmp;
156           }
157           if (tmp == "n") {
158             std::cout << "LAST "; NEWTON<T>::output();
159             throw std::string("STOPPING NEWTON BECAUSE OF SLOW CONVERGENCE!");
160           }
161
162           ignore_warnings = true;
163         }
164       }
165     }
166     void lin_solve() {
167       Eigen::BiCGSTAB<MT> solver;
168       solver.setTolerance(ny);
169       solver.compute(JR);
170       X = solver.solve((-1)*R);
171       if (X != X) {
172         std::cout << "LAST "; NEWTON<T>::output();
173         throw std::string("STOPPING NEWTON BECAUSE OF NON CONVERGENCE! TRY DIFFERENT
174   BOUNDARY VALUES OR METHOD!");
175       }
176     }
177   };
178
179   template <class T>
180   class NEWTON_CLASSIC : public NEWTON<T> {
181     using DISCRETIZATION_CORE<T>::m; using DISCRETIZATION_CORE<T>::s;
```

```
182     using NEWTON<T>::JR; using NEWTON<T>::Z; using NEWTON<T>::X; using NEWTON<T>::R;
183   using NEWTON<T>::DR;
184     using NEWTON<T>::norm_Fz_prev; using NEWTON<T>::norm_Fz_JRX_prev;
185     using NEWTON<T>::ny; using NEWTON<T>::ny_prev; using NEWTON<T>::ny_max;
186     using NEWTON<T>::epsilon; using NEWTON<T>::delta;
187     using NEWTON<T>::max_newton_iterations;  using NEWTON<T>::k_tilde; using
188   NEWTON<T>::safety_output;
189     typedef Eigen::SparseMatrix<T, Eigen::RowMajor> MT;
190     typedef Eigen::Matrix<T, Eigen::Dynamic, 1> VT;
191   public:
192     NEWTON_CLASSIC(std::vector<std::string>& input) : NEWTON<T>(input) { start(); }
193     NEWTON_CLASSIC(std::vector<std::string>& input, char * vtk_filename) :
194   NEWTON<T>(input, vtk_filename) { start(); }
195     NEWTON_CLASSIC(DISCRETIZATION_CORE<T>* dis, VT Z) : NEWTON<T>(dis, Z) { start(); }
196   private:
197     void start() {
198       DISCRETIZATION_CORE<T>::solve_core(); std::cout << "SUCCESS! - LAST ";
199   NEWTON<T>::output();
200     }
201     virtual void solve() {
202       static bool first_iter = true;
203
204       if (DISCRETIZATION_CORE<T>::count() % safety_output == 0) {
205         NEWTON<T>::output();
206       }
207     JACOBIAN_HARD_3x3<T>::calcualteSYSTEM_MATRIX(DISCRETIZATION_CORE<T>::surface());
208       if (!first_iter) {
209         NEWTON<T>::calculateNY();
210       }
211       NEWTON<T>::lin_solve();
212       NEWTON<T>::updateDR();
213       Z = Z + X;
214       NEWTON<T>::updateNY_calculateK();
215       first_iter = false;
216     }
217   };
218
219   template <class T>
220   class NEWTON_DAMPED_TRIVIAL : public NEWTON<T> {
221     using DISCRETIZATION_CORE<T>::m; using DISCRETIZATION_CORE<T>::s;
222     using NEWTON<T>::JR; using NEWTON<T>::Z; using NEWTON<T>::X; using NEWTON<T>::R;
223   using NEWTON<T>::DR;
224     using NEWTON<T>::norm_Fz_prev; using NEWTON<T>::norm_Fz_JRX_prev;
225     using NEWTON<T>::ny; using NEWTON<T>::ny_prev; using NEWTON<T>::ny_max;
226     using NEWTON<T>::epsilon; using NEWTON<T>::delta;
227     using NEWTON<T>::max_newton_iterations;  using NEWTON<T>::k_tilde; using
228   NEWTON<T>::safety_output;
229     typedef Eigen::SparseMatrix<T, Eigen::RowMajor> MT;
230     typedef Eigen::Matrix<T, Eigen::Dynamic, 1> VT;
231     T lambda, min_lambda;
232   public:
233     NEWTON_DAMPED_TRIVIAL(std::vector<std::string>& input) : NEWTON<T>(input), lambda(1)
234   { start(); }
235     NEWTON_DAMPED_TRIVIAL(std::vector<std::string>& input, char * vtk_filename) :
236   NEWTON<T>(input, vtk_filename), lambda(1) { start(); }
237     NEWTON_DAMPED_TRIVIAL(DISCRETIZATION_CORE<T>* dis, VT Z) : NEWTON<T>(dis, Z),
238   lambda(1) { start(); }
239   private:
240     void start() {
```

```
241       min_lambda = test_on_input_value<T>("MIN LAMBDA", 13,
242   DISCRETIZATION_CORE<T>::input()); // TEST ON MIN LAMBDA
243       DISCRETIZATION_CORE<T>::solve_core(); std::cout << "SUCCESS! - LAST ";
244   NEWTON<T>::output();
245     }
246     virtual void solve() {
247       static bool first_iter = true;
248       if (DISCRETIZATION_CORE<T>::count() % safety_output == 0) {
249         NEWTON<T>::output();
250       }
251     JACOBIAN_HARD_3x3<T>::calcualteSYSTEM_MATRIX(DISCRETIZATION_CORE<T>::surface());
252
253       if (!first_iter) {
254         NEWTON<T>::calculateNY();
255       }
256       NEWTON<T>::lin_solve();
257       if (first_iter) {
258         DAMPER();
259       }
260       else {
261         if (R.norm() > DR.norm()) {
262           Z = Z + lambda * X;
263           if (lambda < 1) {
264             lambda = 2 * lambda;
265           }
266         }
267         else {
268           DAMPER();
269         }
270       }
271
272       NEWTON<T>::updateNY_calculateK();
273
274       first_iter = false;
275     }
276     void DAMPER() {
277       do {
278         updateDR();
279         lambda = lambda / 2;
280       } while (R.norm() < DR.norm() && 2 * lambda > min_lambda);
281       lambda = 2 * lambda;
282       Z = Z + lambda * X;
283     }
284     void updateDR() {
285   #pragma omp parallel for
286       for (int i = 1; i < s() - 1; i++) {
287         for (unsigned int j = 1; j < s() - 1; j++) {
288           unsigned int pos = (i - 1) * m() + (j - 1);
289           DISCRETIZATION_CORE<T>::surface()[i][j] = Z(pos) + lambda * X(pos);
290         }
291       }
292       NEWTON<T>::updateDRhelper();
293     }
294   };
295   template <class T>
296   class NEWTON_DAMPED_BACKTRACKING : public NEWTON<T> {
297     using DISCRETIZATION_CORE<T>::n; using DISCRETIZATION_CORE<T>::m; using
298   DISCRETIZATION_CORE<T>::s;
299     using NEWTON<T>::JR; using NEWTON<T>::Z; using NEWTON<T>::X; using NEWTON<T>::R;
300   using NEWTON<T>::DR;
```

```
301     using NEWTON<T>::norm_Fz_prev; using NEWTON<T>::norm_Fz_JRX_prev;
302     using NEWTON<T>::ny; using NEWTON<T>::ny_prev; using NEWTON<T>::ny_max;
303     using NEWTON<T>::epsilon; using NEWTON<T>::delta;
304     using NEWTON<T>::max_newton_iterations;  using NEWTON<T>::k_tilde; using
305   NEWTON<T>::safety_output;
306     typedef Eigen::SparseMatrix<T, Eigen::RowMajor> MT;
307     typedef Eigen::Matrix<T, Eigen::Dynamic, 1> VT;
308     VT DReps, DR0;
309     T, thetha, thetha_max, thetha_min;
310   public:
311     NEWTON_DAMPED_BACKTRACKING(std::vector<std::string>& input) : NEWTON<T>(input),
312   t(0.0001), thetha(1.), thetha_min(0.1), thetha_max(0.5) { start(); }
313     NEWTON_DAMPED_BACKTRACKING(std::vector<std::string>& input, char * vtk_filename) :
314   NEWTON<T>(input, vtk_filename), t(0.0001), thetha(1.), thetha_min(0.1),
315   thetha_max(0.5) { start(); }
316     NEWTON_DAMPED_BACKTRACKING(DISCRETIZATION_CORE<T>* dis, VT Z) : NEWTON<T>(dis, Z),
317   t(0.0001), thetha(1.), thetha_min(0.1), thetha_max(0.5) { start(); }
318   private:
319     void start() {
320       DReps.resize(n());
321       DR0.resize(n());
322       DISCRETIZATION_CORE<T>::solve_core(); std::cout << "SUCCESS! - LAST ";
323   NEWTON<T>::output();
324     }
325     virtual void solve() {
326       static bool first_iter = true;
327
328       if (DISCRETIZATION_CORE<T>::count() % safety_output == 0) {
329         NEWTON<T>::output();
330       }
331
332       JACOBIAN_HARD_3x3<T>::calcualteSYSTEM_MATRIX(DISCRETIZATION_CORE<T>::surface());
333
334       if (!first_iter) {
335         NEWTON<T>::calculateNY();
336       }
337
338       NEWTON<T>::lin_solve();
339
340       NEWTON<T>::updateDR();
341       T p_0 = R.squaredNorm() / 2.;
342       while (DR.norm() > ((1. - t * (1. - ny)) * R.norm())) {
343         updateDReps();
344         updateDR0();
345         T p_prime_0 = (DReps.squaredNorm() - DR0.squaredNorm()) / (2. * 0.000001);
346         T p_1 = DR.squaredNorm() / 2.;
347         thetha = -p_prime_0 / (2. * (p_1 - p_0 - p_prime_0));
348         if (thetha < thetha_min) { thetha = thetha_min; }
349         if (thetha > thetha_max) { thetha = thetha_max; }
350         X = thetha * X;
351         ny = 1. - thetha * (1. - ny);
352         NEWTON<T>::updateDR();
353       }
354       Z = Z + X;
355
356       NEWTON<T>::updateNY_calculateK();
357
358       first_iter = false;
359     }
360
```

```
361     void updateDReps() {
362   #pragma omp parallel for
363       for (int i = 1; i < s() - 1; i++) {
364         for (unsigned int j = 1; j < s() - 1; j++) {
365           unsigned int pos = (i - 1) * m() + (j - 1);
366           DISCRETIZATION_CORE<T>::surface()[i][j] = Z(pos) + 0.000001 * X(pos);
367         }
368       }
369   #pragma omp parallel for
370       for (int i = 0; i < m(); i++) {
371         for (unsigned int j = 0; j < m(); j++) {
372           unsigned int pos = i * m() + j;
373           std::vector<T> stencil = NEWTON<T>::calculateSTENCIL(i, j);
374           DISCRETIZATION_CORE<T>::stencils()[pos] = stencil;
375           DReps(pos) = NEWTON<T>::calculateRESIDUAL(stencil);
376         }
377       }
378     }
379     void updateDR0() {
380   #pragma omp parallel for
381       for (int i = 1; i < s() - 1; i++) {
382         for (unsigned int j = 1; j < s() - 1; j++) {
383           unsigned int pos = (i - 1) * m() + (j - 1);
384           DISCRETIZATION_CORE<T>::surface()[i][j] = Z(pos);
385         }
386       }
387   #pragma omp parallel for
388       for (int i = 0; i < m(); i++) {
389         for (unsigned int j = 0; j < m(); j++) {
390           unsigned int pos = i * m() + j;
391           std::vector<T> stencil = NEWTON<T>::calculateSTENCIL(i, j);
392           DISCRETIZATION_CORE<T>::stencils()[pos] = stencil;
393           DR0(pos) = NEWTON<T>::calculateRESIDUAL(stencil);
394         }
395       }
396     }
397   };
398
399   template <class T>
400   class LAPLACE : public DISCRETIZATION_CORE<T>, public RESIDUAL_STAR<T>, public
401   LAPLACIAN_OPERATOR<T> {
402     using DISCRETIZATION_CORE<T>::h; using DISCRETIZATION_CORE<T>::m; using
403   DISCRETIZATION_CORE<T>::n; using DISCRETIZATION_CORE<T>::s;
404     typedef Eigen::SparseMatrix<T, Eigen::RowMajor> MT;
405     typedef Eigen::Matrix<T, Eigen::Dynamic, 1> VT;
406     MT A;
407     VT R, Z, X;
408   public:
409     LAPLACE(std::vector<std::string>& input) : DISCRETIZATION_CORE<T>(input),
410   RESIDUAL_STAR<T>(h()), LAPLACIAN_OPERATOR<T>(m()) {
411       A.resize(n(), n());
412       A.reserve(Eigen::Matrix<T, Eigen::Dynamic, 1>::Constant(n(), 5));
413       R.resize(n());
414       X.resize(n());
415       Z.resize(n());
416
417       DISCRETIZATION_CORE<T>::solve_core();
418
419       if (DISCRETIZATION_CORE<T>::input()[0].size() == 1) {
420         DISCRETIZATION_CORE<T>::output();
```

```
421         return;
422       }
423       if (!std::isdigit(static_cast<unsigned
424   char>(DISCRETIZATION_CORE<T>::input()[0][1]))) {
425         throw std::string("SELECTION DIGIT FOR SOLVER IS NON DIGIT!");
426       }
427       int D = std::stoi(DISCRETIZATION_CORE<T>::input()[0].substr(1,
428   DISCRETIZATION_CORE<T>::input()[0].size() - 1));
429       if (D == 2) {
430         NEWTON_CLASSIC<T>(static_cast<DISCRETIZATION_CORE<T>*>(this), Z);
431       }
432       else if (D == 3) {
433         NEWTON_DAMPED_TRIVIAL<T>(static_cast<DISCRETIZATION_CORE<T>*>(this), Z);
434       }
435       else if (D == 4) {
436         NEWTON_DAMPED_BACKTRACKING<T>(static_cast<DISCRETIZATION_CORE<T>*>(this), Z);
437       }
438       else {
439         std::stringstream tmp;
440         tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!";
441         throw tmp.str();
442       }
443     }
444   private:
445     //GETTER-METHODS:
446     virtual std::vector<T> getSTENCILS(unsigned int i) { return
447   DISCRETIZATION_CORE<T>::stencils()[i]; }
448     virtual T getZ(unsigned int i) { return Z(i); }
449     virtual bool stop() { return false; }
450
451     //SETTER-METHODS:
452     virtual void insertRESIDUAL(unsigned int i, T value) { R(i) = value; }
453     virtual void insertSYSTEM_MATRIX(unsigned int i, unsigned int j, T value) {
454   A.coeffRef(i, j) = value; }
455
456     //CALCULATION-METHODS:
457     virtual std::vector<T> calculateSTENCIL(unsigned int i, unsigned int j) {
458       return { DISCRETIZATION_CORE<T>::surface()[i][j + 1],
459         DISCRETIZATION_CORE<T>::surface()[i + 1][j], DISCRETIZATION_CORE<T>::surface()[i
460   + 1][j + 1], DISCRETIZATION_CORE<T>::surface()[i + 1][j + 2],
461         DISCRETIZATION_CORE<T>::surface()[i + 2][j + 1] };
462     }
463     virtual T calculateRESIDUAL(const std::vector<T> & stencil) { return
464   RESIDUAL_STAR<T>::calculateRESIDUAL(stencil); }
465     virtual void solve() {
466       LAPLACIAN_OPERATOR<T>::calcualteSYSTEM_MATRIX(DISCRETIZATION_CORE<T>::surface());
467
468       A *= 1 / (h() * h());
469
470       Eigen::BiCGSTAB<MT> solver;
471       solver.setTolerance(10e-3);
472       solver.compute((-1)*A);
473       X = solver.solve(R);
474       if (X != X) {
475         std::cout << "LAST "; NEWTON<T>::output();
476         throw std::string("STOPPING LAPLACE BECAUSE OF NON CONVERGENCE! TRY DIFFERENT
477   BOUNDARY VALUES OR METHOD!");
478       }
479
480       Z = X;
```

```
481     }
482   };
```

7.6 INTERPRETER.h

```
1   #pragma once
2
3   #include <exception>
4   #include <vector>
5   #include <stack>
6   #include <string>
7   #include <cmath>
8   #include <algorithm>
9   #include <sstream>
10  #include <memory>
11
12  #include "TOKEN.h"
13
14  class INTERPRETER {
15    std::vector<std::shared_ptr<TOKEN>> tokens;
16    std::vector<std::shared_ptr<TOKEN>> rpn; //reverse polish notation
17    std::string expression;
18
19  public:
20    INTERPRETER(std::string expression) : expression(expression) {
21      interpret();
22    }
23    void interpret() {
24      //std::cout << "INPUT: " << expression << std::endl;
25
26      removeBlanks();
27      interpretExpression();
28      cleanUpForShuntingYard();
29
30      std::cout << "TOKENS: \t";
31      for (unsigned int i = 0; i < tokens.size(); i++) {
32        std::cout << *(tokens[i]) << " ";
33      }
34      std::cout << std::endl;
35
36      shuntingYard();
37
38      /*std::cout << "RPN: ";
39      for (unsigned int i = 0; i < rpn.size(); i++) {
40        std::cout << *(rpn[i]) << " ";
41      }
42      std::cout << std::endl;*/
43    }
44
45  private:
46    void error(std::string errorstring, unsigned int pos) {
47      std::stringstream tmp;
48      tmp << errorstring << " at position: " << std::to_string(pos + 1);
49      throw tmp.str();
50    }
51
52    void error(std::string errorstring) {
53      throw errorstring;
54    }
55
56    void removeBlanks() {
```

```
57        expression.assign(expression.begin(), remove_if(expression.begin(),
58    expression.end(), &isspace));
59      }
60
61      bool isOperator(char token) {
62        return token == '+' || token == '-' || token == '*' || token == '/' || token ==
63    '^';
64      }
65
66      bool isNumber(char token) {
67        return token == '0' || token == '1' || token == '2' || token == '3' || token ==
68    '4' || token == '5' || token == '6' || token == '7' || token == '8' || token == '9';
69      }
70
71      bool isLetter(char token) {
72        return  token == 'a' || token == 'b' || token == 'c' || token == 'd' || token ==
73    'e' || token == 'f' || token == 'g' || token == 'h' || token == 'i' || token == 'j' ||
74    token == 'k' ||
75          token == 'l' || token == 'm' || token == 'n' || token == 'o' || token == 'p' ||
76    token == 'q' || token == 'r' || token == 's' || token == 't' || token == 'u' || token
77    == 'v' ||
78          token == 'w' || token == 'x' || token == 'y' || token == 'z' || token == 'A' ||
79    token == 'B' || token == 'C' || token == 'D' || token == 'E' || token == 'F' || token
80    == 'G' ||
81          token == 'H' || token == 'I' || token == 'J' || token == 'K' || token == 'L' ||
82    token == 'M' || token == 'N' || token == 'O' || token == 'P' || token == 'Q' || token
83    == 'R' ||
84          token == 'S' || token == 'T' || token == 'U' || token == 'V' || token == 'W' ||
85    token == 'X' || token == 'Y' || token == 'Z';
86      }
87
88      bool isFunction(std::string test) {
89        return test == "exp" || test == "sin" || test == "cos" || test == "tan" || test ==
90    "arcsin" || test == "arccos" || test == "arctan" || test == "abs" || test == "sinh" ||
91    test == "cosh" || test == "tanh" || test == "arsinh" || test == "arcossh" || test ==
92    "artanh" || test == "min" || test == "max" || test == "ln" || test == "log10" || test
93    == "log2";
94      }
95
96      bool interpretExpression() {
97        char token;
98        for (unsigned int i = 0; i < expression.size(); i++) {
99          token = expression[i];
100
101         //OPERATOR:
102         if (isOperator(token)) {
103           tokens.push_back(std::shared_ptr<OPERATOR>(new OPERATOR(token)));
104           continue;
105         }
106
107         //SCALAR:
108         if (isNumber(token) || token == '.') {
109           std::string numberstring;
110           bool decimal_separator = false;
111           do {
112             if (token == '.') {
113               if (!decimal_separator) {
114                 decimal_separator = true;
115                 if (numberstring.empty()) {
116                   numberstring.push_back('0');
```

```
117                 }
118               }
119               else {
120                 error("SCALAR HAS MORE THAN ONE DECIMAL SEPERATOR!", tokens.size());
121                 return false;
122               }
123             }
124             numberstring.push_back(token);
125             if (++i >= expression.size()) {
126               break;
127             }
128             token = expression[i];
129           } while (isNumber(token) || token == '.');
130           if (token == '.') {
131             error("SCALAR ENDS WITH A DECIMAL SEPERATOR!", tokens.size());
132             return false;
133           }
134
135           tokens.push_back(std::shared_ptr<SCALAR>(new
136   SCALAR(std::stod(numberstring))));
137
138           i--;
139           continue;
140         }
141
142         //FUNCTION or VARIABLE:
143         if (isLetter(token)) {
144           std::string letterstring;
145           do {
146             letterstring.push_back(token);
147             if (++i >= expression.size()) {
148               break;
149             }
150             token = expression[i];
151           } while (isLetter(token));
152
153           //FUNCTION:
154           if (isFunction(letterstring)) {
155             tokens.push_back(std::shared_ptr<FUNCTION>(new FUNCTION(letterstring)));
156           }
157
158           //VARIABLE:
159           else {
160             tokens.push_back(std::shared_ptr<VARIABLE>(new VARIABLE(letterstring)));
161           }
162
163           i--;
164           continue;
165         }
166
167         //ARGUMENT_SEPARATOR:
168         if (token == ',') {
169           tokens.push_back(std::shared_ptr<ARGUMENT_SEPARATOR>(new ARGUMENT_SEPARATOR));
170           continue;
171         }
172
173         //BRACKETS:
174         if (token == '(') {
175           tokens.push_back(std::shared_ptr<OPENING_BRACKET>(new OPENING_BRACKET));
176           continue;
```

```
177         }
178         if (token == ')') {
179           tokens.push_back(std::shared_ptr<CLOSING_BRACKET>(new CLOSING_BRACKET));
180           continue;
181         }
182 
183         std::stringstream tmp;
184         tmp << "TOKEN '" << token << "' DOES NOT BELONG TO ANY CATEGORY!";
185         error(tmp.str(), tokens.size());
186         return false;
187       }
188       return true;
189     }
190 
191     bool cleanUpForShuntingYard() {
192       if (tokens.empty()) {
193         tokens.push_back(std::shared_ptr<SCALAR>(new SCALAR(0)));
194       }
195 
196       //TEST ON SIGN OPERATOR AT POSITION 0: like -2..., -x..., -(..., -sin..., --...
197       bool rerun = true;
198       while (rerun) {
199         rerun = false;
200         if (tokens[0]->getType() == 0) {
201           std::shared_ptr<OPERATOR> a = std::static_pointer_cast<OPERATOR,
202     TOKEN>(tokens[0]);
203           if (a->getOPERATOR() == '-' || a->getOPERATOR() == '+') {
204             if (tokens.size() > 1) {
205               rerun = true;
206               if (a->getOPERATOR() == '-') {
207                 if (tokens[1]->getType() == 1) {
208                   std::shared_ptr<SCALAR> b = std::static_pointer_cast<SCALAR,
209     TOKEN>(tokens[1]);
210                   tokens[1] = std::shared_ptr<SCALAR>(new SCALAR(-b->getSCALAR()));
211                   b.reset();
212                 }
213                 else if (tokens[1]->getType() == 2 || tokens[1]->getType() == 3 ||
214     tokens[1]->getType() == 10) {
215                   tokens.insert(tokens.begin() + 1, std::shared_ptr<SCALAR>(new SCALAR(-
216     1)));
217                   tokens.insert(tokens.begin() + 2, std::shared_ptr<OPERATOR>(new
218     OPERATOR('*')));
219                 }
220                 else if (tokens[1]->getType() == 0) {
221                   std::shared_ptr<OPERATOR> b = std::static_pointer_cast<OPERATOR,
222     TOKEN>(tokens[1]);
223                 if (b->getOPERATOR() == '-') {
224                     tokens.insert(tokens.begin() + 1, std::shared_ptr<OPERATOR>(new
225     OPERATOR('+')));
226                     b.reset();
227                     tokens.erase(tokens.begin() + 2);
228                   }
229                   else if (b->getOPERATOR() == '+') {
230                     tokens.insert(tokens.begin() + 1, std::shared_ptr<OPERATOR>(new
231     OPERATOR('-')));
232                     b.reset();
233                     tokens.erase(tokens.begin() + 2);
234                   }
235                   else {
236                     std::stringstream tmp;
```

```
237                   tmp << "OPERATOR '" << b->getOPERATOR() << "' IS NOT A SIGN OPERATOR
238   / TWO NOT COMPATIBLE TOKENS '" << a->getOPERATOR() << "' AND '" << b->getOPERATOR() <<
239   "' MEET!";
240                   error(tmp.str(), 0);
241                   return false;
242                 }
243               }
244               else if (tokens[1]->getType() == 5 || tokens[1]->getType() == -10) {
245                 std::stringstream tmp;
246                 tmp << "TWO NOT COMPATIBLE TOKENS '" << a->getOPERATOR() << "' AND '"
247   << (tokens[1]->getType() == 5 ? ',' : ')') << "' MEET!";
248                 error(tmp.str(), 0);
249                 return false;
250               }
251               else {
252                 error("NOT REACHABLE!");
253                 return false;
254               }
255             }
256             a.reset();
257             tokens.erase(tokens.begin() + 0);
258           }
259           else {
260             std::stringstream tmp;
261             tmp << "EXPRESSION ENDS WITH A SIGN OPERATOR: '" << a->getOPERATOR() <<
262   "'!";
263             error(tmp.str(), 0);
264             return false;
265           }
266         }
267         else {
268           std::stringstream tmp;
269           tmp << "OPERATOR '" << a->getOPERATOR() << "' IS NOT A SIGN OPERATOR!";
270           error(tmp.str(), 0);
271           return false;
272         }
273       }
274     }
275
276     for (unsigned int i = 0; i < tokens.size() - 1; i++) {
277
278       //TEST ON SIGN OPERATOR AFTER OTHER OPERATORS OR AFTER OPENING BRACKETS OR AFTER
279   ARGUMENT SEPERATOR: like ...*-2..., ...^-x..., .../-(..., ...+-sin... or like ...(-
280   2..., ...(-x..., ...(-(..., ...(-sin... or like ...,-(...
281       rerun = true;
282       while (rerun) {
283         rerun = false;
284         if ((tokens[i]->getType() == 0 || tokens[i]->getType() == 10 || tokens[i]-
285   >getType() == 5) && tokens[i + 1]->getType() == 0) {
286           std::shared_ptr<OPERATOR> b = std::static_pointer_cast<OPERATOR,
287   TOKEN>(tokens[i + 1]);
288           if (b->getOPERATOR() == '-' || b->getOPERATOR() == '+') {
289             if (tokens.size() >(i + 2)) {
290               rerun = true;
291               if (b->getOPERATOR() == '-') {
292                 if (tokens[i + 2]->getType() == 1) {
293                   std::shared_ptr<SCALAR> c = std::static_pointer_cast<SCALAR,
294   TOKEN>(tokens[i + 2]);
295                   tokens[i + 2] = std::shared_ptr<SCALAR>(new SCALAR(-c-
296   >getSCALAR()));
```

```
297                   c.reset();
298                 }
299                 else if (tokens[i + 2]->getType() == 2 || tokens[i + 2]->getType() ==
300   3 || tokens[i + 2]->getType() == 10) {
301                   tokens.insert(tokens.begin() + i + 2, std::shared_ptr<SCALAR>(new
302   SCALAR(-1)));
303                   tokens.insert(tokens.begin() + i + 3, std::shared_ptr<OPERATOR>(new
304   OPERATOR('*')));
305                 }
306                 else if (tokens[i + 2]->getType() == 0) {
307                   std::shared_ptr<OPERATOR> c = std::static_pointer_cast<OPERATOR,
308   TOKEN>(tokens[i + 2]);
309                   if (c->getOPERATOR() == '-') {
310                     tokens.insert(tokens.begin() + i + 2,
311   std::shared_ptr<OPERATOR>(new OPERATOR('+')));
312                     c.reset();
313                     tokens.erase(tokens.begin() + i + 3);
314                   }
315                   else if (c->getOPERATOR() == '+') {
316                     tokens.insert(tokens.begin() + i + 2,
317   std::shared_ptr<OPERATOR>(new OPERATOR('-')));
318                     c.reset();
319                     tokens.erase(tokens.begin() + i + 3);
320                   }
321                   else {
322                     std::stringstream tmp;
323                     tmp << "OPERATOR '" << c->getOPERATOR() << "' IS NOT A SIGN
324   OPERATOR / TWO NOT COMPATIBLE TOKENS '" << b->getOPERATOR() << "' AND '" << c-
325   >getOPERATOR() << "' MEET!";
326                     error(tmp.str(), i + 1);
327                     return false;
328                   }
329                 }
330                 else if (tokens[i + 2]->getType() == 5 || tokens[i + 2]->getType() ==
331   -10) {
332                   std::stringstream tmp;
333                   tmp << "TWO NOT COMPATIBLE TOKENS '" << b->getOPERATOR() << "' AND
334   '" << (tokens[i + 2]->getType() == 5 ? ',' : ')') << "' MEET!";
335                   error(tmp.str(), i + 1);
336                   return false;
337                 }
338                 else {
339                   error("NOT REACHABLE!");
340                   return false;
341                 }
342               }
343               b.reset();
344               tokens.erase(tokens.begin() + i + 1);
345             }
346             else {
347               std::stringstream tmp;
348               tmp << "EXPRESSION ENDS WITH A SIGN OPERATOR: '" << b->getOPERATOR() <<
349   "'!";
350               error(tmp.str(), 0);
351               return false;
352             }
353           }
354           else {
355             std::stringstream tmp;
```

```
356             tmp << "OPERATOR '" << b->getOPERATOR() << "' IS NOT A SIGN OPERATOR / TWO
357   NOT COMPATIBLE TOKENS '" << (tokens[i]->getType() == 0 ?
358   std::static_pointer_cast<OPERATOR, TOKEN>(tokens[i])->getOPERATOR() : tokens[i]-
359   >getType() == 10 ? '(' : ',') << "' AND '" << b->getOPERATOR() << "' MEET!";
360             error(tmp.str(), i);
361             return false;
362           }
363         }
364       }
365
366       //ADD MULTIPLICATION SIGNS:
367       if ((tokens[i]->getType() == 1 || tokens[i]->getType() == 2 || tokens[i]-
368   >getType() == -10) && (tokens[i + 1]->getType() == 1 || tokens[i + 1]->getType() == 2
369   || tokens[i + 1]->getType() == 3 || tokens[i + 1]->getType() == 10)) {
370         tokens.insert(tokens.begin() + i + 1, std::shared_ptr<OPERATOR>(new
371   OPERATOR('*')));
372       }
373
374       //IF YOU USE FUNCTIONS WITHOUT BRACKETS: like ...sin2..., ...cos-2...
375       rerun = true;
376       while (rerun) {
377         rerun = false;
378         if (tokens[i]->getType() == 3) {
379           if (tokens[i + 1]->getType() == 1 || tokens[i + 1]->getType() == 2) {
380             tokens.insert(tokens.begin() + i + 1, std::shared_ptr<OPENING_BRACKET>(new
381   OPENING_BRACKET));
382             tokens.insert(tokens.begin() + i + 3, std::shared_ptr<CLOSING_BRACKET>(new
383   CLOSING_BRACKET));
384           }
385           else if (tokens[i + 1]->getType() == 0) {
386             std::shared_ptr<OPERATOR> b = std::static_pointer_cast<OPERATOR,
387   TOKEN>(tokens[i + 1]);
388             if (b->getOPERATOR() == '-' || b->getOPERATOR() == '+') {
389               if (tokens.size() > (i + 2)) {
390                 rerun = true;
391                 if (b->getOPERATOR() == '-') {
392                   if (tokens[i + 2]->getType() == 1) {
393                     std::shared_ptr<SCALAR> c = std::static_pointer_cast<SCALAR,
394   TOKEN>(tokens[i + 2]);
395                     tokens[i + 2] = std::shared_ptr<SCALAR>(new SCALAR(-c-
396   >getSCALAR()));
397                     c.reset();
398                   }
399                   else if (tokens[i + 2]->getType() == 2 || tokens[i + 2]->getType()
400   == 3 || tokens[i + 2]->getType() == 10) {
401                     tokens.insert(tokens.begin() + i + 2,
402   std::shared_ptr<OPENING_BRACKET>(new OPENING_BRACKET));
403                     tokens.insert(tokens.begin() + i + 3, std::shared_ptr<SCALAR>(new
404   SCALAR(-1)));
405                     tokens.insert(tokens.begin() + i + 4,
406   std::shared_ptr<OPERATOR>(new OPERATOR('*')));
407                     tokens.insert(tokens.begin() + i + 6,
408   std::shared_ptr<CLOSING_BRACKET>(new CLOSING_BRACKET));
409                   }
410                   else if (tokens[i + 2]->getType() == 0) {
411                     std::shared_ptr<OPERATOR> c = std::static_pointer_cast<OPERATOR,
412   TOKEN>(tokens[i + 2]);
413                     if (c->getOPERATOR() == '-') {
414                       tokens.insert(tokens.begin() + i + 2,
415   std::shared_ptr<OPERATOR>(new OPERATOR('+')));
```

```
416                       c.reset();
417                       tokens.erase(tokens.begin() + i + 3);
418                     }
419                     else if (c->getOPERATOR() == '+') {
420                       tokens.insert(tokens.begin() + i + 2,
421   std::shared_ptr<OPERATOR>(new OPERATOR('-')));
422                       c.reset();
423                       tokens.erase(tokens.begin() + i + 3);
424                     }
425                     else {
426                       std::stringstream tmp;
427                       tmp << "OPERATOR '" << c->getOPERATOR() << "' IS NOT A SIGN
428   OPERATOR / TWO NOT COMPATIBLE TOKENS '" << b->getOPERATOR() << "' AND '" << c-
429   >getOPERATOR() << "' MEET!";
430                       error(tmp.str(), i + 1);
431                       return false;
432                     }
433                   }
434                   else if (tokens[i + 2]->getType() == 5 || tokens[i + 2]->getType()
435   == -10) {
436                     std::stringstream tmp;
437                     tmp << "TWO NOT COMPATIBLE TOKENS '" << b->getOPERATOR() << "' AND
438   '" << (tokens[i + 2]->getType() == 5 ? ',' : ')') << "' MEET!";
439                     error(tmp.str(), i + 1);
440                     return false;
441                   }
442                   else {
443                     error("NOT REACHABLE!");
444                     return false;
445                   }
446                 }
447                 b.reset();
448                 tokens.erase(tokens.begin() + i + 1);
449               }
450               else {
451                 std::stringstream tmp;
452                 tmp << "EXPRESSION ENDS WITH A SIGN OPERATOR: '" << b->getOPERATOR()
453   << "'!";
454                 error(tmp.str(), 0);
455                 return false;
456               }
457             }
458             else {
459               std::stringstream tmp;
460               tmp << "OPERATOR '" << b->getOPERATOR() << "' IS NOT A SIGN OPERATOR /
461   TWO NOT COMPATIBLE TOKENS '" << std::static_pointer_cast<FUNCTION, TOKEN>(tokens[i])-
462   >getFUNCTION() << "' AND '" << b->getOPERATOR() << "' MEET!";
463               error(tmp.str(), i);
464               return false;
465             }
466           }
467         }
468       }
469     }
470     //ADD BRACKETS AT THE END:
471     int bracketsToAdd = 0;
472     for (unsigned int i = 0; i < tokens.size(); i++) {
473       if (tokens[i]->getType() == 10) {
474         bracketsToAdd++;
475       }
```

```
476         if (tokens[i]->getType() == -10) {
477           bracketsToAdd--;
478         }
479       }
480       for (; bracketsToAdd > 0; bracketsToAdd--) {
481         tokens.push_back(std::shared_ptr<CLOSING_BRACKET>(new CLOSING_BRACKET));
482       }
483
484       return true;
485     }
486
487     bool isLeftAssociative(std::shared_ptr<OPERATOR> op) {
488       if (op->getOPERATOR() == '+' || op->getOPERATOR() == '-' || op->getOPERATOR() ==
489   '*' || op->getOPERATOR() == '/') {
490         return true;
491       }
492       return false;
493     }
494
495     bool isPrecedenceLessOrEqual(std::shared_ptr<OPERATOR> op1,
496   std::shared_ptr<OPERATOR> op2) {
497       int precedence1;
498       int precedence2;
499       if (op1->getOPERATOR() == '+' || op1->getOPERATOR() == '-') {
500         precedence1 = 2;
501       }
502       else if (op1->getOPERATOR() == '*' || op1->getOPERATOR() == '/') {
503         precedence1 = 3;
504       }
505       else if (op1->getOPERATOR() == '^') {
506         precedence1 = 4;
507       }
508       if (op2->getOPERATOR() == '+' || op2->getOPERATOR() == '-') {
509         precedence2 = 2;
510       }
511       else if (op2->getOPERATOR() == '*' || op2->getOPERATOR() == '/') {
512         precedence2 = 3;
513       }
514       else if (op2->getOPERATOR() == '^') {
515         precedence2 = 4;
516       }
517       if (precedence1 <= precedence2) {
518         return true;
519       }
520       return false;
521     }
522
523     bool shuntingYard() {
524       std::stack<std::shared_ptr<TOKEN>> operatorStack;
525       for (unsigned int i = 0; i < tokens.size(); i++) {
526         if (tokens[i]->getType() == 1) {
527           rpn.push_back(std::static_pointer_cast<SCALAR, TOKEN>(tokens[i]));
528           continue;
529         }
530
531         if (tokens[i]->getType() == 2) {
532           rpn.push_back(std::static_pointer_cast<VARIABLE, TOKEN>(tokens[i]));
533           continue;
534         }
535
```

```
536         if (tokens[i]->getType() == 3) {
537           operatorStack.push(std::static_pointer_cast<FUNCTION, TOKEN>(tokens[i]));
538           continue;
539         }
540 
541         if (tokens[i]->getType() == 5) {
542           while (true) {
543             if (operatorStack.empty()) {
544               error("THERE IS AN INCORRECTLY PLACED ARGUMENT SEPARATOR OR THE CLOSING
545   BRACKET ISN'T PRECEDED BY AN OPENING BRACKET!", i);
546               return false;
547             }
548             if (operatorStack.top()->getType() == 10) {
549               break;
550             }
551             else {
552               if (operatorStack.top()->getType() == 0) {
553                 rpn.push_back(std::static_pointer_cast<OPERATOR,
554   TOKEN>(operatorStack.top()));
555               }
556               else if (operatorStack.top()->getType() == 1) {
557                 rpn.push_back(std::static_pointer_cast<SCALAR,
558   TOKEN>(operatorStack.top()));
559               }
560               else if (operatorStack.top()->getType() == 2) {
561                 rpn.push_back(std::static_pointer_cast<VARIABLE,
562   TOKEN>(operatorStack.top()));
563               }
564               else if (operatorStack.top()->getType() == 3) {
565                 rpn.push_back(std::static_pointer_cast<FUNCTION,
566   TOKEN>(operatorStack.top()));
567               }
568               else {
569                 error("NOT REACHABLE!");
570                 return false;
571               }
572               operatorStack.pop();
573             }
574           }
575           continue;
576         }
577 
578         if (tokens[i]->getType() == 0) {
579           while (true) {
580             if (operatorStack.empty()) {
581               break;
582             }
583             if (operatorStack.top()->getType() == 0) {
584               if (isLeftAssociative(std::static_pointer_cast<OPERATOR,
585   TOKEN>(tokens[i])) && isPrecedenceLessOrEqual(std::static_pointer_cast<OPERATOR,
586   TOKEN>(tokens[i]), std::static_pointer_cast<OPERATOR, TOKEN>(operatorStack.top()))) {
587                 rpn.push_back(std::static_pointer_cast<OPERATOR,
588   TOKEN>(operatorStack.top()));
589                 operatorStack.pop();
590               }
591               else {
592                 break;
593               }
594             }
595             else {
```

```
596               break;
597             }
598           }
599           operatorStack.push(std::static_pointer_cast<OPERATOR, TOKEN>(tokens[i]));
600           continue;
601         }
602
603         if (tokens[i]->getType() == 10) {
604           operatorStack.push(std::static_pointer_cast<OPENING_BRACKET,
605   TOKEN>(tokens[i]));
606           continue;
607         }
608
609         if (tokens[i]->getType() == -10) {
610           while (true) {
611             if (operatorStack.empty()) {
612               error("THE CLOSING BRACKET ISN'T PRECEDED BY AN OPENING BRACKET!", i);
613               return false;
614             }
615             if (operatorStack.top()->getType() == 10) {
616               operatorStack.pop();
617               break;
618             }
619             else {
620               rpn.push_back(operatorStack.top());
621               operatorStack.pop();
622             }
623           }
624           if (!operatorStack.empty()) {
625             if (operatorStack.top()->getType() == 3) {
626               rpn.push_back(std::static_pointer_cast<FUNCTION,
627   TOKEN>(operatorStack.top()));
628               operatorStack.pop();
629             }
630           }
631           continue;
632         }
633         error("NOT REACHABLE!");
634         return false;
635       }
636       while (true) {
637         if (operatorStack.empty()) {
638           break;
639         }
640         if (operatorStack.top()->getType() == 10) {
641           error("THERE ARE MORE OPENING THAN CLOSING BRACKETS!");
642           return false;
643         }
644         else {
645           rpn.push_back(operatorStack.top());
646           operatorStack.pop();
647         }
648       }
649       return true;
650     }
651
652   public:
653     double calculate(std::vector<double> x) {
654       double totalresult = NAN;
655       std::stack<std::shared_ptr<TOKEN>> evaluationStack;
```

```
656       std::vector<std::string> varnames;
657       for (unsigned int i = 0; i < rpn.size(); i++) {
658         if (rpn[i]->getType() == 1) {
659           evaluationStack.push(std::static_pointer_cast<SCALAR, TOKEN>(rpn[i]));
660           continue;
661         }
662
663         if (rpn[i]->getType() == 2) {
664           unsigned int pos;
665           std::string varname = std::static_pointer_cast<VARIABLE, TOKEN>(rpn[i])-
666   >getVARIABLE();
667           bool registered = false;
668           for (unsigned int i = 0; i < varnames.size(); i++) {
669             if (varname == varnames[i]) {
670               registered = true;
671               pos = i;
672               break;
673             }
674           }
675           if (!registered) {
676             varnames.push_back(varname);
677             pos = varnames.size() - 1;
678           }
679           if (varnames.size() > x.size()) {
680             std::stringstream tmp;
681             tmp << "THERE ARE MORE VARIABLES THAN INPUT VALUES FOR THIS FUNCTION,
682   VARAIBLES ARE: ";
683             for (unsigned int i = 0; i < varnames.size(); i++) {
684               tmp << "'" << varnames[i] << "', ";
685             }
686             tmp << "INPUT VALUES ARE: ";
687             for (unsigned int i = 0; i < x.size(); i++) {
688               tmp << "'" << x[i] << "', ";
689             }
690             tmp << "!";
691             error(tmp.str());
692             //return false;
693           }
694           evaluationStack.push(std::shared_ptr<SCALAR>(new SCALAR(x[pos])));
695           continue;
696         }
697
698         if (rpn[i]->getType() == 0) {
699           double result = 0;
700
701           char op = std::static_pointer_cast<OPERATOR, TOKEN>(rpn[i])->getOPERATOR();
702
703           //TEST ON TOKEN TO OPERATE:
704           if (evaluationStack.empty()) {
705             std::stringstream tmp;
706             tmp << "OPERATOR '" << op << "' HAS NO TOKEN TO OPERATE!";
707             error(tmp.str(), i);
708             //return false;
709           }
710           //TEST ON SCALAR TO OPERATE:
711           else if (evaluationStack.top()->getType() != 1) {
712             std::stringstream tmp;
713             tmp << "OPERATOR '" << op << "' HAS NO SCALAR TO OPERATE!";
714             error(tmp.str(), i);
715             //return false;
```

```
716           }
717
718           std::shared_ptr<SCALAR> right = std::static_pointer_cast<SCALAR,
719   TOKEN>(evaluationStack.top());
720           evaluationStack.pop();
721
722           //TEST ON TOKENS TO OPERATE:
723           if (evaluationStack.empty()) {
724             std::stringstream tmp;
725             tmp << "OPERATOR '" << op << "' NEEDS 2 TOKENS TO OPERATE!";
726             error(tmp.str(), i);
727             //return false;
728           }
729           //TEST ON SCALARS TO OPERATE:
730           else if (evaluationStack.top()->getType() != 1) {
731             std::stringstream tmp;
732             tmp << "OPERATOR '" << op << "' NEEDS 2 SCALARS TO OPERATE!";
733             error(tmp.str(), i);
734             //return false;
735           }
736
737           std::shared_ptr<SCALAR> left = std::static_pointer_cast<SCALAR,
738   TOKEN>(evaluationStack.top());
739           evaluationStack.pop();
740
741           if (op == '+') {
742             result = left->getSCALAR() + right->getSCALAR();
743           }
744           else if (op == '-') {
745             result = left->getSCALAR() - right->getSCALAR();
746           }
747           else if (op == '*') {
748             result = left->getSCALAR() * right->getSCALAR();
749           }
750           else if (op == '/') {
751             result = left->getSCALAR() / right->getSCALAR();
752           }
753           else if (op == '^') {
754             result = pow(left->getSCALAR(), right->getSCALAR());
755           }
756           else {
757             error("NOT REACHABLE!");
758             //return false;
759           }
760           evaluationStack.push(std::shared_ptr<SCALAR>(new SCALAR(result)));
761           continue;
762         }
763
764         if (rpn[i]->getType() == 3) {
765           double result = 0;
766
767           std::string f = std::static_pointer_cast<FUNCTION, TOKEN>(rpn[i])-
768   >getFUNCTION();
769
770           //TEST ON INPUT VALUE:
771           if (evaluationStack.empty()) {
772             std::stringstream tmp;
773             tmp << "FUNCTION '" << f << "' HAS NO INPUT VALUE!";
774             error(tmp.str(), i);
775             //return false;
```

```
776           }
777           //TEST ON SCALAR INPUT VALUE:
778           else if (evaluationStack.top()->getType() != 1) {
779             std::stringstream tmp;
780             tmp << "FUNCTION '" << f << "' HAS NO SCALAR INPUT VALUE!";
781             error(tmp.str(), i);
782             //return false;
783           }
784
785           std::shared_ptr<SCALAR> right = std::static_pointer_cast<SCALAR,
786   TOKEN>(evaluationStack.top());
787           evaluationStack.pop();
788
789           if (f == "exp") {
790             result = std::exp(right->getSCALAR());
791           }
792           else if (f == "ln") {
793             result = std::log(right->getSCALAR());
794           }
795           else if (f == "log10") {
796             result = std::log10(right->getSCALAR());
797           }
798           else if (f == "log2") {
799             result = std::log2(right->getSCALAR());
800           }
801           else if (f == "sin") {
802             result = std::sin(right->getSCALAR());
803           }
804           else if (f == "cos") {
805             result = std::cos(right->getSCALAR());
806           }
807           else if (f == "tan") {
808             result = std::tan(right->getSCALAR());
809           }
810           else if (f == "arcsin") {
811             result = std::asin(right->getSCALAR());
812           }
813           else if (f == "arccos") {
814             result = std::acos(right->getSCALAR());
815           }
816           else if (f == "arctan") {
817             result = std::atan(right->getSCALAR());
818           }
819           else if (f == "sinh") {
820             result = std::sinh(right->getSCALAR());
821           }
822           else if (f == "cosh") {
823             result = std::cosh(right->getSCALAR());
824           }
825           else if (f == "tanh") {
826             result = std::tanh(right->getSCALAR());
827           }
828           else if (f == "arsinh") {
829             result = std::asinh(right->getSCALAR());
830           }
831           else if (f == "arcosh") {
832             result = std::acosh(right->getSCALAR());
833           }
834           else if (f == "artanh") {
835             result = std::atanh(right->getSCALAR());
```

```
836           }
837           else if (f == "abs") {
838             result = std::abs(right->getSCALAR());
839           }
840           else if (f == "min" || f == "max") {
841
842             //TEST ON ENOUGH INPUT VALUES:
843             if (evaluationStack.empty()) {
844               std::stringstream tmp;
845               tmp << "FUNCTION '" << f << "' HAS NOT ENOUGH INPUT VALUES (NEEDS 2)!";
846               error(tmp.str(), i);
847               //return false;
848             }
849             //TEST ON ENOUGH SCALAR INPUT VALUES:
850             else if (evaluationStack.top()->getType() != 1) {
851               std::stringstream tmp;
852               tmp << "FUNCTION '" << f << "' HAS NOT ENOUGH SCALAR INPUT VALUES (NEEDS
853   2)!";
854               error(tmp.str(), i);
855               //return false;
856             }
857
858             std::shared_ptr<SCALAR> left = std::static_pointer_cast<SCALAR,
859   TOKEN>(evaluationStack.top());
860             evaluationStack.pop();
861
862             if (f == "min") {
863               result = std::min<double>(left->getSCALAR(), right->getSCALAR());
864             }
865             else if (f == "max") {
866               result = std::max<double>(left->getSCALAR(), right->getSCALAR());
867             }
868             else {
869               error("NOT REACHABLE!");
870               //return false;
871             }
872           }
873           else {
874             std::stringstream tmp;
875             tmp << "FUNCTION '" << f << "' IS NOT SUPPORTED!";
876             error(tmp.str(), i);
877             //return false;
878           }
879
880           evaluationStack.push(std::shared_ptr<SCALAR>(new SCALAR(result)));
881           continue;
882         }
883
884         error("NOT REACHABLE!");
885         //return false;
886       }
887
888       //TEST TOTAL RESULT ON SCALAR:
889       if (evaluationStack.empty()) {
890         error("NO RESULT!");
891         //return false;
892       }
893       else if (evaluationStack.top()->getType() != 1) {
894         error("RESULT IS NOT SCALAR!");
895         //return false;
```

```
896       }
897
898       totalresult = std::static_pointer_cast<SCALAR, TOKEN>(evaluationStack.top())-
899   >getSCALAR();
900       if (totalresult != totalresult) {
901         std::stringstream tmp;
902         tmp << "RESULT IS: '" << totalresult << "', FOR FUNCTION: '";
903         for (unsigned int i = 0; i < tokens.size(); i++) {
904           tmp << *(tokens[i]) << " ";
905         }
906         tmp << "'!";
907         error(tmp.str());
908       }
909       return totalresult;
910     }
911
912     ~INTERPRETER() {
913       for (unsigned int i = 0; i < tokens.size(); i++) {
914         tokens[i].reset();
915       }
916       for (unsigned int i = 0; i < rpn.size(); i++) {
917         //delete rpn[i]; da in rpn nur zeiger von tokens sind;
918       }
919     }
920   };
921
```

7.7 TOKEN.h

```
1   #pragma once
2
3   #include <string>
4   #include <iostream>
5
6   class TOKEN {
7     int type;
8   public:
9     TOKEN(int type) : type(type) {}
10    int getType() { return type; }
11
12    virtual std::ostream & output(std::ostream & os) = 0;
13    friend std::ostream & operator<<(std::ostream&, const TOKEN&);
14
15    virtual ~TOKEN() {}
16  };
17
18  std::ostream & operator<<(std::ostream & os, TOKEN & obj) {
19    return obj.output(os);
20  }
21
22  /*
23  TOKEN:              |  TYPE:
24  --------------------|-------
25  OPERATOR            |  0
26  SCALAR              |  1
27  VARIABLE            |  2
28  FUNCTION            |  3
29  ARGUMENT_SEPARATOR  |  5
30  OPENING_BRACKET     |  10
31  CLOSING_BRACKET     |  -10
32  */
33
```

```
34  class OPERATOR : public TOKEN {
35    char o;
36  public:
37    OPERATOR(char o) : o(o), TOKEN(0) {}
38    char getOPERATOR() { return o; }
39
40    std::ostream & output(std::ostream & os) { return os << o; }
41
42    virtual ~OPERATOR() {}
43  };
44
45  class SCALAR : public TOKEN {
46    double s;
47  public:
48    SCALAR(double s) : s(s), TOKEN(1) {}
49    double getSCALAR() { return s; }
50
51    std::ostream & output(std::ostream & os) { return os << s; }
52
53    virtual ~SCALAR() {}
54  };
55
56  class VARIABLE : public TOKEN {
57    std::string v;
58  public:
59    VARIABLE(std::string v) : v(v), TOKEN(2) {}
60    std::string getVARIABLE() { return v; }
61
62    std::ostream & output(std::ostream & os) { return os << v; }
63
64    virtual ~VARIABLE() {}
65  };
66
67  class FUNCTION : public TOKEN {
68    std::string f;
69  public:
70    FUNCTION(std::string f) : f(f), TOKEN(3) {}
71    std::string getFUNCTION() { return f; }
72
73    std::ostream & output(std::ostream & os) { return os << f; }
74
75    virtual ~FUNCTION() {}
76  };
77
78  class ARGUMENT_SEPARATOR : public TOKEN {
79  public:
80    ARGUMENT_SEPARATOR() : TOKEN(5) {}
81
82    std::ostream & output(std::ostream & os) { return os << ","; }
83
84    virtual ~ARGUMENT_SEPARATOR() {}
85  };
86
87  class OPENING_BRACKET : public TOKEN {
88  public:
89    OPENING_BRACKET() : TOKEN(10) {}
90
91    std::ostream & output(std::ostream & os) { return os << "("; }
92
93    virtual ~OPENING_BRACKET() {}
```

```
94  };
95
96  class CLOSING_BRACKET : public TOKEN {
97  public:
98    CLOSING_BRACKET() : TOKEN(-10) {}
99
100   std::ostream & output(std::ostream & os) { return os << ")"; }
101
102   virtual ~CLOSING_BRACKET() {}
103 };
```

7.8 main.cpp

```
1   #include <cstdio>
2   #include <vector>
3   #include <cmath>
4   #include <iostream>
5   #include <fstream>
6   #include <exception>
7
8
9
10  #include <omp.h>
11
12  #include "DISCRETIZATIONS.h"
13
14
15
16  void prepare_input(std::vector<std::string>& input, char * filename) {
17    try {
18      std::ifstream ifs(filename, std::ifstream::in);
19      while (ifs.good()) {
20        std::string tmp;
21        char c = ifs.get();
22        while (c != EOF && c != '\n') {
23          tmp.push_back(c);
24          c = ifs.get();
25        }
26        input.push_back(tmp);
27      }
28    }
29    catch (std::exception e) {
30      std::cout << e.what() << std::endl;
31    }
32
33    //KOMMERNTARE + LEERE ZEILEN ABFANGEN:
34    for (unsigned int i = 0; i < input.size();) {
35      for (unsigned int j = 0; j < input[i].size(); j++) {
36        if (input[i][j] == '/') {
37          if (++j < input[i].size()) {
38            if (input[i][j] == '/') {
39              input[i].erase(input[i].begin() + (j - 1), input[i].end());
40              continue;
41            }
42          }
43        }
44        if (input[i][j] == ' '|| input[i][j] == '\r') {
45          input[i].erase(input[i].begin() + j);
46          j--;
47        }
48      }
49      if (input[i].empty()) {
```

```
50          input.erase(input.begin() + i);
51          continue;
52        }
53        i++;
54      }
55    }
56
57    int main(int argc, char *argv[]) {
58      std::vector<std::string> input;
59      double timestamp = omp_get_wtime();
60      if (argc == 2) {
61        prepare_input(input, argv[1]);
62        //DISCRETIZATION_CORE:
63        try {
64          if (input.empty()) {
65            throw std::string("INPUTFILE IS EMPTY!");
66          }
67          if (!std::isdigit(static_cast<unsigned char>(input[0][0]))) {
68            throw std::string("SELECTION DIGIT FOR SOLVER IS NON DIGIT!");
69          }
70          int D = input[0][0] - '0';
71          if (input.size() <= 1) {
72            throw std::string("DATA TYPE NOT ENTERED!");
73          }
74          std::string type = input[1];
75          if (type == "float") {
76            if (D == 2) {
77              NEWTON_CLASSIC<float> x(input);
78            }
79            else if (D == 0) {
80              LAPLACE<float> x(input);
81            }
82            else if (D == 3) {
83              NEWTON_DAMPED_TRIVIAL<float> x(input);
84            }
85            else if (D == 4) {
86              NEWTON_DAMPED_BACKTRACKING<float> x(input);
87            }
88            else {
89              std::stringstream tmp;
90              tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
91              throw tmp.str();
92            }
93          }
94          else if (type == "double") {
95            if (D == 2) {
96              NEWTON_CLASSIC<double> x(input);
97            }
98            else if (D == 0) {
99              LAPLACE<double> x(input);
100           }
101           else if (D == 3) {
102             NEWTON_DAMPED_TRIVIAL<double> x(input);
103           }
104           else if (D == 4) {
105             NEWTON_DAMPED_BACKTRACKING<double> x(input);
106           }
107           else {
108             std::stringstream tmp;
109             tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
```

```
110             throw tmp.str();
111           }
112         }
113         else if (type == "longdouble") {
114           if (D == 2) {
115             NEWTON_CLASSIC<long double> x(input);
116           }
117           else if (D == 0) {
118             LAPLACE<long double> x(input);
119           }
120           else if (D == 3) {
121             NEWTON_DAMPED_TRIVIAL<long double> x(input);
122           }
123           else if (D == 4) {
124             NEWTON_DAMPED_BACKTRACKING<long double> x(input);
125           }
126           else {
127             std::stringstream tmp;
128             tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
129             throw tmp.str();
130           }
131         }
132         else {
133           std::stringstream tmp;
134           tmp << "DATA TYPE '" << type << "' IS NOT SUPPORTED!" << std::endl;
135           throw tmp.str();
136         }
137       }
138       catch (std::string e) {
139         std::cout << e << std::endl;
140       }
141     }
142     else if (argc == 3) {
143       prepare_input(input, argv[1]);
144       //DISCRETIZATION_CORE:
145       try {
146         if (input.empty()) {
147           throw std::string("INPUTFILE IS EMPTY!");
148         }
149         if (!std::isdigit(static_cast<unsigned char>(input[0][0]))) {
150           throw std::string("SELECTION DIGIT FOR SOLVER IS NON DIGIT!");
151         }
152         int D = input[0][0] - '0';
153 
154         if (input.size() <= 1) {
155           throw std::string("DATA TYPE NOT ENTERED!");
156         }
157         std::string type = input[1];
158         if (type == "float") {
159           if (D == 2) {
160             NEWTON_CLASSIC<float> x(input, argv[2]);
161           }
162           else if (D == 3) {
163             NEWTON_DAMPED_TRIVIAL<float> x(input, argv[2]);
164           }
165           else if (D == 4) {
166             NEWTON_DAMPED_BACKTRACKING<float> x(input, argv[2]);
167           }
168           else {
169             std::stringstream tmp;
```

```
170             tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
171             throw tmp.str();
172           }
173         }
174         else if (type == "double") {
175           if (D == 2) {
176             NEWTON_CLASSIC<double> x(input, argv[2]);
177           }
178           else if (D == 3) {
179             NEWTON_DAMPED_TRIVIAL<double> x(input, argv[2]);
180           }
181           else if (D == 4) {
182             NEWTON_DAMPED_BACKTRACKING<double> x(input, argv[2]);
183           }
184           else {
185             std::stringstream tmp;
186             tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
187             throw tmp.str();
188           }
189         }
190         else if (type == "longdouble") {
191           if (D == 2) {
192             NEWTON_CLASSIC<long double> x(input, argv[2]);
193           }
194           else if (D == 3) {
195             NEWTON_DAMPED_TRIVIAL<long double> x(input, argv[2]);
196           }
197           else if (D == 4) {
198             NEWTON_DAMPED_BACKTRACKING<long double> x(input, argv[2]);
199           }
200           else {
201             std::stringstream tmp;
202             tmp << "DISCRETIZATION CORE '" << D << "' IS NOT SUPPORTED!" << std::endl;
203             throw tmp.str();
204           }
205         }
206         else {
207           std::stringstream tmp;
208           tmp << "DATA TYPE '" << type << "' IS NOT SUPPORTED!" << std::endl;
209           throw tmp.str();
210         }
211       }
212       catch (std::string e) {
213         std::cout << e << std::endl;
214       }
215     }
216     else {
217       std::cout << "MISSING INPUT FILE, TRY: ./main <config-FILE> OR ./main <config-
218   FILE> <output-FILE>" << std::endl;
219     }
220     std::cout << "TOTAL TIME: \t" << omp_get_wtime() - timestamp << std::endl;
221     return 0;
222   }
223
```