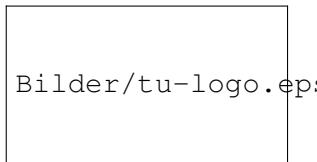


=5

=1

=



Fakultät 5
Institut für Mechanik

Simulation der diffusionsbedingten Medikamentenfreisetzung aus Nanopharmazeutika auf Polymerbasis unter Berücksichtigung der Matrixdegradation

vorgelegt von:
Jan Pieter 327 571

Gutachter: Prof. Dr. rer. nat. W. H. Müller
Betreuer: Dr.-Ing. Aleksandr Morozov

Technische Universität Berlin, Fakultät 5 – Institut für Mechanik,
Fachgebiet für Kontinuumsmechanik und Materialtheorie
Marburg, 26. Februar 2025

Eigenständigkeitserklärung

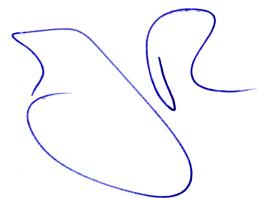
Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen die den benutzten Quellen und Hilfsmitteln unverändert oder sinngemäß entnommen sind, habe ich als solche kenntlich gemacht.

Sofern generative KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die jeweiligen Einsatzzwecke (z.B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 15. Februar 2023. https://www.static.tu.berlin/fileadmin/www/10002457/K3-AMB1/Amtsblatt_2023/Amtliches_Mitteilungsblatt_Nr._16_vom_30.05.2023.pdf habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Marburg, den 26. Februar 2025



Jan Pieter

Inhaltsverzeichnis

1 Einleitung	1
2 Experiment	5
2.1 Versuchsaufbau und Durchführung	5
2.2 Freisetzungskurve	6
3 Polymerträgermatrix	9
3.1 Matrixdegradations-/Wirkstofffreisetzungsmodell	9
3.2 Wirkstoffverteilung zum Startzeitpunkt	10
3.2.1 Oberfläche	10
3.2.2 Wirkstoffverteilung innerhalb der Matrix	14
4 Physikalische Modellbildung	17
4.1 Homogene Diffusionsgleichung	17
4.2 Degradationsmodell	18
4.3 Randbedingungen	19
4.4 Wirkstoffkonzentration in der Umgebungslösung	21
4.5 Freisetzungskurve	22
5 Numerische Betrachtungen	25
5.1 Einheitenlose Darstellung	25
5.1.1 Homogene Diffusionsgleichung	26
5.1.2 Degradationsmodell	26
5.1.3 Randbedingungen	27
5.1.4 Stoffübergangskoeffizient h''	27
5.1.5 Diffusionsgleichung der Umgebungslösung	27
5.1.6 Freisetzungskurve	27
5.2 Zeitliche Diskretisierung	28
5.2.1 Homogene Diffusionsgleichung	30
5.2.2 Degradationsmodell	32
5.2.3 Randbedingungen	33
5.2.4 Stoffübergangskoeffizient \tilde{h}''	33
5.2.5 Diffusionsgleichung der Umgebungslösung	33
5.2.6 Freisetzungskurve	35

5.3	Räumliche Diskretisierung	35
5.3.1	Polymerträgermatrix	35
5.4	Konvergenzanalyse	36
5.4.1	<i>Method of Manufactured Solutions</i>	36
5.4.2	Diskretisierungsfehler	39
5.4.3	Konvergenzrate	39
5.5	Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt	40
5.6	Softwarelösung	41
5.7	Inverse Analyse	41
6	Ergebnisse	43
6.1	Konvergenzanalyse	43
6.1.1	Explizites EULER-Verfahren	43
6.1.2	CRANK-NICOLSON-Verfahren	45
6.2	Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt	47
6.3	Inverse Analyse	48
7	Diskussion	51
7.1	Konvergenzanalyse	51
7.1.1	Explizites EULER-Verfahren	51
7.1.2	CRANK-NICOLSON-Verfahren	52
7.2	Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt	54
7.3	Wahl der Diskretisierungparameter	54
7.4	Inverse Analyse	55
7.5	Degradationsmodell	57
7.6	Randbedingung	57
8	Zusammenfassung	59
9	Ausblick	63
	Abbildungsverzeichnis	XI
	Tabellenverzeichnis	XV
	Algorithmen und Quelltexte	XVII
	Literatur	XIX
A	GNU-Octave-Programm	i
B	Python-Quelltext für FEniCS Legacy	v
	B.1 paramters_units.py	viii

B.2 paramters.py	x
B.3 main.py	xiv
B.4 menu.py	xvi
B.5 model.py	xxvii
B.6 mesh_domains_boundaries.py	xxxv
B.7 boundary_conditions.py	xxxvii
B.8 u_0.py	xxxviii
B.9 source_term.py	xl
B.10 surrounding_solution.py	xli
B.11 degradation.py	xliv
B.12 solver_Euler.py	xlv
B.13 solver_CN_low.py	xlvii
B.14 solver_CN_high.py	xlix
B.15 utility.py	li
B.16 optimization.py	lviii
B.17 convergence_analysis.py	lxii
B.18 testing.py	lxvii
B.19 save.py	lxix
B.20 plot_settings_inits.py	lxxx
B.21 plot_funcs.py	lxxxvi

1 Einleitung

Der Ausbruch des Coronavirus SARS-CoV-2 hat der Gesellschaft nicht zuletzt die Notwendigkeit einer beschleunigten Medikamentenentwicklung vor Augen geführt. Aktuell vergehen im Durchschnitt mehr als 13 Jahre bis zur erfolgreichen Zulassung eines neuen Medikamentes, siehe Abbildung 1.1, [Paul *et al.* (2010)] und [Tamimi *et al.* (2009)].

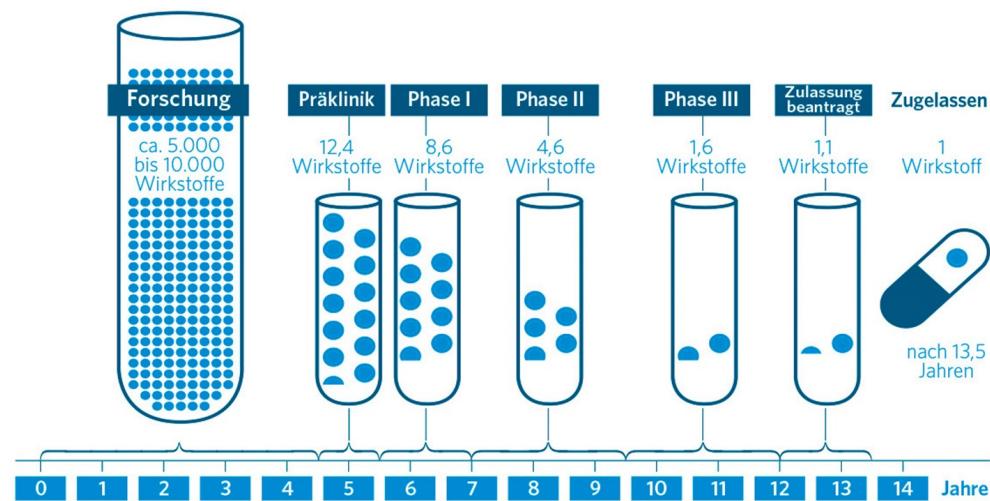


Abb. 1.1: Entwicklungsschritte eines neuen Medikaments und deren Dauer, siehe [Tamimi *et al.* (2009)] und [Paul *et al.* (2010)]: Identifikation einer nicht oder zumindest nicht zufriedenstellend behandelbaren Krankheit, Ermittlung der zugrunde liegenden Wirkmechanismen, Bestimmung des Angriffspunkts und Auswahl der Wirkstoffkandidaten, die erfolgreich am Angriffspunkt binden können im Rahmen der Forschung. Untersuchung der Wirkstoffkandidaten im Reagenzglas (in vitro) und an Tieren (in vivo) auf Wirksamkeit, Verträglichkeit und Wechselwirkungen während der Präklinik. Erprobung des Medikaments an 60–80 gesunden Probanden (Klinische Phase I), an 100–500 erkrankten (Klinische Phase II) und an tausenden erkrankten (Klinische Phase III); Grafische Aufbereitung [Herzog (2022)]

Zunächst muss ein medizinischer Bedarf, also z.B. eine nicht oder nur unzureichend behandelbare Krankheit, identifiziert und gleichzeitig festgestellt werden, ob neue Erkenntnisse über die zugrunde liegenden Abläufe der Krankheit bekannt geworden sind. Ist dies der Fall, wird versucht einen Angriffspunkt

während dieses Ablaufs zu ermitteln, um dort mit einem Wirkstoff anzusetzen. Anschließend werden Millionen infrage kommende Moleküle daraufhin untersucht, ob sie an diesem Angriffspunkt binden können. Dieser durchschnittlich viereinhalb Jahre andauernde Ablauf wird in Abbildung 1.1 unter dem Zeitabschnitt Forschung zusammengefasst. Die ermittelten Wirkstoffkandidaten werden im Rahmen der darauffolgenden Präklinik im Reagenzglas u.a. an Zellkulturen (in vitro) auf Wirksamkeit, Verträglichkeit und Wechselwirkungen hin untersucht. Um diese Aspekte ebenfalls am Organismus (in vivo) zu untersuchen, sind zu diesem Zeitpunkt auch Tierversuche nötig. Zuletzt werden die klinischen Phasen durchlaufen, wobei während der ersten Phase das Medikament an 60–80 gesunden Probanden, während der zweiten Phase an 100–500 erkrankten und zuletzt in der dritten Phase an tausenden erkrankten untersucht wird. Über den gesamten Prozess hinweg entstehen Kosten von 0.161–4.54 Milliarden US-Dollar pro Medikament, siehe [Schlander *et al.* (2021)].

Ein wichtiges Untersuchungsfeld während der präklinischen und klinischen Phase eines Medikamentes ist die Pharmakokinetik. Sie beschreibt die Wechselwirkung zwischen Arzneimittel und Organismus. Hierbei werden in die Phasen der Freisetzung des Wirkstoffs aus seiner Darreichungsform, der Aufnahme, der Verteilung, der Verstoffwechselung sowie der Ausscheidung unterschieden, siehe [Ruiz-Garcia *et al.* (2008)]. Abbildung 1.2 zeigt eine Positronen-Emissions-Tomographie (PET)-Aufnahme, worauf sich, nach Injektion des Wirkstoffes (blau) in den Arm, dessen Verteilung über die Blutbahn und die darauffolgende Ausscheidung über Niere und Blase beobachten lässt. Bevor solche

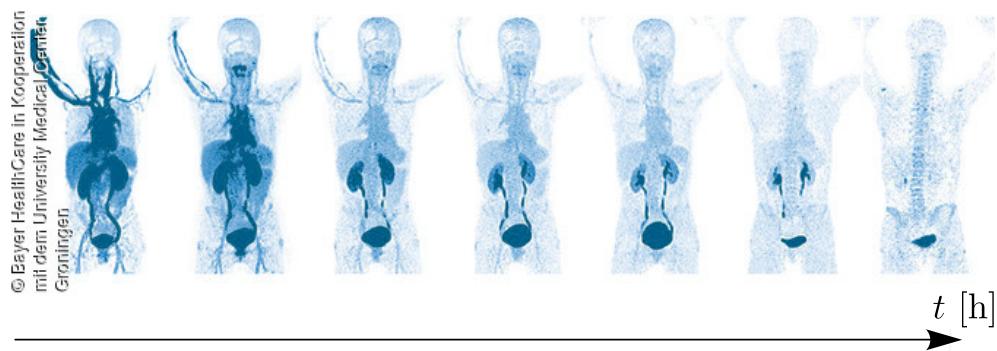


Abb. 1.2: Positronen-Emissions-Tomographie-Ganzkörperaufnahme: Verteilung des Wirkstoffes (blau) über die Blutbahn nach Injektion in den Arm sowie darauffolgende Ausscheidung über Niere und Blase für fortschreitende Zeitpunkte, siehe [Bayer HealthCare *et al.* (o. D.)].

Versuche am Menschen durchgeführt werden können, gehören Versuche zur Freisetzung des Wirkstoffes im Reagenzglas zum Bestandteil der präklinischen

Versuchsphase. Solch ein in vitro-Experiment liegt auch dieser Arbeit zugrunde. Dabei wird der Wirkstoff aus seiner Darreichungsform, einer biologisch abbaubaren Polymerträgermatrix, unter ständigem Rühren in eine körperähnliche Flüssigkeit freigesetzt. Ziel dieser Arbeit ist es, ein Modell (*in silico*) zu entwickeln, dass die gemessene Freisetzungskurve vorhersagen kann. Die Durchführung des Experiments nimmt dabei mehrere Monate in Anspruch, wohingegen die entsprechende Simulation in wenigen Stunden durchgeführt werden kann. Da die durch die Simulation gewonnenen Materialparameter geometrieunabhängig sind, können auch andere Matrixgeometrien und Wirkstoffverteilungen daraufhin simuliert werden ohne dazu erneut mehrmonatige Experimente durchführen zu müssen. Darüber hinaus besteht die Möglichkeit, technisch noch nicht umsetzbare Verteilungen und Geometrien zu simulieren und so den Nutzen von deren Realisierung aufzuzeigen. So können Zeit und Geld eingespart werden, was die Medikamentenentwicklung beschleunigt, die Chance der erfolgreichen Markteinführung erhöht und so schlussendlich dem erkrankten Patienten zugutekommt.

2 Experiment

An dieser Stelle wird das von [Macha *et al.* (2019)] durchgeführte Experiment zur Bestimmung der Freisetzungskurve des Wirkstoffs Gentamicin (GM), der in eine Medikamentenmatrix eingebettet wurde, kurz vorgestellt.

2.1 Versuchsaufbau und Durchführung

GM diffundierte während des Experiments aus der Medikamentenmatrix Ω_m in die matrixumgebende Lösung Ω_s hinein. In der Umgebungslösung Ω_s wurde die Konzentration von GM in wöchentlichen Abständen mit Hilfe eines UV/Vis-Spektralphotometers gemessen.

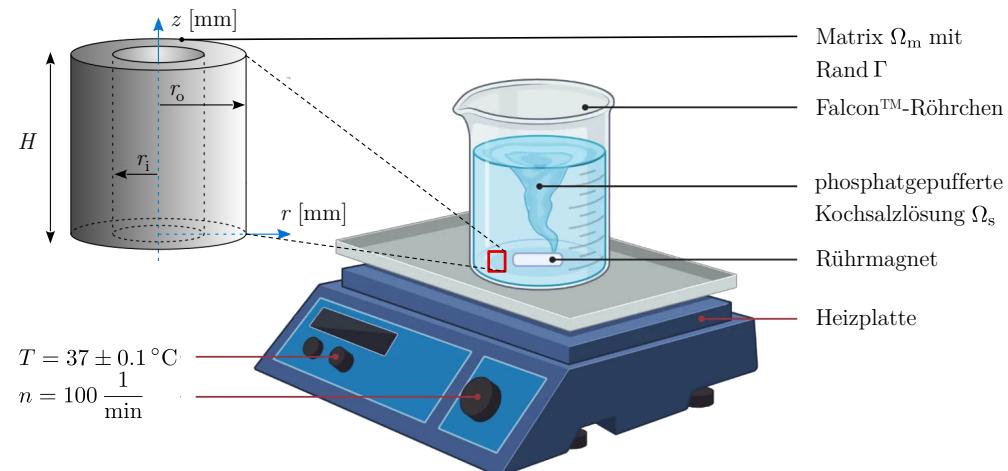


Abb. 2.1: Versuchsaufbau: Mit 10 % GM-Massenanteil versehene, biokompatible, biologisch abbaubare, nanoporöse, hohlzylinderförmige ($r_o = 3.4 \text{ mm}$, $r_i = 3.2 \text{ mm}$, $H = 20 \text{ mm}$) PLA-Matrix Ω_m in Umgebungslösung Ω_s ($V_s = 1.5 \times 10^4 \text{ mm}^3$, $pH = 7.4$, $T = 37 \pm 0.1^\circ\text{C}$) auf Magnetrührer bei $n = 100 \frac{1}{\text{min}}$ Umdrehungen. [Macha *et al.* (2019)], [MicrobeNotes (o.D.)]

Die Medikamentenmatrix Ω_m ist eine aus Polylactid (PLA) gefertigte, biokompatible, biologisch abbaubare, nanoporöse Matrix, welche in Streifen geschnitten

und gerollt wurde. Daraus resultiert ein Hohlzylinder mit einem AuSSenradius von $r_o = 3.4 \text{ mm}$, einem Innenradius von $r_i = 3.2 \text{ mm}$ und einer Höhe von $H = 20 \text{ mm}$, siehe Abbildung 2.1. Diese gerollten Streifen, die zuvor mit einem GM-Massenanteil von 10 % versehen wurden, wurden zusammen mit der Umgebungslösung Ω_s in ein konisches FalconTM-Röhrchen gegeben. Als Umgebungslösung Ω_s , welche physikalische Eigenschaften ähnlich denen von Körperflüssigkeiten inne hat, wurde phosphatgepufferte Kochsalzlösung verwendet. Die Lösung hatte ein Volumen von $V_s = 1.5 \times 10^4 \text{ mm}^3$, einen pH -Wert von $pH = 7.4$, wurde über das Experiment hinweg auf Körpertemperatur ($T = 37 \pm 0.1^\circ\text{C}$) gehalten und mit Hilfe eines Magnetrührers bei $n = 100 \frac{1}{\text{min}}$ Umdrehungen gerührt.

2.2 Freisetzungskurve

Durch dieses Experiments wurde die Freisetzung $F(t) [-]$ von GM bestimmt, welche nach

$$F(t) = \frac{c_s(t)}{c_{\max}}, \quad 0 \leq F(t) \leq 1, \quad (2.1)$$

definiert ist. $c_s(t) \left[\frac{\text{g}}{\text{mm}^3} \right]$ beschreibt die Konzentration von GM, die in der Umgebungslösung zum Zeitpunkt $t [\text{s}]$ gemessen wurde, und $c_{\max} \left[\frac{\text{g}}{\text{mm}^3} \right]$ die höchstmögliche Konzentration des Wirkstoffs in ebendieser. Dabei berechnet sich die Konzentration von GM in der Umgebungslösung $c_s(t)$ wie folgt

$$c_s(t) = \frac{m(t)}{V_s}, \quad (2.2)$$

wobei $m(t) [\text{g}]$ die Masse von GM in der Umgebungslösung zum Zeitpunkt t angibt. Die höchstmögliche Konzentration c_{\max} von GM in der Umgebungslösung ist durch

$$c_{\max} = \frac{m_d}{V_s} \quad (2.3)$$

gegeben. $m_d [\text{g}]$ beschreibt die Masse von GM, die zum Zeitpunkt $t = 0 \text{ s}$ in der Matrix eingebettet ist und beläuft sich auf $m_d = 7.5 \times 10^{-3} \text{ g}$. Daraus resultiert für Gleichung 2.3 $c_{\max} = 5 \times 10^{-7} \frac{\text{g}}{\text{mm}^3}$. Aus der per UV/Vis-Spektralphotometer gemessenen Konzentration $c_s(t)$ von GM in der Umgebungslösung und deren maximal mögliche Konzentration c_{\max} resultiert die Freisetzungskurve $F(t)$

von GM, siehe Abbildung 2.2. Zwischen den gemessenen Datenpunkten, siehe Tabelle 2.1, wurde linear interpoliert. Der Freisetzungsverlauf wird hierbei in

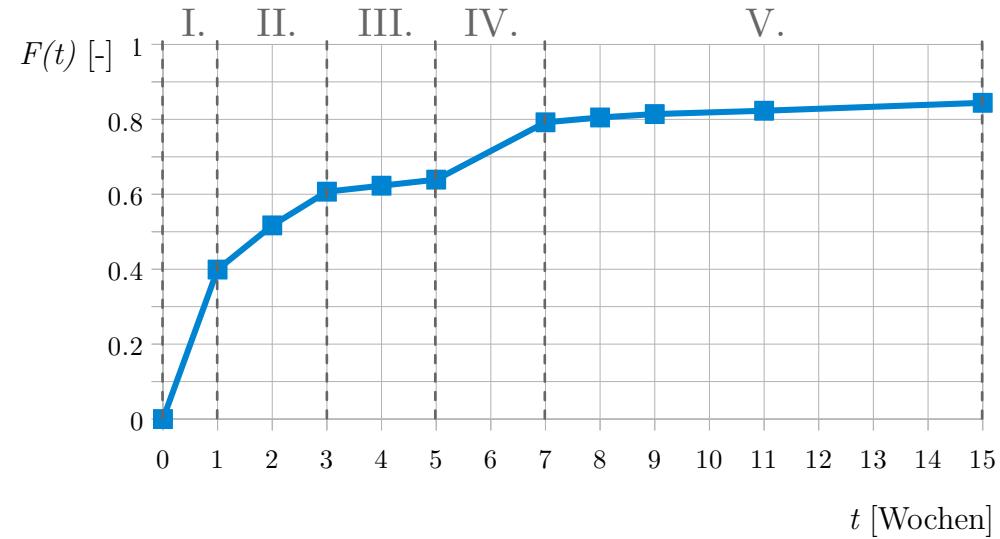


Abb. 2.2: Fünfstufiger GM-Freisetzungsverlauf $F(t)$ [-] über 15 Wochen. Blaue Rechtecke beschreiben die gemessenen Konzentrationswerte. Dazwischen wurde linear interpoliert. I. Stufe: stoSSartige Freisetzung; II. Stufe: starke, aber leicht reduzierte Freisetzung; III. Stufe: Stagnation; IV. Stufe: Freisetzungssteigerung durch Matrixbruch; V. Stufe: geringe Restfreisetzung

Anlehnung an [Macha et al. (2019)] in fünf Stufen unterteilt. Die erste Stufe (1. Woche), die durch eine stoSSartige Wirkstofffreisetzung gekennzeichnet ist, die zweite Stufe (2.–3. Woche), die eine starke, aber leicht reduzierte Freisetzung zeigt, die dritte Stufe (4.–5. Woche) in der nahezu kein Wirkstoff freigesetzt wird (Stagnation) und die vierte Stufe (6.–7. Woche), in der die Wirkstofffreisetzung wieder deutlich steigt bis sie in Stufe fünf (8.–15. Woche) auf geringem Niveau verbleibt.

Woche i	0	1	2	3	4	5	7	8	9	11	15
$F(t_i)$	0	0.399	0.517	0.607	0.623	0.639	0.792	0.805	0.814	0.823	0.844

Tab. 2.1: Experimentell bestimmte Datenpunkte $F(t_i)$ [-] der GM-Freisetzung. Dazu wird die Konzentration c_s [$\frac{\text{g}}{\text{mm}^3}$] von GM in der Umgebungslösung an den Wochen i gemessen und durch die maximal mögliche GM-Konzentration c_{\max} [$\frac{\text{g}}{\text{mm}^3}$] in ebendieser geteilt. [Macha et al. (2019)]

Im folgenden Unterkapitel 3.1 wird der Verlauf durch zugrunde liegende physikalische Prozesse erklärt.

3 Polymerträgermatrix

3.1 Matrixdegradations-/Wirkstofffreisetzungsmodell

An dieser Stelle soll ein fünfstufiges Matrixdegradations-/Wirkstofffreisetzungsmodell vorgestellt werden. Dabei handelt es sich um das vierstufige Modell von [Macha *et al.* (2019)], wobei die letzte Stufe in zwei Stufen aufgeteilt wird, um den Matrixbruch zu akzentuieren. Auf diese Weise lässt sich der Verlauf der Freisetzungskurve durch physikalische Phänomene erklären. Dabei werden die Erkenntnisse bezüglich der Wirkstoffpartikel auf der Matrixoberfläche, siehe Unterkapitel 3.2.1, berücksichtigt. Abbildung 3.1 illustriert diesen Prozess.

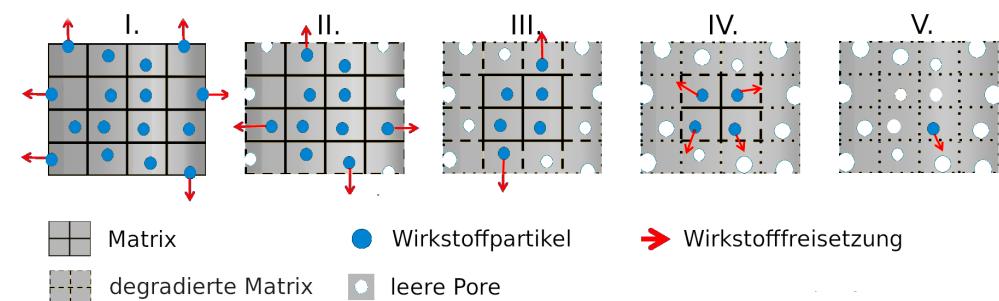


Abb. 3.1: Fünfstufiges Matrixdegradations-/Wirkstofffreisetzungsmodell basierend auf dem vierstufigen Modell nach [Macha *et al.* (2019)]

- Stufe I (1. Woche):
Die stoßartige Freisetzung des Wirkstoffs wird durch die im äußeren Bereich der Matrix liegenden Wirkstoffpartikel ermöglicht. Sie können, ohne weite Wege durch die Matrix zurücklegen zu müssen, in Lösung gehen und in die Umgebungslösung diffundieren.
- Stufe II (2.-3. Woche):
Die Freisetzung ist weiterhin stark, aber leicht reduziert, da der Wirkstoff im äußeren Bereich bereits zum großen Teil in die Umgebungslösung diffundiert ist. Gleichzeitig beginnt die Degradation der Matrix in diesem Bereich. Weiter innen gelegene Wirkstoffpartikel gehen in Lösung

und diffundieren durch die bereits nahezu geleerten Poren und leicht degradierten Matrixanteile in die Umgebungslösung.

- Stufe III (4.-5. Woche):

Die Freisetzung stagniert nahezu. Dies ist darauf zurückzuführen, dass nahezu alle Wirkstoffpartikel in der Nähe der Matrixoberfläche in Lösung gegangen und in die Umgebungslösung diffundiert sind. Gleichzeitig degradiert die Matrix weiterhin. Die Matrixdegradation im inneren Teil ist nicht ausreichend umfangreich, damit dort in Lösung gegangene Partikel deutlich beschleunigt in die Umgebungslösung diffundieren können.

- Stufe IV (6.-7. Woche):

Die Freisetzung steigt deutlich. Dies ist auf den hohen Grad der Matrixdegradation zurückzuführen. Die Matrix bricht zusätzlich entlang der leeren Poren, wodurch diese sich verbinden. Dies erlaubt in Lösung gegangenen Wirkstoffpartikeln aus dem Inneren der Matrix deutlich schneller, durch die Matrix hindurch, in die Umgebungslösung zu diffundieren.

- Stufe V (8.-15. Woche):

Die Freisetzung geht ab der achten Woche wieder stark zurück, da der Großteil der Wirkstoffpartikel bereits in Lösung gegangen und durch die Matrix in die Umgebungslösung diffundiert ist.

3.2 Wirkstoffverteilung zum Startzeitpunkt

Die Gesamtmasse des Wirkstoffs m_d zum Startzeitpunkt $t = 0\text{ s}$ unterteilt sich auf die Wirkstoffmasse auf der Matrixoberfläche $m_{d,\Gamma} [\text{g}]$ und die Wirkstoffmasse innerhalb der Matrix $m_{d,\Omega_m} [\text{g}]$ und ist dementsprechend definiert nach

$$m_d = m_{d,\Gamma} + m_{d,\Omega_m} . \quad (3.1)$$

3.2.1 Oberfläche

Abbildung 3.2 zeigt eine Rasterelektronenmikroskopaufnahme (REM) einer mit GM besetzten PLA-Matrixoberfläche. Dabei sind die helleren Bereiche GM-Partikel. In der vorliegenden Arbeit wird die Hypothese aufgestellt, dass der sprunghafte Anstieg der Freisetzungskurve in der ersten Woche (Stufe I) in hohem Maße auf die schnell in die Umgebungslösung diffundierenden GM-Partikel auf der Oberfläche zurückzuführen ist, da das aufgelöste GM nicht erst durch die Matrix diffundieren muss. Ziel ist es, die sich auf der

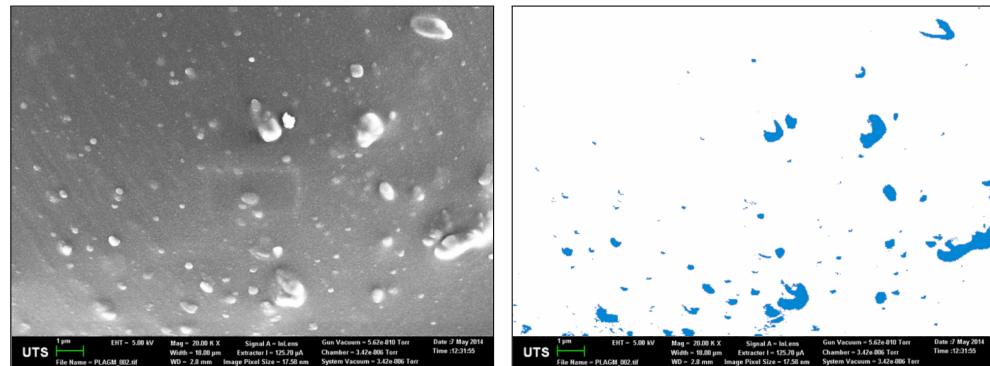


Abb. 3.2: REM-Aufnahme der Oberfläche einer mit GM besetzten PLA-Matrix. Die helleren Bereiche sind GM-Partikel. [Macha *et al.* (2019)]

Abb. 3.3: Nebenstehende REM-Aufnahme der Matrixoberfläche wird mit Hilfe eines GNU-Octave-Programmes in ein Schwarz-WeiSS-Bild umgewandelt. Die blauen Bereiche sind GM-Partikel.

REM-Aufnahme befindliche GM-Masse zu bestimmen, um so Rückschlüsse auf die Gesamtmasse des Wirkstoffs, der sich auf der Oberfläche der Matrix befindet, zu ziehen. Die sich auf der REM-Aufnahme befindliche Wirkstoffmasse $m_{d, \text{REM}}$ [mg] berechnet sich nach

$$m_{d, \text{REM}} = n_{p, \text{REM}} \rho_d V_{p, \text{REM}}^\varnothing , \quad (3.2)$$

wobei ρ_d mit $\rho_d = \rho_{\text{GM}} = 1.3 \times 10^{-9} \frac{\text{mg}}{\mu\text{m}^3}$ die Dichte von GM angibt, siehe [ChemSpider ([o. D.](#))]. Hierbei beschreibt $n_{p, \text{REM}}$ [–] die Anzahl der sich auf der REM-Aufnahme befindlichen Wirkstoffpartikel und $V_{p, \text{REM}}^\varnothing$ [μm^3] das durchschnittliche Volumen eines solchen Partikels. Unter der Annahme, dass alle Wirkstoffpartikel kugelförmig sind, berechnet sich das durchschnittliche Volumen eines sich auf der REM-Aufnahme befindlichen Wirkstoffpartikel nach

$$V_{p, \text{REM}}^\varnothing = \frac{4}{3} \pi r_{p, \text{REM}}^\varnothing {}^3 , \quad (3.3)$$

wobei $r_{p, \text{REM}}^\varnothing$ [μm] den durchschnittlichen Radius eines auf der REM-Aufnahme abgebildeten Wirkstoffpartikels beschreibt. Mit Hilfe eines GNU-Octave-Programmes wird die REM-Aufnahme in ein Schwarz-WeiSS-Bild umgewandelt, siehe Anhang A und [Eaton *et al.* (2023)]. Die schwarzen Bereiche werden daraufhin blau eingefärbt, siehe Abbildung 3.3. Diese blauen Bereiche stellen die GM-Partikel dar. Unter der Vereinfachung, dass es sich bei dem betrachteten Ausschnitt um eine plane Ebene, anstatt um eine Mantelfläche eines Zylinders handelt, ergibt sich für den betrachteten Bereich bei einer Höhe von

$h_{\text{REM}} \approx 18 \mu\text{m}$ und einer Breite $w_{\text{REM}} \approx 12.31 \mu\text{m}$ eine Fläche A_{REM} [μm^2] von

$$A_{\text{REM}} = h_{\text{REM}} w_{\text{REM}} = 2.22 \times 10^2 \mu\text{m}^2 . \quad (3.4)$$

Die REM-Aufnahme besitzt eine Auflösung von 1072×730 Pixeln. Dementsprechend nimmt ein Pixel die Fläche von

$$\begin{aligned} 1072 \times 730 \text{ Pixel} &\doteq 2.22 \times 10^2 \mu\text{m}^2 \\ \Leftrightarrow 1 \times 1 \text{ Pixel} &\approx 2.83 \times 10^{-4} \mu\text{m}^2 \end{aligned} \quad (3.5)$$

ein. Auf diese Weise kann die kumulierte Querschnittsfläche der Partikel auf $A_{p,\text{REM}}^{\text{kum}} \approx 7.72 \mu\text{m}^2$ bestimmt werden. Darüber hinaus wird die Anzahl der Partikel ebenfalls mit Hilfe des Programmes auf $n_{p,\text{REM}} = 2.17 \times 10^2$ ermittelt. Unter der Annahme, dass alle Wirkstoffpartikel auf der Matrixoberfläche den selben Radius haben, resultiert daraus eine durchschnittliche Querschnittsfläche von

$$A_{p,\text{REM}}^\varnothing = \frac{A_{p,\text{REM}}^{\text{kum}}}{n_{p,\text{REM}}} \approx 3.56 \times 10^{-2} \mu\text{m}^2 . \quad (3.6)$$

Dementsprechend ergibt sich ein durchschnittlicher Radius der Wirkstoffpartikel $r_{p,\text{REM}}^\varnothing$ auf der Medikamentenmatrixoberfläche von

$$r_{p,\text{REM}}^\varnothing = \sqrt{\frac{A_{p,\text{REM}}^\varnothing}{\pi}} \approx 1.06 \times 10^{-1} \mu\text{m} . \quad (3.7)$$

Setzt man Gleichung 3.7 zusammen mit $n_{p,\text{REM}} = 2.17 \times 10^2$ in Gleichung 3.3 ein, ergibt sich

$$V_{p,\text{REM}}^\varnothing = \frac{4}{3}\pi r_{p,\text{REM}}^\varnothing {}^3 \approx 5.05 \times 10^{-3} \mu\text{m}^3 . \quad (3.8)$$

Somit ist

$$m_{d,\text{REM}} = \rho_d n_{p,\text{REM}} V_{p,\text{REM}}^\varnothing \approx 1.42 \times 10^{-9} \text{ mg} \quad (3.9)$$

das Resultat für Gleichung 3.2. Nun wird die sich auf der REM-Aufnahme befindliche Wirkstoffmasse $m_{d,\text{REM}}$ auf die gesamte Matrixoberfläche $m_{d,\Gamma}$ hochgerechnet. Diese berechnet sich nach

$$m_{d,\Gamma} = m_{d,\text{REM}} \frac{A_\Gamma}{A_{\text{REM}}} , \quad (3.10)$$

wobei $A_\Gamma [\mu\text{m}^2]$ die Mantelfläche der Medikamentenmatrix beschreibt. Aufgrund der geringen Dicke $d_{\Omega_m} = r_o - r_i = 2 \times 10^2 \mu\text{m}$ der Matrix werden die Stirnflächen bei der Berechnung der Medikamentenmatrixoberfläche A_Γ vernachlässigt, siehe [Rickert et al. (2019)]. Daher ergibt sich

$$A_\Gamma = A_{\Gamma_o} + A_{\Gamma_i} = 2\pi H (r_o + r_i) \approx 8.29 \times 10^8 \mu\text{m}^2 , \quad (3.11)$$

wobei $A_{\Gamma_o} [\mu\text{m}^2]$ die äußere bzw. $A_{\Gamma_i} [\mu\text{m}^2]$ die innere Mantelfläche der Medikamentenmatrix Ω_m beschreibt. Daraus resultiert für Gleichung 3.10 mit $A_\Gamma \approx 8.29 \times 10^8 \mu\text{m}^2$

$$m_{d,\Gamma} \approx 5.33 \times 10^{-3} \text{ mg} . \quad (3.12)$$

Somit befinden sich

$$q_{d,\Gamma} = \frac{m_{d,\Gamma}}{m_d} \times 10^2 \approx 7.11 \times 10^{-2} \% \quad (3.13)$$

des Wirkstoffs auf der Oberfläche der Medikamentenmatrix A_Γ . Mit $F(t = 1 \text{ Woche}) = 0.399$, siehe Tabelle 2.1, ergibt sich mit Gleichung 2.1, 2.2 und 2.3 für $m(t = 1 \text{ Woche})$

$$m(t = 1 \text{ Woche}) = F(t = 1 \text{ Woche}) m_d \approx 2.99 \text{ mg} . \quad (3.14)$$

Dies sind

$$q_{d,1 \text{ Woche}} = \frac{m(t = 1 \text{ Woche})}{m_d} \times 10^2 \approx 39.9 \% \quad (3.15)$$

der gesamten Wirkstoffmasse m_d .

Daraus wird geschlussfolgert, dass 0.07 % der Gesamtirkstoffmasse, die sich

auf der Matrixoberfläche befindet, vernachlässigbar zum sprunghaften Anstieg während der ersten Woche beträgt. Somit wird die zu Beginn dieses Unterkapitels aufgestellte Hypothese widerlegt. Daraus resultiert

$$m_d = m_{d,\Gamma} + m_{d,\Omega_m} \approx m_{d,\Omega_m} \quad (3.16)$$

als vereinfachte Form von Gleichung 3.1.

3.2.2 Wirkstoffverteilung innerhalb der Matrix

[Macha *et al.* (2019)] und [Rickert *et al.* (2019)] gehen davon aus, dass die Verteilung der Wirkstoffkonzentration im Querschnitt der Matrix $c(r, t = 0 \text{ s}) \left[\frac{\text{g}}{\text{mm}^3} \right]$ zu Beginn des Experiments konstant ist, siehe Abbildung 3.4. Dabei berechnet sich die konstante Wirkstoffverteilung $c_0 \left[\frac{\text{g}}{\text{mm}^3} \right]$ mit $m_{d,\Omega_m} \approx m_d$, siehe Gleichung 3.16, nach

$$c_0 = \frac{m_d}{H\pi(r_o^2 - r_i^2)} \approx 9.04 \times 10^{-5} \frac{\text{g}}{\text{mm}^3}. \quad (3.17)$$

In dieser Arbeit wird die Hypothese aufgestellt, dass aus dem Prozess, bei dem der Wirkstoff in die Matrix eingebettet wird, eine parabelförmige Wirkstoffverteilung $c(r, t = 0 \text{ s})$ resultiert. Diese Verteilung wird durch Abbildung 3.5 illustriert.

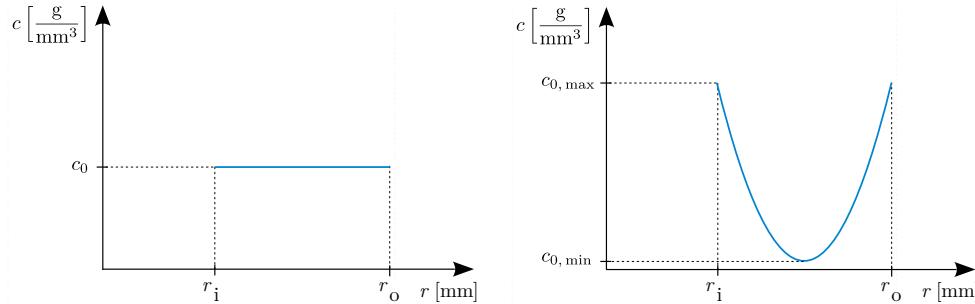


Abb. 3.4: Konstante Wirkstoffverteilung $c(r, t = 0 \text{ s}) = c_0 \left[\frac{\text{g}}{\text{mm}^3} \right]$ über den Matrixquerschnitt zum Startzeitpunkt $t = 0 \text{ s}$ nach [Macha *et al.* (2019)] und [Rickert *et al.* (2019)].

Abb. 3.5: Parabelförmige Wirkstoffverteilung $c(r, t = 0 \text{ s}) \left[\frac{\text{g}}{\text{mm}^3} \right]$ über den Matrixquerschnitt zum Startzeitpunkt $t = 0 \text{ s}$. $c_0, \text{min} \left[\frac{\text{g}}{\text{mm}^3} \right]$ und $c_0, \text{max} \left[\frac{\text{g}}{\text{mm}^3} \right]$ beschreiben die minimale bzw. maximale Konzentration im Matrixquerschnitt.

Davon ausgehend wird argumentiert, dass die stoßartige Freisetzung des Wirkstoffs während Stufe I (1. Woche) in erheblichem Maße durch die hohe Wirkstoffkonzentration im randnahen Bereich begründet ist. Dort können die Wirkstoffpartikel, ohne weite Wege durch die Matrix zurücklegen zu müssen, schneller in Lösung gehen.

Um beide Startkonzentrationsverteilungen des Wirkstoffs quantitativ verglichen zu können, wird die parabelförmige Verteilung mathematisch konkretisiert. Diese Wirkstoffkonzentrationsverteilung zum Startzeitpunkt $c(r, t = 0 \text{ s})$ mit dem minimalen und maximalen Konzentrationswert $c_{0,\min} \left[\frac{\text{g}}{\text{mm}^3} \right]$ bzw. $c_{0,\max} \left[\frac{\text{g}}{\text{mm}^3} \right]$ ist durch die Parabelfunktion

$$c(r, t = 0 \text{ s}) = p \left(r - \frac{r_i + r_o}{2} \right)^2 + v \quad \forall r \in [r_i, r_o] \quad (3.18)$$

gegeben. Hierbei ist $p \left[\frac{\text{g}}{\text{mm}^5} \right]$ der Streckfaktor der Parabel und $v \left[\frac{\text{g}}{\text{mm}^3} \right]$ die Konstante, die die horizontale Verschiebung der Parabel charakterisiert. Damit gilt für den Parabelscheitelpunkt $c_{0,\min} = v$, welcher mit $v \approx 9.04 \times 10^{-9} \frac{\text{g}}{\text{mm}^3}$ gegeben ist. Darüber hinaus gilt für die Wirkstoffmasse innerhalb der hohlzyllinderförmigen Matrix m_{d,Ω_m} mit $m_{d,\Omega_m} \approx m_d$ nach Gleichung 3.16 für den Startzeitpunkt des Experiments $t = 0 \text{ s}$

$$m_d(r, \phi, z, t = 0 \text{ s}) = \int_0^{2\pi} \int_0^H \int_{r_i}^{r_o} c(r, \phi, z) r dr dz d\phi. \quad (3.19)$$

Wird die Unabhängigkeit der Gleichung 3.18 von z und die Rotationssymmetrie bezüglich des Winkels ϕ der Matrix berücksichtigt, ergibt sich für Gleichung 3.19

$$m_d(r, t = 0 \text{ s}) = 2\pi H \int_{r_i}^{r_o} c(r) r dr. \quad (3.20)$$

Durch Einsetzen von Gleichung 3.18 in Gleichung 3.20 resultiert

$$m_d = \frac{-\pi H p}{12} (r_i - r_o)^3 (r_i + r_o) + \pi H v (r_o^2 - r_i^2). \quad (3.21)$$

Somit ist das Resultat für den Streckfaktor p der Parabel mit $m_d = 7.5 \times 10^{-3} \text{ g}$ und $v \approx 9.04 \times 10^{-9} \frac{\text{g}}{\text{mm}^3}$

$$p = \frac{-12(m_d - \pi H v (r_o^2 - r_i^2))}{\pi H (r_i - r_o)^3 (r_i + r_o)} \approx 2.71 \times 10^{-2} \frac{\text{g}}{\text{mm}^5}. \quad (3.22)$$

Für den maximalen Konzentrationswert der parabelförmigen Wirkstoffverteilung ergibt sich $c_{0,\max} = c(r_i, t = 0 \text{ s}) = c(r_o, t = 0 \text{ s}) \approx 2.71 \times 10^{-4} \frac{\text{g}}{\text{mm}^3}$.

In Unterkapitel 6.3 werden die Freisetzungsvorläufe $F(t)$ für eine konstante sowie für eine parabelförmige Startkonzentrationsverteilung des Wirkstoffs $c(r, t = 0 \text{ s})$ präsentiert und in Unterkapitel 7.4 erläutert, ob sich durch die Annahme einer parabelförmigen Verteilung eine verbesserte Vorhersage des Freisetzungsvorlaufs $F(t)$ realisieren lässt.

4 Physikalische Modellbildung

An dieser Stelle soll ein physikalisches Modell vorgestellt werden, das die in Kapitel 2.2 gezeigte experimentell bestimmte Freisetzungskurve, siehe Abbildung 2.2, prädictieren kann.

4.1 Homogene Diffusionsgleichung

Hier soll zunächst eine Modellierung für die Diffusion des Wirkstoffs innerhalb der Matrix vorgestellt werden. Die homogene Diffusionsgleichung wird dazu ausgehend von der Kontinuitätsgleichung in der allgemeinen Schreibweise

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{j} = 0 , \quad (4.1)$$

bei der $\rho \left[\frac{\text{g}}{\text{mm}^3} \right]$ die Dichte und $\mathbf{j} \left[\frac{\text{g}}{\text{mm}^2 \text{s}} \right]$ die Stromdichte beschreibt, hergeleitet. Für eine Konzentration $c(\mathbf{x}, t) \left[\frac{\text{g}}{\text{mm}^3} \right]$, wobei $\mathbf{x} = [x \ y \ z]_i \mathbf{e}_i$ den Ortsvektor des betrachteten Punktes zum Zeitpunkt t [s] symbolisiert, lässt sich die allgemeine Kontinuitätsgleichung 4.1 folgendermaßen umschreiben

$$\frac{\partial c(\mathbf{x}, t)}{\partial t} + \nabla \cdot \mathbf{j} = 0 . \quad (4.2)$$

In dieser Arbeit wird die Konzentration von GM betrachtet. Durch Einsetzen des ersten FICKSchen Gesetz [Fick (1855)] für die Stromdichte \mathbf{j}

$$\mathbf{j} = -D \nabla c(\mathbf{x}, t) , \quad (4.3)$$

wobei $D \left[\frac{\text{mm}^2}{\text{s}} \right]$ den Diffusionskoeffizient von GM beschreibt, in Gleichung 4.2 ergibt sich

$$\frac{\partial c(\mathbf{x}, t)}{\partial t} - D \nabla^2 c(\mathbf{x}, t) = 0 . \quad (4.4)$$

Gleichung 4.4 ist die homogene Diffusionsgleichung.

4.2 Degradationsmodell

Die Degradation, also der Zerfall, einer PLA-Matrix ist prinzipiell durch den Einfluss von

- hydrolytischen
- photochemischen
- mikrobiellen
- enzymatischen

Prozessen bzw. einer Kombination dieser möglich, siehe [Zaaba *et al.* (2020)]. Die im Experiment verwendeten FalconTM-Röhrchen waren lediglich mit dem in die PLA-Matrix eingebetteten GM und der phosphatgepufferten Kochsalzlösung befüllt. Eine mikrobielle, enzymatische sowie photochemische Degradation der Matrix kann somit aufgrund des unter Laborbedingungen und im Innenraum durchgeföhrten Experiments ausgeschlossen werden. Daher wird die Annahme getroffen, dass die Matrixdegradation ausschließlich auf Hydrolyse zurückzuföhren ist, siehe [Lopes *et al.* (2012)]. Da die Hydrolyse ein komplexer Prozess, siehe [Limsukon *et al.* (2023)], der darüber hinaus nach [Gorrasí *et al.* (2017)] von vielen Faktoren, wie z.B. Temperatur, pH-Wert des Lösemittels, Molekülmasse und Kristallinität des PLAs abhängig ist, wird der Degradationsprozess durch ein Modell angenähert, das die Argumentation aus Kapitel 3.1 aufgreift. Daher wird eine Funktion $\kappa(\mathbf{x}, t)$ [–] gewählt, die zu Beginn eine geringe Matrixdegradation aufweist, dann stark ansteigt, um den Matrixbruch abzubilden und daraufhin wieder abflacht. Diese ist nach

$$\kappa(\mathbf{x}, t) = 1 - e^{-\xi \left(1 - \frac{c(\mathbf{x}, t)}{c(\mathbf{x}, t=0)}\right)^2} \quad (4.5)$$

definiert. Die jeweilige Ausprägung kann über die Konstante ξ [–] gesteuert werden. Diese wird im Rahmen der inversen Analyse im Ergebnisteil präsentiert. Zusätzlich wird angenommen, dass je niedriger die Konzentration $c(\mathbf{x}, t)$ innerhalb der Matrix bzw. umso mehr GM bereits aus der Matrix herausdiffundiert ist, desto höher fällt die Matrixdegradation $\kappa(\mathbf{x}, t)$ aus. Dieser Zusammenhang findet im Exponenten der Gleichung Berücksichtigung, der die durch die Startkonzentration $c(\mathbf{x}, t = 0)$ geteilte Differenz zwischen Startkonzentration $c(\mathbf{x}, t = 0)$ und aktueller Konzentration $c(\mathbf{x}, t)$ enthält. Abbildung 4.1 zeigt

den Degradationsverlauf der PLA-Matrix ausgehend von der konstanten Anfangskonzentration $c(\mathbf{x}, t = 0) = c_0 \approx 9.04 \times 10^{-5} \frac{\text{g}}{\text{mm}^3}$ beispielhaft für die Degradationskonstante $\xi [-]$ von $\xi = 1, \dots, 4$.

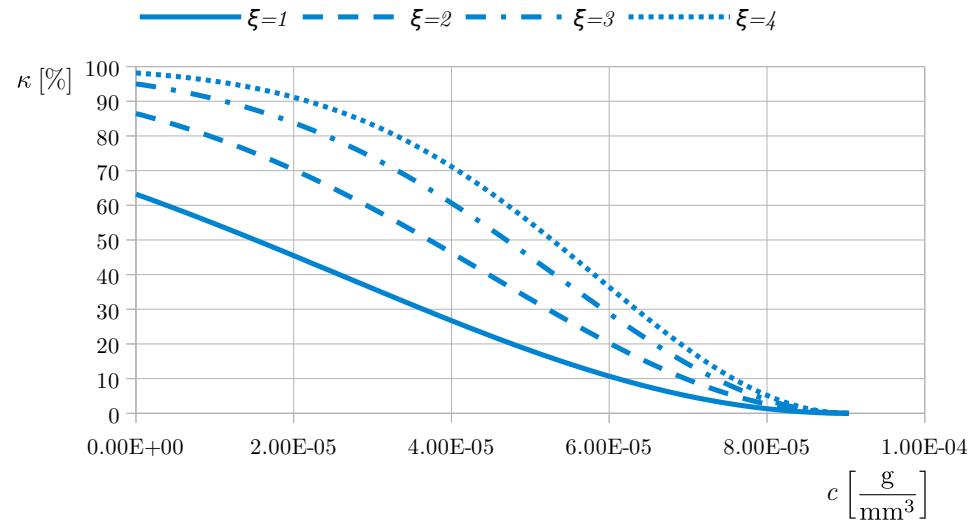


Abb. 4.1: Degradationsverlauf $\kappa(\mathbf{x}, t)$ der Matrix ausgehend von der konstanten Anfangskonzentration $c(\mathbf{x}, t = 0) = c_0 \approx 9.04 \times 10^{-5} \frac{\text{g}}{\text{mm}^3}$ für die Degradationskonstante ξ von $\xi = 1, \dots, 4$.

Aufgrund der nichtkonstanten Anfangskonzentration $c(\mathbf{x}, t = 0)$ der parabelförmigen Wirkstoffverteilung wird auf eine grafische Aufbereitung dieses Degradationsverlaufs verzichtet. Er berechnet sich jedoch analog nach Gleichung 4.5. Die Matrixdegradation $\kappa(\mathbf{x}, t)$ hat Einfluss auf die im folgenden Unterkapitel beschriebene Randbedingung und zwar in der Form, dass bei Erreichen eines bestimmten Werts $\kappa(\mathbf{x}, t)$ der Matrixbruch ausgelöst wird. Dies äußert sich durch eine sprunghafte Steigung des Stoffübergangskoeffizienten h'' , siehe Unterkapitel 4.3.

4.3 Randbedingungen

Bei der Beschreibung von Diffusionsprozessen werden unter anderem konvektive Randbedingungen verwendet. Diese werden analog bei der Modellierung von Wärmeübertragungsprozessen eingesetzt, siehe [Kolesnikov (1987)]. Wie in [Rickert *et al.* (2019)] erläutert, haben diese ROBIN-Randbedingungen im Kontext der Diffusion die Form

$$\mathbf{j}_m \cdot \mathbf{n}_m = h(c(\mathbf{x}, t) - c_\infty) \quad \forall \mathbf{x} \in \Gamma , \quad (4.6)$$

wobei $h \left[\frac{\text{mm}}{\text{s}} \right]$ den Stoffübergangskoeffizienten und $c_\infty \left[\frac{\text{g}}{\text{mm}^3} \right]$ die Konzentration weit entfernt jeglicher Turbulenzen beschreibt. Weiterhin wird in [Rickert *et al.* (2019)] erläutert, dass „diese Randbedingung auf der Idee des FICKSchen Gesetzes basiert, wonach der Massefluss proportional zum Konzentrationsgradienten ist. Da die Konzentration im Umgebungsmedium meistens unbekannt ist und sich der Massefluss zwischen verschiedenen Medien komplexer als FICKSche Diffusion darstellt, wird der Gradient durch die Differenz $c(\mathbf{x}, t) - c_\infty$ angenähert. Um die Diffusion zwischen dem untersuchten Bereich und dem Umgebungsmedium zu charakterisieren, wird der Materialparameter h eingeführt. Dieser kann von einer Vielzahl von Faktoren, wie z.B. Temperatur, Konzentration oder Geschwindigkeit, abhängig sein. Dementsprechend muss er an die entsprechenden Bedingungen angepasst werden. c_∞ kann auch als Sättigungskonzentration im Umgebungsmedium interpretiert werden, da der Massefluss um so geringer wird, je näher die Konzentration im Umgebungsmedium an der Sättigungskonzentration liegt. Dies ist jedoch keine physikalisch fundierte Interpretation, da die Konzentration im Umgebungsmedium nicht berücksichtigt wird.“ Aufgrund der eingeschränkten Eignung eines konstanten Stoffübergangskoeffizienten h und den oben genannten Gründen, wurde in [Macha *et al.* (2019)] eine Erweiterung von Gleichung 4.6 vorgeschlagen. Da sich nach [Macha *et al.* (2019)] in Abbildung 2.2 eine Sättigung der Wirkstoffkonzentration c_s in der Umgebungslösung beobachten lässt, ergibt sich für die Erweiterung der ROBIN-Randbedingung

$$\mathbf{j}_m \cdot \mathbf{n}_m = h'(t)(c(\mathbf{x}, t) - c_s(t)) \quad \forall \mathbf{x} \in \Gamma , \quad (4.7)$$

wobei $h'(t) \left[\frac{\text{mm}}{\text{s}} \right]$ der nach [Macha *et al.* (2019)] modifizierte Stoffübergangskoeffizient ist. Dieser ist nach

$$h'(t) = \begin{cases} \frac{D}{l_b} & , \text{wenn } \alpha(c_* - c_s(t)) \geq 1 \\ \frac{D}{l_b}\alpha(c_* - c_s(t)) & , \text{wenn } \alpha(c_* - c_s(t)) < 1 \end{cases} \quad (4.8)$$

definiert, wobei l_b [mm] die Dicke des Übergangs zwischen Matrix und Umgebungslösung charakterisiert, $\alpha \left[\frac{\text{mm}^3}{\text{g}} \right]$ ein weiterer Stoffübergangskoeffizient ist und $c_* \left[\frac{\text{g}}{\text{mm}^3} \right]$ die Sättigungslöslichkeit beschreibt. Die Konzentration c_∞ , die weit entfernt von der Matrix vorliegt, wurde durch die Sättigungslöslichkeit c_* ersetzt. Der Faktor $\alpha(c_* - c_s(t))$ wird also nur aktiviert, wenn die Wirkstoffkonzentration in der Lösung c_s über der Sättigungslöslichkeit c_* liegt.

Zwischen der 5. und 7. Woche (Stufe IV) steigt die Freisetzung des Wirkstoffs jedoch sprunghaft an, siehe Abbildung 2.2 und Unterkapitel 3.1. Um diesen Prozess abzubilden, wird der Stoffübergangskoeffizient h' von [Macha *et al.* (2019)] wiederum erweitert. Somit ist die neue ROBIN-Randbedingung definiert nach

$$\mathbf{j}_m \cdot \mathbf{n}_m = h''(\mathbf{x}, t)(c(\mathbf{x}, t) - c_s(t)) \quad \forall \mathbf{x} \in \Gamma. \quad (4.9)$$

Für den neuen Stoffübergangskoeffizient $h''(\mathbf{x}, t)$ [$\frac{\text{mm}}{\text{s}}$], der den Bruch der Matrix berücksichtigt, gilt

$$h''(\mathbf{x}, t) = \begin{cases} \frac{D}{l_b} & , \text{ wenn } \alpha(c_* - c_s(t)) \geq 1 \text{ und } \kappa_{\min}(\mathbf{x}, t) \leq \kappa_b \\ \frac{A_{m,b} D}{A_{m,0} l_b} & , \text{ wenn } \alpha(c_* - c_s(t)) \geq 1 \text{ und } \kappa_{\min}(\mathbf{x}, t) > \kappa_b \\ \frac{D}{l_b} \alpha(c_* - c_s(t)) & , \text{ wenn } \alpha(c_* - c_s(t)) < 1 \text{ und } \kappa_{\min}(\mathbf{x}, t) \leq \kappa_b \\ \frac{A_{m,b} D}{A_{m,0} l_b} \alpha(c_* - c_s(t)) & , \text{ wenn } \alpha(c_* - c_s(t)) < 1 \text{ und } \kappa_{\min}(\mathbf{x}, t) > \kappa_b \end{cases}, \quad (4.10)$$

wobei $A_{m,0}$ [mm^2] die Oberfläche der hohlzylinderförmigen Matrix Ω_m im unbeschädigten Ausgangszustand $t = 0\text{s}$ beschreibt. Die Grund- und Deckfläche wird dabei vernachlässigt, da die Vereinfachung getroffen wurde, dass die Diffusion lediglich über die Mantelflächen hinaus erfolgt, siehe [Rickert *et al.* (2019)]. $A_{m,b}$ [mm^2] wiederum beschreibt die durch das Aufbrechen der Matrix Ω_m entlang der leeren Poren vergrößerte Matrixoberfläche zum Zeitpunkt des Bruchs t_b [s]. Dies geschieht, wenn der bruchspezifische Degradationswert $\kappa(\mathbf{x}, t = t_b) = \kappa_b$ [–] erreicht wird. Dieser wird im Rahmen der Inversen Analyse bestimmt. Bricht die Matrix Ω_m also auf, steigt der Stoffübergangskoeffizient h'' sprunghaft an, wodurch sich der sprunghafte Anstieg der Wirkstofffreisetzung modellieren lässt.

4.4 Wirkstoffkonzentration in der Umgebungslösung

Unter der Annahme, dass die Diffusion innerhalb der Umgebungslösung Ω_s sehr viel schneller als in der Matrix Ω_m abläuft, der Wirkstoff sich also sofort nach Verlassen der Matrix Ω_m in der Umgebungslösung Ω_s auflöst, kann die Wirkstoffkonzentration c_s in der Umgebungslösung Ω_s durch eine alleinig von der Zeit abhängigen Funktion $c_s := c_s(t)$ modelliert werden, siehe [Rickert *et al.* (2019)]. Die Massenbilanz der Umgebungslösung Ω_s lautet

$$\frac{dc_s(t)}{dt} = -\nabla \cdot \mathbf{j}_s . \quad (4.11)$$

Wird das Flussintegral für Gleichung 4.11 formuliert, ergibt sich

$$V_s \frac{dc_s(t)}{dt} = - \int_{\delta\Omega_s} \mathbf{j}_s \cdot \mathbf{n}_s \, dA . \quad (4.12)$$

Da GM nicht durch das FalconTM-Röhrchen ($\delta\Omega_s \setminus \Gamma$), in der sich die Umgebungsflüssigkeit Ω_s befindet, diffundieren kann, muss nur der Rand Γ zwischen Matrix Ω_m und Umgebungsflüssigkeit Ω_s betrachtet werden. Daraus folgt

$$V_s \frac{dc_s(t)}{dt} = - \int_{\Gamma} \mathbf{j}_s \cdot \mathbf{n}_s \, dA . \quad (4.13)$$

Mit $\mathbf{n}_s = -\mathbf{n}_m$ bzw. $\mathbf{j}_s \cdot \mathbf{n}_s = -\mathbf{j}_m \cdot \mathbf{n}_m$ ergibt sich

$$V_s \frac{dc_s(t)}{dt} = \int_{\Gamma} \mathbf{j}_m \cdot \mathbf{n}_m \, dA \quad (4.14)$$

und unter Berücksichtigung der neuen ROBIN-Randbedingung $h''(\mathbf{x}, t)$ nach Gleichung 4.9 resultiert für Gleichung 4.14

$$V_s \frac{dc_s(t)}{dt} = h''(\mathbf{x}, t) \int_{\Gamma} (c(\mathbf{x}, t) - c_s(t)) \, dA . \quad (4.15)$$

Gleichung 4.15 ist also die Gleichung für die Konzentration $c_s(t)$ von GM in der Umgebungslösung Ω_s .

4.5 Freisetzungskurve

Analog zu Gleichung 2.1 berechnet sich die numerisch berechnete Freisetzungskurve nach

$$F_{\text{num}}(t) = \frac{c_s(t)}{c_{\max}} . \quad (4.16)$$

Wird darüber hinaus Gleichung 2.2 in Gleichung 4.16 eingesetzt, ergibt sich

$$F_{\text{num}}(t) = \frac{m(t)}{V_s c_{\max}} \quad (4.17)$$

bzw. unter Verwendung der Integralschreibweise resultiert daraus

$$F_{\text{num}}(t) = \frac{1}{V_s c_{\max}} \int_{\tau=0}^{\tau=t} \dot{m}(\tau) d\tau. \quad (4.18)$$

Wird in Gleichung 4.18 wiederum die Gleichung für den Massenstrom

$$\dot{m}(\tau) = \oint_{\Gamma} \mathbf{j}_m \cdot \mathbf{n}_m dA \quad (4.19)$$

eingesetzt, ist das Resultat hierfür

$$F_{\text{num}}(t) = \frac{1}{V_s c_{\max}} \int_{\tau=0}^{\tau=t} \oint_{\Gamma} \mathbf{j}_m \cdot \mathbf{n}_m dAd\tau. \quad (4.20)$$

Durch Einsetzen von Gleichung 4.3 in Gleichung 4.20 ergibt sich schlussendlich

$$F_{\text{num}}(t) = \frac{1}{V_s c_{\max}} \int_{\tau=0}^{\tau=t} \oint_{\Gamma} (-D \nabla c(\mathbf{x}, t)) \cdot \mathbf{n}_m dAd\tau. \quad (4.21)$$

Überführt man Gleichung 4.21 mit $dA = rd\phi dz$ in Zylinderkoordinaten, resultiert daraus

$$F_{\text{num}}(t) = \frac{1}{V_s c_{\max}} \int_{\tau=0}^{\tau=t} \int_0^{2\pi} \int_0^H (-D \nabla c((r, \phi, z), t)) \cdot \mathbf{n}_m r d\phi dz d\tau. \quad (4.22)$$

Unter Berücksichtigung der Unabhängigkeit von z und der Rotationssymmetrie bezüglich des Winkels ϕ der hohlzylinderförmigen Matrix Ω_m vereinfacht sich Gleichung 4.22 zu

$$F_{\text{num}}(t) = -\frac{2\pi HD}{V_s c_{\max}} \int_{\tau=0}^{\tau=t} \nabla c(r, t) \cdot \mathbf{n}_m r d\tau. \quad (4.23)$$

Gleichung 4.23 ist also die Gleichung für die Freisetzung von GM aus der Matrix Ω_m in die Umgebungslösung Ω_s .

5 Numerische Betrachtungen

In diesem Kapitel soll das in Kapitel 4 entworfene physikalische Modell zeitlich und räumlich diskretisiert werden. Methoden zur Bestimmung der dabei entstehenden Diskretisierungsfehler werden vorgestellt. Anschließend wird die Methode der inversen Analyse präsentiert. Mit deren Hilfe lassen sich die bislang unbekannten Materialparameter Diffusionskoeffizient D , Stoffübergangskoeffizient α , Längenparameter l_b , Sättigungslöslichkeit c_* , Degradationskonstante ξ , bruchspezifischer Degradationswert κ_b und Oberfläche der aufgebrochenen Matrix $A_{m,b}$ bestimmen.

5.1 Einheitenlose Darstellung

Die Berechnungen werden mit dem Programm FEniCS durchgeführt, siehe [Alnæs *et al.* (2015)]. Einheitenbehaftete Konstanten und Variablen werden dort nicht unterstützt. Zudem reduzieren Werte, die optimalerweise die Größen eins haben, den numerischen Fehler, siehe [Langtangen *et al.* (2016b)]. Daher werden die in Kapitel 4 vorgestellten Konstanten und Variablen mit Hilfe von Referenzkonstanten in die einheitenlose Form überführt. Diese werden wie folgt gewählt

$$c_{\text{ref}} = c_0 = \frac{m_d}{V_c} \approx 9.04 \times 10^{-5} \frac{\text{g}}{\text{mm}^3}, \quad t_{\text{ref}} = 1 \text{ Woche} = 604800 \text{ s}, \\ l_{\text{ref}} = r_i = 3.2 \text{ mm}, \quad D_{\text{ref}} = D_{\text{GM, expect}} = 2.6 \times 10^{-7} \frac{\text{mm}^2}{\text{s}}, \quad (5.1)$$

wobei $D_{\text{GM, expect}}$ einen erwartbaren Wert für den zu bestimmenden Diffusionskoeffizienten D darstellt. Ursprünglich wurde ein Wert von $D'_{\text{ref}} = 10^{-9} \frac{\text{mm}^2}{\text{s}}$ angenommen, da die Ergebnisse für die Bestimmung des Diffusionskoeffizienten D von [Macha *et al.* (2019)], [Rickert *et al.* (2019)] und [Morozova *et al.* (2021)], denen dasselbe Experiment zugrunde liegt, in diesem Bereich lagen. Erste Simulationen des hier vorgestellten Modells zeigten jedoch, dass Ergebnisse für den Diffusionskoeffizienten D im Bereich $D_{\text{GM, expect}} = 2.6 \times 10^{-7} \frac{\text{mm}^2}{\text{s}}$ zu

erwarten sind. Deswegen wird die Referenzkonstante für den Diffusionskoeffizienten $D_{\text{ref}} = D_{\text{GM, expect}} = 2.6 \times 10^{-7} \frac{\text{mm}^2}{\text{s}}$ gewählt. Die bereits vorgestellten Konstanten und Variablen werden in die einheitenlose Form überführt

$$\begin{aligned} c &= c_{\text{ref}} \tilde{c}, \quad c_s = c_{\text{ref}} \tilde{c}_s, \quad c_\infty = c_{\text{ref}} \tilde{c}_\infty, \quad c_* = c_{\text{ref}} \tilde{c}_*, \quad c_{\max} = c_{\text{ref}} \tilde{c}_{\max}, \\ D &= D_{\text{ref}} \tilde{D}, \quad \mathbf{j} = \frac{D_{\text{ref}} c_{\text{ref}}}{l_{\text{ref}}} \tilde{\mathbf{j}}, \quad h'' = \frac{D_{\text{ref}}}{l_{\text{ref}}} \tilde{h}'', \quad \alpha = \frac{\tilde{\alpha}}{c_{\text{ref}}}, \quad t = t_{\text{ref}} \tilde{t}, \\ \mathbf{x} &= l_{\text{ref}} \tilde{\mathbf{x}}, \quad l = l_{\text{ref}} \tilde{l}, \quad r = l_{\text{ref}} \tilde{r}, \quad r_i = l_{\text{ref}} \tilde{r}_i, \quad r_o = l_{\text{ref}} \tilde{r}_o, \quad H = l_{\text{ref}} \tilde{H}, \\ A_m &= l_{\text{ref}}^2 \tilde{A}_m, \quad V_s = l_{\text{ref}}^3 \tilde{V}_s, \end{aligned} \tag{5.2}$$

wobei Konstanten und Variablen mit hochgestelltem Tildesymbol einheitenlos sind.

Die grundlegenden Gleichungen, die ebenfalls in Kapitel 4 eingeführt wurden, werden jetzt im Folgenden durch Einsetzen der Konstanten und Variablen aus Gleichung 5.1 und 5.2 in die einheitenlose Form überführt.

5.1.1 Homogene Diffusionsgleichung

Die einheitenlose Darstellung der homogenen Diffusionsgleichung 4.4 liest sich demnach wie folgt

$$\frac{\partial \tilde{c}(\tilde{\mathbf{x}}, \tilde{t})}{\partial \tilde{t}} - \beta \tilde{D} \tilde{\Delta} \tilde{c}(\tilde{\mathbf{x}}, \tilde{t}) = 0, \tag{5.3}$$

wobei $\beta = \frac{D_{\text{ref}} t_{\text{ref}}}{l_{\text{ref}}^2}$ gilt.

5.1.2 Degradationsmodell

$$\tilde{\kappa}(\tilde{\mathbf{x}}, \tilde{t}) = 1 - e^{-\xi \left(1 - \frac{\tilde{c}(\tilde{\mathbf{x}}, \tilde{t})}{\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}=0)} \right)^2} \tag{5.4}$$

ist die einheitenlose Form von Gleichung 4.5.

5.1.3 Randbedingungen

$$\tilde{\mathbf{j}}_m \cdot \mathbf{n}_m = \tilde{h}''(\tilde{\mathbf{x}}, \tilde{t}) (\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}) - \tilde{c}_s(\tilde{t})) \quad (5.5)$$

stellt dementsprechend die in die einheitenlose Form überführte Gleichung 4.9 dar.

5.1.4 Stoffübergangskoeffizient h''

$$\tilde{h}''(\tilde{\mathbf{x}}, \tilde{t}) = \begin{cases} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] \geq 1 \text{ und } \tilde{\kappa}_{\min}(\tilde{\mathbf{x}}, \tilde{t}) \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_{m,b}}{\tilde{A}_{m,0}} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] \geq 1 \text{ und } \tilde{\kappa}_{\min}(\tilde{\mathbf{x}}, \tilde{t}) > \tilde{\kappa}_b \\ \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] < 1 \text{ und } \tilde{\kappa}_{\min}(\tilde{\mathbf{x}}, \tilde{t}) \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_{m,b}}{\tilde{A}_{m,0}} \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s(\tilde{t})] < 1 \text{ und } \tilde{\kappa}_{\min}(\tilde{\mathbf{x}}, \tilde{t}) > \tilde{\kappa}_b \end{cases} \quad (5.6)$$

ist mit $l_b = l_{\text{ref}}$ die einheitenlose Darstellung von Gleichung 4.10 für den neuen Stoffübergangskoeffizient h'' .

5.1.5 Diffusionsgleichung der Umgebungslösung

Gleichung 4.15 in einheitenloser Notation ergibt

$$\frac{\partial \tilde{c}_s(\tilde{t})}{\partial \tilde{t}} = \gamma \tilde{h}''(\tilde{\mathbf{x}}, \tilde{t}) \int_{\tilde{\Gamma}} (\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}) - \tilde{c}_s(\tilde{t})) d\tilde{A}, \quad (5.7)$$

wobei $\gamma = \frac{D_{\text{ref}} t_{\text{ref}} l_{\text{ref}}}{V_s}$ zutrifft.

5.1.6 Freisetzungskurve

$$F_{\text{num}}(\tilde{t}) = -\frac{2\pi \tilde{H} \tilde{D} \gamma}{\tilde{c}_{\max}} \int_{\tilde{\tau}=0}^{\tilde{\tau}=\tilde{t}} \tilde{\nabla} \tilde{c}(\tilde{r}, \tilde{\tau}) \cdot \mathbf{n}_m \tilde{r} d\tilde{\tau} \quad (5.8)$$

ist die einheitenlose Darstellung von Gleichung 4.23.

5.2 Zeitliche Diskretisierung

Zur zeitlichen Diskretisierung wird das implizite CRANK-NICOLSON-Verfahren verwendet, siehe [Crank *et al.* (1947)]. Dieses wird aus dem gewichteten Mittel des expliziten Vorwärts- und des impliziten Rückwärts-EULER-Verfahrens gebildet. Es hat den Vorteil, dass es für die Diffusionsgleichung numerisch stabil und zweiter Ordnung $O(\Delta t^2) + O(\Delta x^2)$ ist, siehe [Thomas (1995)]. Jedoch tendiert die Lösung bei unstetigen Startbedingungen oder wie in diesem Fall bei Unstetigkeit zwischen Randbedingung und Startbedingung zu Oszillationen, siehe [Østerby (2003)]. Aus diesem Grund werden die ersten $n_{t,EULER} = 100$ Berechnungsschritte bis zum Zeitpunkt $t_{EULER,end} \approx 4.23 \times 10^{-2}$ s mit dem expliziten Vorwärts-EULER-Verfahren berechnet. Dieses ist für eindimensionale, partielle Differentialgleichungen für eine COURANT-FRIEDRICHS-LEWY-Zahl von $C_{\max} = 1$ numerisch stabil, solange

$$D \frac{\Delta t}{\Delta x} \leq C_{\max} \quad (5.9)$$

gilt, siehe [Courant *et al.* (1928)]. Δt [s] beschreibt dabei die Länge des Zeitschritts, Δx [mm] die Länge des zur räumlichen Diskretisierung verwendeten Elements, siehe Unterkapitel 5.3. Um in den nachfolgenden Abschnitten eine universelle Darstellung der homogenen Diffusionsgleichung in Vorwärts-EULER- sowie CRANK-NICOLSON-Notation zu ermöglichen, wird an dieser Stelle das Θ -Schema eingeführt

$$\frac{\partial u}{\partial t} = f(t) \quad \Rightarrow \quad \frac{u^{n+1} - u^n}{\Delta t} = \Theta f^{n+1} + (1 - \Theta) f^n. \quad (5.10)$$

u^n beschreibt dabei die numerische Lösung zum Zeitpunkt t_n am Ort \mathbf{x} bzw. u^{n+1} zum Zeitpunkt t_{n+1} , also $u^n := u(\mathbf{x}, t_n)$ bzw. $u^{n+1} := u(\mathbf{x}, t_{n+1})$. Analog wird $f^n := f(\mathbf{x}, t_n)$ und $f^{n+1} := f(\mathbf{x}, t_{n+1})$ als Notation gewählt. Weiterhin gilt

$$\Theta = \begin{cases} 0 & \text{Vorwärts-EULER} \\ \frac{1}{2} & \text{CRANK-NICOLSON} \end{cases}. \quad (5.11)$$

Um sich die Konvergenzeigenschaften und die numerische Stabilität des CRANK-NICOLSON-Verfahrens zu eigen zu machen und damit die Berechnungszeit zu reduzieren, wird nach Durchführung der $n_{t,EULER}$ Berechnungsschritte ein adaptives CRANK-NICOLSON-Verfahren implementiert, siehe [Blechta

et al. (2018)]. Dazu wird jeweils eine Lösung mit hoher Präzision u_{high} sowie mit niedriger Präzision u_{low} berechnet, die sich dadurch unterscheiden, dass sie mit halber bzw. doppelter Zeitschrittänge berechnet werden, also $\Delta t_{\text{high}} = \Delta t_{\text{low}}/2 = \Delta t/2$ [s]. Für den Zeitschritt Δt wird die Abweichung zwischen den beiden Lösungen mit Hilfe der RICHARDSON-Extrapolation η [–]

$$\eta = \frac{\|u_{\text{high}}^{n+1} - u_{\text{low}}^{n+1}\|_{L^2(\Omega_m)}}{2^p - 1} \quad (5.12)$$

berechnet, siehe [Richardson (1911)]. $\| \dots \|_{L^2(\Omega_m)}$ entspricht der Norm auf dem L^2 -Hilbertraum und $p = 2$ gilt für das CRANK-NICOLSON-Verfahren. Liegt das Ergebnis der RICHARDSON-Extrapolation η unter der Toleranz tol [–] wird die Zeitschrittänge Δt^* [s] für den nächsten Zeitschritt nach

$$\Delta t^* = \left(\frac{s \, tol}{\eta} \right)^{\frac{1}{p}} \Delta t \quad (5.13)$$

berechnet, wobei $0 < s \leq 1$ ein Sicherheitsfaktor ist. Dabei wird der Sicherheitsfaktor s mit $s = 0.9$ und die Toleranz tol mit $tol = 10^{-3}$ gewählt, siehe [Blechta *et al.* (2018)]. Gilt jedoch $\eta > tol$ wird die Zeit t auf den eben verwendeten Zeitschrittstartpunkt zurückgesetzt und dann mit einem Zeitschritt, der einem Zehntel der ursprünglichen Zeitschrittänge entspricht, berechnet. Algorithmus 5.1 fasst die Implementierung des adaptiven CRANK-NICOLSON-Verfahrens zusammen.

Algorithmus 5.1 Adaptives CRANK-NICOLSON-Verfahren

```

while  $t < t_{\text{end}}$  do
     $t+ = \Delta t$ 
     $u_{\text{low}}^{n+1}$  berechnen
     $u_{\text{high}}^{n+1}$  berechnen
     $\eta$  berechnen
    if  $\eta < tol$  then
         $\Delta t^*$  berechnen
         $\Delta t \leftarrow \Delta t^*$ 
    else
         $t- = \Delta t$ 
         $\Delta t = \Delta t/10$ 

```

Im Folgenden werden die in Unterkapitel 5.1 vorgestellten einheitenlosen Formeln zeitlich diskretisiert.

5.2.1 Homogene Diffusionsgleichung

Mit $u^n := \tilde{c}^n = \tilde{c}(\tilde{\mathbf{x}}, \tilde{t}_n)$ bzw. $u^{n+1} := \tilde{c}^{n+1} = \tilde{c}(\tilde{\mathbf{x}}, \tilde{t}_{n+1})$ und $f^n := -\beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m = -\beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m(\tilde{\mathbf{x}}, \tilde{t}_n)$ bzw. $f^{n+1} := -\beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^{n+1} = -\beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m(\tilde{\mathbf{x}}, \tilde{t}_{n+1})$ ergibt sich für die einheitenlose, homogene Diffusionsgleichung 5.3 unter Verwendung des Θ -Schemas nach Gleichung 5.10

$$\begin{aligned} \frac{\partial \tilde{c}}{\partial \tilde{t}} &= -\beta \tilde{\nabla} \cdot \tilde{\mathbf{j}} \\ \Rightarrow \frac{\tilde{c}^{n+1} - \tilde{c}^n}{\Delta \tilde{t}} &= -\left(\Theta \beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^{n+1} + (1 - \Theta) \beta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^n \right) \quad (5.14) \\ \Leftrightarrow \tilde{c}^{n+1} - \tilde{c}^n + \beta \Delta \tilde{t} &\left(\Theta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^{n+1} + (1 - \Theta) \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^n \right) = 0 . \end{aligned}$$

Zunächst wird Gleichung 5.14 mit der Testfunktion δc multipliziert und daraufhin über das einheitenlose Volumen der Medikamentenmatrix $\tilde{\Omega}_m$ integriert

$$0 = \int_{\tilde{\Omega}_m} (\tilde{c}^{n+1} - \tilde{c}^n) \delta c \, d\tilde{V} + \beta \Delta \tilde{t} \int_{\tilde{\Omega}_m} \left(\Theta \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^{n+1} + (1 - \Theta) \tilde{\nabla} \cdot \tilde{\mathbf{j}}_m^n \right) \delta c \, d\tilde{V} . \quad (5.15)$$

Durch Anwendung des Satzes von GAUSS auf den zweiten Term der rechten Seite von Gleichung 5.15 ergibt sich

$$\begin{aligned} 0 &= \int_{\tilde{\Omega}_m} (\tilde{c}^{n+1} - \tilde{c}^n) \delta c \, d\tilde{V} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{\Omega}_m} \left(\Theta \tilde{\nabla} \tilde{c}^{n+1} + (1 - \Theta) \tilde{\nabla} \tilde{c}^n \right) \cdot \tilde{\nabla} \delta c \, d\tilde{V} \\ &\quad + \beta \Delta \tilde{t} \int_{\tilde{\Gamma}} \delta c \left(\Theta \tilde{\mathbf{j}}_m^{n+1} + (1 - \Theta) \tilde{\mathbf{j}}_m^n \right) \cdot \mathbf{n}_m \, d\tilde{A} . \quad (5.16) \end{aligned}$$

Und unter Berücksichtigung der einheitenlosen ROBIN-Randbedingung $\tilde{\mathbf{j}}_m \cdot \mathbf{n}_m = \tilde{h}''(\tilde{\mathbf{x}}, \tilde{t})(\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}) - \tilde{c}_s(\tilde{t}))$ bzw. $\tilde{\mathbf{j}}_m^{n+1} \cdot \mathbf{n}_m = \tilde{h}''^{n+1}(\tilde{c}^{n+1} - \tilde{c}_s^{n+1})$ und $\tilde{\mathbf{j}}_m^n \cdot \mathbf{n}_m = \tilde{h}''^n(\tilde{c}^n - \tilde{c}_s^n)$ resultiert

$$0 = \int_{\tilde{\Omega}_m} \delta c (\tilde{c}^{n+1} - \tilde{c}^n) d\tilde{V} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{\Omega}_m} (\Theta \tilde{\nabla} \tilde{c}^{n+1} + (1 - \Theta) \tilde{\nabla} \tilde{c}^n) \cdot \tilde{\nabla} \delta c d\tilde{V} \\ + \beta \Delta \tilde{t} \int_{\tilde{\Gamma}} \delta c (\Theta \tilde{h}''^{n+1} (\tilde{c}^{n+1} - \tilde{c}_s^{n+1}) + (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n)) d\tilde{A}. \quad (5.17)$$

Durch Überführung von Gleichung 5.17 in Zylinderkoordinaten mit $d\tilde{V} = \tilde{r} d\tilde{r} d\tilde{z} d\phi$ und $d\tilde{A} = \tilde{r} d\phi d\tilde{z}$ ergibt sich

$$0 = \int_0^{2\pi} \int_0^{\tilde{H}} \int_{\tilde{r}_i}^{\tilde{r}_o} \delta c (\tilde{c}^{n+1} - \tilde{c}^n) \tilde{r} d\tilde{r} d\tilde{z} d\phi \\ + \beta \Delta \tilde{t} \tilde{D} \int_0^{2\pi} \int_0^{\tilde{H}} \int_{\tilde{r}_i}^{\tilde{r}_o} (\Theta \tilde{\nabla} \tilde{c}^{n+1} + (1 - \Theta) \tilde{\nabla} \tilde{c}^n) \cdot \tilde{\nabla} \delta c \tilde{r} d\tilde{r} d\tilde{z} d\phi \\ + \beta \Delta \tilde{t} \int_0^{\tilde{H}} \int_0^{2\pi} \delta c (\Theta \tilde{h}''^{n+1} (\tilde{c}^{n+1} - \tilde{c}_s^{n+1}) + (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n)) \tilde{r} d\phi d\tilde{z}. \quad (5.18)$$

Wird die Rotationssymmetrie bezüglich des Winkels ϕ der hohlzylinderförmigen Matrix Ω_m und die Unabhängigkeit der Gleichung 5.18 von z berücksichtigt, resultiert daraus

$$0 = \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} (\tilde{c}^{n+1} - \tilde{c}^n) \delta c d\tilde{r} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \left(\Theta \frac{\partial \tilde{c}^{n+1}}{\partial \tilde{r}} + (1 - \Theta) \frac{\partial \tilde{c}^n}{\partial \tilde{r}} \right) \frac{\partial \delta c}{\partial \tilde{r}} d\tilde{r} \\ + \beta \Delta \tilde{t} \left(\tilde{r} \Theta \tilde{h}''^{n+1} (\tilde{c}^{n+1} - \tilde{c}_s^{n+1}) \delta c + \tilde{r} (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \delta c \right). \quad (5.19)$$

Für $\mathcal{A}(\tilde{c}^{n+1}, \delta c)$ und $\mathcal{L}(\tilde{c}^n, \delta c)$ ergibt sich somit

$$\begin{aligned}
\mathcal{A}(\tilde{c}^{n+1}, \delta c) &= \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \tilde{c}^{n+1} \delta c \, d\tilde{r} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \Theta \frac{\partial \tilde{c}^{n+1}}{\partial \tilde{r}} \frac{\partial \delta c}{\partial \tilde{r}} \, d\tilde{r} \\
&\quad + \Theta \tilde{h}''^{n+1} \beta \Delta \tilde{t} (\tilde{r} \tilde{c}^{n+1} \delta c) \\
&= \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \tilde{c}^{n+1} \delta c \, d\tilde{r} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \Theta \frac{\partial \tilde{c}^{n+1}}{\partial \tilde{r}} \frac{\partial \delta c}{\partial \tilde{r}} \, d\tilde{r} \\
&\quad + \Theta \tilde{h}''^{n+1} \beta \Delta \tilde{t} (\tilde{r}_i \tilde{c}^{n+1} \delta c) + \Theta \tilde{h}''^{n+1} \beta \Delta \tilde{t} (\tilde{r}_o \tilde{c}^{n+1} \delta c)
\end{aligned} \tag{5.20}$$

und

$$\begin{aligned}
\mathcal{L}(\tilde{c}^n, \delta c) &= \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \tilde{c}^n \delta c \, d\tilde{r} - \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} (1 - \Theta) \frac{\partial \tilde{c}^n}{\partial \tilde{r}} \frac{\partial \delta c}{\partial \tilde{r}} \, d\tilde{r} \\
&\quad + \beta \Delta \tilde{t} (\tilde{r} \Theta \tilde{h}''^{n+1} \tilde{c}_s^{n+1} \delta c - \tilde{r} (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \delta c) \\
&= \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \tilde{c}^n \delta c \, d\tilde{r} - \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} (1 - \Theta) \frac{\partial \tilde{c}^n}{\partial \tilde{r}} \frac{\partial \delta c}{\partial \tilde{r}} \, d\tilde{r} \\
&\quad + \beta \Delta \tilde{t} (\tilde{r}_i \Theta \tilde{h}''^{n+1} \tilde{c}_s^{n+1} \delta c - \tilde{r}_i (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \delta c) \\
&\quad + \beta \Delta \tilde{t} (\tilde{r}_o \Theta \tilde{h}''^{n+1} \tilde{c}_s^{n+1} \delta c - \tilde{r}_o (1 - \Theta) \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \delta c) .
\end{aligned} \tag{5.21}$$

Gleichung 5.20 und 5.21 ist die bilineare bzw. lineare Darstellung der schwachen Form der homogenen Diffusionsgleichung.

5.2.2 Degradationsmodell

$$\tilde{\kappa}^n = 1 - \epsilon^{-\xi \left(1 - \frac{\tilde{c}^{n-1}}{\tilde{c}^0}\right)^2} \tag{5.22}$$

bzw.

$$\tilde{\kappa}^{n+1} = 1 - \epsilon^{-\xi \left(1 - \frac{\tilde{c}^n}{\tilde{c}^0}\right)^2} \tag{5.23}$$

ist die zeitlich diskretisierte Form von Gleichung 5.4.

5.2.3 Randbedingungen

$$\tilde{\mathbf{j}}_m^n \cdot \mathbf{n}_m = \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \quad (5.24)$$

bzw.

$$\tilde{\mathbf{j}}_m^{n+1} \cdot \mathbf{n}_m = \tilde{h}''^{n+1} (\tilde{c}^{n+1} - \tilde{c}_s^{n+1}) \quad (5.25)$$

ist das Ergebnis für die zeitliche Diskretisierung von Gleichung 5.5.

5.2.4 Stoffübergangskoeffizient \tilde{h}''

Mit $\tilde{A}_m^0 = \tilde{A}_m (\tilde{t} = 0)$ bzw. $\tilde{A}_m^b = \tilde{A}_m (\tilde{t} = \tilde{t}_b)$ ergibt sich

$$\tilde{h}''^n = \begin{cases} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] \geq 1 \text{ und } \tilde{\kappa}_{\min}^n \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_m^b}{\tilde{A}_m^0} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] \geq 1 \text{ und } \tilde{\kappa}_{\min}^n > \tilde{\kappa}_b \\ \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] < 1 \text{ und } \tilde{\kappa}_{\min}^n \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_m^b}{\tilde{A}_m^0} \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^n] < 1 \text{ und } \tilde{\kappa}_{\min}^n > \tilde{\kappa}_b \end{cases} \quad (5.26)$$

bzw.

$$\tilde{h}''^{n+1} = \begin{cases} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] \geq 1 \text{ und } \tilde{\kappa}_{\min}^{n+1} \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_m^b}{\tilde{A}_m^0} \tilde{D} & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] \geq 1 \text{ und } \tilde{\kappa}_{\min}^{n+1} > \tilde{\kappa}_b \\ \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] < 1 \text{ und } \tilde{\kappa}_{\min}^{n+1} \leq \tilde{\kappa}_b \\ \frac{\tilde{A}_m^b}{\tilde{A}_m^0} \tilde{D} \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] & , \text{ wenn } \tilde{\alpha} [\tilde{c}_* - \tilde{c}_s^{n+1}] < 1 \text{ und } \tilde{\kappa}_{\min}^{n+1} > \tilde{\kappa}_b \end{cases} \quad (5.27)$$

für die zeitlich diskretisierte Darstellung von Gleichung 5.6.

5.2.5 Diffusionsgleichung der Umgebungslösung

Zur zeitlichen Diskretisierung der homogenen Diffusionsgleichung, die die Diffusion innerhalb der Matrix Ω_m beschreibt, wird das adaptive, implizite CRANK-NICOLSON-Verfahren angewendet, siehe Kapitel 5.2. Für die Diffusionsgleichung der Umgebungslösung Ω_s würde bei Verwendung eines impliziten

Verfahrens zur zeitlichen Diskretisierung eine nicht lineare Randbedingung entstehen. Darüber hinaus müssten beide Diffusionsgleichungen gleichzeitig gelöst werden, da sie über die ROBIN-Randbedingung miteinander gekoppelt sind. Um Rechenleistung einzusparen wurde in [Rickert *et al.* (2019)] eine Vereinfachung vorgeschlagen. Dort wurde argumentiert, dass die Diffusion in der Umgebungslösung Ω_s sehr viel schneller als in der Matrix Ω_m abläuft. So wird davon ausgegangen, dass der Wirkstoff, der aus der Matrix Ω_m herausdiffundiert, sich sofort – verglichen mit der zeitlichen GröSSenordnung der Diffusion innerhalb der Matrix Ω_m – homogen in der Umgebungslösung Ω_s verteilt. So kann die Umgebungslösungskonzentration als eine alleine von der Zeit abhängige Funktion dargestellt werden, also $\tilde{c}_s = \tilde{c}_s(\tilde{t})$. Daher wird das explizite Vorwärts-EULER-Verfahren nach Gleichung 5.10 für $\Theta = 0$ zur zeitlichen Diskretisierung der Diffusionsgleichung der Umgebungslösung Ω_s angewendet. Dabei werden folgende Notationen verwendet

$$\begin{aligned} u^n &:= \tilde{c}_s^n = \tilde{c}_s(\tilde{t}_n) , \quad u^{n+1} := \tilde{c}_s^{n+1} = \tilde{c}_s(\tilde{t}_{n+1}) , \\ f^n &:= \gamma \tilde{h}''^n \int_{\tilde{\Gamma}} (\tilde{c}^n - \tilde{c}_s^n) d\tilde{A} = \gamma \tilde{h}''(\tilde{\mathbf{x}}, \tilde{t}_n) \int_{\tilde{\Gamma}} (\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}_n) - \tilde{c}_s(\tilde{t}_n)) d\tilde{A} . \end{aligned} \quad (5.28)$$

Somit ergibt sich für die zeitlich diskretisierte Form von Gleichung 5.7

$$\begin{aligned} \frac{\partial \tilde{c}_s(\tilde{t})}{\partial \tilde{t}} &= \gamma \tilde{h}''(\tilde{\mathbf{x}}, \tilde{t}) \int_{\tilde{\Gamma}} (\tilde{c}(\tilde{\mathbf{x}}, \tilde{t}) - \tilde{c}_s(\tilde{t})) d\tilde{A} \\ \Rightarrow \quad \frac{\tilde{c}_s^{n+1} - \tilde{c}_s^n}{\Delta \tilde{t}} &= \gamma \tilde{h}''^n \int_{\tilde{\Gamma}} (\tilde{c}^n - \tilde{c}_s^n) d\tilde{A} \\ \Leftrightarrow \quad \tilde{c}_s^{n+1} &= \tilde{c}_s^n + \Delta \tilde{t} \gamma \tilde{h}''^n \int_{\tilde{\Gamma}} (\tilde{c}^n - \tilde{c}_s^n) d\tilde{A} \end{aligned} \quad (5.29)$$

Überführt man Gleichung 5.29 mit $d\tilde{A} = \tilde{r} d\phi d\tilde{z}$ in Zylinderkoordinaten, ergibt sich

$$\tilde{c}_s^{n+1} = \tilde{c}_s^n + \Delta \tilde{t} 2\pi \tilde{H} \gamma \tilde{h}''^n (\tilde{c}^n - \tilde{c}_s^n) \tilde{r} . \quad (5.30)$$

Bei Gleichung 5.30 wurde bereits die Rotationssymmetrie bezüglich des Winkels ϕ des hohlzylinderförmigen Randes $\tilde{\Gamma}$ und die Unabhängigkeit der Gleichung 5.29 von z berücksichtigt.

5.2.6 Freisetzungskurve

$$\tilde{F}_{\text{num}}^{n+1} = \tilde{F}_{\text{num}}^n - \Delta \tilde{t} \frac{2\pi \tilde{H} \tilde{D} \gamma}{\tilde{c}_{\max}} \tilde{\nabla} \tilde{c}^n \cdot n_m \tilde{r} \quad \text{mit } \tilde{F}_{\text{num}}^{n=0} = 0 . \quad (5.31)$$

ist die mit Hilfe des EULER-Vorwärts-Verfahrens diskretisierte Form von Gleichung 5.8.

5.3 Räumliche Diskretisierung

5.3.1 Polymerträgermatrix

Um die Rechendauer zu minimieren, wird die Rotationssymmetrie bezüglich des Winkels ϕ der hohlzylinderförmigen Matrix Ω_m und die Unabhängigkeit von z der schwachen Form der Diffusionsgleichung 5.18, die die Diffusion innerhalb der Matrix Ω_m beschreibt, ausgenutzt. So reduziert sich die Matrixgeometrie auf einen eindimensionalen Raum entlang des Hohlzylinderradius \tilde{r} , siehe Abbildung 2.1 links oben bzw. Abbildung 3.4 und 3.5. Dieses Intervall mit $\tilde{r} \in [\tilde{r}_i, \tilde{r}_o]$ wird in $n P_j^- \Lambda^0$ -Elemente mit Grad j aufgeteilt, siehe Abbildung 5.1 und [Arnold *et al.* (2014)].



Abb. 5.1: Eindimensionales $P_j^- \Lambda^0$ -Element mit Grad $j = 1$ (links) bzw. $j = 2$ (rechts)

Um einen Kompromiss zwischen Rechendauer und ausreichender Genauigkeit zu gewährleisten, wird mit Hilfe der Konvergenzanalyse aus dem Folgekapitel 5.4 die Anzahl n und der Grad j der $P_j^- \Lambda^0$ -Elemente in Kapitel 6.1 ermittelt.

Umgebungslösung

Wie bereits in Unterkapitel 5.2.5 erläutert, wird die Umgebungslösungskonzentration \tilde{c}_s als eine alleine von der Zeit abhängige Funktion, also $\tilde{c}_s = \tilde{c}_s(\tilde{t})$, dargestellt. Somit wird die Umgebungslösungskonzentration \tilde{c}_s , die im konischen FalconTM-Röhrchen vorliegt, nur für einen Punkt berechnet und daraufhin für jeden Punkt innerhalb des konischen Röhrchens angenommen.

5.4 Konvergenzanalyse

Die Konvergenzanalyse wird unter Zuhilfenahme der *Method of Manufactured Solutions* (MMS) durchgeführt. Die MMS wird unter vielen Verifikations- & Validierungsexperten als Goldstandard angesehen, siehe [Roache (2019)]. Bei der MMS wird zunächst eine Lösung für das Problem angenommen, damit ein Quellterm berechnet werden kann, mit welchem das Problem daraufhin geprüft wird.

5.4.1 Method of Manufactured Solutions

Angenommene Lösung

Zunächst wird die Lösung $\tilde{U}(\tilde{r}, \tilde{t})$ mit

$$\tilde{U}(\tilde{r}, \tilde{t}) = \tilde{U}_0 - \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \sin(\tilde{r}) \quad (5.32)$$

angenommen, wobei $\tilde{U}_0 = 2$ und $\tilde{U}_1 = 1$ beliebig gewählte Konzentrationen in einheitenloser Darstellung und $\tilde{t}_{\text{end}} = 15$ die Gesamtversuchsdauer ebenfalls in einheitenloser Form repräsentieren. Die angenommene Lösung $\tilde{U}(\tilde{r}, \tilde{t})$ muss nach [Roache (2001)] keine physikalische Sinnhaftigkeit gewährleisten, jedoch wird sie hier so gewählt, dass sie für $\tilde{t} = 0$ durch die Startkonzentration \tilde{U}_0 (analog zu $\tilde{c}(\tilde{r}, \tilde{t} = 0)$) charakterisiert wird und über den Zeitraum \tilde{t}_{end} durch den Term rechts vom Minuszeichen in Gleichung 5.32 abnimmt.

Forcing function

Darüber hinaus wird der Quellterm $\tilde{Q}(\tilde{\mathbf{x}}, \tilde{t})$ eingeführt und zur rechten Seite der starken Form der einheitenlosen, homogenen Diffusionsgleichung 5.3 addiert, um den Quellterm $\tilde{Q}(\tilde{\mathbf{x}}, \tilde{t})$ zu berechnen. So ergibt sich

$$\frac{\partial \tilde{u}(\tilde{\mathbf{x}}, \tilde{t})}{\partial \tilde{t}} - \beta \tilde{D} \tilde{\nabla}^2 \tilde{u}(\tilde{\mathbf{x}}, \tilde{t}) = \tilde{Q}(\tilde{\mathbf{x}}, \tilde{t}) . \quad (5.33)$$

Überführt man Gleichung 5.33 in Zylinderkoordinaten in eindimensionaler Form ergibt sich

$$\tilde{r}\tilde{Q}(\tilde{r}, \tilde{t}) = \tilde{r} \frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{t}} - \beta \tilde{D} \frac{\partial}{\partial \tilde{r}} \left(\tilde{r} \frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{r}} \right). \quad (5.34)$$

Mit $\tilde{u}(\tilde{r}, \tilde{t}) := \tilde{U}(\tilde{r}, \tilde{t})$ resultiert für Gleichung 5.34

$$\tilde{Q}(\tilde{r}, \tilde{t}) = \frac{\partial \tilde{U}(\tilde{r}, \tilde{t})}{\partial \tilde{t}} - \frac{1}{\tilde{r}} \beta \tilde{D} \frac{\partial}{\partial \tilde{r}} \left(\tilde{r} \frac{\partial \tilde{U}(\tilde{r}, \tilde{t})}{\partial \tilde{r}} \right). \quad (5.35)$$

Durch Einsetzen der angenommenen Lösung nach Gleichung 5.32 ergibt sich

$$\tilde{Q}(\tilde{r}, \tilde{t}) = -\tilde{U}_1 \frac{\sin(\tilde{r})}{\tilde{t}_{\text{end}}} + \beta \tilde{D} \frac{\tilde{U}_1 \tilde{t}}{\tilde{t}_{\text{end}}} \left(\frac{\cos(\tilde{r})}{\tilde{r}} - \sin(\tilde{r}) \right). \quad (5.36)$$

Gleichung 5.36 ist die *forcing function*. Wird Gleichung 5.36 in Gleichung 5.35 eingesetzt, ergibt sich die zu lösende Gleichung

$$\frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{t}} - \frac{1}{\tilde{r}} \beta \tilde{D} \frac{\partial}{\partial \tilde{r}} \left(\tilde{r} \frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{r}} \right) = -\tilde{U}_1 \frac{\sin(\tilde{r})}{\tilde{t}_{\text{end}}} + \beta \tilde{D} \frac{\tilde{U}_1 \tilde{t}}{\tilde{t}_{\text{end}}} \left(\frac{\cos(\tilde{r})}{\tilde{r}} - \sin(\tilde{r}) \right), \quad (5.37)$$

die mit der exakten angenommenen Lösung nach Gleichung 5.32 zu vergleichen ist.

Anfangsbedingung

$$\tilde{U}(\tilde{r}, \tilde{t} = 0) = \tilde{U}_0 \quad (5.38)$$

ist die Anfangsbedingung für die angenommene Lösung nach Gleichung 5.32 für den Anfangszeitpunkt $\tilde{t} = 0$.

Randbedingungen

Nun werden die Randbedingungen analog zu Gleichung 5.5 nach der Form

$$\tilde{\mathbf{j}}_{\text{MMS}} \cdot \mathbf{n} = \tilde{h}_{\text{MMS}}''(\tilde{r}, \tilde{t})(\tilde{u}(\tilde{r}, \tilde{t}) - \tilde{c}_{s, \text{MMS}}(\tilde{r}, \tilde{t})) \quad (5.39)$$

aufgestellt und $\tilde{h}_{\text{MMS}}''(\tilde{r}, \tilde{t})$ bzw. $\tilde{c}_{s, \text{MMS}}(\tilde{r}, \tilde{t})$ für den linken und rechten Rand bestimmt.

Angewendet auf den linken Rand bei $\tilde{r} = \tilde{r}_i$ mit $\tilde{\mathbf{j}}_{\text{MMS}} \cdot \mathbf{n} = \tilde{D} \frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{r}}$ und $\tilde{u}(\tilde{r}, \tilde{t}) := \tilde{U}(\tilde{r}, \tilde{t})$ resultiert

$$\begin{aligned} \tilde{D} \frac{\partial \tilde{U}(\tilde{r}, \tilde{t})}{\partial \tilde{r}} &= \tilde{h}_{\text{MMS}}''(\tilde{r}, \tilde{t}) (\tilde{U}(\tilde{r}, \tilde{t}) - \tilde{c}_{s, \text{MMS}}(\tilde{r}, \tilde{t})) \\ \Rightarrow -\tilde{D} \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \cos(\tilde{r}_i) &= \tilde{h}_{\text{MMS}}''(\tilde{r}_i, \tilde{t}) \left(\tilde{U}_0 - \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \sin(\tilde{r}_i) - \tilde{c}_{s, \text{MMS}}(\tilde{r}_i, \tilde{t}) \right). \end{aligned} \quad (5.40)$$

Für $\tilde{h}_{\text{MMS}}''(\tilde{r}_i, \tilde{t}) = \tilde{D} \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \cos(\tilde{r}_i)$ und $\tilde{c}_{s, \text{MMS}}(\tilde{r}_i, \tilde{t}) = \tilde{U}_0 - \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \sin(\tilde{r}_i) + 1$ wird Gleichung 5.40 erfüllt.

Für den rechten Rand bei $\tilde{r} = \tilde{r}_o$ gilt mit $\tilde{\mathbf{j}}_{\text{MMS}} \cdot \mathbf{n} = -\tilde{D} \frac{\partial \tilde{u}(\tilde{r}, \tilde{t})}{\partial \tilde{r}}$ und $\tilde{u}(\tilde{r}, \tilde{t}) := \tilde{U}(\tilde{r}, \tilde{t})$

$$\begin{aligned} -\tilde{D} \frac{\partial \tilde{U}(\tilde{r}, \tilde{t})}{\partial \tilde{r}} &= \tilde{h}_{\text{MMS}}''(\tilde{r}, \tilde{t}) (\tilde{U}(\tilde{r}, \tilde{t}) - \tilde{c}_{s, \text{MMS}}(\tilde{r}, \tilde{t})) \\ \Rightarrow \tilde{D} \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \cos(\tilde{r}_o) &= \tilde{h}_{\text{MMS}}''(\tilde{r}_o, \tilde{t}) \left(\tilde{U}_0 - \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \sin(\tilde{r}_o) - \tilde{c}_{s, \text{MMS}}(\tilde{r}_o, \tilde{t}) \right). \end{aligned} \quad (5.41)$$

Gleichung 5.41 ist für $\tilde{h}_{\text{MMS}}''(\tilde{r}_o, \tilde{t}) = \tilde{D} \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \cos(\tilde{r}_o)$ und $\tilde{c}_{s, \text{MMS}}(\tilde{r}_o, \tilde{t}) = \tilde{U}_0 - \tilde{U}_1 \frac{\tilde{t}}{\tilde{t}_{\text{end}}} \sin(\tilde{r}_o) - 1$ erfüllt.

Schwache Form

Analog zu Gleichung 5.19 wird die schwache Form aufgestellt und um die *forcing function* mit $\tilde{Q}^n := \tilde{Q}(\tilde{r}, \tilde{t}_n)$ bzw. $\tilde{Q}^{n+1} := \tilde{Q}(\tilde{r}, \tilde{t}_{n+1})$ als Quellterm ergänzt. Dabei wird erneut das Θ -Schema verwendet, da die Konvergenzeigenschaften des Vorwärts-EULER-Verfahrens aber auch des CRANK-NICOLSON-Verfahrens mit niedriger und hoher Präzision geprüft werden sollen.

$$0 = \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} (\tilde{u}^{n+1} - \tilde{u}^n) \delta u d\tilde{r} + \beta \Delta \tilde{t} \tilde{D} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \left(\Theta \frac{\partial \tilde{u}^{n+1}}{\partial \tilde{r}} + (1 - \Theta) \frac{\partial \tilde{u}^n}{\partial \tilde{r}} \right) \frac{\partial \delta u}{\partial \tilde{r}} d\tilde{r} \\ + \beta \Delta \tilde{t} \left(\tilde{r} \Theta \tilde{h}_{\text{MMS}}''^{n+1} (\tilde{u}^{n+1} - \tilde{c}_{s, \text{MMS}}^{n+1}) \delta u + \tilde{r} (1 - \Theta) \tilde{h}_{\text{MMS}}''^n (\tilde{u}^n - \tilde{c}_{s, \text{MMS}}^n) \delta u \right) \\ - \Delta \tilde{t} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{r} \left(\Theta \tilde{Q}^{n+1} + (1 - \Theta) \tilde{Q}^n \right) \delta u d\tilde{r} \quad (5.42)$$

ist dementsprechend die schwache Form zur Verwendung der MMS.

5.4.2 Diskretisierungsfehler

Der durch die zeitliche Diskretisierung entstandene Fehler err_{n_t} zwischen der angenommenen analytischen Lösung \tilde{U} , siehe Gleichung 5.32, und der nach Gleichung 5.42 numerisch bestimmten Lösung \tilde{u} , hier zur besseren Unterscheidung \tilde{u}_{n_t} genannt, berechnet sich nach

$$err_{n_t} = \frac{1}{n_t} \sum_1^{n_t} \sqrt{\int_{\tilde{r}_i}^{\tilde{r}_o} (\tilde{U}(\tilde{\tau}) - \tilde{u}_{n_t}(\tilde{\tau}))^2 d\tilde{r}}, \quad (5.43)$$

wobei n_t die Anzahl der Teilungen des untersuchten Zeitraums angibt. Dabei wird die Anzahl n_s der $P_1^- \Lambda^0$ bzw. $P_2^- \Lambda^0$ -Elemente festgehalten und die Anzahl der Teilungen n_t des untersuchten Zeitraums schrittweise erhöht. Analog berechnet sich der durch die räumliche Diskretisierung entstandene Fehler err_{n_s} für die numerisch bestimmte Lösung \tilde{u}_{n_s} nach

$$err_{n_s} = \frac{1}{n_t} \sum_1^{n_t} \sqrt{\int_{\tilde{r}_i}^{\tilde{r}_o} (\tilde{U}(\tilde{\tau}) - \tilde{u}_{n_s}(\tilde{\tau}))^2 d\tilde{r}}. \quad (5.44)$$

Hierbei wird jedoch die Anzahl der Teilungen n_t des untersuchten Zeitraums festgehalten und die Anzahl der Elemente n_s schrittweise erhöht.

5.4.3 Konvergenzrate

Um die Konvergenzrate R zu bestimmen, wird die Lösung \tilde{u} bzw. \tilde{u}_{n_s} der MMS bei immer feinerer Vernetzung berechnet und die korrespondieren Zeitschritte

zweier Vernetzungen per L^2 -Norm miteinander verglichen. Hierbei wird die ElementgröSSe g_{n_s} nach

$$g_{n_s} = \frac{1}{n_s} \quad (5.45)$$

verwendet. Die L^2 -Norm E_k kann für die verschiedenen Vernetzungen n_s auch nach

$$E_k = C g_{n_s}^R \quad (5.46)$$

definiert werden, wobei C eine unbekannte Konstante ist. So können zwei Lösungen mit jeweils feinerer Vernetzung bezüglich ihrer L^2 -Norm miteinander verglichen und nach der Konvergenzrate R zu

$$R_{k-1/k} = \frac{\log\left(\frac{E_k}{E_{k-1}}\right)}{\log\left(\frac{g_{n_s}}{g_{n_{s-1}}}\right)} \quad (5.47)$$

aufgelöst werden.

5.5 Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt

Um den Diskretisierungsfehler der Wirkstoffverteilung \tilde{c}^0 zum Startzeitpunkt zu minimieren, wird eine Prüflogik implementiert. Dazu wird zunächst von der diskretisierten Wirkstoffverteilung ausgehend die Wirkstoffmasse \tilde{m}_d^{num} , die sich zu $\tilde{t} = 0$ innerhalb der Matrix $\tilde{\Omega}_m$ befindet, numerisch berechnet, siehe

$$\tilde{m}_d^{\text{num}} = 2\pi \tilde{H} \int_{\tilde{r}_i}^{\tilde{r}_o} \tilde{c}^0 \tilde{r} d\tilde{r} \quad (5.48)$$

und daraufhin mit

$$m_d^{\text{num}} = c_{\text{ref}} l_{\text{ref}}^3 \tilde{m}_d^{\text{num}} \quad (5.49)$$

in die einheitenbehaftete Form überführt, um sie mit der gegebenen Wirkstoffmasse m_d nach

$$err_{m_d} = \frac{m_d^{\text{num}} - m_d}{m_d} 100 \quad (5.50)$$

zu vergleichen. Der relative Fehler err_{m_d} [%] wird im Diskussionsteil zu Rate gezogen, um die Diskretisierungsparameter im Rahmen eines Kompromisses zwischen Diskretisierungsfehlerminimierung und Berechnungsgeschwindigkeit zu bestimmen. Dies wird für die konstante und die parabelförmige, initiale Wirkstoffverteilung durchgeführt und zwar unter Verwendung von $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elementen.

5.6 Softwarelösung

Das hier vorgestellte Modell wird mit Hilfe der Finiten-Elemente-Methode (FEM) unter Verwendung des Programms FEniCS Legacy berechnet, siehe [Alnæs *et al.* (2015)]. Dazu wird der Quellcode in der Programmiersprache Python (Version: 3.10.12) verfasst, siehe [Van Rossum *et al.* (2009)], [Python Software Foundation ([o.D.](#))] und Anhang B.

5.7 Inverse Analyse

Ziel der inversen Analyse für die ersten fünf Experimentswochen ist es, die unbekannten Materialparameter D , α und l_b zu bestimmen. Für die Sättigungskonzentration c_* lässt sich davon ausgehen, dass sie der finalen Umgebungslösungskonzentration c_s ($t = 15$ Wochen) des Experiments entspricht, siehe [Rickert *et al.* (2019)]. Die dazu zu minimierende Verlustfunktion $L_{p,q}$ wird durch

$$L_{p,q} = 0.5 \sum_{j=p}^q (F_{\text{exp}}(t_j) - F_{\text{num}}(t_j))^2 \quad (5.51)$$

beschrieben, wobei t_j mit $j \in \{0, \dots, 6\}$ den diskreten Messzeitpunkten für die ersten fünf Wochen aus Tabelle 2.1 entsprechen. $F_{\text{exp}}(t_j)$ sind dementsprechend die in dieser Tabelle angegebenen Messwerte der experimentell bestimmten Freisetzungskurve. $F_{\text{num}}(t_j)$ wird nach Gleichung 5.8 bzw. 5.31 bestimmt. Auf Gleichung 5.51 wird mit Hilfe des *trust region reflective*-Algorithmus die Methode der kleinsten Quadrate angewendet, um bei kleinstmöglicher Summe der quadrierten Residuen die unbekannten Materialparameter zu bestimmen. Während des Optimierungsprozesses werden Untersuchungen mit unterschiedlichen

Algorithmen (*trust region reflective*, *dogleg*), Parametergrenzen und Residuengewichtungen durchgeführt. Anhang B.16 gibt Aufschluss über die finale Parameterwahl bei der Implementierung der Methode der kleinsten Quadrate mit Hilfe des SciPy *least_squares*-Paketes, siehe [Virtanen *et al.* (2020)]. Aufgrund der weiten Verbreitung des BestimmtheitsmaSS $R_{p,q}^2$ zur Bestimmung der Anpassungsgüte eines Modells wird für die nach Gleichung 5.51 optimierten Materialparameter ebenfalls das BestimmtheitsmaSS $R_{p,q}^2$ nach

$$R_{p,q}^2 = 1 - \frac{\sum_{j=p}^q (F_{\text{exp}}(t_j) - F_{\text{num}}(t_j))^2}{\sum_{j=p}^q (F_{\text{exp}}(t_j) - \bar{F}_{\text{exp}})^2} \quad (5.52)$$

bestimmt. Dabei beschreibt \bar{F}_{exp} den Mittelwert aller experimentell bestimmten Freisetzungswerte $F_{\text{exp}}(t_j)$.

Daraufhin wird die spezifische Bruchkonstante κ_b händisch für den Matrixbruch zum Zeitpunkt $t_b = 5$ Wochen optimiert. Um eine bessere Vergleichbarkeit zu ermöglichen, wird die Degradationskonstante ξ für die konstante sowie parabelförmige Startkonzentrationsverteilung bei $\xi = 1$ gehalten. Für die Oberfläche der aufgebrochenen Matrix $A_{m,b}$ wird im Folgenden eine Optimierung, analog zur Optimierung für die ersten fünf Wochen, für den Zeitraum von der sechsten bis zur letzten Woche durchgeführt. Die entsprechenden Ergebnisse sind in Kapitel 6.3 einzusehen.

6 Ergebnisse

6.1 Konvergenzanalyse

In diesem Kapitel werden zunächst die bei der zeitlichen und räumlichen Diskretisierung des physikalischen Modells entstehenden numerischen Fehler und die Konvergenzrate bestimmt. Daraufhin werden die Ergebnisse der unbekannten Materialparameter, die im Rahmen der inversen Analyse gewonnen werden, samt Freisetzungskurven präsentiert.

6.1.1 Explizites EULER-Verfahren

Das explizite EULER-Verfahren wird von $t = 0 \text{ s}$ bis $t \approx 4.23 \times 10^{-2} \text{ s}$ verwendet und dementsprechend für diesen Zeitraum mit Hilfe der MMS getestet.

Zeitliche Konvergenz

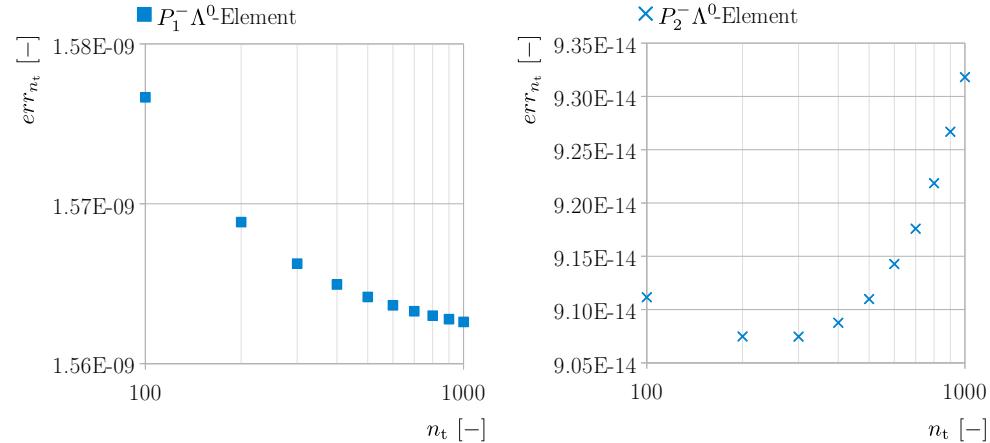


Abb. 6.1: Durch zeitliche Diskretisierung entstandener numerischer Fehler err_{n_t} des expliziten EULER-Verfahrens unter Verwendung der MMS bei Änderung der Teilungsanzahl n_t des untersuchten Zeitraums. Hierbei wird die Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts) bei 100 gehalten.

Abbildung 6.1 zeigt den durch die zeitliche Diskretisierung entstandenen numerischen Fehler err_{n_t} des expliziten EULER-Verfahrens. Hierbei wird die Teilungszahl n_t des untersuchten Zeitraums von 100 bis 1000 in Hunderterstufen erhöht und die Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts) bei 100 gehalten.

Räumliche Konvergenz

Abbildung 6.2 zeigt den durch die räumliche Diskretisierung entstandenen numerischen Fehler err_{n_s} . Die Anzahl n_s der $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elemente wird

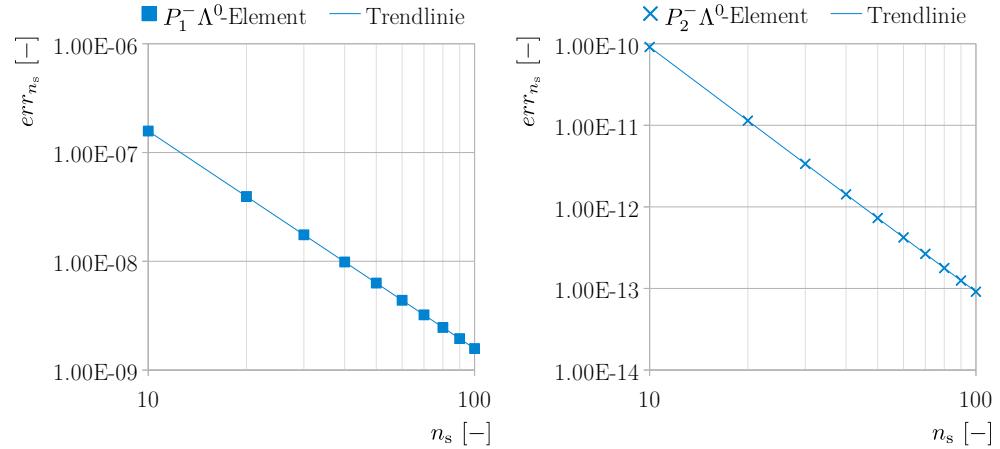


Abb. 6.2: Durch räumliche Diskretisierung entstandener numerischer Fehler err_{n_s} des expliziten EULER-Verfahrens unter Verwendung der MMS bei Änderung der Anzahl n_s der $P_1^- \Lambda^0$ -Elemente (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts). Die Teilungszahl n_t des untersuchten Zeitraums wird bei 100 festgehalten. Zusätzlich wird eine Trendlinie durch die Fehlerwerte angenähert.

dabei von 10 bis 100 in Zehnerschritten erhöht. Die Anzahl n_t der Teilungen des Zeitraums wird bei 100 festgehalten. Die in Abbildung 6.2 angenäherten Trendlinien f werden bei einem BestimmtheitsmaSS von $\mathfrak{R}_{\text{EULER}, P_1^- \Lambda^0}^2 \approx 1$ bzw. $\mathfrak{R}_{\text{EULER}, P_2^- \Lambda^0}^2 \approx 1$ durch folgende Gleichungen beschrieben

$$f_{\text{EULER}, P_1^- \Lambda^0}(n_s) \approx 1.58 \times 10^{-5} n_s^{-2} \quad (6.1)$$

bzw.

$$f_{\text{EULER}, P_2^- \Lambda^0}(n_s) \approx 9.11 \times 10^{-8} n_s^{-3}. \quad (6.2)$$

Tabelle 6.1 gibt einen Überblick über die korrespondierenden Konvergenzraten $R_{k-1/k}$ der berechneten Vernetzungspaare $k - 1/k$.

expl. EULER	n_s	10	20	30	40	50	60	70	80	90	100
$P_1^- \Lambda^0$ -Element		2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00	2.00
$P_2^- \Lambda^0$ -Element		2.99	2.99	2.99	2.99	2.99	3.00	2.99	3.00	2.99	2.99

Tab. 6.1: Konvergenzrate $R_{k-1/k}$ je Paar $k - 1/k$ der Vernetzungsgrade n_s für das explizite EULER-Verfahren für $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elemente. Dabei wird die Teilungsanzahl des Zeitraums n_t bei $n_t = 100$ gehalten.

Anzumerken ist, dass die Nachkommastellen der Konvergenzraten nach der zweiten Stelle abgeschnitten und nicht gerundet werden. Ab der vierten Nachkommastelle können Werte ungleich null vorliegen.

6.1.2 CRANK-NICOLSON-Verfahren

Der Anteil mit niedriger und hoher Präzision des adaptiven CRANK-NICOLSON-Verfahrens wird für den Zeitraum von $\tilde{t} = 0$ bis $\tilde{t}_{\text{end}} = 15$ mit Hilfe der MMS untersucht.

Zeitliche Konvergenz

Abbildung 6.3 zeigt den durch die zeitliche Diskretisierung entstandenen numerischen Fehler err_{n_t} für $P_1^- \Lambda^0$ - (links) und $P_2^- \Lambda^0$ -Elemente (rechts).

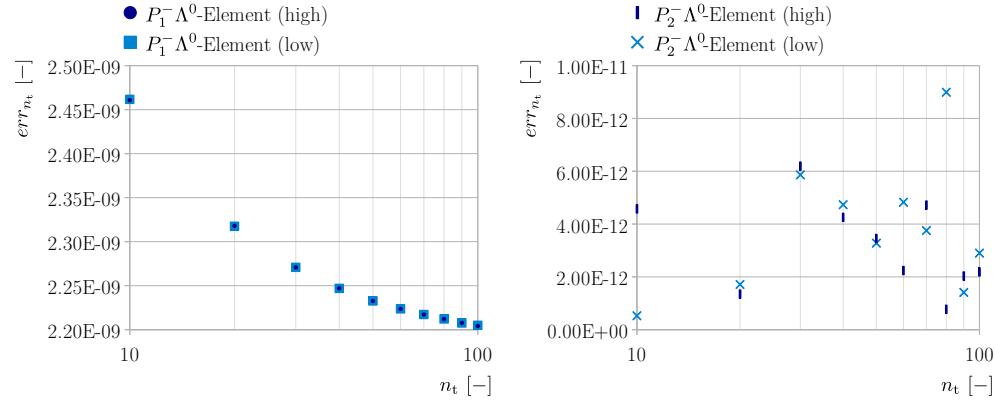


Abb. 6.3: Durch zeitliche Diskretisierung entstandener numerischer Fehler err_{n_t} des CRANK-NICOLSON-Verfahrens mit niedriger und hoher Präzision unter Verwendung der MMS bei Änderung der Teilungsanzahl n_t des untersuchten Zeitraums. Hierbei wird die Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts) bei 100 gehalten.

Der variierte Parameter n_t wird von 10 bis 100 in Zehnerschritten erhöht, die Anzahl der Elemente n_s bei 100 festgehalten. Bedingt durch die grafische Darstellung entsteht der Eindruck, dass die bestimmten Diskretisierungsfehler err_{n_t} für $P_1^- \Lambda^0$ -Elemente identisch sind. Jedoch besteht eine Abweichung ab der dritten Nachkommastelle.

Räumliche Konvergenz

Abbildung 6.4 und zeigt den durch die räumliche Diskretisierung entstandenen numerischen Fehler err_{n_s} für $P_1^- \Lambda^0$ - (links) und $P_2^- \Lambda^0$ -Elemente (rechts).

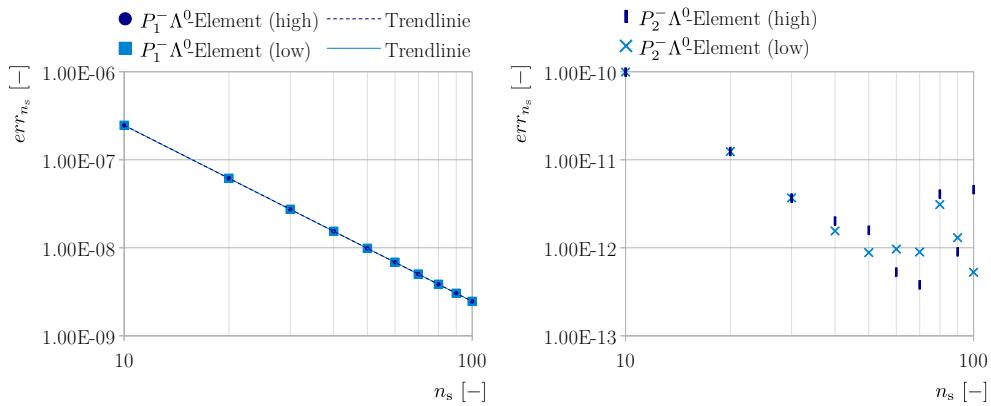


Abb. 6.4: Durch räumliche Diskretisierung entstandener numerischer Fehler err_{n_s} des CRANK-NICOLSON-Verfahrens mit niedriger (low) und hoher (high) Präzision unter Verwendung der MMS bei Änderung der Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts). Die Teilungsanzahl n_t des untersuchten Zeitraums wird bei 10 gehalten. Zusätzlich wird für $P_1^- \Lambda^0$ -Elemente eine Trendlinie durch die Fehlerwerte angenähert.

Die Anzahl der Elemente n_s wird von 10 bis 100 in Zehnerschritten erhöht, wobei die Anzahl der Zeitschritte n_t bei 10 festgehalten wird. Bedingt durch die grafische Darstellung entsteht ebenfalls der Eindruck, dass die bestimmten Fehler err_{n_s} zwischen niedriger und hoher Präzision identisch sind. Jedoch besteht auch hier eine Abweichung ab der zweiten Nachkommastelle. Die in Abbildung 6.4 angenäherten Trendlinien f werden durch folgende Gleichungen mit einem BestimmtheitsmaSS von $\Re_{\text{CN (niedrig), } P_1^- \Lambda^0}^2 \approx 1$ bzw. $\Re_{\text{CN (hoch), } P_1^- \Lambda^0}^2 \approx 1$ beschrieben

$$f_{\text{CN (niedrig), } P_1^- \Lambda^0}(n_s) \approx 2.46 \times 10^{-5} n_s^{-2}, \quad (6.3)$$

$$f_{\text{CN}(\text{hoch}), P_1^- \Lambda^0}(n_s) \approx 2.46 \times 10^{-5} n_s^{-2}. \quad (6.4)$$

Tabelle 6.2 gibt einen Überblick über die korrespondierenden Konvergenzraten $R_{k-1/k}$ der berechneten Vernetzungspaare $k - 1/k$.

n_s	10	20	30	40	50	60	70	80	90	100
$P_1^- \Lambda^0$ -Element										
CN (niedrig)	2.00	2.00	2.00	2.00	1.99	2.00	1.99	2.00	1.99	
CN (hoch)	2.00	2.00	1.99	2.00	1.99	1.99	2.00	2.00	2.00	
$P_2^- \Lambda^0$ -Element										
CN (niedrig)	2.99	2.98	2.24	1.63	-3.64	13.62	-22.40	11.30	-4.98	
CN (hoch)	2.99	2.98	2.78	2.68	-4.42	5.82	-12.57	6.35	9.98	

Tab. 6.2: Konvergenzrate $R_{k-1/k}$ je Paar $k - 1/k$ der Vernetzungsgrade n_s für das CRANK-NICOLSON-Verfahren für $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elemente. Die Teilungsanzahl n_t des untersuchten Zeitraums wird für beide Präzisionsstufen bei $n_t = 10$ festgehalten.

Anzumerken ist, dass die Nachkommastellen der Konvergenzraten nach der zweiten Stelle abgeschnitten und nicht gerundet werden. Ab der dritten Nachkommastelle liegen für $R_{k-1/k} \approx 2$ Werte ungleich null vor.

6.2 Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt

Zur Minimierung der Diskretisierungsfehler wird die Diskretisierung der initialen Wirkstoffverteilung $c(r, t = 0 \text{ s})$ betrachtet. Dazu wird der relative Fehler err_{m_d} zwischen dem gegebenen Wert $m_d = 7.5 \times 10^{-3} \text{ g}$ und der numerisch berechneten Wirkstoffmasse m_d^{num} zum Startzeitpunkt $t = 0 \text{ s}$ bestimmt. Während für eine konstante Wirkstoffverteilung für $n_s = 10, 20, \dots, 100$ $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elemente lediglich relative Fehler err_{m_d} im Bereich der Maschinengenauigkeit für doppelte Genauigkeit von 10^{-14} beobachtet werden, ergeben sich für eine parabelförmige Wirkstoffverteilung bei der Verwendung von $P_1^- \Lambda^0$ -Elementen deutlich höhere relative Fehler, siehe Abbildung 6.5. Werden $P_2^- \Lambda^0$ -Elemente eingesetzt, reduzieren sich die relativen Fehler zu Fehlern im Bereich der Maschinengenauigkeit. Abbildung 6.6 illustriert beispielhaft den Diskretisierungsfehler zwischen analytischer und numerischer Darstellung einer initialen, parabelförmigen Wirkstoffverteilung $c(r, t = 0 \text{ s})$ unter Verwendung von $n_s = 10$ $P_1^- \Lambda^0$ -Elementen. Die Fläche zwischen den beiden Graphen stellt den absoluten Diskretisierungsfehler dar.

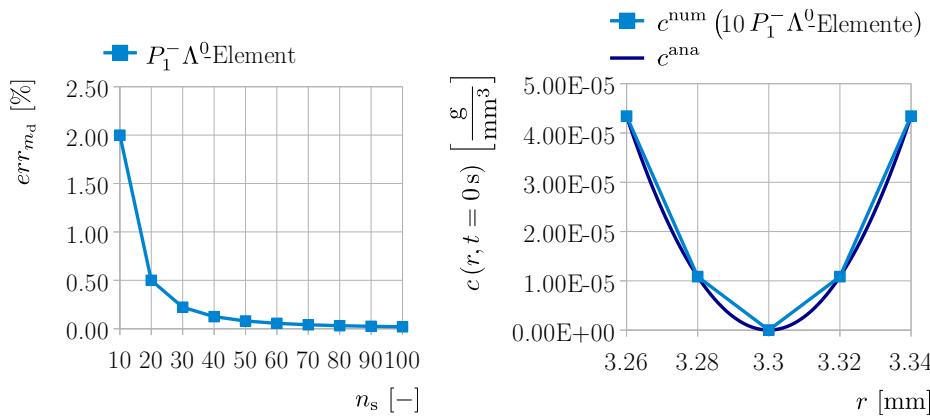


Abb. 6.5: Relativer Fehler err_{m_d} [%] zwischen gegebenem Wert m_d und numerisch berechneter Wirkstoffmasse m_d^{num} zum Startzeitpunkt $t = 0$ s für eine konstante Wirkstoffverteilung unter Verwendung von $P_1^- \Lambda^0$ -Elementen in Abhängigkeit von der Elementanzahl n_s

Abb. 6.6: Analytische und numerische Darstellung der initialen Wirkstoffverteilung $c(r, t = 0$ s); Zur Verdeutlichung des absoluten Diskretisierungsfehlers wird lediglich ein Teilbereich ($r = 3.26$ – 3.34 mm) des Hohlzylinderradius r dargestellt.

Um diesen besser zu verdeutlichen, wird nur ein Teilbereich ($r = 3.26$ – 3.34 mm) des Hohlzylinderradius r dargestellt.

6.3 Inverse Analyse

Für die Sättigungskonzentration c_* lässt sich davon ausgehen, dass sie der finalen Umgebungslösungskonzentration c_s des Experiments entspricht, siehe [Rickert *et al.* (2019)]. Somit gilt

$$c_* = F(t = 15 \text{ Wochen}) c_{\max} \approx 4.22 \times 10^{-7} \frac{\text{g}}{\text{mm}^3}. \quad (6.5)$$

Wird mit Hilfe des *trust region reflective*-Algorithmus die Methode der kleinsten Quadrate auf die Verlustfunktion nach Gleichung 5.51 angewendet, ergeben sich mit $c_* \approx 4.22 \times 10^{-7} \frac{\text{g}}{\text{mm}^3}$ für die konstante bzw. parabelförmige Startkonzentrationsverteilung $c(r, t = 0$ s) die in Tabelle 6.3 aufgelisteten Materialparameter Diffusionskoeffizient D , Stoffübergangskoeffizient α und Längenparameter l_b für die ersten fünf Wochen. Anzumerken ist, dass der bruchspezifische Degradationswert κ_b per händischer Optimierung für den Matrixbruch bestimmt wird. Daraufhin wird die Degradationskonstante ξ bei $\xi = 1$ festgehalten und die

Oberfläche der aufgebrochenen Matrix $A_{m,b}$, analog zur Optimierung während den ersten fünf Wochen, gewonnen. Die entsprechenden BestimmtheitsmaSSe $\mathfrak{R}_{0,10}^2$ für den gesamten Zeitraum sind in obengenannter Tabelle einzusehen. Dabei werden die BestimmtheitsmaSSwerte $\mathfrak{R}_{0,10}^2$ auf die vierte Nachkommastelle gerundet, die Materialparameter auf die zweite.

	$D \left[\frac{\text{mm}^2}{\text{s}} \right]$	$\alpha \left[\frac{\text{mm}^3}{\text{g}} \right]$	$l_b \text{ [mm]}$	$\xi [-]$	$\kappa_b [-]$	$\frac{A_b}{A_m^0} [-]$	$\mathfrak{R}_{0,10}^2 [-]$
konstant	2.6×10^{-7}	2.62×10^6	3.2	1	0.36	2.86	0.9899
parabelförmig	2.6×10^{-7}	3.02×10^6	3.2	1	0.5	4.51	0.9895

Tab. 6.3: Durch inverse Analyse bestimmte Materialparameter Diffusionskoeffizient D , Stoffübergangskoeffizient α und Längenparameter l_b mit $c_* \approx 4.22 \times 10^{-7} \frac{\text{g}}{\text{mm}^3}$ für konstante bzw. parabelförmige Startkonzentrationsverteilung $c(r, t = 0 \text{ s})$. Der bruchspezifische Degradationswert κ_b wird händisch optimiert, die Degradationskonstante ξ bei $\xi = 1$ festgehalten und die Oberfläche der aufgebrochenen Matrix $A_{m,b}$ im Rahmen einer zweiten inversen Analyse ermittelt. Für den gesamten Zeitraum wird das BestimmtheitsmaSS $\mathfrak{R}_{0,10}^2$ ermittelt.

Abbildung 6.7 zeigt für die oben genannten Materialparameter die simulierte

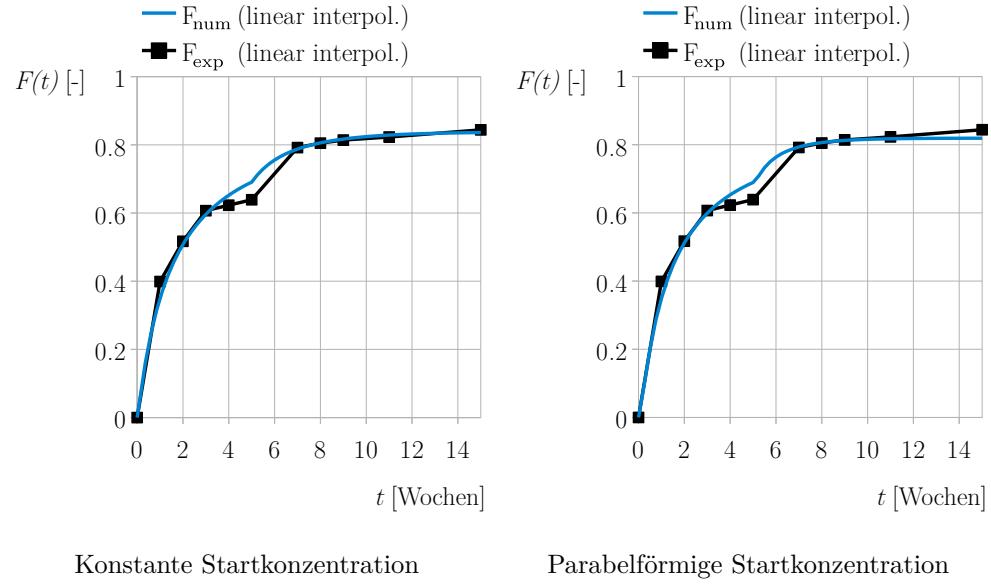


Abb. 6.7: Simulierte Freisetzungskurve $F_{\text{num}}(t)$ für die optimierten Materialparameter, siehe Tabelle 6.3, sowie experimentell bestimmte Freisetzungskurve $F_{\text{exp}}(t)$. Zwischen den experimentell bestimmten bzw. numerisch berechneten Werten wird linear interpoliert.

Freisetzungskurve $F_{\text{num}}(t)$ bei konstanter (links) bzw. parabelförmiger (rechts) Startkonzentrationsverteilung $c(r, t = 0 \text{ s})$ sowie die experimentell bestimmte Freisetzungskurve $F_{\text{exp}}(t)$. Zwischen den experimentell bestimmten und numerisch berechneten Werten wird linear interpoliert.

7 Diskussion

In diesem Kapitel werden die bei der zeitlichen und räumlichen Diskretisierung des physikalischen Modells entstehenden numerischen Fehler und die bestimmten Konvergenzraten diskutiert. Daraufhin werden die im Rahmen der inversen Analyse gewonnenen Ergebnisse für die Materialparameter samt Freisetzungskurve eingeordnet. Dies erfolgt für die konstante Startkonzentration sowie für die parabelförmige. Abschließend wird das Degradationsmodell und die Randbedingung beurteilt.

7.1 Konvergenzanalyse

7.1.1 Explizites EULER-Verfahren

Die zeitliche und räumliche Konvergenz des auf die homogene Diffusionsgleichung angewandte explizite EULER-Verfahren wird hier für $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elemente eingeordnet.

Zeitliche Konvergenz

Die bei der zeitlichen Diskretisierung des physikalischen Modells durch Verwendung des expliziten EULER-Verfahrens entstehenden Diskretisierungsfehler err_{n_t} für $P_1^- \Lambda^0$ -Elemente konvergieren bei steigender Teilungszahl n_t des untersuchten Zeitraums, siehe Abbildung 6.1 links. Die Fehler err_{n_t} sind jedoch durch die räumliche Diskretisierung dominiert, was am geringen Fehlerbereich von 1.56×10^{-9} bis 1.58×10^{-9} zu erkennen ist. Eine lineare Konvergenz bei doppelt logarithmischer Achsenauftragung liegt nicht vor. Das CFL-Kriterium nach Gleichung 5.9 ist also für die durch das explizite EULER-Verfahren diskretisierte homogene Diffusionsgleichung nicht aussreichend. Ein entsprechendes Kriterium müsste entwickelt werden, was den Umfang dieser Arbeit übersteigt. Aufgrund einer Vielzahl von untersuchten Parametersätzen, die alle konvergierende, aber räumliche dominierte, Diskretisierungsfehler zum Ergebnis haben, wird angenommen, dass das physikalische Modell für $P_1^- \Lambda^0$ -Elemente erfolgreich zeitlich diskretisiert ist.

Für $P_2^- \Lambda^0$ -Elemente liegt keine zeitliche Konvergenz der Fehler err_{n_t} vor, siehe Abbildung 6.1 rechts. Dies lässt sich durch das Erreichen der Maschinengenauigkeit der Diskretisierungsfehler err_n für das hier verwendete Gleitkomma-Zahlenformat mit doppelter Genauigkeit erklären. Eine Untersuchung mit einer geringeren Anzahl an $P_2^- \Lambda^0$ -Elementen würde das Erreichen der Maschinengenauigkeit verhindern. Auch hier wäre bei Nichterreichen der Maschinengenauigkeit die Verwendung eines spezifischen Stabilitätskriteriums von Vorteil. Darauf wird verzichtet, da die Fehler im Bereich der Maschinengenauigkeit als ausreichend gering befunden werden. Daher wird die erfolgreiche zeitliche Diskretisierung für $P_2^- \Lambda^0$ -Elemente als gegeben angenommen.

Räumliche Konvergenz

Die bei der räumlichen Diskretisierung entstehenden Fehler err_{n_s} durch Verwendung von $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elementen konvergieren bei steigender Elementanzahl n_s und doppelt logarithmischer Darstellung unter Nutzung des expliziten EULER-Verfahrens linear, siehe Abbildung 6.2. Die Exponenten der dort eingezeichneten Trendlinien korrespondieren mit den berechneten Konvergenzraten, siehe Gleichung 6.1 bzw. 6.2 und Tabelle 6.1. Bei erfolgreicher räumlicher Diskretisierung entspricht die Konvergenzrate $R_{k-1/k}$ bei der hier verwendeten L^2 -Norm dem um eins erhöhten Polynomgrad j des verwendeten $P_j^- \Lambda^0$ -Elements, also $R_{k-1/k} = j + 1$ [Logg et al. (2012)]. Für die hier verwendeten Polynomgrade $j = 1$ mit $R_{k-1/k} = 2$ bzw. $j = 2$ mit $R_{k-1/k} = 3$ ist dies der Fall. Die gering unter oder über dem zu erwartenden Wert liegenden Konvergenzraten lassen sich durch Rundungsfehler und „die ungenaue Lösung des linearen Gleichungssystems“ erklären, siehe [Langtangen et al. (2016a)]. Zusammenfassend lässt sich also sagen, dass das physikalische Modell unter Verwendung des expliziten EULER-Verfahrens mit beiden Elementtypen räumlich erfolgreich diskretisiert wird.

7.1.2 CRANK-NICOLSON-Verfahren

Die zeitlichen und räumlichen Konvergenzergebnisse des CRANK-NICOLSON-Verfahrens mit niedriger und hoher Präzision werden für $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elemente in den folgenden Unterkapiteln eingeordnet.

Zeitliche Konvergenz

Hier zeigen sich dieselben Phänomene, die bereits für die zeitliche Konvergenz des expliziten EULER-Verfahrens zu beobachten waren. Eine durch die

räumliche Diskretisierung dominierte Konvergenz für $P_1^- \Lambda^0$ -Elemente, siehe Abbildung 6.3 links, und eine Nichtkonvergenz für $P_2^- \Lambda^0$ -Elemente aufgrund des Erreichens der Maschinengenauigkeit der Diskretisierungsfehler err_{n_t} , siehe Abbildung 6.3 rechts. Ebenfalls wird aufgrund einer Vielzahl von untersuchten Parametersätzen, die alle konvergierende, aber räumliche dominierte, Diskretisierungsfehler zum Ergebnis haben, angenommen, dass das physikalische Modell für $P_1^- \Lambda^0$ -Elemente erfolgreich zeitlich diskretisiert ist. Auch hier werden Fehler im Bereich der Maschinengenauigkeit für $P_2^- \Lambda^0$ -Elemente als ausreichend gering befunden, weshalb die erfolgreiche zeitliche Diskretisierung als gegeben angesehen wird.

Räumliche Konvergenz

Wiederum zeigen sich Parallelen zu den räumlichen Konvergenzeigenschaften des expliziten EULER-Verfahrens, nämlich eine lineare Konvergenz bei doppelt-logarithmischer Achsenaufteilung für $P_1^- \Lambda^0$ -Elemente, siehe Abbildung 6.4 links, mit übereinstimmenden Konvergenzraten, konkret $R_{k-1/k} = j + 1 = 2$ in Tabelle 6.2, und Exponenten der Trendlinien nach Gleichung 6.3 und 6.4. Bezuglich der gering unter oder über dem zu erwartenden Wert liegenden Konvergenzraten siehe Unterkapitel 7.1.1. Somit ist die erfolgreiche räumliche Diskretisierung mit $P_1^- \Lambda^0$ -Elementen für das CRANK-NICOLSON-Verfahren bewiesen.

Für $P_2^- \Lambda^0$ -Elemente lässt sich für den Anteil mit niedriger Präzision von $n_s = 10 - 30$ und für den Anteil mit hoher Präzision von $n_s = 10 - 20$ lineare Konvergenz in doppelt-logarithmischer Achsenauftragung beobachten, siehe Abbildung 6.4 rechts. Ebenfalls stimmen für diese Elementanzahlbereiche die Konvergenzraten $R_{k-1/k} = 3$ in Tabelle 6.2 mit dem bei erfolgreicher räumlicher Diskretisierung zu erwartenden Wert von $R_{k-1/k} = j + 1 = 3$ nach [Logg *et al.* (2012)] überein. Bezuglich der gering unter oder über dem zu erwartenden Wert liegenden Konvergenzraten siehe ebenfalls Unterkapitel 7.1.1. Danach geht die lineare Konvergenz aufgrund des Erreichens der Maschinengenauigkeit der Diskretisierungsfehler err_{n_s} verloren. Bei der Verwendung kleinerer Elementanzahlen wird davon ausgegangen, dass für beide Präzisionen erneut lineare Konvergenz bei doppelt-logarithmischer Darstellung erreicht wird. Daher wird die räumliche Diskretisierung für $P_2^- \Lambda^0$ -Elemente bei der Verwendung des CRANK-NICOLSON-Verfahrens mit niedriger und hoher Präzision als erfolgreich eingestuft.

7.2 Diskretisierungsfehler der Wirkstoffverteilung zum Startzeitpunkt

Da die berechneten Konzentrationen „mit experimentell bestimmten Daten verglichen werden, ist ein Fehler von 10^{-3} akzeptabel“ [Rickert *et al.* (2019)]. Dies ist für eine durch $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elemente diskretisierte konstante Startwirkstoffverteilung der Fall. In diesen Fällen ist die Wirkstoffverteilung vorzugsweise durch $P_1^- \Lambda^0$ -Elemente zu diskretisieren, da diese ausreichend geringe relative Fehler err_{m_d} im Bereich von $10^{-14} \%$ garantieren. Zusätzlich ermöglichen $P_1^- \Lambda^0$ -Elemente eine geringere Berechnungsdauer als $P_2^- \Lambda^0$ -Elemente, da diese eine größere Anzahl an Elementknoten und damit auch Freiheitsgraden besitzen, siehe Abbildung 5.1. Für eine parabelförmige Startwirkstoffverteilung müssen wiederum $P_2^- \Lambda^0$ -Elemente eingesetzt werden, um einen ausreichend geringen Fehler zu garantieren. Denn für $P_1^- \Lambda^0$ -Elemente liegen die Diskretisierungsfehler über der geforderten Grenze von 10^{-3} , siehe Abbildung 6.5. Für $P_1^- \Lambda^0$ -Elemente bei konstanter Startwirkstoffverteilung und für $P_2^- \Lambda^0$ -Elemente bei parabelförmiger Startwirkstoffverteilung liegen für $n_s = 10$ Elemente ausreichend geringe Fehler vor. Aus diesem Grund werden lediglich $n_s = 10$ Elemente verwendet, um die Rechendauer möglichst gering zu halten.

7.3 Wahl der Diskretisierungparameter

Aus der Konvergenzanalyse wird ersichtlich, dass die Diskretisierungsfehler für das explizite EULER- sowie für das CRANK-NICOLSON-Verfahren für $P_1^- \Lambda^0$ - und $P_2^- \Lambda^0$ -Elemente bei allen untersuchten Elementanzahlen n_s und Zeitschrittanzahlen n_t unter 10^{-3} liegen. Daher wäre die Implementierung der Diskretisierungsverfahren unter dem Aspekt einer Rechendauerminimierung mit den niedrigsten untersuchten Parametern, sprich $n_s = 10$ $P_1^- \Lambda^0$ -Elemente zur räumlichen Diskretisierung und $n_t = 100$ Zeitschritte zur zeitlichen Diskretisierung mit Hilfe des expliziten EULER- und $n_t = 10$ Zeitschritte mit Hilfe des CRANK-NICOLSON-Verfahrens, durchzuführen. Aufgrund der Ergebnisse aus Kapitel 6.2 müssten jedoch für Berechnungen mit einer parabelförmigen Startkonzentration $n_s = 10$ $P_2^- \Lambda^0$ -Elemente verwendet werden, um den Diskretisierungsfehler unter 10^{-3} und gleichzeitig die Berechnungsdauer durch die Wahl des niedrigsten untersuchten Zeitschrittparameters möglichst gering zu halten. Da jedoch anhand des BestimmtheitsmaSS $\mathfrak{R}_{0,10}^2$ untersucht werden soll, ob die Freisetzungskurve $F(t)$ besser durch eine konstante oder eine parabelförmige Startkonzentrationsverteilung zu prognostizieren ist, werden auch für Untersuchungen mit einer konstanten Startkonzentrationsverteilung

$n_s = 10 P_2^- \Lambda^0$ -Elemente zur räumlichen Diskretisierung verwendet, um die Ergebnisse für das BestimmtheitsmaSS $\mathfrak{R}_{0,10}^2$ nicht durch eine abweichende räumliche Diskretisierung zu verfälschen. Dementsprechend ergibt sich die in Tabelle 7.1 gezeigte Diskretisierungsparameterwahl. Zu beachten ist jedoch, dass durch Implementierung eines adaptiven CRANK-NICOLSON-Verfahrens, welches die Zeitschrittänge und somit die Zeitschrittzahl n_t anhand der RICHARDSON-Extrapolationsbedingung nach Gleichung 5.12 bzw. 5.13 bestimmt, real $n_{t,\text{low}} = 64$ bei niedriger bzw. $n_{t,\text{high}} = 2 n_{t,\text{low}} = 128$ bei hoher Präzision für die konstante und $n_{t,\text{low}} = 71$ bei niedriger bzw. $n_{t,\text{high}} = 142$

	t_{start} [s]	t_{end} [s]	n_t [-]	Elementtyp	n_s [-]
expliziter EULER	0	4.23×10^{-2}	100	$P_2^- \Lambda^0$	10
CRANK-NICOLSON (niedrig)	4.23×10^{-2}	15	10	$P_2^- \Lambda^0$	10
CRANK-NICOLSON (hoch)	4.23×10^{-2}	15	20	$P_2^- \Lambda^0$	10

Tab. 7.1: Diskretisierungsparameter der verwendeten Verfahren

Zeitschritte bei hoher Präzision für die parabelförmige Startkonzentrationsverteilung verwendet werden. Die Abweichung zur berechneten Zeitschrittzahl n_t ist durch eine im Programm implementierte Zeitschrittlimitierung von $\Delta t \leq 0.25$ Wochen begründet. Auf diese Weise wird eine stetige Darstellung der numerisch bestimmten Freisetzungskurven F_{num} ermöglicht.

7.4 Inverse Analyse

Wie das berechnete BestimmtheitsmaSS von $\mathfrak{R}_{0,10}^2 = 0.9899$ für eine konstante und ein BestimmtheitsmaSS von $\mathfrak{R}_{0,10}^2 = 0.9895$ für eine parabelförmige Startkonzentrationsverteilung zeigt, prädiert das hier vorgestellte Modell die diffusionsbedingte Freisetzung von GM aus einer nanoporösen PLA-Matrix in eine körperflüssigkeitenähnliche Umgebungslösung gut. Lediglich für den Bereich der stagnierenden Freisetzung während der vierten und fünften Woche (Stufe III) sagt das Modell eine zu hohe Wirkstofffreisetzung voraus, was sich für diesen Zeitraum durch ein BestimmtheitsmaSS von $\mathfrak{R}_{3,5}^2 = -6.1014$ für eine konstante und ein BestimmtheitsmaSS von $\mathfrak{R}_{3,5}^2 = -5.8412$ für eine parabelförmige Startkonzentrationsverteilung äußert. Ein negatives BestimmtheitsmaSS \mathfrak{R}^2 bedeutet, dass das hier vorgestellte Modell die Stagnation schlechter als der Mittelwert der experimentell bestimmten Freisetzungswerte in diesem Zeitraum prognostiziert.

Die in Unterkapitel 3.2.2 aufgestellte Hypothese, dass eine parabelförmige

Wirkstoffverteilung die stoSSartige Wirkstofffreisetzung während der ersten Woche durch die hohe Wirkstoffkonzentration im randnahen Bereich besser prädikiert, muss jedoch widerlegt werden, da das BestimmtheitsmaSS von $\mathfrak{R}_{0,1}^2 = 0.9683$ für die parabelförmige Startwirkstoffverteilung sich nicht maSSgeblich vom BestimmtheitsmaSS von $\mathfrak{R}_{0,1}^2 = 0.9678$ für die konstante Startwirkstoffverteilung unterscheidet. Auch für den gesamten Zeitraum lässt sich keine maSSgeblich verbesserte Vorhersage des Experiments feststellen, siehe die oben genannten BestimmtheitsmaSSwerte $\mathfrak{R}_{0,10}^2$. Aus diesem Grund wird für weitere Berechnungen empfohlen, die bereits von [Macha *et al.* (2019)] und [Rickert *et al.* (2019)] getroffene Annahme einer konstanten Startwirkstoffverteilung zu wählen. Darüber hinaus lässt sich von der geringeren Berechnungsdauer profitieren, da hier der Einsatz von $P_1^- \Lambda^0$ -Elementen ausreichend geringe Diskretisierungsfehler ermöglicht.

Für das hier untersuchte Experiment wurden bereits Modelle zur Prädiktion der Wirkstofffreisetzung entwickelt, siehe [Macha *et al.* (2019)], [Rickert *et al.* (2019)] und [Morozova *et al.* (2021)].

	Diffusionskoeffizient D [$\frac{\text{mm}^2}{\text{s}}$]
[Macha <i>et al.</i> (2019)]	2.33×10^{-9}
[Rickert <i>et al.</i> (2019)]	1.4×10^{-9}
[Morozova <i>et al.</i> (2021)]	$2.4-5 \times 10^{-9}$
diese Arbeit	2.6×10^{-7}

Tab. 7.2: Ermittelte Diffusionskoeffizienten D verschiedener Autoren für das hier betrachtete Experiment

Tabelle 7.2 zeigt eine Übersicht über die dabei bestimmten Diffusionskoeffizienten für eine konstante Startkonzentrationsverteilung des Wirkstoffs. Anzumerken ist der um etwa zwei Größenordnungen kleinere Diffusionskoeffizient D von

[Macha *et al.* (2019)], [Rickert *et al.* (2019)] und [Morozova *et al.* (2021)]. Der Diffusionskoeffizient D ist jedoch von vielen Faktoren abhängig. Für den hier betrachteten Fall ist die Diffusion durch eine nanoporöse, degradierende Matrix zu nennen, welche sich komplexer als einfache Diffusion darstellt. Das eigens für diese Arbeit entwickelte Degradationsmodell sowie die Erweiterung der von [Macha *et al.* (2019)] und [Rickert *et al.* (2019)] vorgestellten ROBIN-Randbedingung wirkt sich auf die simulierte Wirkstofffreisetzung und somit auf das Diffusionsverhalten aus. Daraus ergibt sich ein für jedes Modell spezifischer Diffusionskoeffizient D . Zudem prädiktieren die genannten Autoren voneinander abweichende Wirkstofffreisetzungskurven $F_{\text{num}}(t)$.

7.5 Degradationsmodell

Auffällig ist, dass die Matrix bei einer konstanten Wirkstoffverteilung bereits nach 35.6 prozentiger Matrixdegradation aufbricht und sich dabei die Matrixoberfläche um das 2.86-fache vergrößert, aber bei einer parabelförmigen erst nach 50.3 % und die Oberfläche sich um das 4.51-fache vergrößert. Ob eines dieser Szenarien im Experiment so erfolgt ist, kann aufgrund nicht vorhandene REM-Aufnahmen nicht bestätigt werden. Jedoch berücksichtigt das in Unterkapitel 4.2 vorgestellte Degradationsmodell keine zugrunde liegende Degradationsphänomene, um den Zerfallsprozess zu prädiktieren. Im Rahmen weiterer Arbeiten würden REM-Aufnahmen, parallel zu den Konzentrationsmessungen der Umgebungslösung, helfen den Degradationsprozess der Matrix besser zu verstehen. Im Rahmen der Arbeit wurden Degradationsmodelle ausgehend von [Noyes *et al.* (1897)], [Nernst (1904)], [Brunner (1904)] und [Cabrerá *et al.* (2006)] entwickelt, die die zunehmende Hohlraumgröße der nanoporösen PLA-Matrix bzw. die abnehmende Wirkstoffpartikeloberfläche berücksichtigen. Hierdurch nimmt die Tragfähigkeit der Matrix ab, was schlussendlich zum Matrixbruch führt. Dazu müsste die als Skalarfeld dargestellte Oberfläche der Hohlräume bzw. die der Wirkstoffpartikel in die Kontinuumsumgebung überführt werden. Da dies den Rahmen dieser Arbeit überschritten hätte, wird anstelle dessen das hier vorgestellte Degradationsmodell, trotz seiner Unzulänglichkeiten, verwendet. Weitere Literaturrecherchen offenbarten den vielversprechenden Ansatz, die Degradation der PLA-Matrix mit Hilfe eines Hydrolyseprozesses zu modellieren, siehe Unterkapitel 4.2. Auf diese Weise würden zugrunde liegenden Prozesse bei der Modellierung berücksichtigt werden.

7.6 Randbedingung

Wie in Abschnitt 7.4 bereits erwähnt, wird die Wirkstofffreisetzung während der vierten und fünften Woche zu hoch vorausgesagt. Dieser Zeitabschnitt wird maßgebend durch die Randbedingung nach Formel 4.9 geprägt. Dabei wird eine Sättigungslösung c_* berücksichtigt, obwohl das Experiment unter SINK-Bedingungen durchgeführt wurde, welche eine Sättigung der Umgebungslösung per Definition ausschließen. Hier besteht bei der Anpassung bzw. Wahl der Randbedingung Handlungsbedarf für zukünftige Arbeiten.

8 Zusammenfassung

Im Rahmen der Arbeit wird ein computergestütztes Modell (*in silico*) präsentiert, dass das entsprechende Reagenzglas-Experiment (*in vitro*) mit einem BestimmtheitsmaSS von $R_{0,10}^2 \approx 0.9899$ prädictiert. Dazu wird auf Basis von [Macha *et al.* (2019)] ein Modell präsentiert, dass die zugrunde liegenden physikalischen Prozesse der Wirkstofffreisetzung und Matrixdegradation beschreibt und erklärt. Das vierstufige Modell von [Macha *et al.* (2019)] wird um eine weitere Stufe - den Matrixbruch - erweitert. So lässt sich die Wirkstofffreisetzung über die 15-wöchige Versuchsdauer hinweg durch folgende Stufen beschreiben:

- I. StoSSartige Freisetzung (1. Woche)
- II. Geminderte Freisetzung (2.–3. Woche)
- III. Stagnation (4.–5. Woche)
- IV. Matrixbruch (6.–7. Woche)
- V. Restfreisetzung (8.–15. Woche)

Zunächst wird der Einfluss der Wirkstoffpartikel, die sich zu Beginn auf der Matrixoberfläche befinden, auf die stoSSartige Freisetzung während der ersten Woche untersucht und als vernachlässigbar eingestuft. Ebenfalls wird der Einfluss einer konstanten bzw. parabelförmigen Wirkstoffverteilung über den Matrixquerschnitt betrachtet. Dass eine parabelförmige Wirkstoffverteilung die stoSSartige Wirkstofffreisetzung während der ersten Woche durch die hohe Wirkstoffkonzentration im randnahen Bereich besser prädictiert, bestätigt sich nicht. Die bereits in der Literatur vorzufindende Annahme einer konstanten Wirkstoffverteilung wird für weitere Untersuchungen empfohlen. Darüber hinaus wird die von [Macha *et al.* (2019)] vorgestellte ROBIN-Randbedingung um einen weiteren Fall ergänzt. Neben dem Sättigungsfall, der die Modellierung der geminderten und darauffolgenden stagnierenden Wirkstofffreisetzung ermöglicht, wird die Randbedingung um den Fall des Matrixbruchs ergänzt. Dabei wird zum Zeitpunkt des Matrixbruchs der Materialtransferkoeffizient der ROBIN-Randbedingung schlagartig erhöht. Dazu wird ein Matrixdegradationsmodell entworfen, dass den Zerfall der Matrix in Abhängigkeit von der Wirkstoffkonzentration innerhalb der Matrix modelliert. Je mehr Wirkstoff aus der Matrix herausdiffunziert, um so höher ist die Matrixdegradation. Das Modell prognostiziert den Zeitpunkt der stark ansteigenden Wirkstofffreisetzung erfolgreich, jedoch wird die vorherige Stagnation zwischen der vierten

und fünften unterprädiktiert.

Die oben genannten Abläufe werden zu einem Gesamtmodell zusammengefasst und mit Hilfe der Finiten-Elemente-Methode durch das Programm FEniCS Legacy berechnet. Dazu wird das Modell, basierend auf der allgemeinen Diffusionsgleichung, in die schwache Form überführt und zeitlich sowie räumlich diskretisiert. Die zeitliche Diskretisierung erfolgt mit Hilfe eines adaptiven CRANK-NICOLSON-Verfahrens. Für die ersten Zeitschritte wird zusätzlich die explizite EULER-Methode implementiert, um Schwingungen der Lösung aufgrund der Unstetigkeit zwischen Randbedingung und Startbedingung zu vermeiden. Die Symmetrien der zylinderförmigen Matrix werden genutzt, wodurch die räumliche Disrektisierung durch eindimensionale Linienelemente über den Matrixradius verwirklicht werden kann. Daraufhin wird die zeitliche und räumliche Diskretisierung mit Hilfe der *Method of Manufactured Solutions*, dem diesbezüglichen Goldstandard, überprüft.

AbschlieSSend wird eine inverse Analyse durchgeführt, womit sich der Diffusionskoeffizient des Wirkstoffs, der Stoffübergangskoeffizient, die kritische Matrixdegradation, die zum Bruch führt, eine für die Degradation spezifische Konstante und die durch den Bruch erhöhte Matrixoberfläche bestimmen lässt. Durch Implementierung dieser Parameter in das hier vorgestellte Modell, lässt sich die Wirkstofffreisetzungskurve des 15-wöchigen Experiments innerhalb von ca. 37 Minuten berechnen. So lassen sich eine Vielzahl zusätzlicher Matrixdesigns und Wirkstoffmassen berechnen ohne die Zeit und Personalkosten der jeweiligen Experimente in Kauf nehmen zu müssen. Auch ist es denkbar, technisch noch nicht umsetzbare Matrixdesigns zu simulieren und so deren Vorteilhaftigkeit offenzulegen. Auf diese Weise kann die präklinische Phase, in der solche in vitro-Experimente klassischerweise durchgeführt werden, zeitlich verkürzt, kostentechnisch optimiert und die Zulassungschancen des neuen Medikamentes erhöht werden. Die Dringlichkeit dieser Aspekte wurde der Gesellschaft nicht zuletzt durch den Coronavirus SARS-CoV-2 vor Augen geführt. Ansatzpunkte für weiterführende Arbeiten sind die zu hoch vorausgesagte Wirkstofffreisetzung während der Stagnationsphase und eine Überarbeitung des Degradationsmodells, da das hier vorgestellte keine tieferliegenden Degradationsphänomene berücksichtigt. Im Rahmen dieser Arbeit wurden zwar Degradationsmodelle, die die Oberfläche der Wirkstoffpartikel bzw. die der Hohlräume innerhalb der nanoporösen Matrix miteinbeziehen, entwickelt, jedoch aufgrund der Komplexität, die aus der entsprechenden Überführung in die Kontinuumsumgebung resultiert, auf deren Implementierung verzichtet. Darüber hinaus ist eine durch Hydrolyse modellierte Matrixdegradation ein vielversprechender Ansatzpunkt zur Entwicklung eines Modells, das die tieferliegenden Degradationsprozesse berücksichtigt. Bezuglich der überprognostizierten Wirkstofffreisetzung während der Stagnationsphase ist ein an die

Degradation gekoppelter und somit ortsabhängiger Diffusionskoeffizient und eine überarbeitete Randbedingung angedacht.

9 Ausblick

Zur Weiterentwicklung des hier vorgestellten Modells sieht der Autor zwei Ansatzpunkte, nämlich die Behebung der überprognostizierten Medikamentenfreisetzung während der Stagnationsphase und eine Implementierung eines Matrixdegradationsmodells, das die zugrunde liegenden Degradationsprozesse berücksichtigt.

Der Zerfall einer PLA-Matrix ist prinzipiell durch hydrolytische, photochemische, mikrobielle und enzymatische Prozesse möglich. Aufgrund des unter Laborbedingungen und im Innenraum durchgeführten Experiments können photochemische, mikrobielle und enzymatische Einflüsse ausgeschlossen werden. Somit sollte das Augenmerk der Recherche zunächst auf die durch Hydrolyse bedingte Degradation der PLA-Matrix gerichtet werden. Die Forschungslage ist diesbezüglich vielversprechend, alleine im Jahr 2018 wurden mehr als 1 200 Artikel zur Degradation von PLA veröffentlicht, siehe [Zaaba *et al.* (2020)]. Nach Möglichkeit sollte ein von Raum und Zeit abhängiges Degradationsmodell identifiziert und in das in dieser Arbeit vorgestellte Gesamtmodell implementiert werden. Darüber hinaus empfiehlt sich eine Koppelung an einen orts- und zeitabhängigen Diffusionskoeffizienten, da eine zunehmende Matrixdegradation zur Folge hat, dass der gelöste Wirkstoff aufgrund abnehmender Hindernisse schneller durch die Matrix diffundieren kann. Der Diffusionskoeffizient ist an diesen Stellen also größer. Die fortwährende Degradation wird schlussendlich einen Matrixbruch bedingen. Hier wird empfohlen den Matrixbruch, nicht wie bisher an die Randbedingung in Form eines sprunghaft ansteigenden Materialübergangskoeffizienten, sondern an einen sprunghaften Anstieg des Diffusionskoeffizienten zu koppeln.

Um die Komplexität der hier eingesetzten Randbedingung zu reduzieren und gleichzeitig die Zuverlässigkeit dieser zu erhöhen, wird empfohlen, die als konvektive ROBIN-Randbedingung bekannte Randbedingung in zukünftigen Modellen zu implementieren, da sich diese in den Ingenieurwissenschaften bereits bewährt hat. Lediglich die in der Literatur als Konzentration weit weg jeglicher Turbulenzen bekannte Konzentration sollte durch die Konzentration der Umgebungslösung ersetzt werden, um durch diese dämpfende Komponente der überprognostizierten Wirkstofffreisetzung während der Stagnationsphase entgegenzuwirken.

Abbildungsverzeichnis

1.1	Entwicklungsschritte eines neuen Medikaments und deren Dauer, siehe [Tamimi <i>et al.</i> (2009)] und [Paul <i>et al.</i> (2010)]: Identifikation einer nicht oder zumindest nicht zufriedenstellend behandelbaren Krankheit, Ermittlung der zugrunde liegenden Wirkmechanismen, Bestimmung des Angriffspunkts und Auswahl der Wirkstoffkandidaten, die erfolgreich am Angriffspunkt binden können im Rahmen der Forschung. Untersuchung der Wirkstoffkandidaten im Reagenzglas (in vitro) und an Tieren (in vivo) auf Wirksamkeit, Verträglichkeit und Wechselwirkungen während der Präklinik. Erprobung des Medikaments an 60–80 gesunden Probanden (Klinische Phase I), an 100–500 erkrankten (Klinische Phase II) und an tausenden erkrankten (Klinische Phase III); Grafische Aufbereitung [Herzog (2022)]	1
1.2	Positronen-Emissions-Tomographie-Ganzkörperaufnahme: Verteilung des Wirkstoffs (blau) über die Blutbahn nach Injektion in den Arm sowie darauffolgende Ausscheidung über Niere und Blase für fortschreitende Zeitpunkte, siehe [Bayer HealthCare <i>et al.</i> (o. D.)].	2
2.1	Versuchsaufbau: Mit 10 % GM-Massenanteil versehene, biokompatible, biologisch abbaubare, nanoporöse, hohlzylinderförmige ($r_o = 3.4 \text{ mm}$, $r_i = 3.2 \text{ mm}$, $H = 20 \text{ mm}$) PLA-Matrix Ω_m in Umgebungslösung Ω_s ($V_s = 1.5 \times 10^4 \text{ mm}^3$, $pH = 7.4$, $T = 37 \pm 0.1^\circ\text{C}$) auf Magnetrührer bei $n = 100 \frac{1}{\text{min}}$ Umdrehungen. [Macha <i>et al.</i> (2019)], [MicrobeNotes (o. D.)]	5
2.2	Fünfstufiger GM-Freisetzungsverlauf $F(t)$ [–] über 15 Wochen. Blaue Rechtecke beschreiben die gemessenen Konzentrationswerte. Dazwischen wurde linear interpoliert. I. Stufe: stoSSartige Freisetzung; II. Stufe: starke, aber leicht reduzierte Freisetzung; III. Stufe: Stagnation; IV. Stufe: Freisetzungssteigerung durch Matrixbruch; V. Stufe: geringe Restfreisetzung	7
3.1	Fünfstufiges Matrixdegradations-/Wirkstofffreisetzungsmodell basierend auf dem vierstufigen Modell nach [Macha <i>et al.</i> (2019)]	9

3.2 REM-Aufnahme der Oberfläche einer mit GM besetzten PLA-Matrix. Die helleren Bereiche sind GM-Partikel. [Macha <i>et al.</i> (2019)]	11
3.3 Nebenstehende REM-Aufnahme der Matrixoberfläche wird mit Hilfe eines GNU-Octave-Programmes in ein Blau-WeiSS-Bild umgewandelt. Die blauen Bereiche sind GM-Partikel.	11
3.4 Konstante Wirkstoffverteilung $c(r, t = 0 \text{ s}) = c_0 \left[\frac{\text{g}}{\text{mm}^3} \right]$ über den Matrixquerschnitt zum Startzeitpunkt $t = 0 \text{ s}$ nach [Macha <i>et al.</i> (2019)] und [Rickert <i>et al.</i> (2019)].	14
3.5 Parabelförmige Wirkstoffverteilung $c(r, t = 0 \text{ s}) \left[\frac{\text{g}}{\text{mm}^3} \right]$ über den Matrixquerschnitt zum Startzeitpunkt $t = 0 \text{ s}$. $c_{0,\min} \left[\frac{\text{g}}{\text{mm}^3} \right]$ und $c_{0,\max} \left[\frac{\text{g}}{\text{mm}^3} \right]$ beschreiben die minimale bzw. maximale Konzentration im Matrixquerschnitt.	14
4.1 Degradationsverlauf $\kappa(\mathbf{x}, t)$ der Matrix ausgehend von der konstanten Anfangskonzentration $c(\mathbf{x}, t = 0) = c_0 \approx 9.04 \times 10^{-5} \frac{\text{g}}{\text{mm}^3}$ für die Degradationskonstante ξ von $\xi = 1, \dots, 4$. . .	19
5.1 Eindimensionales $P_j^- \Lambda^0$ -Element mit Grad $j = 1$ (links) bzw. $j = 2$ (rechts)	35
6.1 Durch zeitliche Diskretisierung entstandener numerischer Fehler err_{n_t} des expliziten EULER-Verfahrens unter Verwendung der MMS bei Änderung der Teilungsanzahl n_t des untersuchten Zeitraums. Hierbei wird die Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts) bei 100 gehalten.	43
6.2 Durch räumliche Diskretisierung entstandener numerischer Fehler err_{n_s} des expliziten EULER-Verfahrens unter Verwendung der MMS bei Änderung der Anzahl n_s der $P_1^- \Lambda^0$ -Elemente (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts). Die Teilungsanzahl n_t des untersuchten Zeitraums wird bei 100 festgehalten. Zusätzlich wird eine Trendlinie durch die Fehlerwerte angenähert.	44
6.3 Durch zeitliche Diskretisierung entstandener numerischer Fehler err_{n_t} des CRANK-NICOLSON-Verfahrens mit niedriger und hoher Präzision unter Verwendung der MMS bei Änderung der Teilungsanzahl n_t des untersuchten Zeitraums. Hierbei wird die Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts) bei 100 gehalten.	45

6.4 Durch räumliche Diskretisierung entstandener numerischer Fehler err_{n_s} des CRANK-NICOLSON-Verfahrens mit niedriger (low) und hoher (high) Präzision unter Verwendung der MMS bei Änderung der Anzahl n_s der $P_1^- \Lambda^0$ - (links) bzw. $P_2^- \Lambda^0$ -Elemente (rechts). Die Teilungsanzahl n_t des untersuchten Zeitraums wird bei 10 gehalten. Zusätzlich wird für $P_1^- \Lambda^0$ -Elemente eine Trendlinie durch die Fehlerwerte angenähert.	46
6.5 Relativer Fehler err_{m_d} [%] zwischen gegebenem Wert m_d und numerisch berechneter Wirkstoffmasse m_d^{num} zum Startzeitpunkt $t = 0$ s für eine konstante Wirkstoffverteilung unter Verwendung von $P_1^- \Lambda^0$ -Elementen in Abhängigkeit von der Elementanzahl n_s	48
6.6 Analytische und numerische Darstellung der initialen Wirkstoffverteilung $c(r, t = 0$ s); Zur Verdeutlichung des absoluten Diskretisierungfehlers wird lediglich ein Teilbereich ($r = 3.26\text{--}3.34$ mm) des Hohlzylindradius r dargestellt.	48
6.7 Simulierte Freisetzungskurve $F_{\text{num}}(t)$ für die optimierten Materialparameter, siehe Tabelle 6.3, sowie experimentell bestimmte Freisetzungskurve $F_{\text{exp}}(t)$. Zwischen den experimentell bestimmten bzw. numerisch berechneten Werten wird linear interpoliert.	49
B.1 Programmablaufplan der Implementierung des räumlich und zeitlich diskretisierten physikalischen Modells	v
B.2 Programmablaufplan des Python-Quelltexts B	vii

Tabellenverzeichnis

2.1 Experimentell bestimmte Datenpunkte $F(t_i)$ [–] der GM-Freisetzung. Dazu wird die Konzentration $c_s \left[\frac{\text{g}}{\text{mm}^3} \right]$ von GM in der Umgebungslösung an den Wochen i gemessen und durch die maximal mögliche GM-Konzentration $c_{\max} \left[\frac{\text{g}}{\text{mm}^3} \right]$ in eben dieser geteilt. [Macha <i>et al.</i> (2019)]	7
6.1 Konvergenzrate $R_{k-1/k}$ je Paar $k - 1/k$ der Vernetzungsgrade n_s für das explizite EULER-Verfahren für $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elemente. Dabei wird die Teilungszahl des Zeitraums n_t bei $n_t = 100$ gehalten.	45
6.2 Konvergenzrate $R_{k-1/k}$ je Paar $k - 1/k$ der Vernetzungsgrade n_s für das CRANK-NICOLSON-Verfahren für $P_1^- \Lambda^0$ - bzw. $P_2^- \Lambda^0$ -Elemente. Die Teilungszahl n_t des untersuchten Zeitraums wird für beide Präzisionsstufen bei $n_t = 10$ festgehalten.	47
6.3 Durch inverse Analyse bestimmte Materialparameter Diffusionskoeffizient D , Stoffübergangskoeffizient α und Längenparameter l_b mit $c_* \approx 4.22 \times 10^{-7} \frac{\text{g}}{\text{mm}^3}$ für konstante bzw. parabelförmige Startkonzentrationsverteilung $c(r, t = 0\text{s})$. Der bruchspezifische Degradationswert κ_b wird händisch optimiert, die Degradationskonstante ξ bei $\xi = 1$ festgehalten und die Oberfläche der aufgebrochenen Matrix $A_{m,b}$ im Rahmen einer zweiten inversen Analyse ermittelt. Für den gesamten Zeitraum wird das BestimmtheitsmaSS $\mathfrak{R}_{0,10}^2$ ermittelt.	49
7.1 Diskretisierungsparameter der verwendeten Verfahren	55
7.2 Ermittelte Diffusionskoeffizienten D verschiedener Autoren für das hier betrachtete Experiment	56

Algorithmen und Quelltexte

5.1 Adaptives CRANK-NICOLSON-Verfahren	29
A.1 GNU-Octave-Programm zur Bestimmung der Medikamentenmasse auf der Matrixoberfläche	i
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/parameters_units.py	viii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/parameters.py	x
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/main.py	xiv
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/menu.py	xvi
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/model.py	xxvii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/mesh_domains_boundaries.py	xxxv
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/boundary_conditions.py	xxxvii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/u_0.py	xxxviii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/source_term.py	xl
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/surrounding_solution.py	xli
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/degradation.py	xliv
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/solver_Euler.py	xlv
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/solver_CN_low.py	xlvii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/solver_CN_high.py	xlix
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/utility.py	li
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/optimization.py	lviii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/convergence_analysis.py	lxii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/testing.py	lxvii
/home/jan/Dropbox/BACHELORARBEIT/01_programmierung/0000_FULLBUILD/20250224_FULLBUILD_thesisrelease/save.py	lxx

/home/jan/Dropbox/BACHELORARBEIT/01Programmierung/0000_{FULLBUILD/20250224}_{FULLBUILD}_{thesisrelease/plot_settings_inits.py} lxxx
/home/jan/Dropbox/BACHELORARBEIT/01Programmierung/0000_{FULLBUILD/20250224}_{FULLBUILD}_{thesisrelease/plot_funcs.py} lxxxvi

Literatur

- Alnæs, Martin, Blechta, Jan, Hake, Johan, Johansson, August, Kehlet, Benjamin, Logg, Anders, Richardson, Chris, Ring, Johannes, Rognes, Marie E. und Wells, Garth N. (2015). "The FEniCS Project Version 1.5". en. In: *Archive of Numerical Software* Vol 3, Starting Point and Frequency: Year: 2013. doi: [10.11588/ANS.2015.100.20553](https://doi.org/10.11588/ANS.2015.100.20553).
- Arnold, D. N. und Logg, A. (Nov. 2014). "Periodic Table of the Finite Elements". In: *SIAM News* 47.9.
- Bayer HealthCare und University Medical Center Groningen (o. D.). *Positronen-Emissions-Tomographie-Ganzkörperaufnahme: Verteilung eines Wirkstoffes über die Blutbahn*.
- Blechta, J., Herzog, R., Hron, J. und Wachsmuth, G. (2018). *FEniCS hands-on*. URL: <https://fenics-handson.readthedocs.io/en/latest/heat/doc.html#adaptive-time-stepping> (besucht am 22.01.2025).
- Brunner, Erich (Jan. 1904). "Reaktionsgeschwindigkeit in heterogenen Systemen". In: *Zeitschrift für Physikalische Chemie* 47U.1, S. 56–102. ISSN: 0942-9352. doi: [10.1515/zpch-1904-4705](https://doi.org/10.1515/zpch-1904-4705).
- Cabrera, María I., Luna, Julio A. und Grau, Ricardo J.A. (Sep. 2006). "Modeling of dissolution-diffusion controlled drug release from planar polymeric systems with finite dissolution rate and arbitrary drug loading". In: *Journal of Membrane Science* 280.12, S. 693–704. ISSN: 0376-7388. doi: [10.1016/j.memsci.2006.02.025](https://doi.org/10.1016/j.memsci.2006.02.025).
- ChemSpider (o. D.). *Properties of GentaMicin (C₂₁H₄₃N₅O₇)*. URL: https://www.chemspider.com/Chemical-Structure.78430428.html?rid=2db4322c-25bb-4ebe-9d7b-e0ea1019ff26&page_num=0 (besucht am 06.01.2025).
- Courant, R., Friedrichs, K. und Lewy, H. (1928). "Über die partiellen Differenzengleichungen der mathematischen Physik". In: *Mathematische Annalen* 100.1, S. 32–74. doi: [10.1007/bf01448839](https://doi.org/10.1007/bf01448839).

- Crank, J. und Nicolson, P. (1947). "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 43.1, S. 50–67. DOI: [10.1017/s0305004100023197](https://doi.org/10.1017/s0305004100023197).
- Eaton, John W., Bateman, David, Hauberg, Søren und Wehbring, Rik (2023). *GNU Octave version 8.3.0 manual: a high-level interactive language for numerical computations*. URL: <https://octave.org/doc/v8.3.0/>.
- Fick, Adolf (1855). "Ueber Diffusion". In: *Annalen der Physik und Chemie* 170.1, S. 59–86. DOI: [10.1002/andp.18551700105](https://doi.org/10.1002/andp.18551700105).
- Gorrasi, Giuliana und Pantani, Roberto (2017). "Hydrolysis and Biodegradation of Poly(lactic acid)". In: *Synthesis, Structure and Properties of Poly(lactic acid)*. Springer International Publishing, S. 119–151. ISBN: 9783319642307. DOI: [10.1007/12_2016_12](https://doi.org/10.1007/12_2016_12).
- Herzog, Alexander (2022). *Daten & Fakten 2022: Arzneimittel und Gesundheitswesen in Österreich*. PHARMIG. URL: https://www.pharmig.at/media/4964/pharmig-daten_und_fakten_2022_deutsch_web.pdf.
- Kolesnikov, P.M. (1987). "Generalized boundary conditions of the heat and mass transfer". In: *International Journal of Heat and Mass Transfer* 30.1, S. 85–92. DOI: [10.1016/0017-9310\(87\)90062-7](https://doi.org/10.1016/0017-9310(87)90062-7).
- Langtangen, Hans Petter und Logg, Anders (2016a). *Solving PDEs in Python*. Springer International Publishing. DOI: [10.1007/978-3-319-52462-7](https://doi.org/10.1007/978-3-319-52462-7).
- Langtangen, Hans Petter und Pedersen, Geir K. (2016b). *Scaling of Differential Equations*. Springer International Publishing. DOI: [10.1007/978-3-319-32726-6](https://doi.org/10.1007/978-3-319-32726-6).
- Limsukon, Wanwarang, Rubino, Maria, Rabnawaz, Muhammad, Lim, Loong-Tak und Auras, Rafael (Nov. 2023). "Hydrolytic degradation of poly(lactic acid): Unraveling correlations between temperature and the three phase structures". In: *Polymer Degradation and Stability* 217, S. 110537. ISSN: 0141-3910. DOI: [10.1016/j.polymdegradstab.2023.110537](https://doi.org/10.1016/j.polymdegradstab.2023.110537).
- Logg, Anders, Mardal, Kent-Andre und Wells, Garth (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Springer Berlin Heidelberg. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).
- Lopes, M. Savioli, Jardini, A.L. und Filho, R. Maciel (2012). "Poly (Lactic Acid) Production for Tissue Engineering Applications". In: *Procedia Engineering* 42, S. 1402–1413. ISSN: 1877-7058. DOI: [10.1016/j.proeng.2012.07.534](https://doi.org/10.1016/j.proeng.2012.07.534).

Macha, Innocent J., Ben-Nissan, Besim, Vilchevskaya, Elena N., Morozova, Anna S., Abali, Bilen Emek, Müller, Wolfgang H. und Rickert, W. (2019). "Drug Delivery From Polymer-Based Nanopharmaceuticals—An Experimental Study Complemented by Simulations of Selected Diffusion Processes". In: *Frontiers in Bioengineering and Biotechnology* 7. DOI: [10.3389/fbioe.2019.00037](https://doi.org/10.3389/fbioe.2019.00037).

MicrobeNotes (o. D.). *Magnetic Stirrer- Principle, Parts, Types, Uses, Examples*. URL: <https://microbenotes.com/magnetic-stirrer-principle-parts-types-uses-examples/> (besucht am 11.02.2025).

Morozova, Anna S., Vilchevskaya, Elena N., Müller, Wolfgang und Bessonov, Nikolay M. (2021). "A holistic continuum model-based approach to drug release". In: *Continuum Mechanics and Thermodynamics* 34.1, S. 81–91. DOI: [10.1007/s00161-021-01046-8](https://doi.org/10.1007/s00161-021-01046-8).

Nernst, W. (Jan. 1904). "Theorie der Reaktionsgeschwindigkeit in heterogenen Systemen". In: *Zeitschrift für Physikalische Chemie* 47U.1, S. 52–55. ISSN: 0942-9352. DOI: [10.1515/zpch-1904-4704](https://doi.org/10.1515/zpch-1904-4704).

Noyes, Arthur A. und Whitney, Willis R. (Dez. 1897). "THE RATE OF SOLUTION OF SOLID SUBSTANCES IN THEIR OWN SOLUTIONS." In: *Journal of the American Chemical Society* 19.12, S. 930–934. ISSN: 1520-5126. DOI: [10.1021/ja02086a003](https://doi.org/10.1021/ja02086a003).

Østerby, Ole (2003). "Five Ways of Reducing the Crank–Nicolson Oscillations". In: *BIT Numerical Mathematics* 43.4, S. 811–822. DOI: [10.1023/b:bitn.0000009942.00540.94](https://doi.org/10.1023/b:bitn.0000009942.00540.94).

Paul, Steven M., Mytelka, Daniel S., Dunwiddie, Christopher T., Persinger, Charles C., Munos, Bernard H., Lindborg, Stacy R. und Schacht, Aaron L. (2010). "How to improve R&D productivity: the pharmaceutical industry's grand challenge". In: *Nature Reviews Drug Discovery* 9.3, S. 203–214. DOI: [10.1038/nrd3078](https://doi.org/10.1038/nrd3078).

Python Software Foundation (o. D.). *Python Language Reference, version 3.10.12*. Available at <http://www.python.org>.

Richardson, L. F. (1911). "IX. The approximate arithmetical solution by finite differences of physical problems involving differential equations, with an application to the stresses in a masonry dam". In: *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character* 210.459-470, S. 307–357. DOI: [10.1098/rsta.1911.0009](https://doi.org/10.1098/rsta.1911.0009).

Rickert, Wilhelm, Morozova, Anna, Müller, Wolfgang H., Vilchevskaya, Elena N., Ben-Nissan, Besim und Macha, Innocent (2019). "Drug Delivery from

- Polymer-Based Nanopharmaceuticals—Simulations of Selected Diffusion Processes". In: *Advanced Structured Materials*. Springer International Publishing, S. 207–226. DOI: [10.1007/978-3-030-21251-3_12](https://doi.org/10.1007/978-3-030-21251-3_12).
- Roache, Patrick J. (2001). "Code Verification by the Method of Manufactured Solutions". In: *Journal of Fluids Engineering* 124.1, S. 4–10. DOI: [10.1115/1.1436090](https://doi.org/10.1115/1.1436090).
- (2019). "The Method of Manufactured Solutions for Code Verification". In: *Simulation Foundations, Methods and Applications*. Springer International Publishing, S. 295–318. DOI: [10.1007/978-3-319-70766-2_12](https://doi.org/10.1007/978-3-319-70766-2_12).
- Ruiz-Garcia, Ana, Bermejo, Marival, Moss, Aaron und Casabo, Vicente G. (2008). "Pharmacokinetics in Drug Discovery". In: *Journal of Pharmaceutical Sciences* 97.2, S. 654–690. DOI: [10.1002/jps.21009](https://doi.org/10.1002/jps.21009).
- Schlander, Michael, Hernandez-Villafuerte, Karla, Cheng, Chih-Yuan, Mestre-Ferrandiz, Jorge und Baumann, Michael (2021). "How Much Does It Cost to Research and Develop a New Drug? A Systematic Review and Assessment". In: *PharmacoEconomics* 39.11, S. 1243–1269. DOI: [10.1007/s40273-021-01065-y](https://doi.org/10.1007/s40273-021-01065-y).
- Tamimi, Nihad A.M. und Ellis, Peter (Aug. 2009). "Drug Development: From Concept to Marketing!" In: *Nephron Clinical Practice* 113.3, S. c125c131. ISSN: 1660-2110. DOI: [10.1159/000232592](https://doi.org/10.1159/000232592).
- Thomas, J. W. (1995). *Numerical Partial Differential Equations: Finite Difference Methods*. Springer New York. DOI: [10.1007/978-1-4899-7278-1](https://doi.org/10.1007/978-1-4899-7278-1).
- Van Rossum, Guido und Drake, Fred L. (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace. ISBN: 1441412697.
- Virtanen, Pauli et al. (Feb. 2020). "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature Methods* 17.3, S. 261–272. ISSN: 1548-7105. DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).
- Zaaba, Nor Fasihah und Jaafar, Mariatti (Sep. 2020). "A review on degradation mechanisms of polylactic acid: Hydrolytic, photodegradative, microbial, and enzymatic degradation". In: *Polymer Engineering & Science* 60.9, S. 2061–2075. ISSN: 1548-2634. DOI: [10.1002/pen.25511](https://doi.org/10.1002/pen.25511).

A GNU-Octave-Programm

Quelltext A.1 zeigt das GNU-Octave-Programm mit dessen Hilfe die Medikamentenmasse auf der Matrixoberfläche bestimmt wird, siehe Kapitel 3.2.1.

```
1 clc
2 close all
3 clear all
4 pkg load image
5 REM=imread('REM_ohne_Legende.png');
6
7 % grayscale
8 REM_gray=rgb2gray(REM);
9 figure,
10 imshow(REM_gray);
11
12 % bw
13 threshold=0.78;
14 REM_bw=im2bw(REM_gray, threshold);
15 figure,
16 imshow(REM_bw);
17
18 % count
19 anzahl = bwconncomp(REM_bw);
20 anzahlPartikel = anzahl.NumObjects;
21 disp('Die Anzahl der einzelnen Medikamentenpartikel
      betraegt');
22 disp(anzahlPartikel)
23 disp(' ');
24 disp(' ');
25
26
27 disp('R E M - A U F N A H M E');
28
29 % area Partikel
30 area = bwarea(REM_bw);
```

```
31 disp('Die Flaeche, die alle auf dem Ausschnitt zu  
      sehenden Partikel zusammen einnehmen, betraegt in  
      Pixeln');  
32 disp(area);  
33 display('');  
34  
35 % area REM  
36 area_Rem_Pixel = 1072*730;  
37 area_Rem_mu2 = 221.58;  
38 disp('Die REM-Aufnahme hat eine Auflösung von 1072x730  
      Pixeln und besteht damit aus ... Pixeln');  
39 disp(area_Rem_Pixel)  
40 display('');  
41 disp('Die Fläche, die alle auf dem Ausschnitt zu  
      sehenden Partikel zusammen einnehmen, betraegt in mu  
      ^2');  
42 area_mu2 = (area_Rem_mu2/area_Rem_Pixel)*area;  
43 disp(area_mu2)  
44 disp('');  
45 disp('');  
46  
47  
48 disp('P A R T I K E L G E O M E T R I E');  
49  
50 # durchschnittlicher Partikelquerschnitt  
51 area_mu2_pro_partikel = area_mu2/anzahlPartikel;  
52 disp('Der durchschnittliche Partikelquerschnitt beträgt  
      ...[mu mš].');  
53 disp(area_mu2_pro_partikel)  
54 disp('');  
55  
56 % area Partikel Prozent  
57 area_prozent = area/area_Rem_Pixel*100;  
58 disp('Damit nehmen die Medikamentenpartikel ... Prozent  
      der Oberflaeche ein.');//  
59 disp(area_prozent)  
60 disp('');  
61  
62 # durchschnittlicher Partikelradius  
63 disp('Der durchschnittliche Partikelradius beträgt ...[  
      mu m].');  
64 r = sqrt(area_mu2_pro_partikel/pi);
```

```
65 disp(r)
66 disp(' ');
67
68 # durchschnittliche Partikeloberfläche
69 disp('Die durchschnittliche Partikeloberfläche beträgt
...[mu m^2].');
70 a = 4*pi*r^2;
71 disp(a)
72 disp(' ');
73 disp(' ');
74
75
76 disp('M E D I K A M E N T E N L A D U N G');
77
78 # Medikamentenvolumen auf REM-Aufnahme
79 disp('Das kummulierte Medikamentenvolumen auf der REM-
Aufnahme beträgt ...[mu m^3].');
80 V_Rem = (4/3)*pi*r^3*anzahlPartikel;
81 disp(V_Rem)
82 disp(' ');
83
84 # Medikamentenvolumen auf gesamten Matrix
85 h_Matrix = 20000;
86 ri = 3200;
87 ro = 3400;
88 area_Matrix = 2*pi*h_Matrix*(ro+ri);
89 disp('Das kummulierte Medikamentenvolumen auf der
gesamten Matrix beträgt ...[mu m^3].');
90 V_Matrix = V_Rem*(area_Matrix/area_Rem_mu2);
91 disp(V_Matrix)
92 disp(' ');
93
94 # Medikamentenmasse auf Matrix
95 rho_GM = 1.3*10^(-9);
96 m_Matrixoberflaeche = V_Matrix*rho_GM;
97 disp('Es befindet sich eine Medikamentenmasse von ...[mg]
auf der Matrixoberfläche.');
98 disp(m_Matrixoberflaeche)
99 disp(' ');
100
101 # prozentualer Anteil der gesamten Medikamentenmasse auf
der Matrixoberfläche
```

```
102 m_d = 7.5;
103 q_GM_Oberflaeche = (m_Matrixoberflaeche/m_d)*100;
104 disp('Es befinden sich ...% des gesamten Medikaments auf
      der Matrixoberfläche.');
105 disp(q_GM_Oberflaeche)
106
107 # Gesamtanzahl an Medikamentenpartikeln auf der
      aeusseren Mantelflaeche
108 A_o = 2*pi*ro*h_Matrix;
109 n_o = A_o/area_Rem_mu2 * anzahlPartikel;
110 disp('Auf der aeusseren Mantelflaeche befinden sich ...
      Partikel.');
111 disp(n_o)
112
113 # Gesamtanzahl an Medikamentenpartikeln auf der inneren
      Mantelflaeche
114 A_i = 2*pi*ri*h_Matrix;
115 n_i = A_i/area_Rem_mu2 * anzahlPartikel;
116 disp('Auf der inneren Mantelflaeche befinden sich ...
      Partikel.');
117 disp(n_i)
```

Qtxt. A.1: GNU-Octave-Programm zur Bestimmung der Medikamentenmasse auf der Matrixoberfläche

B Python-Quelltext für FEniCS Legacy

Implementierungsaufbau des physikalischen Modells

Abbildung B.1 verdeutlicht den Aufbau der Implementierung des in dieser Arbeit präsentierten physikalischen Modells, siehe Kapitel 5.2 und 5.3. Im Zentrum steht der Quelltext *model.py* (vgl. Anhang B.5). Darüber hinaus findet sich die Implementierung der konstanten bzw. parabelförmigen Startkonzentrationsverteilung $\tilde{c}(\tilde{r}, \tilde{t} = 0)$ (vgl. Kapitel 3.2.2 in einheitenbehafteter Darstellung) in *u_0.py* (vgl. Anhang B.8). Die Implementierung der ROBIN-Randbedingung (vgl. Kapitel 5.2.3 und 5.2.4) ist im Quelltext *boundary_conditions.py* (vgl. Anhang B.7) hinterlegt, die der Diffusionsgleichung der Umgebungslösung (vgl. Kapitel 5.2.5) im Quelltext *surrounding_solution.py* (vgl. Anhang B.10). In *mesh_domains_boundaries.py* (vgl. Anhang B.6)

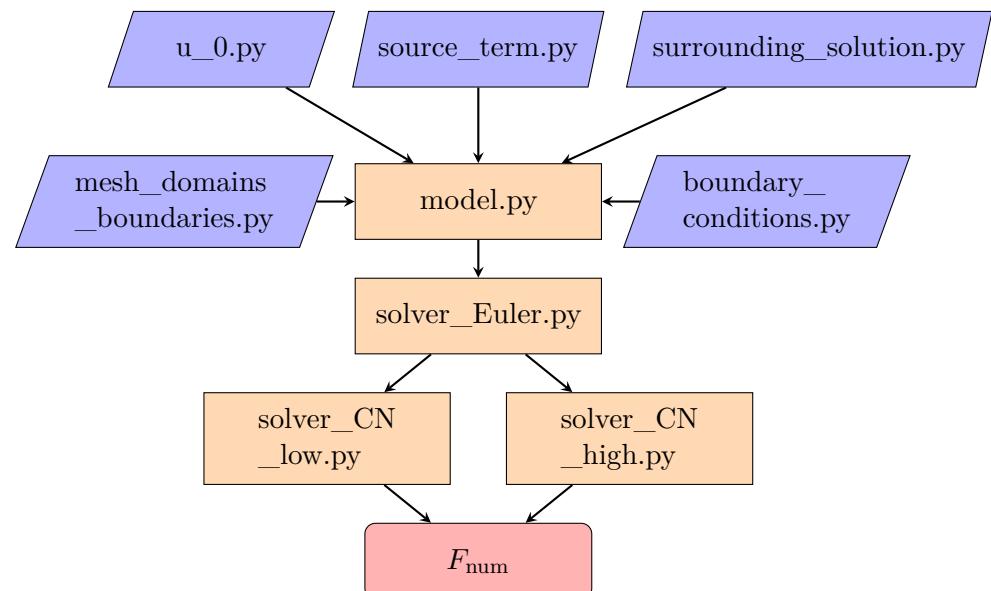


Abb. B.1: Programmablaufplan der Implementierung des räumlich und zeitlich diskretisierten physikalischen Modells

findet sich die Implementierung der räumlichen Diskretisierung der hohlzyllinderförmigen Medikamentenmatrix $\tilde{\Omega}_m$ (vgl. Kapitel 5.3.1). Im Falle der Konvergenzanalyse (vgl. Kapitel 5.4) wird ein Quellterm aus *source_term.py* (vgl. Angang B.9) zugeschaltet. Ansonsten ist dieser Wert, entsprechend einer homogenen Diffusionsgleichung, null.

model.py bekommt *u_0.py*, *source_term.py*, *surrounding_solution.py*, *mesh_domains_boundaries.py*, *boundary_conditions.py* und den aus Übersichtsgründen nicht dargestellten Quelltext *degradation.py* (vgl. Anhang B.11 und Kapitel 5.2.2) als Input übergeben.

Daraufhin wird die homogene Diffusionsgleichung (vgl. Kapitel 5.2.1) mit Hilfe von *solver_Euler.py* (vgl. Anhang B.12) für $n_t = 100$ Zeitschritte berechnet. Nach Ablauf dieser Zeitschritte wird die homogene Diffusionsgleichung durch ein adaptives CRANK-NICOLSON-Verfahren (vgl. Kapitel 5.2) berechnet. Dazu werden *solver_CN_low.py* (vgl. Anhang B.13) und *solver_CN_high.py* (vgl. Anhang B.14) verwendet. Als Output gibt *model.py* die numerisch bestimmte Freisetzungskurve F_{num} (vgl. Gleichung 5.31) zurück. Sie wird mit Hilfe der in *utility.py* (vgl. Anhang B.15) implementierten Funktion *calc_F()* berechnet.

Programmaufbau

Abbildung B.2 illustriert den Ablauf des Python-Quelltext B. In *parameters_units.py* (vgl. Anhang B.1) sind die Material- und Geometrieparameter in einheitenbehafteter Darstellung sowie die Referenzparameter zur Überführung dieser in die einheitenlose Form hinterlegt. Diese einheitenlosen Parameter werden in *parameters.py* (vgl. Anhang B.2), neben weiteren Hilfsparametern, gespeichert und zusammen an *main.py* (vgl. Anhang B.3) übergeben. Dort kann der Nutzer zwischen einer Simulation des physikalischen Modells (vgl. Anhang B.5 und Abbildung B.1), einer inversen Analyse (vgl. Kapitel 5.7) und der Konvergenzanalyse (vgl. Kapitel 5.4) wählen. Diese Nutzerwahl ist im Quelltext *menu.py* (vgl. Anhang B.4) hinterlegt.

Im Fall einer Simulation des physikalischen Modells wird an *model.py* (vgl. Anhang B.5) übergeben. Wählt der Nutzer eine Konvergenzanalyse, greift Quelltext *convergence_analysis* (vgl. Anhang B.17). Hier kann zwischen einer Optimierung für den Diffusionskoeffizienten \tilde{D} , des Stoffübergangsparameters $\tilde{\alpha}$ und des Längenparameters \tilde{l}_b während der ersten fünf Wochen oder einer Optimierung für die Matrixoberfläche nach dem Bruch \tilde{A}_m^b während der weiteren zehn Wochen gewählt werden. Diese Parameter werden iteriert und jeweils zur erneuten Berechnung an *model.py* übergeben. Möchte der Nutzer eine Konvergenzanalyse ausführen, wird für eine zeitliche Konvergenzanalyse die

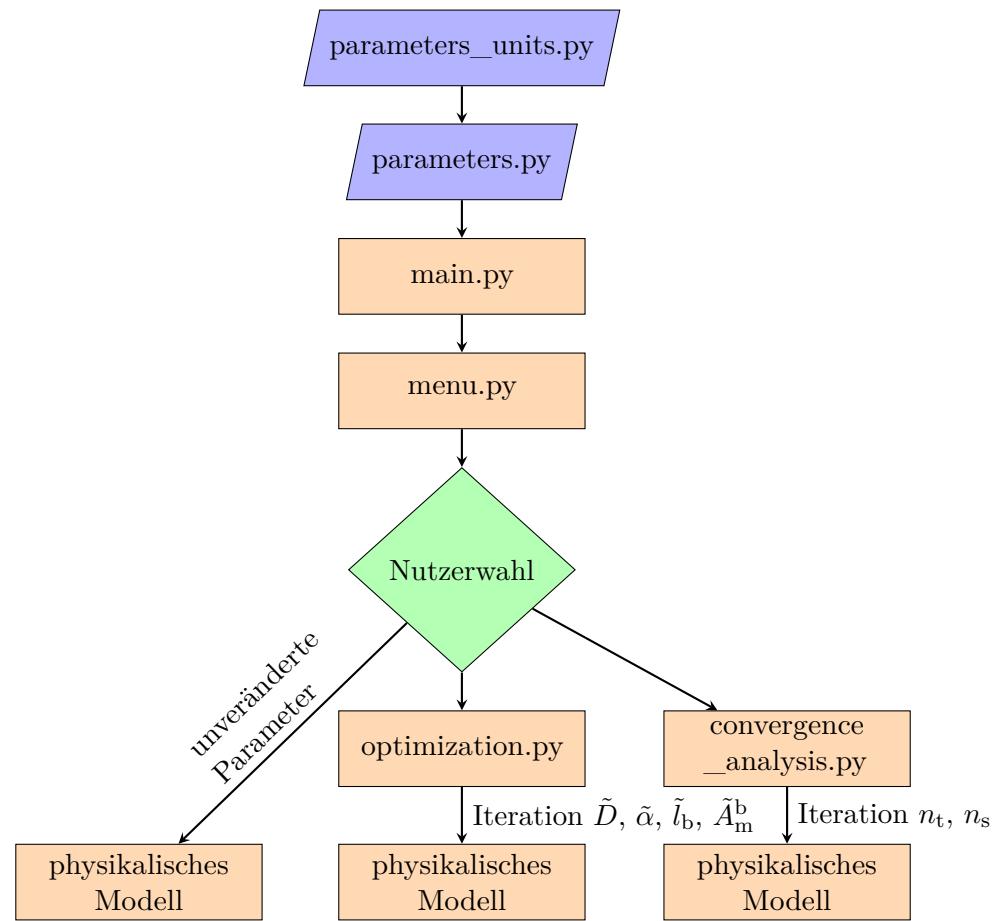


Abb. B.2: Programmablaufplan des Python-Quelltexts B

Anzahl der Zeitschritte n_t iteriert, für eine räumliche Konvergenzanalyse analog die Anzahl der Elemente n_s und jeweils an `model.py` übergeben.

B.1 parameters_units.py

```

1 # -----
2 # P A R A M E T E R S (units)
3 # -----
4 import numpy as np
5
6 # Geometrieparameter Hohlzylindermatrix
7 r_i_units = 3.2
8          # [mm]      - Innenradius r_i der Hohlzylindermatrix
9 r_o_units = 3.4
10         # [mm]     - AuSSenradius r_o der Hohlzylindermatrix
11 H_units = 20.0
12          # [mm]     - Höhe H der Hohlzylindermatrix
13 V_c_units = np.pi * (r_o_units**2 - r_i_units**2)
14          * H_units # [mm$]   - Volumen V_c der Hohlzylindermatrix
15 A_m_t0_units = 2.0*np.pi*H_units*(r_i_units+
16          r_o_units) # [mm$]   - Mantelfläche der
17          Hohlzylindermatrix
18
19 # Gentamicin-Parameter
20 D_GM_exp_units = 2.6 * 10**-7
21          # [mm$/s] - erwartbare Groessenordnung des
22          Diffusionskoeffizientens
23 m_d_t0_units = 0.0075
24          # [g]       - Gesamtmasse GM, die in die Matrix
25          eingebracht wurde (m_d = 7.5 mg)
26
27 # Umgebungslösungs-Parameter
28 V_s_units = 15000.0
29          # [mm$]   - Volumen der Umgebungslösung (V_s = 15 ml
30          )
31 c_max_units = m_d_t0_units/V_s_units
32          # [g/mm$] - maximal mögliche Konzentration innerhalb
33          der Umgebungslösung
34
35 #-----
36 # U N B E K A N N T E P A R A M E T E R (hier bereits optimiert)
37 #
38 D_expect_constant_units = 2.6032998653581 * 10**-7
39          # [mm$/s] - Diffusion coefficient
40 D_expect_parabolic_units = 2.5993487812531 * 10**-7
41          # [mm$/s] - Diffusion coefficient
42 alpha_expect_constant_units = 2.61919530791703 * 10**6
43          # [mm$/g] - transfer coefficient
44 alpha_expect_parabolic_units = 3.02044125873031 * 10**6
45          # [mm$/g] - transfer coefficient
46 l_b_expect_constant_units = 3.19623827676102
47          # [mm]      - Diffusionsübergangsbereich
48 l_b_expect_parabolic_units = 3.20057233085569
49          # [mm]      - Diffusionsübergangsbereich

```

```
30 A_m_degrad_constant_units    = 2.85545640764786 * A_m_t0_units
     # [mm$] - Mantelfläche der Hohlzylindermatrix
     nachdem sie aufgebrochen ist
31 A_m_degrad_parabolic_units   = 4.5053678672349 * A_m_t0_units
     # [mm$] - Mantelfläche der Hohlzylindermatrix
     nachdem sie aufgebrochen ist
32
33 #-----
34 # R E F E R E N Z K O N S T A N T E N
35 #-----
36 t_ref                      = 604800.0
     # [s]      - = 1 Woche
37 c_ref                       = m_d_t0_units/V_c_units
     # [g/mm$] - = c(t=0)
38 l_ref                       = r_i_units
     # [mm]     - Innenradius der zylinderförmigen Matrix
39 D_ref                        = D_GM_exp_units
     # [mm$/s] - erwartbare Groessenordnung des
     Diffusionskoeffizientens
```

B.2 parameters.py

```

1 # -----
2 # P A R A M E T E R S (unitless)
3 # -----
4 from parameters_units import D_expect_constant_units,
   alpha_expect_constant_units, l_b_expect_constant_units,
   D_expect_parabolic_units, alpha_expect_parabolic_units,
   l_b_expect_parabolic_units, r_i_units, r_o_units, H_units,
   A_m_t0_units, A_m_degrad_constant_units,
   A_m_degrad_parabolic_units, V_c_units, m_d_t0_units, c_max_units
   , V_s_units, D_ref, c_ref, l_ref, t_ref
5 import numpy as np
6
7 #-----
8 # U N B E K A N N T E P A R A M E T E R (hier bereits optimiert)
9 #-----
10 D_constant           = D_expect_constant_units / D_ref      #
11   [-]     - Diffusionskoeffizient (D = D_GM_exp/D_ref)
12 D_parabolic          = D_expect_parabolic_units / D_ref      #
13   [-]     - Diffusionskoeffizient (D = D_GM_exp/D_ref)
14 alpha_constant       = alpha_expect_constant_units * c_ref  #
15   [-]     - another transfer coefficient
16 alpha_parabolic     = alpha_expect_parabolic_units * c_ref  #
17   [-]     - another transfer coefficient
18 l_b_constant        = l_b_expect_constant_units/l_ref      #
19   [-]     - Diffusionsübergangsbereich
20 l_b_parabolic        = l_b_expect_parabolic_units/l_ref      #
21   [-]     - Diffusionsübergangsbereich
22 A_m_degrad_constant = A_m_degrad_constant_units/l_ref**2    #
23   [-]     - Mantelfläche der Hohlzylindermatrix nachdem sie
24   aufgebrochen ist
25 A_m_degrad_parabolic = A_m_degrad_parabolic_units/l_ref**2  #
26   [-]     - Mantelfläche der Hohlzylindermatrix nachdem sie
27   aufgebrochen ist
28 const_degrad_constant = 1.0                                     #
29   [-]     - degrad = (1-exp(-const_degrad*(1-u_n)*š))*100
30 const_degrad_parabolic = 1.0                                     #
31   [-]     - degrad = (1-exp(-const_degrad*(1-u_n)*š))*100
32 degrad_break_constant = 0.356                                    #
33   [-]     - matrix breaks, if value [0,1] is reached
34 degrad_break_parabolic = 0.503                                    #
35   [-]     - matrix breaks, if value [0,1] is reached
36
37 # Zeitparameter
38 t                   = 0.0                                      #
39   [-]     - Startzeitpunkt
40 T                   = 15.0                                     #
41   [-]     - Gesamtversuchsdauer = 15 Wochen (T = 15 W[s]/t_ref =
42   15 W[s]/1 W[s] = 15)

```

```

26 num_steps          = 0                                #
27 [-]      - Startwert des Berechnungsschrittzählers
28
29 # Gentamicin
30 m_d_t0           = m_d_t0_units/(c_ref*l_ref**3)    #
31 [-]      - Gesamtmasse GM, die in die Matrix eingebracht wurde (
32     m_d_units = 7.5 mg)
33
34 # Geometrie Hohlzylindermatrix
35 r_i              = r_i_units/l_ref                 #
36 [-]      - Innenradius r_i der Hohlzylindermatrix
37 r_o              = r_o_units/l_ref                 #
38 [-]      - Außenumradius r_o der Hohlzylindermatrix
39 l_c              = r_o-r_i                         #
40 [-]      - Wandstärke des Hohlzylinders
41 H                = H_units/l_ref                 #
42 [-]      - Höhe H der Hohlzylindermatrix
43 V_c              = V_c_units/l_ref**3             #
44 [-]      - Volumen V_c der Hohlzylindermatrix
45 A_m_t0           = A_m_t0_units/l_ref**2          #
46 [-]      - Mantelfläche der Hohlzylindermatrix
47
48 # Konzentration Hohlzylindermatrix
49 const_distribution = m_d_t0/(np.pi*H*(r_o**2 - r_i**2)) #
50 [-]      - u(t=0) = constant = m_d_t0 / (pi*H*r_o - r_i)
51 offset_parabola  = 0.0001                          #
52 [-]      - u(t=0) = parabel * (x[0] - ((r_i + r_o) / 2.0)) +
53     offset
54 parabolic_distribution = ((-12.0*(m_d_t0-np.pi*H*offset_parabola*(
55     r_o**2 - r_i**2)))/ # [-]      - Parameter, um parabelförmige
56     Medikamentenverteilung zu spezifizieren
57     (np.pi*H*(r_i-r_o)**3 * (r_i+r_o)))
58
59 # Parameter Umgebungslösung c_s
60 c_s_t0           = 0.0                                #
61 [-]      - Medikamentenkonzentration in der Umgebungsloesung zum
62     Startzeitpunkt t=0
63 c_max            = c_max_units/c_ref                 #
64 [-]      - maximal mögliche Konzentration innerhalb der
65     Umgebungslösung
66 F_15             = 0.844                            #
67 [-]      - F(t=15 W) = c_s/c_max = 0.844
68 c_sat            = F_15*c_max                      #
69 [-]      - Sättigungsloeslichkeit der Umgebungsloeslichkeit
70     fuer Gentamicin
71 V_s              = V_s_units/l_ref**3             #
72 [-]      - Volumen der Umgebungslösung (V_s_units = 15 ml)
73
74 # Weitere Degradationsparameter
75 t_break           = 5.0                                #
76 [-]      - Matrix bricht in Woche 5 (t=5)

```

```

54 t_reached_5          = False                      #
55   [-]      - wird auf 'True' gesetzt m sobald t=5 erreicht wird bzw
56   . die Matrix gebrochen ist
55 const_degrad_deactivated = 0                      #
56   [-]      - matrix break deactivated (for optimization before
57   break): degrad = (1-exp(-const_degrad*(1-u_n)))*100 = 0
56 limiter_break       = 1                          #
57   [-]      - if limiter_break != 1: matrix is broken
57 limiter_sat         = 1                          #
58   [-]      - if limiter_sat != 1: "saturation" started
58
59 # Hilfsparameter
60 beta              = (D_ref*t_ref)/l_ref**2        #
61   [-]      - Hilfskonstante zur besseren Lesbarkeit
61 gamma             = D_ref*t_ref*l_ref/V_s_units    #
62   [-]      - Hilfskonstante zur besseren Lesbarkeit (wird für c_s
62   verwendet)
63
63 # Elementparameter
64 element_family     = 'P'
65 degree            = [1,2]
66   # [-]      - Grad des gewählten Elementtyps
66 nx = nr            = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
67   # [-]      - Anzahl der Elemente der Intervalldomain
67 h                 = []
68   # [-]      - h = 1/nx bzw. h = 1/nt
68
69 # mesh Parameter
70 boundary_tol       = 1E-14                      #
71   [-]
71
72 # Expliziter EULER
73 T_explicit_Euler   = 70 * 10 ** -9
74 num_steps_explicit_Euler = [100, 200, 300, 400, 500, 600, 700, 800,
75   900, 1000]
75
76 # adaptiver CN
77 fineness           = 2                          #
78   [-]      - dt_coarse = fineness * dt_fine
78 runs_CN_high       = fineness                  #
79   [-]      - solver Durchläufe pro Zeitschritt
79 p                 = 2                          #
80   [-]      - CN-Wert für Richardson-Extrapolation
80 tol_adaptiv_dt    = 10**-3                     #
81   [-]      - besserer Wert? --> Bezug zur Messgenauigkeit?
81 safety            = 0.9                      #
82   [-]      - 0 < safety < 1 (Quelle: Fenics hands-on)
82
83
84 # Inverse Analyse

```

```
85 F_num_komplex_n_Euler      = 0.0                      #
86 [-]      - Startwert: F_num(t=0)=0
87 kind_of_1d_interpolation = 'linear'                   #
88 [-]      - Wie soll F_exp interpoliert werden?
89 t_exp          = [0., 1., 2., 3., 4., 5., 7.,
90     8., 9., 11., 15.]
91 F_exp          =
92     [0., 0.399, 0.517, 0.607, 0.623, 0.639, 0.792, 0.805, 0.814, 0.823, 0.844]
93 t_num          = [0]                      #
94 [-]      - erster Wert: t=0
95 F_num          = [0]                      #
96 [-]      - erster Wert: F(t=0) = 0
97 opt_success    = False                     #
98 [-]      - Status der Optimierung
99
100 # MMS
101 U_0            = 2.0                      #
102 [-]      - Konzentration zur Berechnung der angenommenen Loesung
103 U_1            = 1.0                      #
104 [-]      - Konzentration zur Berechnung der angenommenen Loesung
105
106 # Konvergenzanalyse
107 discretization_scheme = 0                  #
108 [-]      - DUMMY-variable [only != 0, if convergence analysis is
109     active]
110 steps_convergence_analysis = len(num_steps_explicit_Euler)
111 L2_error_avg_list       = [[] for _ in range(len(degree))]
112
113 # Zeitschrittparameter
114 dt              = len(num_steps_explicit_Euler)* [0]
115 for i in range(len(num_steps_explicit_Euler)):
116     dt[i]= T_explicit_Euler/num_steps_explicit_Euler[i]
117
118 nt_CN          = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
119     # [-] - Anzahl der Unterteilungen des untersuchten Zeitraums
120 dt_CN          = len(nt_CN)* [0]
121     # [-] - Wird nur für die Konvergenzanalyse verwendet
122 for j in range(len(nt_CN)):
123     dt_CN[j] = T/nt_CN[j]
```

B.3 main.py

```

1 # -----
2 # M A I N
3 # -----
4 from parameters import *
5 from menu import main_menu, u_t0_distribution_menu, timer_menu,
   verbose_mode, compute_results
6 from utility import stop_timer
7 from plot_funcs import print_plot_all_results
8 import matplotlib.pyplot as plt # version: 2.1.1
9
10
11 # -----
12 #   main M E N U
13 # -----
14 menu = main_menu()
15
16 # -----
17 #   D R U G   D I S T R I B U T I O N   u(t=0)
18 # -----
19 D, alpha, l_b, A_m_degrad, const_degrad, degrad_break,
   distribution_menu = u_t0_distribution_menu(D_constant,
   D_parabolic,
20
   alpha_constant, alpha_parabolic,
21
   l_b_constant, l_b_parabolic,
22
   A_m_degrad_constant,
23
   A_m_degrad_parabolic,
24
   const_degrad_constant,
25
   const_degrad_parabolic,
26
   degrad_break_constant,
27
   degrad_break_parabolic,
28
   menu)
29
30 # -----
31 #   toggle V E R B O S E mode
32 # -----
33 #   timer M E N U
34 # -----
35 timer, tic = timer_menu()
36
37 # -----

```

```
38 # C O M P U T E R E S U L T S according to user choice
39 # -----
40 results = compute_results(D, alpha, l_b, A_m_degrad, const_degrad,
41     degrad_break,
41     t, t_break, t_reached_5, T,
41     T_explicit_Euler, dt, dt_CN, c_sat, c_s_t0, c_max, V_s,
41     V_s_units, V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0,
41     const_distribution, parabolic_distribution, offset_parabola, U_0
41     , U_1, beta, gamma, const_degrad_deactivated, limiter_break,
41     limiter_sat,
42     nx, nr, h, degree, element_family,
42     num_steps, num_steps_explicit_Euler, nt_CN,
42     steps_convergence_analysis, L2_error_avg_list,
42     F_num_komplex_n_Euler, t_num, F_num, runs_CN_high, fineness,
42     tol_adaptiv_dt, safety, p, boundary_tol,
43     menu, distribution_menu, verbose,
43     discretization_scheme, opt_success, t_ref, c_ref, D_ref, l_ref,
44     t_exp, F_exp, kind_of_1d_interpolation)
45
46 stop_timer(tic, timer)
47
48 # -----
49 # P R I N T R E S U L T S
50 #
51 print_plot_all_results(results, menu, t_exp, F_exp, T,
51     kind_of_1d_interpolation, element_family, degree, nx, h, D,
51     alpha, l_b, const_degrad, degrad_break, A_m_t0, A_m_t0_units,
51     A_m_degrad_constant_units, A_m_degrad_parabolic_units,
51     opt_success, c_ref, D_ref, l_ref)
52 plt.show() # show plots
```

B.4 menu.py

```

1 # -----
2 # M E N U
3 # -----
4 from mesh_domains_boundaries import collect_mesh_parameters
5 from model import run_model
6 from convergence_analysis import run_convergence_analysis
7 from optimization import run_optimization
8 from utility import check_CFL_conditions,
9     add_fitted_points_to_experimental_data,
10    adjust_t_exp_and_F_exp_to_optimization_time_frame
11 from testing import run_testing
12 import time
13 import sys
14
15
16 def compute_results(D, alpha, l_b, A_m_degrad, const_degrad,
17     degrad_break,
18         t, t_break, t_reached_5, T, T_explicit_Euler, dt
19     , dt_CN, c_sat, c_s_t0, c_max, V_s, V_s_units, V_c_units, r_i,
20     r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
21     parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
22     const_degrad_deactivated, limiter_break, limiter_sat,
23         nx, nr, h, degree, element_family, num_steps,
24     num_steps_explicit_Euler, nt_CN, steps_convergence_analysis,
25     L2_error_avg_list, F_num_komplex_n_Euler, t_num, F_num,
26     runs_CN_high, fineness, tol_adaptiv_dt, safety, p, boundary_tol,
27         menu, distribution_menu, verbose,
28     discretization_scheme, opt_success, t_ref, c_ref, D_ref, l_ref,
29         t_exp, F_exp, kind_of_1d_interpolation):
30
31
32 if menu == 1: # S I M U L A T I O N
33
34     # Mesh parameters
35     mesh, V, ds, r, n_facet = collect_mesh_parameters(r_i, r_o,
36     nx[0], degree[1], element_family, boundary_tol)
37
38     # Check CFL-conditions
39     dt[0] = check_CFL_conditions(dt[0], D, mesh, verbose,
40     discretization_scheme, menu) # Set dt, that dt meets CFL-
41     condition: dt<=dx^2/(2*D)
42
43     # Run model
44     _tnum, _Fnum = run_model(D, alpha, l_b, A_m_degrad,
45     const_degrad, degrad_break,
46         t, t_break, t_reached_5, T,
47     T_explicit_Euler, dt[0], c_sat, c_s_t0, c_max, V_s, V_s_units,
48     V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
49     parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
50     limiter_break, limiter_sat,
```

```
31             V, mesh, nx[0], nr[0], ds, degree
32             [1], r, n_facet, num_steps, num_steps_explicit_Euler[0],
33             F_num_komplex_n_Euler, runs_CN_high, fineness, tol_adaptiv_dt,
34             safety, p, boundary_tol,
35             menu, distribution_menu, verbose,
36             discretization_scheme, t_ref, c_ref, D_ref, l_ref, t_num, F_num)
37
38
39     return [_tnum, _Fnum, distribution_menu]
40
41
42 elif menu == 2: # I N V E R S E A N A L Y S I S
43
44     # User chooses duration of optimization and the option to
45     # add fitted data points to experimental data
46     parameters_to_optimize, parameters_treated_as_constant,
47     T_opt_end, t_exp_opt, F_exp_opt, opt_time_frame =
48     choose_optimization_time_frame(D, alpha, l_b, A_m_degrad,
49     const_degrad, const_degrad_deactivated, degrad_break,
50     kind_of_1d_interpolation, T, t_exp, F_exp)
51
52     # Mesh parameters
53     mesh, V, ds, r, n_facet = collect_mesh_parameters(r_i, r_o,
54     nx[0], degree[1], element_family, boundary_tol)
55
56     # Check CFL-conditions
57     dt[0] = check_CFL_conditions(dt[0], D, mesh, verbose,
58     discretization_scheme, menu) # Set dt, that dt meets CFL-
59     condition: dt<=dx^2/(2*D)
60
61     # Run inverse analysis (matrix break deactivated)
62     _res_opt = run_optimization(parameters_to_optimize,
63     parameters_treated_as_constant,
64             t, t_break, t_reached_5,
65             T_opt_end, T_explicit_Euler, dt[0], c_sat, c_s_t0, c_max,V_s,
66             V_s_units, V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0,
67             const_distribution, parabolic_distribution, offset_parabola, U_0
68             , U_1, beta, gamma, limiter_break, limiter_sat,
69             V, mesh, nx[0], nr[0], ds,
70             degree[1], r, n_facet, num_steps, num_steps_explicit_Euler[0],
71             F_num_komplex_n_Euler, runs_CN_high, fineness, tol_adaptiv_dt,
72             safety, p, boundary_tol,
73             menu, distribution_menu, verbose
74             , discretization_scheme, t_ref, c_ref, D_ref, l_ref, t_num,
75             F_num,
76             t_exp_opt, F_exp_opt,
77             opt_time_frame, opt_success, kind_of_1d_interpolation)
78
79     if _res_opt.success == True:
```

```

59     # Extract parameters
60     if parameters_to_optimize == [D, alpha, l_b]: #
61         Optimization for week 0-5
62
63         D_opt      = _res_opt.x[0]
64         alpha_opt   = _res_opt.x[1]
65         l_b_opt    = _res_opt.x[2]
66         A_m_degrad_opt = A_m_degrad # A_m_degrad wasn't
67         optimized
68
68     elif parameters_to_optimize == [A_m_degrad]: #
69         Optimization for week 5-15
70
71         A_m_degrad_opt = _res_opt.x[0]
72         D_opt = D      # D      wasn't optimized
73         alpha_opt = alpha # alpha wasn't optimized
74         l_b_opt = l_b      # l_b      wasn't optimized
75
76     # Run model with optimized parameters
77     _tnum, _Fnum = run_model(D_opt, alpha_opt, l_b_opt,
78     A_m_degrad_opt, const_degrad, degrad_break,
79     t, t_break, t_reached_5, T,
80     T_explicit_Euler, dt[0], c_sat, c_s_t0, c_max, V_s, V_s_units,
81     V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
82     parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
83     limiter_break, limiter_sat,
84     V, mesh, nx[0], nr[0], ds, degree
85     [1], r, n_facet, num_steps, num_steps_explicit_Euler[0],
86     F_num_komplex_n_Euler, runs_CN_high, fineness, tol_adaptiv_dt,
87     safety, p, boundary_tol,
88     menu, distribution_menu, verbose,
89     discretization_scheme, t_ref, c_ref, D_ref, l_ref, t_num, F_num)
90
91
92     return [_res_opt, _tnum, _Fnum, opt_time_frame,
93     distribution_menu]
94
95
96 elif menu == 4: # C O N V E R G E N C E   A N A L Y S I S
97
98     _nx, _num_steps_explicit_Euler, \
99     _L2_error_avg, _rate, _diff_F_num_total_avg, \
100    _space_or_time, _discretization_scheme =
101    run_convergence_analysis(D, alpha, l_b, A_m_degrad, const_degrad
102    , degrad_break,
103
103    t, t_break, t_reached_5, T, T_explicit_Euler, dt, dt_CN,
104    c_sat, c_s_t0, c_max, V_s, V_s_units, V_c_units, r_i, r_o, l_c,
105    H, A_m_t0, m_d_t0, const_distribution, parabolic_distribution,
106

```

```
92     offset_parabola, U_0, U_1, beta, gamma, limiter_break,
93     limiter_sat,
94
95     degree, element_family, num_steps, nx, nr, h,
96     steps_convergence_analysis, num_steps_explicit_Euler, nt_CN,
97     L2_error_avg_list, F_num_komplex_n_Euler, runs_CN_high, fineness
98     , tol_adaptiv_dt, safety, p, boundary_tol,
99
100    menu, distribution_menu, verbose, t_ref, c_ref, D_ref,
101    l_ref, t_num, F_num)
102
103
104    return [_nx, _num_steps_explicit_Euler, _L2_error_avg, _rate
105    , _diff_F_num_total_avg, _space_or_time, _discretization_scheme]
106
107
108
109
110
111
112 def main_menu():
113
114     _menu = 0
115
116     while _menu < 1 or _menu > 6:
117
118         print('')
119         print('-----')
120         print('1. Simulation')
121         print('2. Optimization')
122         print('3' + '\u0336' + '.' + '\u0336' + ' ' + 'M' + '\u0336'
123         + 'M' + '\u0336' + 'S' + '\u0336')
124         print('4. Convergence analysis')
125         print('5. Testing')
126         print('')
127         print('6. EXIT')
128         print('-----')
```

```
128     print('')
129
130     _menu = int(input())
131
132     if _menu < 1 or _menu > 6:
133         print('You did not choose between "1-6"!')
134         print('')
135
136     if _menu == 6:
137         sys.exit()
138
139
140     return _menu
141
142
143
144
145 def u_t0_distribution_menu(D_constant, D_parabolic,
146                             alpha_constant, alpha_parabolic,
147                             l_b_constant, l_b_parabolic,
148                             A_m_degrad_constant, A_m_degrad_parabolic
149                             ,
150                             const_degrad_constant,
151                             const_degrad_parabolic,
152                             degrad_break_constant,
153                             degrad_break_parabolic,
154                             menu):
155
156     # Initialize distribution menu with dummy variable,
157     # which is also used for convergence analysis, because there u(t
158     =0) is given by the manufactured solution
159     _distribution_menu = 2
160
161     while _distribution_menu != 0 and _distribution_menu != 1:
162
163         print('')
164         print('-----')
165         print('  G E N E R A L   O P T I O N S')
166         print('-----')
167         print('0. Constant drug distribution [u(t=0)]')
168         print('1. Parabolic drug distribution [u(t=0)]')
169         print('-----')
170         print('')
171
172         _distribution_menu = int(input())
173
174         if _distribution_menu != 0 and _distribution_menu != 1:
175             print('You did not choose "1" or "0"!')
```

```
172     print('')
173
174     if _distribution_menu == 0:
175         _D = D_constant
176         _alpha = alpha_constant
177         _l_b = l_b_constant
178         _A_m_degrad = A_m_degrad_constant
179         _const_degrad = const_degrad_constant
180         _degrad_break = degrad_break_constant
181
182     elif _distribution_menu == 1:
183         _D = D_parabolic
184         _alpha = alpha_parabolic
185         _l_b = l_b_parabolic
186         _A_m_degrad = A_m_degrad_parabolic
187         _const_degrad = const_degrad_parabolic
188         _degrad_break = degrad_break_parabolic
189
190
191     return _D, _alpha, _l_b, _A_m_degrad, _const_degrad,
192         _degrad_break, _distribution_menu
193
194
195 def verbose_mode():
196
197     _verbose = 3 # initialize menu with dummy variables
198
199     while _verbose != 0 and _verbose!= 1 and _verbose != 2:
200
201         print('')
202         print('-----')
203         print('0. Verbose mode: SILENT')
204         print('1. Verbose mode: BALANCED')
205         print('2. Verbose mode: FULL')
206         print('-----')
207         print('')
208
209         _verbose = int(input())
210
211         if _verbose != 0 and _verbose!= 1 and _verbose != 2:
212             print('You did not choose "0", "1" or "2"!')
213             print('')
214
215
216     return _verbose
217
218
219
```

```
220 def timer_menu():
221
222     # Run menu
223     _timer_menu = 2
224
225     while _timer_menu != 0 and _timer_menu != 1:
226
227         print('')
228         print('-----')
229         print(' Do you want to time the chosen calculation?')
230         print('-----')
231         print('0. No')
232         print('1. Yes')
233         print('-----')
234         print('')
235
236
237     _timer_menu = int(input())
238
239     if _timer_menu != 0 and _timer_menu != 1:
240         print('You did not choose "0" or "1"!')
241         print('')
242
243     if _timer_menu == 0:
244
245         _timer_menu = 'timer_off'
246         tic = 'dummy'
247
248     elif _timer_menu == 1:
249
250         _timer_menu = 'timer_on'
251         tic = time.perf_counter()
252
253     return _timer_menu, tic
254
255
256
257 def choose_optimization_time_frame(D, alpha, l_b, A_m_degrad,
258     const_degrad, const_degrad_deactivated, degrad_break,
259     kind_of_1d_interpolation, T, t_exp, F_exp):
260
261     _menu_opt_time_frame = 0
262
263     while _menu_opt_time_frame != 1 and _menu_opt_time_frame != 2:
264
265         print('')
266         print('-----')
```

```
265     print(' 1. Optimization from start (t=0) until matrix break  
(t=5)')
266     print(' 2. Optimization from matrix break (t=5) until end (t=15)')
267     print('-----')
268     print('')
269
270     _menu_opt_time_frame = int(input())
271
272     if _menu_opt_time_frame != 1 and _menu_opt_time_frame != 2:
273         print('You did not choose between "1-2"!')
274         print('')
275
276
277     if _menu_opt_time_frame == 1:
278
279         _opt_time_frame = 'until_break'
280         _T_opt_end = 5.0
281         _parameters_to_optimize = [D, alpha, l_b]
282         _parameters_treated_as_constant = [A_m_degrad,
283         const_degrad_deactivated, degrad_break]
284         _t_exp_opt, _F_exp_opt =
285         menu__add_20_fitted_points_to_experimental_data(t_exp, F_exp,
286         _T_opt_end, T, kind_of_1d_interpolation)
287
288     elif _menu_opt_time_frame == 2:
289
290         _opt_time_frame = 'after_break'
291         _T_opt_end = T
292         _parameters_to_optimize = [A_m_degrad]
293         _parameters_treated_as_constant = [D, alpha, l_b,
294         const_degrad, degrad_break]
295         _t_exp_opt = t_exp
296         _F_exp_opt = F_exp
297
298
299     return _parameters_to_optimize, _parameters_treated_as_constant,
300     _T_opt_end, _t_exp_opt, _F_exp_opt, _opt_time_frame
301
302
303 def menu__add_20_fitted_points_to_experimental_data(t_exp, F_exp,
304     T_opt_end, T, kind_of_1d_interpolation):
305
306
307     # Adjust t_exp & F_exp to optimization time frame (t = 0-5)
308     _t_exp_opt, _F_exp_opt =
309     adjust_t_exp_and_F_exp_to_optimization_time_frame(t_exp, F_exp,
310     T_opt_end)
```

```
305 # Run menu
306 _add_20_points_menu = 2
307
308 while _add_20_points_menu != 0 and _add_20_points_menu != 1:
309
310     print('')
311     print('-----')
312     print(' Do you want to add fitted points to the')
313     print(' experimental data?')
314     print('-----')
315     print('0. No')
316     print('1. Yes')
317     print('-----')
318     print('')
319
320     _add_20_points_menu = int(input())
321
322     if _add_20_points_menu != 0 and _add_20_points_menu!= 1:
323         print('You did not choose "0" or "1"!')
324         print('')
325
326     if _add_20_points_menu == 0:
327
328         return _t_exp_opt, _F_exp_opt
329
330
331 elif _add_20_points_menu == 1:
332
333     _add_20_points_menu_when = 0
334
335     while _add_20_points_menu_when < 1 or
336     _add_20_points_menu_when > 5:
337
338         print('')
339         print('-----')
340         print('      When and how many?')
341         print('-----')
342         print('      1. 20 points @ t = 3.0 - 5.0')
343         print('      2. 20 points @ t = 4.9 - 5.0')
344         print('      3. 60 points @ t = 4.9 - 5.0')
345         print('      4. 100 points @ t = 4.9 - 5.0')
346         print('      5. 100 points @ t = 0.0 - 5.0')
347         print('-----')
348
349         print('-----')
```

```
348     _add_20_points_menu__when = int(input())
349
350     if _add_20_points_menu__when < 1 or
351     _add_20_points_menu__when > 5:
352
353         print('You did not choose between "1-5"!')
354         print('')
355
356     if _add_20_points_menu__when == 1:
357
358         t_add_start = 3.0
359         t_add_stop   = 5.0
360         n_add = 20
361
362     elif _add_20_points_menu__when == 2:
363
364         t_add_start = 4.9
365         t_add_stop   = 5.0
366         n_add = 20
367
368     elif _add_20_points_menu__when == 3:
369
370         t_add_start = 4.9
371         t_add_stop   = 5.0
372         n_add = 60
373
374     elif _add_20_points_menu__when == 4:
375
376         t_add_start = 4.9
377         t_add_stop   = 5.0
378         n_add = 100
379
380     elif _add_20_points_menu__when == 5:
381
382         t_add_start = 0.0
383         t_add_stop   = 5.0
384         n_add = 100
385
386
387     _t_exp_artificial, _F_exp_artificial =
388     add_fitted_points_to_experimental_data(t_add_start, t_add_stop,
389     n_add, _t_exp_opt, _F_exp_opt, T_opt_end, T,
390     kind_of_1d_interpolation)
391
392
393
394     return _t_exp_artificial, _F_exp_artificial
```

```
395 def choose_testing_method():
396
397     _menu_testing = 0
398
399     while _menu_testing < 1 or _menu_testing > 2:
400
401         print('')
402         print('-----')
403         print('1. Convergence analysis (TO BE IMPLEMENTED SOON)')
404         print('2. Relative error of drug mass m_d depending on')
405         print('spatial discretization')
406         print('')
407         print('-----')
408
409     _menu_testing = int(input())
410
411     if _menu_testing < 1 or _menu_testing > 2:
412         print('You did not choose between "1-2"!')
413         print('')
414
415 return _menu_testing
```

B.5 model.py

```
1 """
2 Diffusion equation with Robin boundary conditions.
3 Test problem is chosen to give an exact solution at all nodes of the
4 mesh.
5
6     u' = D*Laplace(u)      on Interval mesh
7 """
8
9 from fenics import *
10 from u_0 import define_u_t0, define_u_e
11 from boundary_conditions import h_modified
12 from degradation import degradation_model
13 from source_term import define_source_term
14 from surrounding_solution import calc_c_s, update_c_s_n,
15     transfer_c_s_n_from_EULER_to_CN
16 from solver_Euler import solver_Euler
17 from solver_CN_low import solver_CN_low
18 from solver_CN_high import solver_CN_high
19 from utility import check_u_t0, choose_F_num, calc_F,
20     adjust_dt_to_dt_fine, update_t_to_current_time
21 from save import collect_results, save_t_num_F_num,
22     calc_plot_print_L2_error, create__save_files, save_solution
23 from plot_funcs import plot_results, show_plot, plot__h_mod,
24     print__t_dt_num_steps
25 from plot_settings_inits import plot_settings
26
27 #set_log_active(False)          # mute console output
28 set_log_level(30)             # adjust level of console output (20
29     := solving linear var problem)
30
31
32 def run_model(D, alpha, l_b, A_m_degrad, const_degrad, degrad_break,
33     t, t_break, t_reached_5, T, T_explicit_Euler, dt,
34     c_sat, c_s_t0, c_max, V_s, V_s_units, V_c_units, r_i, r_o, l_c,
35     H, A_m_t0, m_d_t0, const_distribution, parabolic_distribution,
36     offset_parabola, U_0, U_1, beta, gamma, limiter_break,
37     limiter_sat,
38     V, mesh, nx, nr, ds, degree, r, n_facet, num_steps,
39     num_steps_explicit_Euler, F_num_komplex_n_Euler, runs_CN_high,
40     fineness, tol_adaptiv_dt, safety, p, boundary_tol,
41     menu, distribution_menu, verbose,
42     discretization_scheme, t_ref, c_ref, D_ref, l_ref, t_num, F_num)
43     :
44
45     # Misc
46     vtk_file, xdmf_file = create__save_files(menu, distribution_menu
47     )                                         # create save files
48     according to menu/verbose-mode user-choice
49     ax1, ax2, ax3, ax4, ax5, ax6 = plot_settings(c_max, c_sat,
```

```

33     m_d_t0, menu, discretization_scheme, verbose) # set plot
34     settings according to menu/verbose-mode user-choice
35     L2_error = []
36             # initialize list,
37             # that will save L2 errornorm for every time step, if convergence
38             # analysis is performed
39     F_num_choice = choose_F_num(menu) # TODO: besser implementieren
40
41     # Define t^0 values
42     u_0 = define_u_t0(t, T, r_i, r_o, distribution_menu,
43     const_distribution, parabolic_distribution, U_0, U_1,
44     offset_parabola, degree, discretization_scheme, menu) # define u
45     (t=0)
46     u_t0 = interpolate(u_0, V)
47             # interpolate u_0
48     check_u_t0(u_t0, m_d_t0, r, H, l_ref, c_ref, verbose, menu) #
49     check, if distribution-value is chosen correctly, that m_d_units
50     = 7.5g
51
52     # Define t^n values
53     u_n = interpolate(u_0, V) # u^n := u
54     (t=0)
55
56     # Define t^(n-1) values [DUMMY variables]
57     u_n_1 = interpolate(u_0, V) # DUMMY
58     variable
59     c_s_n_1 = c_s_t0 # DUMMY
60     variable
61     h_mod_n_1 = D # DUMMY
62     variable
63     F_num_komplex_n_1_Euler = 0 # DUMMY
64     variable
65
66     # Define manufactured solution
67     u_e = define_u_e(t, T, U_0, U_1, degree+3+1) #
68     manufactured solution/exact solution; degree=element_degree+
69     degree_rise+1 with degree_rise=3 (used for errornorm)
70
71
72     # -----
73     # Explicit E U L E R
74     # -----
75
76     # Initialize t^n Euler-values
77     c_s_n_Euler = calc_c_s(t, u_n_1, c_s_n_1, c_s_t0, h_mod_n_1, T,
78     dt, ds, H, gamma, U_0, U_1, r, r_i, r_o, degree, boundary_tol,
79     discretization_scheme, menu) # Konzentration der
80     Umgebungslösung
81     degrad_max_n_Euler = degradation_model(t, u_t0, u_n_1,
82     const_degrad, r_i, l_c, nx, nr, verbose, ax2[0], menu)
83             #
84     Matrixdegradation

```

```
60     h_mod_n_Euler = h_modified(t, D, alpha, l_b, c_s_n_Euler, c_sat,
61         c_max, r, A_m_t0, A_m_degrad, degrad_max_n_Euler, degrad_break,
62         U_1, T, discretization_scheme, degree, menu) # mod.
63         Stoffübergangsparameter h'
64     f_n_Euler = define_source_term(t, T, D, U_0, U_1, beta, degree,
65         discretization_scheme, menu)
66             # source term
67     # TODO: t=0 case implementieren
68     F_num_simpel_n_Euler, F_num_komplex_n_Euler = calc_F(dt, D,
69         u_n_1, c_s_n_Euler, F_num_komplex_n_1_Euler, c_max, V_s_units,
70         V_c_units, H, beta, gamma, r, n_facet, ds, discretization_scheme
71         , menu)
72
73     # EULER time stepping
74     while t < T_explicit_Euler - dt + (dt/100): # tol_dt = dt/100
75
76         # Update num step counter
77         num_steps += 1
78
79
80         # Compute solution
81         u_Euler = solver_Euler(t, dt, D, u_n, c_s_n_Euler,
82             h_mod_n_Euler, f_n_Euler, beta, V, r, ds,
83                 degree, linear_solver='direct')
84
85
86         # Update t to current time
87         t += dt
88
89         # Update analytical solution
90         u_e.t = t
91
92         # Calc & Plot/Print L2-error (only during convergence
93         analysis)
94         L2_error = calc_plot_print_L2_error(t, T_explicit_Euler, u_e
95         , u_Euler, L2_error, num_steps, num_steps_explicit_Euler, ax5,
96         verbose, discretization_scheme)
97
98         # Update t^n values for next timestep
99         # TODO: muss hier nicht die update_c_s_n-function genutzt
100        werden?
101        c_s_n_Euler = calc_c_s(t, u_n, c_s_n_Euler, c_s_t0,
102            h_mod_n_Euler, T, dt, ds, H, gamma, U_0, U_1, r, r_i, r_o,
103            degree, boundary_tol, discretization_scheme, menu)
104            degrad_max_n_Euler = degradation_model(t, u_t0, u_n,
105            const_degrad, r_i, l_c, nx, nr, verbose, ax2[0], menu)
106            h_mod_n_Euler = h_modified(t, D, alpha, l_b, c_s_n_Euler,
107            c_sat, c_max, r, A_m_t0, A_m_degrad, degrad_max_n_Euler,
108            degrad_break, U_1, T, discretization_scheme, degree, menu)
109            F_num_simpel_n_Euler, F_num_komplex_n_Euler = calc_F(dt, D,
110            u_n, c_s_n_Euler, F_num_komplex_n_Euler, c_max, V_s_units,
```

```

V_c_units, H, beta, gamma, r, n_facet, ds, discretization_scheme
, menu)
92     print__t_dt_num_steps(t, dt, num_steps, T_explicit_Euler,
menu, verbose)
93     f_n_Euler.t=t
94     u_n.assign(u_Euler)

95
96
97     # Save
98     save_solution(t, u_n, vtk_file, xdmf_file, menu)    # save to
vtk- and XDMF-file
99
100    # Plot
101    plot_results(t, T_explicit_Euler, m_d_t0, V_s, H, r, ax1,
ax2, ax3, ax6, verbose, discretization_scheme, menu,
102                  u_n, c_s_n_Euler, degrad_max_n_Euler,
F_num_simpel_n_Euler, F_num_komplex_n_Euler,
103                  0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
104
105
106
107 # -----
108 # Adaptive C R A N K - N I C O L S O N - time-stepping
109 # -----
110 # Initialize t^n CN-values by passing Euler values
111 c_s_n_CN_low = transfer_c_s_n_from_EULER_to_CN(c_s_n_Euler,
degree, menu)
112 c_s_n_CN_high = transfer_c_s_n_from_EULER_to_CN(c_s_n_Euler,
degree, menu)
113 h_mod_n_CN_low = h_mod_n_Euler
114 h_mod_n_CN_high = h_mod_n_Euler
115 f_n_CN_low = define_source_term(t, T, D, U_0, U_1, beta, degree
, discretization_scheme, menu)
116 f_n_CN_high = define_source_term(t, T, D, U_0, U_1, beta, degree
, discretization_scheme, menu)
117 F_num_komplex_n_CN_low = F_num_komplex_n_Euler
118 F_num_komplex_n_CN_high = F_num_komplex_n_Euler
119
120
121 # CN time stepping
122 num_steps_real = num_steps # only time steps, that conform to
Richardson interpolation
123 while t>=T_explicit_Euler-(dt/100) and t<T: # tol_dt = dt/100
124
125     # Update total step counter
126     num_steps += 1
127
128     # Adjust dt for special cases (t=15(end) and t=5(matrix
break))
129     if t+dt>T: # adjust last time-step, that t_num[last]=T
130         dt=T-t

```

```
131     elif t + dt > t_break and t_reached_5 == False and menu !=  
132     4: # adjust dt, that one exactly hits t=5 (not relevant for  
convergence analysis)  
133         dt = t_break - t  
134         t_reached_5 = True  
135  
136  
137     # Compute L O W precision solution  
138     u_CN_low, c_s_CN_low, degrad_max_CN_low, h_mod_CN_low,  
F_num_simpel_CN_low, F_num_komplex_CN_low = solver_CN_low(t, dt,  
u_n, c_s_n_CN_low, h_mod_n_CN_low, f_n_CN_low,  
F_num_komplex_n_CN_low,  
139                                         u_t0, c_sat  
, c_max, c_s_t0, V_s_units, V_c_units, r_i, r_o, H, l_c, A_m_t0,  
A_m_degrad, beta, gamma, U_0, U_1, T, const_degrad,  
degrad_break, degree, nx, nr, V, n_facet, r, ds, boundary_tol,  
verbose, ax2[0], discretization_scheme, menu,  
140                                         D, alpha,  
l_b,  
141                                         degree,  
linear_solver='direct')  
142     # Adjust dt to dt_fine  
143     dt = adjust_dt_to_dt_fine(dt, fineness,  
discretization_scheme) # dt = dt/finess [exception:  
convergence analysis for CN, then dt=dt]  
144  
145     # Compute H I G H precision solution  
146     u_CN_high, c_s_CN_high, degrad_max_CN_high, h_mod_CN_high,  
F_num_simpel_CN_high, F_num_komplex_CN_high = solver_CN_high(t,  
dt, u_n, c_s_n_CN_high, h_mod_n_CN_high, f_n_CN_high,  
F_num_komplex_n_CN_high,  
147                                         u_t0  
, c_sat, c_max, c_s_t0, V_s_units, V_c_units, r_i, r_o, H, l_c,  
A_m_t0, A_m_degrad, beta, gamma, U_0, U_1, T, const_degrad,  
degrad_break, runs_CN_high, degree, nx, nr, V, n_facet, r, ds,  
boundary_tol, verbose, ax2[0], discretization_scheme, menu,  
148                                         D,  
alpha, l_b,  
149                                         degree, linear_solver='direct')  
150  
151     # Update t to current time  
152     t = update_t_to_current_time(t, dt, fineness,
```

```

153
154
155     # Adaptive CN-time-stepping
156     richardson = sqrt(assemble(((u_CN_high - u_CN_low)**2)*dx))
157     / (2**p - 1) # compute Richardson extrapolation
158
159     if richardson < tol_adaptiv_dt and (discretization_scheme != 2 and
160     discretization_scheme != 3):
161
162         # Update step counter, if Richardson extrapolation is
163         passed
164         num_steps_real += 1
165
166         # Update dt according to Richardson extrapolation
167         dt = ((safety * tol_adaptiv_dt / richardson) ** (1 / p)) *
168         dt
169
170         # dt limiter
171         if dt > 0.125: # if dt grows too fast, reduce dt to dt
172             = 0.25
173             dt = 0.125
174
175         # Print dt
176         print_t_dt_num_steps(t, dt, num_steps_real,
177         T_explicit_Euler, menu, verbose)
178
179
180         # Update analytical solution
181         u_e.t = t
182
183         # Reset dt to dt_coarse
184         dt = fineness * dt
185
186
187         # Update t^n values for next timestep (CN_low values are
188         updated with CN_high results, because dt conformed to
189         Richardson interpolation)
190         c_s_n_CN_low = update_c_s_n(t, c_s_CN_high, degree,
191         menu)
192         c_s_n_CN_high = update_c_s_n(t, c_s_CN_high, degree,
193         menu)
194         h_mod_n_CN_low = h_mod_CN_high
195         h_mod_n_CN_high = h_mod_CN_high
196         f_n_CN_low.t=t
197         f_n_CN_high.t=t
198         F_num_komplex_n_CN_low = F_num_komplex_CN_high
199         F_num_komplex_n_CN_high = F_num_komplex_CN_high
200         u_n.assign(u_CN_high)
201
202

```

```
192     # Save
193     save_solution(t, u_n, vtk_file, xdmf_file, menu)  # save
194     to vtk- and XDMF-file
195
196     t_num, F_num = save_t_num_F_num(t, t_num, F_num,
197     F_num_simpel_CN_high, F_num_komplex_CN_high, F_num_choice)
198
199     # Plot
200     limiter_break, limiter_sat = plot__h_mod(t,
201     h_mod_CN_high, degrad_max_CN_high, degrad_break, alpha, c_sat,
202     c_s_CN_high, limiter_break, limiter_sat, ax4, verbose, menu,
203     discretization_scheme)
204     plot_results(t, T_explicit_Euler, m_d_t0, V_s, H, r, ax1
205     , ax2, ax3, ax6, verbose, discretization_scheme, menu,
206     0, 0, 0, 0,
207     u_CN_low, c_s_CN_low, degrad_max_CN_low,
208     F_num_simpel_CN_low, F_num_komplex_CN_low,
209     u_CN_high, c_s_CN_high, degrad_max_CN_high,
210     F_num_simpel_CN_high, F_num_komplex_CN_high)
211
212
213     elif richardson > tol_adaptiv_dt and (discretization_scheme
214     != 2 and discretization_scheme != 3):
215
216         t -= fineness*dt # t wieder zurücksetzen, um wieder
217         einen neuen Berechnungsschritt für selben Zeitraum durchzuführen
218         , aber diesmal mit feinerem dt
219         dt = dt/10      # adjust dt to higher precision/smaller
220         timestep
221
222
223     elif discretization_scheme == 2: # only active during
224     convergence analysis (CN low precision)
225
226         # Update
227         num_steps_real += 1 # step counter
228         u_e.t = t           # analytical solution
229
230         # Calc & plot/print L2-errornorm (convergence analysis)
231         L2_error = calc_plot_print_L2_error(t, T_explicit_Euler,
232         u_e, u_CN_low, L2_error, num_steps_real,
233         num_steps_explicit_Euler, ax5, verbose, discretization_scheme)
234
235         # Update CN low variables for next timestep
236         c_s_n_CN_low = update_c_s_n(t, c_s_CN_low, degree, menu)
237         h_mod_n_CN_low = h_mod_CN_low
238         f_n_CN_low.t = t
239         u_n.assign(u_CN_low)
```

```
228
229     elif discretization_scheme == 3: # only active during
convergence analysis (CN high precision)
230
231         # Update
232         num_steps_real += 1 # step counter
233         u_e.t = t           # analytical solution
234
235         # Calc & plot/print L2-errornorm (convergence analysis)
236         L2_error = calc_plot_print_L2_error(t, T_explicit_Euler,
u_e, u_CN_high, L2_error, num_steps_real,
num_steps_explicit_Euler, ax5, verbose, discretization_scheme)
237
238         # Update CN high variables for next timestep
239         c_s_n_CN_high = update_c_s_n(t, c_s_CN_high, degree,
menu)
240         h_mod_n_CN_high = h_mod_CN_high
241         f_n_CN_high.t = t
242         u_n.assign(u_CN_high)
243
244         # Reset dt to dt_coarse
245         dt = fineness * dt
246
247
248
249
250     show_plot(menu) # display final plot per optimization step for 5
sec
251     results = collect_results(t_num, F_num, L2_error, menu,
distribution_menu) # collect results depending on menu choice
252
253     return results
```

B.6 mesh_domains_boundaries.py

```
1 # -----
2 # M E S H , D O M A I N S & B O U N D A R I E S
3 # -----
4 from fenics import *
5
6 def define_mesh(nx, r_i, r_o, element_degree, element_family):
7     """ Mesh is defined by number of knots, left & right boundary
8        coordinate, element family & degree. """
9
10    # Create mesh and define function space
11    mesh = IntervalMesh(nx, r_i, r_o)
12    space_dim = mesh.geometry().dim()
13    V = FunctionSpace(mesh, element_family, element_degree)
14
15    return mesh, space_dim, V
16
17
18
19 def define_boundaries(mesh, space_dim):
20     """ Boundaries are defined. """
21
22     # Initialize mesh function for boundary domains
23     boundaries = MeshFunction("size_t", mesh, space_dim - 1)
24     boundaries.set_all(0)
25
26     # Define new measures associated with the exterior boundaries
27     ds = Measure('ds', domain=mesh, subdomain_data=boundaries)
28
29
30     return ds
31
32
33
34 def define_r(element_degree):
35     """ r defines 1d-cylinder coordinates. """
36
37     return Expression('x[0]', degree=element_degree)
38
39
40
41 def collect_mesh_parameters(r_i, r_o, nx, element_degree,
42     element_family, boundary_tol):
43
44     _mesh, _space_dim, _V = define_mesh(nx, r_i, r_o, element_degree,
45     element_family)
46     _ds = define_boundaries(_mesh, _space_dim)
47     _r = define_r(element_degree)
48     _n_facet = FacetNormal(_mesh) # define normal vector
```

```
47  
48  
49     return _mesh, _V, _ds, _r, _n_facet
```

B.7 boundary_conditions.py

```
1 # -----
2 # B O U N D A R Y   C O N D I T I O N S
3 # -----
4 from fenics import Expression
5
6
7 def h_modified(t, D, alpha, l_b, c_s, c_sat, c_max, r, A_m_t0,
8     A_m_degrad, degrad_max, degrad_break, U_1, T,
9     discretization_scheme, degree, menu):
10
11    if menu == 1 or menu == 2:
12
13        if t > 0:
14            if alpha * (c_sat - c_s.val) >= 1: # "unsaturated"
15
16                if degrad_max > degrad_break: # matrix breaks
17                    print('M A T R I X B R U C H')
18                    return (A_m_degrad/A_m_t0)*(D/l_b)
19
20            else:
21                return D/l_b
22
23
24        elif alpha * (c_sat - c_s.val) < 1: # "saturated"
25
26            if degrad_max > degrad_break: # matrix breaks
27                print('S Ä T T I G U N G + M A T R I X B R U C H')
28                return (A_m_degrad/A_m_t0) * (D/l_b) * alpha * (
29                    c_sat - c_s.val)
30
31            else:
32                return (D/l_b) * alpha * (c_sat - c_s.val)
33                print('S Ä T T I G U N G')
34
35
36    else:
37        return D/l_b
38
39
40    elif menu == 4:
41
42        return Expression('D*U_1*t*cos(r)/T', t=t, T=T, D=D, U_1=U_1
43        , r=r, degree=degree)
```

B.8 u_0.py

```

1 # -----
2 # u (t=0)
3 # -----
4 from fenics import Expression
5
6
7 def define_u_t0(t, T, r_i, r_o, distribution_menu,
8     const_distribution, parabolic_distribution, U_0, U_1,
9     offset_parabola, degree, discretization_scheme, menu):
10    """ Define initial drug distribution of the matrix. """
11
12    # Define initial drug distribution with real parameters
13    if menu == 1 or menu == 2 or ((menu == 3 or menu == 4) and (
14        discretization_scheme == 6 or discretization_scheme == 7)) or
15        menu == 5:
16
17        # Constant, initial drug distribution
18        if distribution_menu == 0:
19            # constant
20            distribution_of_u_t0 =
21                _u_0 = Expression('const_distribution',
22                    const_distribution=const_distribution, degree=degree)
23
24        # Parabolic, initial drug distribution
25        elif distribution_menu == 1:
26            _u_0 = Expression("parabolic_distribution * (x[0] - ((r_i + r_o) / 2.0)) * (x[0] - ((r_i + r_o) / 2.0))" # parabolic
27            distribution_of_u_t0 =
28                " + offset_parabola",
29                parabolic_distribution=
30                    parabolic_distribution, r_i=r_i, r_o=r_o, offset_parabola=
31                    offset_parabola, degree=degree)
32
33        return _u_0
34
35    # Define initial drug distribution for the convergence analysis
36    # based on the manufactured solution (MMS)
37    elif ((menu == 3 or menu == 4) and (discretization_scheme != 6
38        or discretization_scheme != 7)):
39
40        _u_0 = Expression('U_0 - U_1*t*sin(x[0])/T', U_0=U_0, U_1=
41            U_1, t=t, T=T, degree=degree)
42
43    return _u_0
44
45
46
47 def define_u_e(t, T, U_0, U_1, degree_u_e):

```

```
35     """ Define the manufactured solution (MMS), To be used for the
36     convergence analysis. """
37
38     _u_e = Expression('U_0 - U_1*t*sin(x[0])/T', U_0=U_0, U_1=U_1, t
39     =t, T=T, degree=degree_u_e)
40
41     return _u_e
```

B.9 source_term.py

```
1 # -----
2 # S O U R C E      T E R M
3 # -----
4 from fenics import *
5
6 def define_source_term(t, T, D, U_0, U_1, beta, degree,
7   discretization_scheme, menu):
8   """ Source term, that is only !=0, if the convergence analysis
9   is active. Then the source term is used as a forcing function.
10  """
11
12  # Source term is set to zero, because diffusion is described by a
13  # homogeneous diffusion equation
14  if menu == 1 or menu == 2 or ((menu == 3 or menu == 4) and (
15    discretization_scheme == 6 or discretization_scheme == 7)):
16
17    return Expression('0', degree=degree)
18
19  # Source term is used as a forcing function for the convergence
20  # analysis
21  elif ((menu == 3 or menu == 4) and (discretization_scheme != 6
22    or discretization_scheme != 7)):
23
24    return Expression('-U_1 * sin(x[0]) / T + beta*D*U_1*t*((cos(
25      (x[0])/x[0])-sin(x[0]))/T', t=t, T=T, U_0=U_0, U_1=U_1, beta=
26      beta, D=D, degree=2)
```

B.10 surrounding_solution.py

```
1 # -----
2 # S U R R O U N D I N G   S O L U T I O N
3 # -----
4 from fenics import *
5 import numpy as np
6
7
8 def calc_c_s(t, u_n, c_s_n, c_s_t0, h_mod_n, T_W, dt, ds, H, gamma,
9     U_0, U_1, r, r_i, r_o, degree, boundary_tol,
10    discretization_scheme, menu):
11    """ Calc the surrounding solution concentration. """
12    if menu == 1 or menu == 2:
13
14        _c_s_n = c_s_SIM(t, u_n, c_s_n, c_s_t0, h_mod_n, dt, ds, H,
15                           gamma, r, degree)
16
17    elif menu == 4:
18
19        _c_s_n = C_s_MMS(t, T_W, U_0, U_1, r_i, r_o, boundary_tol)
20
21
22
23 def c_s_SIM(t, u_n, c_s_n, c_s_t0, h_mod_n, dt, ds, H, gamma, r,
24     degree):
25    """ Calc the surrounding solution concentration for the
26    simulation. Real parameters are used. """
27
28    if t > 0:
29
30        _c_s_val = c_s_n.val + dt * 2.0 * np.pi * H * gamma *
31        h_mod_n * assemble((u_n - c_s_n.val) * r * ds)
32
33        return Expression("val", degree=degree, val=_c_s_val)
34
35
36
37 class C_s_MMS(UserExpression):
38    """ Calc the surrounding solution concentration for the
39    convergence analysis. The manufactured solution (MMS) is used.
40    """
41
42    def __init__(self, t, T_W, U_0, U_1, r_i, r_o, boundary_tol, **
43                 kwargs):
```

```

41     super().__init__(**kwargs)
42     self.t = t
43     self.T_W = T_W
44     self.r_i = r_i
45     self.r_o = r_o
46     self.U_0 = U_0
47     self.U_1 = U_1
48     self.boundary_tol = boundary_tol
49
50
51 #TODO: c_s.update(t,...) oder c_s.t=t, c_s.u=u, ... verwenden?
52 def update(self, t):
53     self.t = t
54
55
56 def eval(self, values, x):
57
58     if near(x[0], self.r_i, self.boundary_tol):
59
60         values[0] = self.U_0 - self.U_1 * self.t * sin(self.r_i) /
61         self.T_W + 1.0
62
63     elif near(x[0], self.r_o, self.boundary_tol):
64
65         values[0] = self.U_0 - self.U_1 * self.t * sin(self.r_o) /
66         self.T_W - 1.0
67
68     else:
69         values[0] = 0
70
71
72     def value_shape(self):
73         return ()
74
75
76 def update_c_s_n(t, c_s, degree, menu):
77
78     if menu == 1 or menu == 2:
79
80         _c_s_n = Expression("val", degree=degree, val=c_s.val) #
81         Konzentration der Umgebungslösung
82
83     elif menu == 4:
84
85         c_s.t = t      # update c_s-Expression
86         _c_s_n = c_s # pass updated c_s-Expression to internal
87         _c_s_n variable, which will be used for the next time-step

```

```
88     return _c_s_n
89
90
91
92 def transfer_c_s_n_from_EULER_to_CN(c_s_n_EULER, degree, menu):
93
94     if menu == 1 or menu == 2:
95
96         _c_s_n_CN = Expression("val", degree=degree, val=c_s_n_EULER
97             .val) # Konzentration der Umgebungslösung
98
99     elif menu == 4:
100
101         _c_s_n_CN = c_s_n_EULER
102
103     return _c_s_n_CN
```

B.11 degradation.py

```

1 # -----
2 # D E G R A D A T I O N   M O D E L
3 # -----
4 import numpy as np
5 from plot_funcs import plot_degrad
6
7
8 def degradation_model(t, u_t0, u_n, const_degrad, r_i, l_c, nx, nr,
9   verbose, ax2, menu):
10  """
11    If the parabolic distribution of u(t=0) is chosen and the drug
12    diffuses to the middle of the matrix,
13    so u_t0<u_n applies, _degrad[i] is set to zero at these
14    positions i.
15  """
16
17  # Reset _degrad
18  _degrad = [0] * (nr + 1)
19
20  # Calc degradation
21  for i in range(nr + 1):
22
23    r_coord = r_i + i * (l_c / nr)
24
25    if u_n(r_coord) > u_t0(r_coord):
26      _degrad[i] = 0.0
27
28    else:
29      _degrad[i] = 1.0-np.exp(-const_degrad*((u_t0(r_coord)-
30        u_n(r_coord))/u_t0(r_coord))*( (u_t0(r_coord)-u_n(r_coord))/u_t0(
31        r_coord))))
32
33  # Calc minimum of degradation
34  _degrad_min_no_zeros = np.min(_degrad[_degrad != 0])
35  _degrad_min = np.min(_degrad)
36
37  # Plot degradation
38  if (menu == 1 or menu == 2) and verbose == 2:
39    plot_degrad(_degrad, nr, r_i, l_c, verbose, ax2, menu)  #
40    # activate for debugging
41
42  return _degrad_min_no_zeros

```

B.12 solver_Euler.py

```
1 # -----
2 # solver EULER
3 # -----
4 from fenics import *
5
6
7 def solver_Euler(t, dt, D, u_n, c_s_n_Euler, h_mod_n_Euler,
8     f_n_Euler,
9         beta, V, r, ds,
10        element_degree=1,
11        linear_solver='Krylov',
12        abs_tol=1E-5,
13        rel_tol=1E-3,
14        max_iter=1000):
15
16     """ Solve u' = D*Laplace(u) on the line interval [r_i, r_o] with
17     Lagrange elements of specified degree and Robin boundary
18     conditions by using the explicit EULER-method. """
19
20     # Set linear solver parameters
21     prm = LinearVariationalSolver.default_parameters()
22     if linear_solver == 'Krylov':
23         prm["linear_solver"] = 'gmres'
24         prm["preconditioner"] = 'ilu'
25         prm["krylov_solver"]["absolute_tolerance"] = abs_tol
26         prm["krylov_solver"]["relative_tolerance"] = rel_tol
27         prm["krylov_solver"]["maximum_iterations"] = max_iter
28     else:
29         prm["linear_solver"] = 'lu'
30
31     # Initialize u_n
32     _u_n_Euler = Function(V)
33     _u_n_Euler.assign(u_n)
34
35     # Define trial & test function
36     _u_Euler = TrialFunction(V)
37     v_Euler = TestFunction(V)
38
39     # Weak Form
40     F = (_u_Euler - _u_n_Euler) * v_Euler * r * dx \
41         + dt * beta * h_mod_n_Euler * (_u_n_Euler - c_s_n_Euler) * \
42             v_Euler * r * ds \
43             + dt * beta * D * Dx(_u_n_Euler, 0) * Dx(v_Euler, 0) * r * dx \
44             \
45             - dt * f_n_Euler * v_Euler * r * dx
46
47     a, L = lhs(F), rhs(F)
```

```
45
46     # Compute solution
47     _u_Euler = Function(V)
48     solve(a == L, _u_Euler, solver_parameters=prm)
49
50
51     return _u_Euler
```

B.13 solver_CN_low.py

```
1 # -----
2 # solver CN (low precision/big timestep)
3 # -----
4 from fenics import *
5 from boundary_conditions import h_modified
6 from degradation import degradation_model
7 from source_term import define_source_term
8 from surrounding_solution import calc_c_s
9 from utility import calc_F
10
11
12 def solver_CN_low(t, dt, u_n, c_s_n_CN_low, h_mod_n_CN_low,
13     f_n_CN_low, F_num_komplex_n_CN_low,
14     u_t0, c_sat, c_max, c_s_t0, V_s_units, V_c_units,
15     r_i, r_o, H, l_c, A_m_t0, A_m_degrad, beta, gamma, U_0, U_1, T,
16     const_degrad, degrad_break, degree, nx, nr, V, n_facet, r, ds,
17     boundary_tol, verbose, ax2, discretization_scheme, menu,
18     D, alpha, l_b,
19     element_degree=1,
20     linear_solver='Krylov',
21     abs_tol=1E-5,
22     rel_tol=1E-3,
23     max_iter=10000):
24
25     """ Solve u' = D*Laplace(u) on the line interval [r_i, r_o] with
26     Lagrange elements of specified degree and Robin boundary
27     conditions by using a low precision CRANK-NICOLSON-method. """
28
29     # Set linear solver parameters
30     prm = LinearVariationalSolver.default_parameters()
31     if linear_solver == 'Krylov':
32         prm["linear_solver"] = 'gmres'
33         prm["preconditioner"] = 'ilu'
34         prm["krylov_solver"]["absolute_tolerance"] = abs_tol
35         prm["krylov_solver"]["relative_tolerance"] = rel_tol
36         prm["krylov_solver"]["maximum_iterations"] = max_iter
37     else:
38         prm["linear_solver"] = 'lu'
39
40     # Initialize u_n
41     _u_n_CN_low= Function(V)
42     _u_n_CN_low.assign(u_n)
43
44     # Update time
45     t += dt
46
47     # Initialize & update t^(n+1) variables
48     c_s_CN_low = calc_c_s(t, _u_n_CN_low, c_s_n_CN_low, c_s_t0,
```

```

h_mod_n_CN_low, T, dt, ds, H, gamma, U_0, U_1, r, r_i, r_o,
degree, boundary_tol, discretization_scheme, menu) #
Konzentration der Umgebungslösung
44 degrad_max_CN_low = degradation_model(t, u_t0, _u_n_CN_low,
const_degrad, r_i, l_c, nx, nr, verbose, ax2, menu)
45 h_mod_CN_low = h_modified(t, D, alpha, l_b, c_s_CN_low, c_sat,
c_max, r, A_m_t0, A_m_degrad, degrad_max_CN_low, degrad_break,
U_1, T, discretization_scheme, degree, menu)
46 f_CN_low = define_source_term(t, T, D, U_0, U_1, beta, degree,
discretization_scheme, menu)
47
48
49 # Define trial & test function
50 _u_CN_low = TrialFunction(V)
51 v_CN_low = TestFunction(V)
52
53 # Weak form
54 F = r * (_u_CN_low - _u_n_CN_low) * v_CN_low * dx \
55 + 0.5 * beta * dt * D * r * (Dx(_u_CN_low, 0) + Dx(_u_n_CN_low
, 0)) * Dx(v_CN_low, 0) * dx \
56 + 0.5 * beta * dt * (h_mod_CN_low * (_u_CN_low - c_s_CN_low) +
h_mod_n_CN_low * (_u_n_CN_low - c_s_n_CN_low)) * v_CN_low * r *
ds \
57 - 0.5 * dt * (f_CN_low + f_n_CN_low) * v_CN_low * r * dx
58
59 a, L = lhs(F), rhs(F)
60
61 # Compute solution
62 _u_CN_low = Function(V)
63 solve(a == L, _u_CN_low, solver_parameters=prm)
64
65
66 # Calc F_num
67 # TODO: 2.) ueberpruefen, ob _F_num_komplex_CN_low mit u oder
u_n berechnet?
68 F_num_simpel_CN_low, F_num_komplex_CN_low = calc_F(dt, D,
_u_n_CN_low, c_s_CN_low, F_num_komplex_n_CN_low, c_max,
V_s_units, V_c_units, H, beta, gamma, r, n_facet, ds,
discretization_scheme, menu)
69
70
71 return _u_CN_low, c_s_CN_low, degrad_max_CN_low, h_mod_CN_low,
F_num_simpel_CN_low, F_num_komplex_CN_low

```

B.14 solver_CN_high.py

```
1 # -----
2 # solver CN (high precision/small timestep)
3 # -----
4 from fenics import *
5 from boundary_conditions import h_modified
6 from degradation import degradation_model
7 from source_term import define_source_term
8 from surrounding_solution import calc_c_s, update_c_s_n
9 from utility import calc_F
10
11
12 def solver_CN_high(t, dt, u_n, c_s_n_CN_high, h_mod_n_CN_high,
13     f_n_CN_high, F_num_komplex_n_CN_high,
14     u_t0, c_sat, c_max, c_s_t0, V_s_units, V_c_units,
15     r_i, r_o, H, l_c, A_m_t0, A_m_degrad, beta, gamma, U_0, U_1, T,
16     const_degrad, degrad_break, runs_CN_high, degree, nx, nr, V,
17     n_facet, r, ds, boundary_tol, verbose, ax2,
18     discretization_scheme, menu,
19     D, alpha, l_b,
20     element_degree=1,
21     linear_solver='Krylov',
22     abs_tol=1E-5,
23     rel_tol=1E-3,
24     max_iter=10000):
25
26     """ Solve u' = D*Laplace(u) on the line interval [r_i, r_o] with
27     Lagrange elements of specified degree and Robin boundary
28     conditions by using a high precision CRANK-NICOLSON-method. """
29
30     # Set linear solver parameters
31     prm = LinearVariationalSolver.default_parameters()
32     if linear_solver == 'Krylov':
33         prm["linear_solver"] = 'gmres'
34         prm["preconditioner"] = 'ilu'
35         prm["krylov_solver"]["absolute_tolerance"] = abs_tol
36         prm["krylov_solver"]["relative_tolerance"] = rel_tol
37         prm["krylov_solver"]["maximum_iterations"] = max_iter
38     else:
39         prm["linear_solver"] = 'lu'
40
41
42     # Initialize u_n
43     _u_n_CN_high = Function(V)
44     _u_n_CN_high.assign(u_n)
45
46     # Time stepping
47     for n in range(runs_CN_high):
48
49         # Update time
```

```

43     t += dt
44
45     # Initialize & update t^(n+1) values
46     c_s_CN_high = calc_c_s(t, _u_n_CN_high, c_s_n_CN_high,
47     c_s_t0, h_mod_n_CN_high, T, dt, ds, H, gamma, U_0, U_1, r, r_i,
48     r_o, degree, boundary_tol, discretization_scheme, menu)
49     degrad_max_CN_high = degradation_model(t, u_t0, _u_n_CN_high
50     , const_degrad, r_i, l_c, nx, nr, verbose, ax2, menu)
51     h_mod_CN_high = h_modified(t, D, alpha, l_b, c_s_CN_high,
52     c_sat, c_max, r, A_m_t0, A_m_degrad, degrad_max_CN_high,
53     degrad_break, U_1, T, discretization_scheme, degree, menu)
54     f_CN_high = define_source_term(t, T, D, U_0, U_1, beta,
55     degree, discretization_scheme, menu)
56     F_num_simpel_CN_high, F_num_komplex_CN_high = calc_F(dt, D,
57     _u_n_CN_high, c_s_CN_high, F_num_komplex_n_CN_high, c_max,
58     V_s_units, V_c_units, H, beta, gamma, r, n_facet, ds,
59     discretization_scheme, menu)
60
61     # Define trial & test function
62     _u_CN_high = TrialFunction(V)
63     v_CN_high = TestFunction(V)
64
65     # Weak form
66     F = (_u_CN_high - _u_n_CN_high) * v_CN_high * r * dx \
67         + 0.5 * dt * beta * (h_mod_CN_high * (_u_CN_high -
68         c_s_CN_high) + h_mod_n_CN_high * (_u_n_CN_high - c_s_n_CN_high))
69         * v_CN_high * r * ds \
70         + 0.5 * dt * beta * D * (Dx(_u_CN_high, 0) + Dx(
71         _u_n_CN_high, 0)) * Dx(v_CN_high, 0) * r * dx \
72         - 0.5 * dt * (f_CN_high + f_n_CN_high) * v_CN_high * r *
73         dx
74
75     a, L = lhs(F), rhs(F)
76
77     # Compute solution
78     _u_CN_high = Function(V)
79     solve(a == L, _u_CN_high, solver_parameters=prm)
80
81
82     # Update t^n values for next timestep
83     c_s_n_CN_high = update_c_s_n(t, c_s_CN_high, degree, menu)
84     h_mod_n_CN_high = h_mod_CN_high
85     f_n_CN_high.t = t
86     F_num_komplex_n_CN_high = F_num_komplex_CN_high
87     _u_n_CN_high.assign(_u_CN_high)
88
89
90     return _u_CN_high, c_s_CN_high, degrad_max_CN_high,
91     h_mod_CN_high, F_num_simpel_CN_high, F_num_komplex_CN_high

```

B.15 utility.py

```
1 # -----
2 # U T I L I T Y   F U N C T I O N S
3 # -----
4 from fenics import *
5 from scipy.interpolate import interp1d
6 import numpy as np
7 import time
8 import sys
9
10
11 def check_CFL_conditions(dt, D, mesh, verbose, discretization_scheme,
12     , menu):
13     """ Checks, if  $CFL = D * dt/dx < 1$  is met and reduces dt by  $dt=dt/5$ 
14     till condition is met. """
15
16     _dt=dt
17     element_size = mesh.hmin()
18     CFL = D * _dt;element_size
19
20     if menu == 1 or menu == 2:
21
22         while CFL > 1.0:
23
24             if CFL > 1.0:
25
26                 if verbose==1 or verbose==2:
27
28                     print("#####")
29                     print(" # ! #     $CFL = D * dt/dx > 1$       (CFL -
30 Bedingungen nicht erfüllt)")
31                     print(" # #")
32                     print(" #")
33                     print('-----')
34
35             _dt=_dt/5
36
37     else:
38
39         _dt=_dt
40
41         if verbose == 1 or verbose == 2:
42             print("CFL conditions satisfied:")
43             print("CFL = ", CFL)
44             print("dx = ", element_size)
45             print('')
46
47     return _dt
```

```
46
47
48     elif menu == 4 and (discretization_scheme != 2 and
49     discretization_scheme != 3):
50
51         if CFL > 1.0:
52
53             print("#####")
54             print(" # ! #   CFL = D * dt/dx > 1      (CFL -
55             Bedingungen nicht erfüllt)")
56             print(" # #")
57             print(" # ")
58             print('
59             -----')
60
61             sys.exit("Choose dt, that it will conform to CFL-
62             conditions!")
63
64
65     else:
66
67         if verbose == 1 or verbose == 2:
68
69             print("CFL conditions satisfied")
70
71         return _dt
72
73
74
75 def check_u_t0(u_t0, m_d_t0, r, H, l_ref, c_ref, verbose, menu):
76     """ Checks, if distribution is chosen correctly, that m_d_units
77     = 7.5g. """
78
79     if (menu == 1 and (verbose == 1 or verbose == 2)) or (menu == 2
80     and verbose == 2) or menu == 5:
81
82         # TODO: use m_d_t0_units from parameters_units.py
83         m_d_t0_units           = 0.0075
84         # [g]      - Gesamtmasse GM, die in die Matrix
85         # eingebracht wurde (m_d = 7.5 mg)
86
87         # Calc m_d_test_units [g]
88         _m_d_test_units = 2.0 * np.pi * H * assemble(u_t0 * r * dx)
89         * (c_ref * l_ref ** 3)
```

```

87     # Calc rel error [%]
88     _rel_err_percent = (_m_d_test_units - m_d_t0_units) /
89     m_d_t0_units) * 100
90
91     if menu == 5:
92
93         return _rel_err_percent
94
95
96
97 def choose_F_num(menu):
98
99     if menu == 1:
100
101         _F_num_option = 0 # initialize menu with dummy variables
102
103         while _F_num_option != 1 and _F_num_option != 2:
104
105             print('1. Using c_s to calculate F_num')
106             print('2. Usin F_num^complex')
107             print('')
108
109             _F_num_option = int(input())
110             print('')
111
112             if _F_num_option != 1 and _F_num_option != 2:
113                 print('You did not choose "1" or "2"!')
114                 print('')
115
116
117         return _F_num_option
118
119
120     elif menu == 2 or menu == 4:
121
122         _F_num_option = 2
123
124         # TODO: später wieder aktivieren, nur zur besseren
125         Lesbarkeit bei der Optimierung deaktiviert
126         """
127         if menu == 2:
128             print('')
129             print('T O - D O: function "choose_F_num" muss für die
130             Konvergenzanalyse noch sauber programmiert werden. Aktuell wird
131             "_F_num_option = 1" automatisch gesetzt.')
132             print('')
133             """
134
135
136         return _F_num_option

```

```
134
135
136 def calc_F(dt, D, u_n, c_s, F_num_komplex_n, c_max, V_s_units,
137   V_c_units, H, beta, gamma, r, n_facet, ds, discretization_scheme
138   , menu):
139   """ Calc drug release value F_num. """
140
141   _F_num_komplex = F_num_komplex_n - dt * ((2.0 * np.pi * H *
142     gamma) / c_max) * assemble(D * inner(grad(u_n), n_facet) * r *
143     ds)
144
145   if menu == 1 or menu == 2:
146     _F_num_simpel = c_s.val / c_max
147
148   elif menu == 4:
149     _F_num_simpel = c_s / c_max
150
151
152 def calc_F_num_at_discrete_time_steps (_tnum, _Fnum, _t_exp, T,
153   kind_of_1d_interpolation):
154   """Calc _F_num at discrete time steps (these are the time points
155   , where measurements were taken during the experiment)"""
156
157   # Initialize
158   _F_num_at_discrete_points = []
159
160   # Interpolate _F_num
161   _F_num_interpolated_1d = interp1d(_tnum, _Fnum,
162     kind_of_1d_interpolation)
163
164   # count user specified week i
165   _t_i_count = _t_exp.index(T) + 1
166
167   # Optimization until user specified week i
168   for i in range(_t_i_count):
169     _F_num_at_discrete_points += [_F_num_interpolated_1d(_t_exp[
170       i])]
171
172
173 def adjust_t_exp_and_F_exp_to_optimization_time_frame(t_exp, F_exp,
174   T_opt_end):
175   # Adjust t_exp & F_exp to optimization time frame (t = 0-5)
```

```
176     t_i_count = t_exp.index(T_opt_end) + 1
177
178     _t_exp_opt = []
179     _F_exp_opt = []
180
181     for i in range(t_i_count):
182         _t_exp_opt += [t_exp[i]]
183         _F_exp_opt += [F_exp[i]]
184
185
186     return _t_exp_opt, _F_exp_opt
187
188
189
190 def adjust_dt_to_dt_fine(dt, fineness, discretization_scheme):
191     """ Changes time step from coarse to fine. """
192
193     # Disabled for convergence analysis
194     if discretization_scheme == 2:
195         return dt
196
197     # Reduce time step
198     else:
199         return dt / fineness
200
201
202
203 def update_t_to_current_time(t, dt, fineness, discretization_scheme):
204     :
205
206     _t=t # initialize internal variable
207
208     if discretization_scheme == 2:
209         _t += dt
210
211     else:
212         _t += fineness * dt
213
214
215
216
217
218 def compute_convergence_rate(L2_error_avg_list, degree, nx, nt, h,
219     space_or_time):
220     """ Compute convergence rate. """
221
222     if space_or_time == 1:
223         for i in range(len(nx)):
224             h += [1 / nx[i]]
```

```
225     elif space_or_time == 2:
226         for i in range(len(nt)):
227             h += [1 / nt[i]]
228
229 # rate = [ [[],[],[],[]], [[],[[],[],[],[]]], [[[],[],[],[]]] ]
230 rate = [[[[] for _ in range(len(h) - 1)] for _ in range(len(
231 degree))]]
232
233     for l in range(len(degree)):
234
235         for m in range(len(h) - 1):
236             rate[l][m] += [np.log(L2_error_avg_list[l][m + 1] /
237 L2_error_avg_list[l][m]) / np.log(h[m + 1] / h[m])]
238
239
240
241
242 def check_cost_function(F_exp, _F_num_at_discrete_points, _weight,
243 verbose):
244     """ Checks cost function during parameter optimization. """
245
246     if verbose == 2:
247
248         # Initialize weighted squared residuals
249         _weighted_squared_residuals = []
250
251         # Calc weighted squared residuals
252         for k in range(len(F_exp)):
253             _weighted_squared_residuals.append(_weight[k] * (F_exp[
254 k] - _F_num_at_discrete_points[k]))**2)
255
256         # Calc cost function
257         _cost_function = 0.5 * sum(_weighted_squared_residuals)
258
259         # Print cost function
260         print('cost function (Check) = %.4e' % _cost_function)
261
262
263 def add_fitted_points_to_experimental_data(t_add_start, t_add_end,
264 n_add, t_exp, F_exp, T_opt_end, T, kind_of_1d_interpolation):
265
266     # F_exp interpolieren
267     _F_exp_interp1d = interp1d(t_exp, F_exp,
268     kind_of_1d_interpolation)
269     if t_add_start == 3.0:
270         #n_add = 20
271         n_plot = int(((T/(t_add_end - t_add_start))*n_add) + 1)
```

```
269     #n_plot = 151           # 5-3 = 2      ; 15/2 = 7.5  ;
270     7.5*20 +1 = 151
271     elif t_add_start == 4.9:
272         #n_add = 20
273         n_plot = int(((T/(t_add_end - t_add_start))*n_add) + 1)
274         #n_plot = 3001           # 5-4.9 = 0.1; 15/0.1 = 150; 150 *
275         20 +1 = 3001
276     elif t_add_start == 0.0:
277         #n_add = 100
278         n_plot = int((T / (t_add_end - t_add_start)) * n_add) + 1
279         #n_plot = 301           # 5-0   = 5.0; 15/5.0 = 3  ;
280         3*100 +1    = 301
281
282     _t_plot = np.linspace(0, T_opt_end, n_plot)
283     _F_exp_interpolated = _F_exp_interpid(_t_plot)
284
285     # Create t_exp_artificial + F_exp_artificial
286     _t_exp_artificial = []
287     _F_exp_artificial = []
288
289     for i in range(len(t_exp)):
290         if t_exp[i] <= t_add_start:
291             _t_exp_artificial.append(t_exp[i])
292             _F_exp_artificial.append(F_exp[i])
293
294     for i in range(len(_t_plot)):
295         if _t_plot[i] > t_add_start and _t_plot[i] <= t_add_end:
296             _t_exp_artificial.append(_t_plot[i])
297             _F_exp_artificial.append(_F_exp_interpolated[i])
298
299     for i in range(len(t_exp)):
300         if t_exp[i] > t_add_end:
301             _t_exp_artificial.append(t_exp[i])
302             _F_exp_artificial.append(F_exp[i])
303
304
305     return _t_exp_artificial, _F_exp_artificial
306
307
308 def stop_timer(tic, menu_timer):
309
310     if menu_timer == 'timer_on':
311
312         toc = time.perf_counter()
313
314         print(f' Time needed to run calculation: {toc - tic:0.4f} seconds')
315         print('')
```

B.16 optimization.py

```

1 # -----
2 # I N V E R S E   A N A L Y S I S
3 # -----
4 from model import run_model
5 from utility import calc_F_num_at_discrete_time_steps,
6     check_cost_function
7 from plot_funcs import plot_F_exp_vs_F_num, print_opt_parameters,
8     print_save_R_squared, print_R_squared_for_given_time_frame,
9     print_residual_at_week_5
10 from scipy.optimize import least_squares
11 import numpy as np
12
13 def run_optimization(parameters_to_optimize,
14     parameters_treated_as_constant,
15         t, t_break, t_reached_5, T_opt_end,
16         T_explicit_Euler, dt, c_sat, c_s_t0, c_max, V_s, V_s_units,
17         V_c_units, r_i, r_o, l_c, H,
18             A_m_t0, m_d_t0, const_distribution,
19             parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
20             limiter_break, limiter_sat,
21                 V, mesh, nx, nr, ds, degree, r, n_facet,
22                 num_steps, num_steps_explicit_Euler, F_num_komplex_n_Euler,
23                 runs_CN_high, fineness, tol_adaptiv_dt, safety, p, boundary_tol,
24                     menu, distribution_menu, verbose,
25                     discretization_scheme, t_ref, c_ref, D_ref, l_ref, t_num, F_num,
26                         t_exp, F_exp, opt_time_frame, opt_success,
27                         kind_of_1d_interpolation):
28     """ Optimizes the unknown material parameters by fitting the
29     numerical results (_t_num, _F_num) to the experimental data (
30     t_exp, F_exp) """
31
32     # 1. Define bounds
33     if opt_time_frame == 'until_break':
34         bnds = [[1e-2, 9.0, 3e-1], [1e4, 9e2, 3.0]]
35     elif opt_time_frame == 'after_break':
36         bnds = ([0.0, np.inf])
37
38     # 2. Collect constants
39     standard_constants = [t, t_break, t_reached_5, T_opt_end,
40         T_explicit_Euler, dt, c_sat, c_s_t0, c_max, V_s, V_s_units,
41         V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
42         parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
43         limiter_break, limiter_sat,
44             V, mesh, nx, nr, ds, degree, r, n_facet,
45             num_steps, num_steps_explicit_Euler, F_num_komplex_n_Euler,
46             runs_CN_high, fineness, tol_adaptiv_dt, safety, p, boundary_tol,
47             ]

```

```
29             menu, distribution_menu, verbose,
30             discretization_scheme, t_ref, c_ref, D_ref, l_ref]
31
32     # 3. Run least squares optimization
33     optimized_parameters = least_squares(object_function,
34     parameters_to_optimize, xtol=1e-08, bounds=bnds, method='trf',
35     loss='linear', verbose=2,
36     args=(parameters_treated_as_constant, standard_constants, t_exp, F_exp
37     , opt_time_frame, opt_success, kind_of_1d_interpolation))
38
39
40
41 def object_function(parameters_to_optimize,
42     parameters_treated_as_constant, standard_constants, t_exp, F_exp
43     , opt_time_frame, opt_success, kind_of_1d_interpolation):
44     """ Calcs weighted residuals by running the model and
45     subtracting F_exp from the calculated _F_num and multiply that
46     term by the weights """
47
48     # 1. Extract needed constants
49     T_opt_end      = standard_constants[3]
50     A_m_t0          = standard_constants[16]
51     l_ref           = standard_constants[-1]
52     D_ref           = standard_constants[-2]
53     c_ref           = standard_constants[-3]
54     verbose         = standard_constants[-6]
55     distribution_menu = standard_constants[-7]
56     menu            = standard_constants[-8]
57
58
59     # 2. Reset t_num & F_num (otherwise they contain the values from
60     # previous optimization steps)
61     t_num = [0]
62     F_num = [0]
63
64
65     # 3. Calc _F_num by model
66     if opt_time_frame == 'until_break':
67
68         D          = parameters_to_optimize[0]
69         alpha       = parameters_to_optimize[1]
70         l_b        = parameters_to_optimize[2]
71         A_m_degrad = parameters_treated_as_constant[0]
72         const_degrad = parameters_treated_as_constant[1]
73         degrad_break = parameters_treated_as_constant[2]
```

```

70     elif opt_time_frame == 'after_break':
71
72         A_m_degrad = parameters_to_optimize[0]
73         D          = parameters_treated_as_constant[0]
74         alpha      = parameters_treated_as_constant[1]
75         l_b        = parameters_treated_as_constant[2]
76         const_degrad = parameters_treated_as_constant[3]
77         degrad_break = parameters_treated_as_constant[4]
78
79         # Print parameters for the current optimization run
80         print_opt_parameters(parameters_to_optimize, const_degrad,
81                               A_m_t0, c_ref, D_ref, l_ref, opt_time_frame, verbose)
82
83         # Run model
84         _t_num, _F_num = run_model(D, alpha, l_b, A_m_degrad,
85                                   const_degrad, degrad_break, *standard_constants, t_num, F_num)
86
87         # 4. Calc weighted residuals
88
89         # Define weights for residuals
90         t_opt_end = T_opt_end
91
92         # Disable weights (_weight = 1.0)
93         _weight = []
94         for i in range(len([1 for i in t_exp if i <= t_opt_end])): # for
95             T_opt_end=5/15 (no weights (1.0))
96             _weight.append(1.0)
97
98         # Choose weighting by removing #
99         #_weight = [0.0001,0.0001,0.0001,0.0001,0.0001,1.0] # for
100        T_opt_end=5
101        #_weight = [1.0,0.01,0.01,0.5,0.5,1.0]                      # for
102        T_opt_end=5
103
104        # Calc _F_num at discrete time steps (these are the time points,
105        # where measurements were taken during the experiment)
106        _F_num_at_discrete_points = calc_F_num_at_discrete_time_steps(
107                               _t_num, _F_num, t_exp, T_opt_end, kind_of_1d_interpolation)
108
109        # Calc Weighted residuals
110        _weighted_residuals = [] # initialize
111        for j in range(len(F_exp)):
112            _weighted_residuals.append(_weight[j]*(F_exp[j] -
113                                         _F_num_at_discrete_points[j]))
114
115        # Convert list of weighted residuals to an array
116        _weighted_residuals_arr = np.array(_weighted_residuals)
117
118        # 5. Print/plot results

```

```

113 #TODO: R(5,15) implementieren
114 R_squared_0_T = print_save__R_squared(_F_num_at_discrete_points,
115                                         F_exp, t_exp, 0.0, T__opt_end, parameters_to_optimize, A_m_t0,
116                                         D_ref, c_ref, l_ref, verbose, menu, distribution_menu,
117                                         opt_time_frame)
118 R_squared_0_1 = print__R_squared_for_given_time_frame(
119                                         _F_num_at_discrete_points, F_exp, t_exp, 0.0, 1.0, verbose,
120                                         opt_time_frame) # RŠ initial burst (week:0-1)
121 R_squared_3_5 = print__R_squared_for_given_time_frame(
122                                         _F_num_at_discrete_points, F_exp, t_exp, 3.0, 5.0, verbose,
123                                         opt_time_frame) # RŠ during stagnation (week:3-5)
124 print_residual_at_week_5(_F_num_at_discrete_points, F_exp,
125                           verbose, opt_time_frame)
126 plot__F_exp_vs_F_num(parameters_to_optimize, D_ref, c_ref, l_ref,
127                         , t_exp, _t_num, F_exp, _F_num, T__opt_end, R_squared_0_T,
128                         kind_of_1d_interpolation, verbose, menu, opt_success,
129                         distribution_menu, opt_time_frame)
130 check_cost_function(F_exp, _F_num_at_discrete_points, _weight,
131                      verbose)
132
133     return _weighted_residuals_arr

```

B.17 convergence_analysis.py

```

1 # -----
2 # C O N V E R G E N C E   A N A L Y S I S
3 # -----
4 from mesh_domains_boundaries import collect_mesh_parameters
5 from model import run_model
6 from utility import check_CFL_conditions, compute_convergence_rate
7 from save import save_error_to_csv, save_rate_to_csv
8
9
10 def run_convergence_analysis(D, alpha, l_b, A_m_degrad, const_degrad,
11     , degrad_break,
12         t, t_break, t_reached_5, T,
13         T_explicit_Euler, dt, dt_CN, c_sat, c_s_t0, c_max, V_s,
14         V_s_units, V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0,
15         const_distribution, parabolic_distribution, offset_parabola, U_0
16         , U_1, beta, gamma, limiter_break, limiter_sat,
17             degree, element_family, num_steps, nx,
18             nr, h, steps_convergence_analysis, num_steps_explicit_Euler,
19             nt_CN, L2_error_avg_list, F_num_komplex_n_Euler, runs_CN_high,
20             fineness, tol_adaptiv_dt, safety, p, boundary_tol,
21                 menu, distribution_menu, verbose,
22                 t_ref, c_ref, D_ref, l_ref, t_num, F_num):
23
24
25     # User-menu
26     space_or_time = menu_convergence_analysis_space_or_time()
27     discretization_scheme =
28     menu_convergence_analysis_discretization_scheme()
29
30     # DUMMY parameter (used for collecting results of the '
31     # discretization_scheme == 6/7'-case: )
32     # TODO: ev. entfernen, falls case=6/7 nicht notwendig
33     diff_F_num_total_avg = 0.0
34
35
36     if discretization_scheme == 1: # explicit EULER
37
38         nt = num_steps_explicit_Euler
39         dt = dt
40         T = T_explicit_Euler
41
42     elif discretization_scheme == 2 or discretization_scheme == 3: # CRANK-NICOLSON (low and high precision)
43
44         nt = nt_CN
45         dt = dt_CN
46         T_explicit_Euler = 0 # Choose T_explicit_Euler=0, that EULER
47         loop will be skipped in model.py

```

```
37     if space_or_time == 1: # spatial discretization
38
39         # Set time step length
40         nt = nt[0]
41         dt = dt[0]
42
43
44         # Run spatial convergence analysis
45         for n in range(len(degree)):
46
47             # Iterate through nx
48             for i in range(len(nx)):
49
50                 print('')
51                 print(element_family, degree[n], ': nx =', nx[i])
52
53                 # Mesh parameters
54                 mesh, V, ds, r, n_facet = collect_mesh_parameters(
55                 r_i, r_o, nx[i], degree[n], element_family, boundary_tol)
56
57                 # Check dt for CFL-conditions (only for EULER's
58                 # method)
59                 dt = check_CFL_conditions(dt, D, mesh, verbose,
60                 discretization_scheme, menu)
61
62                 # Compute average L2-error by running model
63                 L2_error_avg = run_model(D, alpha, l_b, A_m_degrad,
64                 const_degrad, degrad_break,
65                     t, t_break, t_reached_5, T,
66                     T_explicit_Euler, dt, c_sat, c_s_t0, c_max, V_s, V_s_units,
67                     V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
68                     parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
69                     limiter_break, limiter_sat,
70                     V, mesh, nx[i], nr[i], ds,
71                     degree[n], r, n_facet, num_steps, nt, F_num_komplex_n_Euler,
72                     runs_CN_high, fineness, tol_adaptiv_dt, safety, p, boundary_tol,
73                     menu, distribution_menu,
74                     verbose, discretization_scheme, t_ref, c_ref, D_ref, l_ref,
75                     t_num, F_num)
76
77                 # Add L2_error to list
78                 L2_error_avg_list[n] += [L2_error_avg]
79
80
81                 # Compute and save convergence rate
82                 rate = compute_convergence_rate(L2_error_avg_list, degree,
83                 nx, nt, h, space_or_time)
84                 save_rate_to_csv(nx, nt, dt, rate, D, D_ref, T,
85                 space_or_time, discretization_scheme)
86
87                 # Save average L2-error for every spatial discretization
```

```

74     save_error_to_csv(nx, nt, dt, L2_error_avg_list, D, D_ref, T
75     , T_explicit_Euler, space_or_time, discretization_scheme)
76
77
78     elif space_or_time == 2: # temporal discretization
79
80         # Set amount of elements
81         nx = nx[-1]
82         nr = nr[-1] # only used for plots
83
84         # Run temporal convergence analysis
85         for n in range(len(degree)):
86
87             # Iterate through dt
88             for i in range(len(nt)):
89                 print('')
90                 print(element_family, degree[n], ': nt =', nt[i])
91
92                 # Mesh parameters
93                 mesh, V, ds, r, n_facet = collect_mesh_parameters(
94                 r_i, r_o, nx, degree[n], element_family, boundary_tol)
95
96                 # Check dt for CFL-conditions (only for EULER's
97                 # method)
98                 dt[i] = check_CFL_conditions(dt[i], D, mesh,
99                 verbose, discretization_scheme, menu)
100
101                 # Compute average L2-error by running model
102                 L2_error_avg = run_model(D, alpha, l_b, A_m_degrad,
103                 const_degrad, degrad_break,
104                 t, t_break, t_reached_5, T,
105                 T_explicit_Euler, dt[i], c_sat, c_s_t0, c_max, V_s, V_s_units,
106                 V_c_units, r_i, r_o, l_c, H, A_m_t0, m_d_t0, const_distribution,
107                 parabolic_distribution, offset_parabola, U_0, U_1, beta, gamma,
108                 limiter_break, limiter_sat,
109                 V, mesh, nx, nr, ds, degree
110                 [n], r, n_facet, num_steps, nt[i], F_num_komplex_n_Euler,
111                 runs_CN_high, fineness, tol_adaptiv_dt, safety, p, boundary_tol,
112                 menu, distribution_menu,
113                 verbose, discretization_scheme, t_ref, c_ref, D_ref, l_ref,
114                 t_num, F_num)
115
116                 # Add L2_error to list
117                 L2_error_avg_list[n] += [L2_error_avg]
118
119
120                 # Compute and save convergence rate
121                 rate = compute_convergence_rate(L2_error_avg_list, degree,
122                 nx, nt, h, space_or_time)

```

```
110     save_rate_to_csv(nx, nt, dt, rate, D, D_ref, T,
111                         space_or_time, discretization_scheme)
112
113     # Save average L2-error for every temporal discretization
114     save_error_to_csv(nx, nt, dt, L2_error_avg_list, D, D_ref, T
115                         , T_explicit_Euler, space_or_time, discretization_scheme)
116
117
118
119
120 def menu_convergence_analysis_space_or_time():
121
122     _space_or_time = 0 # initialize menu with dummy value
123
124     while _space_or_time != 1 and _space_or_time != 2:
125
126         print('')
127         print('
-----')
128         print('      C O N V E R G E N C E   A N A L Y S I S')
129         print('')
130         print('      1. in SPACE')
131         print('      2. in TIME')
132         print('
-----')
133         print('')
134
135     _space_or_time = int(input())
136
137     if _space_or_time != 1 and _space_or_time != 2:
138         print('    You did not choose "1" or "2"!')
139         print('')
140
141
142     return _space_or_time
143
144
145
146 def menu_convergence_analysis_discretization_scheme():
147
148     _discretization_scheme = 0 # initialize menu with dummy value
149
150     while _discretization_scheme < 1 or _discretization_scheme > 3:
151
152         print('')
153         print('
```

```
-----  
154     ')  
155     print('      D I S C R E T I Z A T I O N   S C H E M E  
to test')  
156     print('')  
157     print('      1. MMS - EULER')  
158     print('      2. MMS - CRANK-NICOLSON (low precision loop)')  
159     print('      3. MMS - CRANK-NICOLSON (high precision loop)')  
160     print('')  
161  
162     _discretization_scheme = int(input())  
163  
164     if _discretization_scheme < 1 or _discretization_scheme > 3:  
165         print('      You did not choose between "1" - "3"!')  
166         print('')  
167  
168  
169     return _discretization_scheme  
-----
```

B.18 testing.py

```
1 # -----
2 # T E S T I N G
3 # -----
4 from fenics import interpolate, plot
5 from mesh_domains_boundaries import define_mesh, define_r
6 from u_0 import define_u_t0
7 from utility import check_u_t0
8 from save import save_rel_err_m_d_to_csv, create_u_t0_save_files,
9     save_u_t0
10 import matplotlib.pyplot as plt
11
12 def run_testing(t, T, r_i, r_o, H, m_d_t0, l_ref, c_ref, menu,
13     const_distribution, parabolic_distribution, U_0, U_1,
14     offset_parabola, element_family, degree, nx, verbose,
15     discretization_scheme, menu_testing, distribution_menu):
16
17     if menu_testing==1:
18         print(f'CONVERGENCE ANALYSIS will be moved here')
19
20     elif menu_testing==2: # Relative error of drug mass m_d
21         # depending on spatial discretization
22
23         # Calc relative errors of m_d in relation to degree and
24         # amount of elements
25         rel_err_m_d = calc_rel_err_m_d(t, T, r_i, r_o, H, m_d_t0,
26             l_ref, c_ref, menu, const_distribution, parabolic_distribution,
27             U_0, U_1, offset_parabola, element_family, degree, nx, verbose,
28             discretization_scheme, distribution_menu)
29
30         # Save relative errors
31         save_rel_err_m_d_to_csv(nx, rel_err_m_d, degree,
32             distribution_menu)
33
34
35     return [rel_err_m_d, distribution_menu]
```



```
72         plt.clf()  # close plot
73
74
75     return rel_err_percent_list
```

B.19 save.py

```
1 # -----
2 # S A V E
3 # -----
4 import csv
5 import math
6 import datetime
7 import os
8 from itertools import zip_longest
9 from dolfin import errornorm, File, XDMFFile
10 from plot_funcs import plot_print_L2_errornorm
11
12
13 def collect_results(t_num, F_num, L2_error, menu, distribution_menu):
14     :
15
16     if menu == 1:
17
18         save_tnum_Fnum_to_csv(t_num, F_num, distribution_menu)
19
20         return [t_num, F_num]
21
22     elif menu == 2:
23
24         return [t_num, F_num]
25
26
27     elif menu == 4:
28         """
29             For floating point numbers the numerical precision of sum (
30             and np.add.reduce) is in general limited by directly adding each
31             number individually
32             to the result causing rounding errors in every step. However
33             , often numpy will use a numerically better approach (partial
34             pairwise summation)
35             leading to improved precision in many use-cases. In contrast
36             to NumPy, Python's math.fsum function uses a slower but more
37             precise approach to summation.
38
39             https://numpy.org/doc/stable/reference/generated/numpy.sum.
40             html
41             """
42
43
44             #Calc
45             #_L2_error_avg = 1.0/len(L2_error) * np.sum(L2_error)
46             _L2_error_avg = 1.0/len(L2_error) * math.fsum(L2_error)
47             #_L2_error_avg = 1.0/len(L2_error) * sum(L2_error)
48
49             # Print
```

```
42     print('len(L2_error) =', len(L2_error))
43     print('avg(L2_error) =', _L2_error_avg)
44
45     return _L2_error_avg
46
47
48
49 # -----
50 #  save t_num, F_num
51 #
52 def save_t_num_F_num(t, t_num, F_num, F_num_simpel_CN_high,
53                      F_num_komplex_CN_high, F_num_choice):
54
55     # Define internal variables
56     _t = t
57     _t_num = t_num
58     _F_num = F_num
59     _F_num_simpel_CN_high = F_num_simpel_CN_high
60     _F_num_komplex_CN_high = F_num_komplex_CN_high
61
62     # Save t_num+F_num
63     if _t - _t_num[len(_t_num) - 1] > 0.2 and _t < 1.0: # exclude
64         values during initial time-step finding
65
66         _t_num += [_t]          # save t_num
67
68         if F_num_choice == 1: # save F_num
69             _F_num += [_F_num_simpel_CN_high]
70
71         elif F_num_choice == 2:
72             _F_num += [_F_num_komplex_CN_high]
73
74     elif _t >= 1.0: # includes values of time-step finding after
75         matrix break
76
77         _t_num += [_t]          # save t_num
78
79         if F_num_choice == 1: # save F_num
80             _F_num += [_F_num_simpel_CN_high]
81
82         elif F_num_choice == 2:
83             _F_num += [_F_num_komplex_CN_high]
84
85
86 # -----
87 #  save L2-error
88 # -----
```

```
89 def calc_plot_print_L2_error(t, T_explicit_Euler, u_e, u, _L2_error,
90     num_steps, nt, ax5, verbose, discretization_scheme):
91
92     if discretization_scheme != 0:
93
94         if num_steps <= nt:
95
96             # Calc & collect L2-errornorm
97             _L2_error += [errornorm(u_e, u, 'L2', degree_rise=3)]
98
99         if verbose == 2:
100
101             # Plot/print L2-errornorm
102             plot_print_L2_errornorm(ax5, t, T_explicit_Euler,
103             num_steps, _L2_error[-1])
104
105         elif num_steps > nt:
106
107             print('WARNING: L2_error[', num_steps, '] was not
108             added to L2_error')
109             print('')
110
111
112 # -----
113 # save pvd, XDMF
114 # -----
115
116 def create_save_files(menu, distribution_menu):
117
118     # Read date
119     date = str(datetime.date.today())    # convert date to str
120
121     # Check distribution
122     if distribution_menu == 0:
123         start_distribution = "/constant_c0"
124     elif distribution_menu == 1:
125         start_distribution = "/parabolic_c0"
126
127     # Create path
128     path = str("results/01_simulation/" + date + start_distribution
129     + "/solution")
130
131     # Create folder
132     if not os.path.exists(path):
133         os.makedirs(path)
134
135     # Save files
136     if menu == 1:
```

```
136     _vtk_file = File(path+"/pvf/results_simulation.pvf")
137 # create vtk-file
138     _xdmf_file = XDMFFile(path+"/xdmf/results_simulation.xdmf")
139 # create XDMF-file
140
141     elif menu == 2 or menu == 4:
142
142         _vtk_file = 0 # dummy
143         _xdmf_file = 0 # dummy
144
145
146     return _vtk_file, _xdmf_file
147
148
149
150 def create_u_t0_save_files(distribution_menu, element_family,
151 degree):
152
152     # Create path
153     if distribution_menu == 0:
154         start_distribution = "constant_c0/"
155     elif distribution_menu == 1:
156         start_distribution = "parabolic_c0/"
157
158     date = str(datetime.date.today()) # convert date to str
159     path = str("results/05_testing/rel_err_m_d/" +
160     start_distribution + date) # create path to .csv-file
161
161     # Create folder, if it doesn't exist
162     if not os.path.exists(path+"/vtk"):
163         os.makedirs(path+"/vtk")
164     elif not os.path.exists(path+"/xdmf"):
165         os.makedirs(path+"/xdmf")
166
167     # Create file
168     _vtk_file_u_t0 = File(path+"/vtk/nx_ut0__"+str(
169     element_family)+str(degree)+"_.pvf") # create vtk-file
170     _xdmf_file_u_t0 = XDMFFile(path+"/xdmf/nx_ut0__"+str(
171     element_family)+str(degree)+"_.xdmf") # create XDMF-file
172
172     return _vtk_file_u_t0, _xdmf_file_u_t0
173
174
175
176 def save_solution(t, u_n, vtk_file, xdmf_file, menu):
177
178     # Initialize internal variables
179     _vtk_file = vtk_file
180     _xdmf_file = xdmf_file
```

```

181
182     if menu == 1:
183
184         # Save u (numerical)
185         _vtk_file << (u_n, t)      # (t, u_high) in vtk -file
186         hinterlegen
187         _xdmf_file.write(u_n, t) # (t, u_high) in XDMF-file
188         hinterlegen
189
190
191     def save_u_t0(nx, u_t0, vtk_file, xdmf_file):
192
193         # Initialize internal variables
194         _vtk_file = vtk_file
195         _xdmf_file = xdmf_file
196
197         # Save u_t0 (numerical)
198         _vtk_file << (u_t0, nx)      # (nx, u_t0) in vtk -file hinterlegen
199         _xdmf_file.write(u_t0, nx) # (nx, u_t0) in XDMF-file hinterlegen
200
201
202 # -----
203 #   save CSV
204 #
205 def save_error_to_csv(nx, nt, dt, L2_error_avg_list, D, D_ref, T,
206                       T_explicit_Euler, space_or_time, discretization_scheme):
207
208     if discretization_scheme == 1 and space_or_time == 1:
209         with open('results/04_convergence_analysis/011
210             __space__EULER/
211             results__convergence_analysis__L2_error__space__EULER.csv', 'w')
212             as f:
213                 writer = csv.writer(f, delimiter='\t')
214                 writer.writerow(['D', 'D_ref', 'T'])
215                 writer.writerow([D, D_ref, T_explicit_Euler])
216                 writer.writerow([''])
217                 writer.writerow(['n_s', 'err_n_s'])
218                 writer.writerow([''])
219                 writer.writerow(nx)
220                 writer.writerows(L2_error_avg_list)
221
222     elif discretization_scheme == 1 and space_or_time == 2:
223         with open('results/04_convergence_analysis/021_time__EULER
224             /results__convergence_analysis__L2_error__time__EULER.csv', 'w')
225             as f:
226                 writer = csv.writer(f, delimiter='\t')
227                 writer.writerow(['D', 'D_ref', 'T', 'n_s'])
228                 writer.writerow([D, D_ref, T_explicit_Euler, nx])
229                 writer.writerow([''])

```

```
224     writer.writerow(['n_t', 'err_n_t'])
225     writer.writerow([''])
226     writer.writerow(nt)
227     writer.writerows(L2_error_avg_list)
228
229     elif discretization_scheme == 2 and space_or_time == 1:
230         with open('results/04_convergence_analysis/012
231             _space_CRANK_NICOLSON_low_precision/
232             results_convergence_analysis_L2_error_space_CRANK_NICOLSON_low_precision
233             .csv','w') as f:
234             writer = csv.writer(f, delimiter='\t')
235             writer.writerow(['D', 'D_ref', 'T'])
236             writer.writerow([D, D_ref, T])
237             writer.writerow([''])
238             writer.writerow(['n_s', 'err_n_s'])
239             writer.writerow([''])
240             writer.writerow(nx)
241             writer.writerows(L2_error_avg_list)
242
243     elif discretization_scheme == 2 and space_or_time == 2:
244         with open('results/04_convergence_analysis/022
245             _time_CRANK_NICOLSON_low_precision/
246             results_convergence_analysis_L2_error_time_CRANK_NICOLSON_low_precision
247             .csv','w') as f:
248             writer = csv.writer(f, delimiter='\t')
249             writer.writerow(['D', 'D_ref', 'T'])
250             writer.writerow([D, D_ref, T])
251             writer.writerow([''])
252             writer.writerow(['n_t', 'err_n_t'])
253             writer.writerow([''])
254             writer.writerow(nt)
255             writer.writerows(L2_error_avg_list)
256
257     elif discretization_scheme == 3 and space_or_time == 1:
258         with open('results/04_convergence_analysis/013
259             _space_CRANK_NICOLSON_high_precision/
260             results_convergence_analysis_L2_error_space_CRANK_NICOLSON_high_precision
261             .csv','w') as f:
262             writer = csv.writer(f, delimiter='\t')
263             writer.writerow(['D', 'D_ref', 'T'])
```

```

264     results__convergence_analysis__L2_error__time__CRANK_NICOLSON__high_precision
265     .csv', 'w') as f:
266         writer = csv.writer(f, delimiter='\t')
267         writer.writerow(['D', 'D_ref', 'T'])
268         writer.writerow([D, D_ref, T])
269         writer.writerow([''])
270         writer.writerow(['n_t', 'dt', 'err_n_t'])
271         writer.writerow([''])
272         writer.writerow(nt)
273         writer.writerows(L2_error_avg_list)
274
275 def save_rate_to_csv(nx, nt, dt, rate, D, D_ref, T, space_or_time,
276                      discretization_scheme):
277
278     if discretization_scheme == 1 and space_or_time == 1:
279         with open(
280             'results/04_convergence_analysis/011
281             __space__EULER/results__convergence_analysis__rate__space__EULER
282             .csv',
283             'w') as f:
284         write = csv.writer(f, delimiter='\t')
285         write.writerow(['D', 'D_ref', 'T'])
286         write.writerow([D, D_ref, T])
287         write.writerow([''])
288         write.writerow(['n_s', 'Konvergenzrate'])
289         write.writerow(nx)
290         write.writerows(rate)
291
292     elif discretization_scheme == 1 and space_or_time == 2:
293         with open(
294             'results/04_convergence_analysis/021
295             __time__EULER/results__convergence_analysis__rate__time__EULER.
296             csv',
297             'w') as f:
298         write = csv.writer(f, delimiter='\t')
299         write.writerow(['D', 'D_ref', 'T'])
300         write.writerow([D, D_ref, T])
301         write.writerow([''])
302         write.writerow(['n_t', 'Konvergenzrate'])
303         write.writerow(nt)
304         write.writerows(rate)
305
306     elif discretization_scheme == 2 and space_or_time == 1:
307         with open(
308             'results/04_convergence_analysis/012
309             __space__CRANK_NICOLSON__low_precision/

```

```
306     results__convergence_analysis__rate__space__CRANK_NICOLSON__low_precision
307     .csv',
308         'w') as f:
309             write = csv.writer(f, delimiter='\t')
310             write.writerow(['D', 'D_ref', 'T'])
311             write.writerow([D, D_ref, T])
312             write.writerow([''])
313             write.writerow(['n_s', 'Konvergenzrate'])
314             write.writerow(nx)
315             write.writerows(rate)
316
317     elif discretization_scheme == 2 and space_or_time == 2:
318         with open(
319             'results/04_convergence_analysis/022
320             __time__CRANK_NICOLSON__low_precision/
321             results__convergence_analysis__rate__time__CRANK_NICOLSON__low_precision
322             .csv',
323                 'w') as f:
324                     write = csv.writer(f, delimiter='\t')
325                     write.writerow(['D', 'D_ref', 'T'])
326                     write.writerow([D, D_ref, T])
327                     write.writerow([''])
328                     write.writerow(['n_t', 'Konvergenzrate'])
329                     write.writerow(nt)
330                     write.writerows(rate)
331
332     elif discretization_scheme == 3 and space_or_time == 1:
333         with open(
334             'results/04_convergence_analysis/013
335             __space__CRANK_NICOLSON__high_precision/
336             results__convergence_analysis__rate__space__CRANK_NICOLSON__high_precision
337             .csv',
338                 'w') as f:
339                     write = csv.writer(f, delimiter='\t')
340                     write.writerow(['D', 'D_ref', 'T'])
341                     write.writerow([D, D_ref, T])
342                     write.writerow([''])
343                     write.writerow(['n_s', 'Konvergenzrate'])
344                     write.writerow(nx)
345                     write.writerows(rate)
346
347     elif discretization_scheme == 3 and space_or_time == 2:
348         with open(
349             'results/04_convergence_analysis/023
350             __time__CRANK_NICOLSON__high_precision/
351             results__convergence_analysis__rate__time__CRANK_NICOLSON__high_precision
352             .csv',
353                 'w') as f:
```

```

346         write = csv.writer(f, delimiter='\t')
347         write.writerow(['D', 'D_ref', 'T'])
348         write.writerow([D, D_ref, T])
349         write.writerow([''])
350         write.writerow(['n_t', 'Konvergenzrate'])
351         write.writerow(nt)
352         write.writerows(rate)
353
354
355
356 def save_tnum_Fnum_to_csv(t_num, F_num, distribution_menu):
357
358     # Read date
359     date = str(datetime.date.today()) # convert date to str
360
361     # Check distribution
362     if distribution_menu == 0:
363         start_distribution = "constant_c0"
364     elif distribution_menu == 1:
365         start_distribution = "parabolic_c0"
366
367     # Create path
368     path = str("results/01__simulation/" + date + "/" +
369     start_distribution + "/F_num")
370
371     # Create folder
372     if not os.path.exists(path):
373         os.makedirs(path)
374
375     # Save (t_num, F_num) in .csv file
376     with open(path+'/' +date+'__'+start_distribution+'__'+tnum_Fnum.
377     csv', 'w') as f:
378         write = csv.writer(f, delimiter='\t')
379         write.writerow(['t_num', 'F_num'])
380         write.writerow(t_num)
381         write.writerow(F_num)
382
383
384 def save_rel_err_m_d_to_csv(nx, rel_err_m_d, degree,
385     distribution_menu):
386
387     # Create path
388     if distribution_menu == 0:
389         start_distribution = "constant_c0/"
390     elif distribution_menu == 1:
391         start_distribution = "parabolic_c0/"
392
393     date = str(datetime.date.today()) # convert date to str
394     path = str("results/05__testing/rel_err_m_d/" + start_distribution
395     + date) # create path to .csv-file
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
22
```

```
393 # Create folder, if it doesn't exist
394 if not os.path.exists("results/05_testing/rel_err_m_d/"+start_distribution+date):
395     os.makedirs("results/05_testing/rel_err_m_d/"+start_distribution+date)
396
397 # Create csv-file, if it doesn't exist
398 if not os.path.isfile("results/05_testing/rel_err_m_d/"+start_distribution+date+"/rel_err_m_d.csv"):
399     with open(path + '/rel_err_m_d.csv', 'w') as f:
400         writer = csv.writer(f, delimiter='\t')
401         writer.writerow(['n_s [-]', 'err_m_d [%] (P1)', 'err_m_d [%] (P2)', '', 'c(t=0)', start_distribution])
402
403 # Prepare lists for processing
404 rel_err_m_d_P1 = rel_err_m_d[0]
405 rel_err_m_d_P2 = rel_err_m_d[1]
406 combined_list = [nx, rel_err_m_d_P1, rel_err_m_d_P2]
407 export_data = zip_longest(*combined_list, fillvalue=' ')
408
409 # Save to csv
410 with open(path+'/rel_err_m_d.csv', 'a') as f:
411     writer = csv.writer(f, delimiter='\t')
412     writer.writerows(export_data)
413     f.close()
```

B.20 plot_settings_inits.py

```

1 # -----
2 # P L O T  ( S E T T I N G S & I N I T I A L I Z A T I O N )
3 # -----
4 from fenics import *
5 import matplotlib.pyplot as plt # version: 2.1.1
6
7
8 # -----
9 # S E T T I N G S
10 # -----
11 def plot_settings(c_max, c_sat, m_d_t0, menu, discretization_scheme,
12   verbose):
13   """ Defines which plots are shown depending on user menu &
14   verbose choice. """
15   if menu == 1 or menu == 2:
16
17     if verbose == 0: # verbose mode OFF
18       _fig1 = 0
19       _fig2 = 0
20       _ax1 = 0
21       _ax2 = [0,0,0,0]
22       _ax3 = 0
23       _ax4 = 0
24       _ax5 = 0
25       _ax6 = 0
26
27     elif verbose == 1: # verbose mode OFF
28       _fig1, _ax1 = initialize_F_num_plots__simpel_vs_complex
29     () # initialize F_num plots
30       _fig2 = 0
31       _ax2 = [0,0,0,0]
32       _ax3 = 0
33       _fig4, _ax4 = intialize_h_mod_plot()
34       _ax5 = 0
35       _ax6 = 0
36
37     elif verbose == 2: # verbose mode ON
38       _vtk_file = File("results/pvd/u.pvd") # create vtk-file
39       _xdmf_file = XDMFFile("results/xdmf/t_u.xdmf") # create
40     XDMF-file
41       _fig1, _ax1 = initialize_F_num_plots__simpel_vs_complex
42     () # initialize F_num plots
43       _fig2, _ax2 = initialize_plot_degrad() # initialize
44     degradation plot
45       _fig3, _ax3 = initialize_c_s_plot(c_max, c_sat)
46       _fig4, _ax4 = intialize_h_mod_plot()
47       _ax5 = 0
48       _fig6, _ax6 = initialize_m_d_plot(m_d_t0)
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93

```

```
44
45     elif menu == 4:
46
47         if verbose == 0 or verbose == 1: # verbose mode OFF
48             _fig1 = 0
49             _ax1 = 0
50             _fig2 = 0
51             _ax2 = [0,0,0,0]
52             _ax3 = 0
53             _ax4 = 0
54             _ax5 = 0
55             _ax6 = 0
56
57         elif verbose == 2: # verbose mode ON
58             _fig1 = 0
59             _ax1 = 0
60             _fig2 = 0
61             _ax2 = [0,0,0,0]
62             _ax3 = 0
63             _ax4 = 0
64             _ax5 = initialize_plot__L2_errornorm(
discretization_scheme)
65             _ax6 = 0
66
67
68     return _ax1, _ax2, _ax3, _ax4, _ax5, _ax6
69
70
71
72 # -----
73 # I N I T I A L I Z A T I O N S
74 # -----
75 def initialize_F_num_plots__simpel_vs_complex():
76     """ Initialization of (t_num, F_num_simpel), (t_num,
F_num_complex) and (t_exp, F_exp). """
77
78     # plots initialisieren
79     fig1, ax1 = plt.subplots(1,2)
80
81     # Achsenbezeichnung
82     # x-Achse
83     ax1[0].set_xlabel(r"\tilde{t}", loc="center")
84     ax1[1].set_xlabel(r"\tilde{t}", loc="center")
85
86     # y-Achse
87     ax1[0].set_ylabel(r"\tilde{F}_{num}", loc="top", rotation="horizontal")
88     ax1[1].set_ylabel(r"\tilde{F}_{num}", loc="top", rotation="horizontal")
89
90     # Titel
```

```

91     ax1[0].set_title(r"$\tilde{F}_{\mathrm{num}} = \frac{\tilde{c}_1 \mathrm{s}}{\tilde{c}_1 \mathrm{max}}$")
92     ax1[1].set_title(r"$\tilde{F}_{\mathrm{num}}$-Normalization")
93
94     # F_exp interpolieren
95     import numpy as np
96     from scipy.interpolate import interp1d
97
98     t_exp = [0, 1, 2, 3, 4, 5, 7, 8, 9, 11, 15]
99     F_exp = [0., 0.399, 0.517, 0.607, 0.623, 0.639, 0.792, 0.805,
100      0.814, 0.823, 0.844]
101    T = 15.0
102
103    kind_of_1d_interpolation = 'quadratic' # [-] - Wie soll
104    F_exp_interp1d = interp1d(t_exp, F_exp,
105      kind_of_1d_interpolation)
106
107    _t_plot = np.linspace(0, T, 100)
108    _F_exp_interpolated = _F_exp_interp1d(_t_plot)
109
110    # F_exp und F_exp interpoliert plotten
111    ax1[1].plot(_t_plot, _F_exp_interpolated, color='black', label='
112      F_exp (quad interpol)')
113    ax1[1].scatter(t_exp, F_exp, color='black', marker='s', label='
114      F_exp')
115
116
117  def initialize_plot_degrad():
118      """ Initialization of degradation plot. """
119
120      # Creating grid for subplots
121      _fig2 = plt.figure()
122      _fig2.set_figheight(6)
123      _fig2.set_figwidth(6)
124
125      # Initialize _ax2 with dummy values
126      _ax2 = [0, 0, 0, 0]
127
128      _ax2[0] = plt.subplot2grid(shape=(3, 3), loc=(0, 0), colspan=3,
129        rowspan=2) # degrad
130      _ax2[1] = plt.subplot2grid(shape=(3, 3), loc=(2, 0))
131        # m_d Euler
132      _ax2[2] = plt.subplot2grid(shape=(3, 3), loc=(2, 1))
133        # m_d CN-low
134      _ax2[3] = plt.subplot2grid(shape=(3, 3), loc=(2, 2))
135        # m_d CN-high

```

```
132     _ax2[0].set_xlabel(r"\tilde{r}", loc="right")
133     _ax2[1].set_xlabel(r"\tilde{t}", loc="right")
134     _ax2[2].set_xlabel(r"\tilde{t}", loc="right")
135     _ax2[3].set_xlabel(r"\tilde{t}", loc="right")
136
137     _ax2[0].set_ylabel(r"$degradation$", loc="top", rotation=
138                         "horizontal")
139     _ax2[1].set_ylabel(r"$degradation_{\mathrm{max}}$", loc="top",
140                         rotation="horizontal")
140     _ax2[2].set_ylabel(r"$degradation_{\mathrm{max}}$", loc="top",
141                         rotation="horizontal")
141     _ax2[3].set_ylabel(r"$degradation_{\mathrm{max}}$", loc="top",
142                         rotation="horizontal")
143
144     return _fig2, _ax2
145
146
147
148 def initialize_c_s_plot(c_max, c_sat):
149     """ Initialization of surrounding solution concentration plot.
150     """
151
152     # Plots initialisieren
153     fig3, ax3 = plt.subplots()
154
155     # Achsenbezeichnung
156     ax3.set_xlabel(r"\tilde{t}", loc="right")
157     ax3.set_ylabel(r"\tilde{c}_{(s)}", loc="top", rotation=
158                         "horizontal")
159
160     # Plot c_max for reference
161     plt.axhline(c_max, label=r"\tilde{c}_{\mathrm{max}}", color='r')
162
163     # Plot c_sat
164     plt.axhline(c_sat, label=r"\tilde{c}_{\mathrm{sat}}", color='y')
165
166     # Show labels
167     ax3.legend()
168
169
170
171
172 def intialize_h_mod_plot():
173     """ Initialization of mass transfer coefficient plot. """
174
175     # Plots initialisieren
176     fig4, ax4 = plt.subplots()
```

```
177     # Achsenbezeichnung
178     ax4.set_xlabel(r"\tilde{t}", loc="right")
179     ax4.set_ylabel(r"\tilde{h}_{mod}", loc="top", rotation="horizontal")
180
181
182     return fig4, ax4
183
184
185
186
187 def initialize_plot__L2_errornorm(discretization_scheme):
188     """ Initialization of L2-norm errors between numerical and
189     experimental release curve value F_num/F_exp. """
190
191     # Initialize plots
192     fig5, ax5 = plt.subplots()
193
194     # Achsenbezeichnung
195     ax5.set_xlabel(r"\tilde{t}", loc="center")
196     ax5.set_ylabel("L2-Errornorm [-]", loc="top", rotation="horizontal")
197
198     # Titel
199     if discretization_scheme == 1:
200         ax5.set_title('Explicit Euler')
201
202     elif discretization_scheme == 2:
203         ax5.set_title('CN (low precision)')
204
205     elif discretization_scheme == 3:
206         ax5.set_title('CN (high precision)')
207
208
209     return ax5
210
211
212 def initialize_m_d_plot(m_d_t0):
213     """ Initialization of total drug mass plot. """
214
215     # Initialize plots
216     _fig6, _ax6 = plt.subplots(2,3)
217
218     # Title
219     _fig6.suptitle(r"\tilde{m}_d_{total} = \tilde{m}_d_{m} + \
220                     \tilde{m}_d_{s}", fontsize=15)
221
222     # Achsenbezeichnung
223     # x-Achse
224     _ax6[0,0].set_xlabel(r"\tilde{t}", loc="right")
```

```
224     _ax6[0,1].set_xlabel(r"\tilde{t}", loc="right")
225     _ax6[0,2].set_xlabel(r"\tilde{t}", loc="right")
226
227     _ax6[1,0].set_xlabel(r"\tilde{t}", loc="right")
228     _ax6[1,1].set_xlabel(r"\tilde{t}", loc="right")
229     _ax6[1,2].set_xlabel(r"\tilde{t}", loc="right")
230
231     # y-Achse
232     _ax6[0,0].set_ylabel(r"\tilde{m}_{d, total}", loc="top",
233                           rotation="horizontal")
233     _ax6[0,1].set_ylabel(r"\tilde{m}_{d, total}", loc="top",
234                           rotation="horizontal")
234     _ax6[0,2].set_ylabel(r"\tilde{m}_{d, total}", loc="top",
235                           rotation="horizontal")
236
236     _ax6[1,0].set_ylabel(r"\Delta \tilde{m}_{d, total} \%; [\%]", loc="top",
237                           rotation="horizontal")
237     _ax6[1,1].set_ylabel(r"\Delta \tilde{m}_{d, total} \%; [\%]", loc="top",
238                           rotation="horizontal")
238     _ax6[1,2].set_ylabel(r"\Delta \tilde{m}_{d, total} \%; [\%]", loc="top",
239                           rotation="horizontal")
240
240     # Titel
241     _ax6[0,0].set_title('Explicit Euler')
242     _ax6[0,1].set_title('CN (low precision)')
243     _ax6[0,2].set_title('CN (high precision)')
244
245     # Plot m_d line for reference
246     _ax6[0,0].axhline(m_d_t0, label=r"\tilde{m}_d(\tilde{t}=0)", color='y')
247     _ax6[0,1].axhline(m_d_t0, label=r"\tilde{m}_d(\tilde{t}=0)", color='y')
248     _ax6[0,2].axhline(m_d_t0, label=r"\tilde{m}_d(\tilde{t}=0)", color='y')
249
250     # Show legend
251     _ax6[0,0].legend()
252     _ax6[0,1].legend()
253     _ax6[0,2].legend()
254
255
256     return _fig6, _ax6
```

B.21 plot_funcs.py

```

1 # -----
2 # P L O T
3 # -----
4 from fenics import *
5 from utility import calc_F_num_at_discrete_time_steps,
6     adjust_t_exp_and_F_exp_to_optimization_time_frame
7 import matplotlib.pyplot as plt # version: 2.1.1
8 from scipy.interpolate import interp1d
9 import numpy as np
10 from sklearn.metrics import r2_score
11 import datetime
12 import os
13 import csv
14
15 def plot_results(t, T_explicit_Euler, m_d_t0, V_s, H, r, ax1, ax2,
16     ax3, ax6, verbose, discretization_scheme, menu,
17     u_n, c_s_n_Euler, degrad_max_n_Euler,
18     F_num_simpel_n_Euler, F_num_komplex_n_Euler,
19     u_CN_low, c_s_CN_low, degrad_max_CN_low,
20     F_num_simpel_CN_low, F_num_komplex_CN_low,
21     u_CN_high, c_s_CN_high, degrad_max_CN_high,
22     F_num_simpel_CN_high, F_num_komplex_CN_high):
23
24     if (menu == 1 or menu == 2):
25
26         if verbose == 1:
27
28             if t < T_explicit_Euler:
29
30                 plot_EULER__F_num__simpel_vs_komplex(ax1,
31                 F_num_simpel_n_Euler, F_num_komplex_n_Euler, t)
32
33             elif t > T_explicit_Euler:
34
35                 plot_CN__F_num__simpel_vs_komplex(t,
36                 F_num_simpel_CN_low, F_num_simpel_CN_high, F_num_komplex_CN_low,
37                 F_num_komplex_CN_high, ax1, verbose, menu,
38                 discretization_scheme)
39
40         elif verbose == 2:
41
42             if t < T_explicit_Euler:
43
44                 plot_EULER__F_num__simpel_vs_komplex(ax1,
45                 F_num_simpel_n_Euler, F_num_komplex_n_Euler, t)
46                 plot_EULER__degrad_max(ax2, t, degrad_max_n_Euler)

```

```
39         plot_EULER__m_d(ax6, t, u_n, c_s_n_Euler, m_d_t0,
40                         V_s, H, r, dx)
41
42     elif t > T_explicit_Euler:
43
43         plot_CN__F_num__simpel_vs_komplex(t,
44             F_num_simpel_CN_low, F_num_simpel_CN_high, F_num_komplex_CN_low,
45             F_num_komplex_CN_high, ax1, verbose, menu,
46             discretization_scheme)
46             plot_CN__degrad_max(ax2, t, degrad_max_CN_low,
47             degrad_max_CN_high)
48             plot_CN__c_s(t, c_s_CN_high, ax3, verbose, menu,
49             discretization_scheme)
50             plot_CN__m_d(ax6, u_CN_low, u_CN_high, c_s_CN_low,
51             c_s_CN_high, m_d_t0, V_s, H, r, dx, t, menu,
52             discretization_scheme)
53
54
55
56 def plot_EULER__F_num__simpel_vs_komplex(ax1, F_num_simpel_Euler,
57     F_num_komplex_Euler, t):
58
59     # plot F_num
60     ax1[0].scatter(t, F_num_simpel_Euler, color='blue',
61                     marker='o')
62     ax1[1].scatter(t, F_num_komplex_Euler, color='orange',
63                     marker='o')
64
65     # irritierende Achsenkalierung fixen
66     ax1[0].get_yaxis().get_major_formatter().set_useOffset(False)
67
68     # Exponent der wissenschaftlichen Darstellung verschieben, um
69     # Überschneidung zu vermeiden
70     ax1[0].get_yaxis().get_offset_text().set_x(-0.25)
71     ax1[1].get_yaxis().get_offset_text().set_x(-0.25)
72
73
74 def plot_CN__F_num__simpel_vs_komplex(t, F_num_simpel_CN_low,
75     F_num_simpel_CN_high, F_num_komplex_CN_low,
76     F_num_komplex_CN_high, ax1, verbose, menu, discretization_scheme
77     ):
78
79     if ((menu==1 and (verbose == 1 or verbose==2)) or (menu==2 and
80         verbose==2)) and discretization_scheme != 1:
81
82         ax1[0].scatter(t, F_num_simpel_CN_high, color
83                         ='purple', marker='x')
84         ax1[1].scatter(t, F_num_komplex_CN_high, color
85                         ='red', marker='x')
```

```

72     if menu==1:
73         plt.pause(0.005)
74
75
76
77 def plot_degrad(_degrad, nr, r_i, l_c, verbose, ax2, menu):
78     """ Plots degradation of the matrix. """
79
80     #if menu == 1 or menu == 2 and verbose == 2:
81
82     r_plot = np.linspace(r_i, l_c+r_i, nr+1) # l_c = r_o-r_i
83     ax2.plot(r_plot, _degrad)
84
85
86
87 def plot_EULER_m_d(ax6, t, u_Euler, c_s_Euler, m_d_t0, V_s, H, r,
88 dx):
89
90     # Calc
91     m_d_s = c_s_Euler.val * V_s
92     m_d_m = assemble(2.0 * np.pi * H * u_Euler * r * dx)
93     m_d_total = m_d_s + m_d_m
94     delta_m_d = (m_d_total - m_d_t0) / m_d_t0
95     delta_m_d_prozent = delta_m_d * 100.0
96
97     # plot
98     ax6[0,0].plot(t, m_d_total, color='purple', marker='x')
99     ax6[1,0].plot(t, delta_m_d_prozent, color='purple', marker='x')
100
101    # irritierende Achsenkalierung fixen
102    ax6[1,0].get_yaxis().get_major_formatter().set_useOffset(False)
103
104    # Exponent der wissenschaftlichen Darstellung verschieben, um
105    # Überschneidung zu vermeiden
106    ax6[0,0].get_yaxis().get_offset_text().set_x(0.0)
107    ax6[0,0].get_xaxis().get_offset_text().set_x(0.9)
108
109
110 def plot_CN_m_d(ax6, u_CN_low, u_CN_high, c_s_CN_low, c_s_CN_high,
111 m_d_t0, V_s, H, r, dx, t, menu, discretization_scheme):
112
113     if discretization_scheme != 1:
114
115         m_d_s_CN_low = c_s_CN_low.val * V_s
116         m_d_s_CN_high = c_s_CN_high.val * V_s
117         m_d_m_CN_low = assemble(2.0 * np.pi * H * u_CN_low * r *
dx)
118         m_d_m_CN_high = assemble(2.0 * np.pi * H * u_CN_high * r *
dx)
119         m_d_total_CN_low = m_d_s_CN_low + m_d_m_CN_low

```

```
118     m_d_total_CN_high = m_d_s_CN_high + m_d_m_CN_high
119
120     delta_m_d_CN_low = (m_d_t0 - m_d_total_CN_low) / m_d_t0
121     delta_m_d_CN_high = (m_d_t0 - m_d_total_CN_high) / m_d_t0
122     delta_m_d_CN_low_prozent = delta_m_d_CN_low * 100.0
123     delta_m_d_CN_high_prozent = delta_m_d_CN_high * 100.0
124
125     # plot m_d_total
126     ax6[0,1].plot(t, m_d_total_CN_low, color='purple', marker='x')
127     ax6[0,2].plot(t, m_d_total_CN_high, color='purple', marker='x')
128     ax6[1,1].plot(t, delta_m_d_CN_low_prozent, color='purple',
129                     marker='x')
129     ax6[1,2].plot(t, delta_m_d_CN_high_prozent, color='purple',
130                     marker='x')
131
132     # irritierende Achsenkalierung fixen
133     ax6[1,1].get_yaxis().get_major_formatter().set_useOffset(
134         False)
135     ax6[1,2].get_yaxis().get_major_formatter().set_useOffset(
136         False)
137
138
139
140 def plot_EULER_degrad_max(ax2, t, degrad_max_n_Euler):
141
142     #plot
143     ax2[1].plot(t, degrad_max_n_Euler, color='purple', marker='x')
144
145     # irritierende Achsenkalierung fixen
146     ax2[1].get_yaxis().get_major_formatter().set_useOffset(False)
147
148     # Exponent der wissenschaftlichen Darstellung verschieben, um
149     # Überschneidung zu vermeiden
150     ax2[1].get_yaxis().get_offset_text().set_x(0.0)
151     ax2[1].get_xaxis().get_offset_text().set_x(0.9)
152
153
154 def plot_CN_degrad_max(ax2, t, degrad_max_CN_low,
155                         degrad_max_CN_high):
156
157     #plot
158     ax2[2].plot(t, degrad_max_CN_low, color='purple', marker='x')
159     ax2[3].plot(t, degrad_max_CN_high, color='purple', marker='x')
```

```

161
162 def plot_F_exp_vs_F_num(parameters_to_optimize, D_ref, c_ref, l_ref
163   , t_exp, tnum, F_exp, Fnum, T, R_squared_0_T,
164   kind_of_1d_interpolation, verbose, menu, opt_success,
165   distribution_menu, opt_time_frame):
166
167   # Extract parameters
168   if menu == 1:
169     [D, alpha, l_b, A_m_degrad_units, A_m_t0_units, const_degrad
170   , degrad_break] = parameters_to_optimize
171
172   elif menu == 2:
173     if opt_time_frame == 'until_break':
174       [D_opt, alpha_opt, l_b_opt] = parameters_to_optimize
175
176     elif opt_time_frame == 'after_break':
177       A_m_degrad_opt = parameters_to_optimize
178
179   # Initialize Plot
180   fig, ax = plt.subplots()
181   ax.set_xlabel(r" $\tilde{t}$ ", loc="right")
182   ax.set_ylabel(r" $\tilde{F}(\tilde{t})$ ", loc="top", rotation="horizontal")
183   ax.yaxis.set_label_coords(-0.05, 0.9)
184
185   # Set title
186   if menu == 1:
187     plt.title(f"D = {D*D_ref:e} [mm/s] , alpha = {alpha/c_ref:e}
188   } [mm/g] , l_b = {l_b*l_ref:e} [mm], A_m_degrad = {
189   A_m_degrad_units/A_m_t0_units}*A_m_t0, const_degrad = {
190   const_degrad}, degrad_break = {degrad_break}", fontsize=6)
191
192   elif menu == 2:
193     if opt_time_frame == 'until_break':
194       plt.title(f"D = {D_opt*D_ref:e} [mm/s] , alpha = {
195   alpha_opt/c_ref:e} [mm/g] , l_b = {l_b_opt*l_ref:e} [mm]",
196   fontsize=6)
197
198     elif opt_time_frame == 'after_break':
199       plt.title(f"A_m_degrad = {A_m_degrad_opt * l_ref**2} [
200   mm]", fontsize=6)
201
202   if opt_success == True:
203     plt.suptitle(f"OPTIMIZATION SUCCESS
204 L:  $\int F \, dt$ ; vs.  $\int F^2 \, dt$ :  $R^2(0, \text{int}(T)) = \{R_squared_0_T\}$ ")
205   else:
206     plt.suptitle(f"$F \, dt$; vs.  $\int F^2 \, dt$ :  $R^2(0, \text{int}(T)
207   }) = \{R_squared_0_T\}$")
208
209   # Set the range of axis$ 
```

```
198     plt.ylim(0, 1)
199
200     # F_exp interpolieren
201     _F_exp_interp1d = interp1d(t_exp, F_exp,
202         kind_of_1d_interpolation)
203     _t_plot = np.linspace(0, T, 100)
204     _F_exp_interpolated = _F_exp_interp1d(_t_plot)
205
206     # F_exp und F_exp interpoliert plotten
207     ax.plot(_t_plot, _F_exp_interpolated, color='black', label=f'F_exp ({kind_of_1d_interpolation} interpol)')
208     ax.scatter(t_exp, F_exp, color='black', marker='x', label='F_exp')
209
210     # F_num interpolieren
211     _F_num_interp1d = interp1d(tnum, Fnum, kind_of_1d_interpolation)
212     _F_num_interpolated = _F_num_interp1d(_t_plot)
213
214     # F_num plotten
215     ax.plot(_t_plot, _F_num_interpolated, color='blue', label=f'F_num ({kind_of_1d_interpolation} interpol)')
216     #ax.plot(tnum, Fnum, 'o')
217
218     # Display legends
219     ax.legend()
220
221     # Save plot
222     if menu == 2:
223
224         # Set time stamp
225         date = str(datetime.date.today())
226         date_plus_time = str(datetime.datetime.now())
227
228         # Check distribution
229         if distribution_menu == 0:
230             start_distribution = "constant_c0"
231         elif distribution_menu == 1:
232             start_distribution = "parabolic_c0"
233
234         # Create path
235         path_2 = str("results/02__optimization/" + date + "/" +
236         start_distribution)
237
238         # Check time frame, which is optimized
239         if opt_time_frame == 'until_break':
240
241             # Create folder, if necessary
242             if not os.path.exists(path_2 + "/until_break"):
243                 os.makedirs(path_2 + "/until_break")
244
245             # Save figure to .png
```

```
244         plt.savefig(path_2 + "/until_break" + "/" +
245                     date_plus_time+".png", dpi=600)
246
247     elif opt_time_frame == 'after_break':
248
249         # Create folder, if necessary
250         if not os.path.exists(path_2 + "/after_break"):
251             os.makedirs(path_2 + "/after_break")
252
253         # Save figure to .png
254         plt.savefig(path_2 + "/after_break" + "/" +
255                     date_plus_time+".png", dpi=600)
256
257     plt.axvline(x=5.0)
258
259     if verbose != 0:
260         plt.show()
261
262     plt.close()
263
264 def plot_CN__c_s(t, c_s_CN_high, ax3, verbose, menu,
265                   discretization_scheme):
266
267     if (verbose == 1 or verbose == 2) and discretization_scheme != 1:
268
269         ax3.scatter(t, c_s_CN_high.val, color='purple', marker='x')
270
271         if menu==1:
272             plt.pause(0.005)
273
274
275 def plot__h_mod(t, h_mod_CN_high, degrad_max_CN_high, degrad_break,
276                 alpha, c_sat, c_s_CN_high, limiter_break, limiter_sat, ax4,
277                 verbose, menu, discretization_scheme):
278
279     _limiter_break = limiter_break
280     _limiter_sat = limiter_sat
281
282     # TODO: später wieder auf orignale if Bedingung zurücksetzen
283     if ((menu==1 or menu == 2) and (verbose == 1 or verbose == 2)) and
284         discretization_scheme != 1:
285         #if ((menu==1 or menu == 2) and verbose == 2) and
286         #discretization_scheme != 1:
287
288         # Plot
289         ax4.scatter(t, h_mod_CN_high, color='blue', marker='x')
```

```
287     if degrad_max_CN_high > degrad_break and _limiter_break <=
288         1:
289             _limiter_break+=1
290             ax4.axvline(t, color='r', label='Matrixbruch')
291             ax4.legend()
292
293     if alpha * (c_sat-c_s_CN_high.val) < 1 and _limiter_sat <=
294         1:
295             _limiter_sat+=1
296             ax4.axvline(t, color='m', label='Saettigung')
297             ax4.legend()
298
299     if menu==1:
300         plt.pause(0.005)
301
302
303     return _limiter_break, _limiter_sat
304
305
306
307 def plot_print__L2_errornorm(ax5, t, T_explicit_Euler, num_steps,
308                             L2_errornorm):
309
310     # Print
311     if t < T_explicit_Euler:
312
313         print(f'{num_steps}. time step (EULER): t = {t}: L2-
314           Errornorm = {L2_errornorm}')
315
316     else:
317
318         print(f'{num_steps}. time step (CN):      t = {t}: L2-
319           Errornorm = {L2_errornorm}')
320
321     # Plot
322     ax5.scatter(t, L2_errornorm, color='red')
323     ax5.get_yaxis().get_major_formatter().set_useOffset(False)
324
325
326
327 def plot__convergence_analysis_results(num_steps_explicit_Euler, nx,
328                                         L2_error_avg_list, degree, space_or_time):
329
330     if space_or_time == 1:
331
332         # Plot results
```

```

332     for i in range(len(degree)):
333         fig, ax = plt.subplots()
334         fig.suptitle('P{}'.format(degree[i]), fontsize=15)
335         plt.yscale("log")
336         plt.xscale("log")
337         ax.set_xlabel(r"$n_{\mathrm{s}}$".format(), loc="right")
338         ax.set_ylabel(r"$\text{avg}(\text{L2-error})$".format(), loc="center")
339         ax.plot(nx, L2_error_avg_list[i])
340
341     elif space_or_time == 2:
342
343         # Plot results
344         for i in range(len(degree)):
345             fig, ax = plt.subplots()
346             fig.suptitle(f'P{degree[i]}-Element (n_s = {nx})',
347             fontsize=15)
347             ax.set_xlabel(r"$n_{\mathrm{t}}$".format(), loc="right", fontsize
348 =15)
348             ax.set_ylabel(r"$\text{avg}(\text{L2-error})$".format(), loc="center",
349             fontsize=15)
349             ax.plot(num_steps_explicit_Euler, L2_error_avg_list[i])
350
351     # Show plots
352     plt.show()
353
354
355
356 def show_plot(menu):
357
358     if menu == 1:
359         plt.show()
360
361     elif menu == 2:
362         plt.show(block=False)
363         ##plt.pause(5)
364         #plt.close()
365         #plt.close()
366
367
368
369 # -----
370 # P R I N T
371 # -----
372
373 def print__R_squared_for_given_time_frame(F_num_at_discrete_points,
374 F_exp, t_exp, t_start, t_end, verbose, opt_time_frame):
374     """ R2 = 1-SRS/SST; SSR = sum of residuals squared, SST = total
375     sum of squares"""
375     # https://www.ncl.ac.uk/webtemplate/ask-assets/external/mathssources/statistics/regression-and-correlation/coefficient-of-determination-r-squared.html
```

```
376
377     if verbose != 0 and opt_time_frame == 'until_break':
378
379         # Initialize internal parameters
380         _t_exp = t_exp
381         _F_exp = F_exp
382         _F_num_at_discrete_points = list(F_num_at_discrete_points)
383
384         # Consider time frame
385         if t_start != 0:
386
387             # Calc position of t_start in t_exp
388             t_start_count = t_exp.index(t_start)
389
390             # Remove elements from list
391             _t_exp = _t_exp[t_start_count:]
392             _F_exp = _F_exp[t_start_count:]
393             _F_num_at_discrete_points = _F_num_at_discrete_points[
394                 t_start_count:]
395
396             if t_end != 5:
397
398                 # Calc position of t_end in t_exp
399                 t_end_count = t_exp.index(t_end)
400
401                 # Remove elements from list
402                 _t_exp = _t_exp[:t_end_count+1]
403                 _F_exp = _F_exp[:t_end_count+1]
404                 _F_num_at_discrete_points = _F_num_at_discrete_points[:t_end_count+1]
405
406                 # Initialize and calc sum of squared residuals (SSR = sum of
407                 # residuals squared
408                 _residuals_squared = []
409
410                 for i in range(len(_F_exp)):
411                     _residuals_squared += [(_F_exp[i]-
412                     _F_num_at_discrete_points[i])**2]
413
414                 _sum_of_residuals_squared = sum(_residuals_squared)
415
416                 # Initialize and calc sum of squared differences between
417                 # F_exp and the average of F_exp (SST = total sum of squares)
418                 _avg_F_exp = sum(_F_exp)/len(_F_exp)
419                 _squared_differences_between_F_exp_and_avg_of_F_exp = []
420
421                 for j in range(len(_F_exp)):
422                     _squared_differences_between_F_exp_and_avg_of_F_exp +=
423                         [(_F_exp[j]-_avg_F_exp)**2]
424
425                 _sum_of_squared_differences_between_F_exp_and_avg_of_F_exp =
```

```
    sum(_squared_differences_between_F_exp_and_avg_of_F_exp) # =
SST

421
422     # Calc R2 = 1-SRS/SST
423     _R_squared = 1.0 - (_sum_of_residuals_squared/
424     _sum_of_squared_differences_between_F_exp_and_avg_of_F_exp)
425     print(f' R2(week: {t_start}-{t_end}) = {_R_squared}')

426     if verbose == 2:
427         # Check R2 by calc R2 with sklearn.metrics r2_score
428         package
429             _r2 = r2_score(_F_exp, _F_num_at_discrete_points)
430             print(f' R2(week: {t_start}-{t_end}) = {_r2} (
431             calculated by sklearn.metrics.r2_score package)')

432
433 def print_save_R_squared(F_num_at_discrete_points, F_exp, t_exp,
434     t_start, t_end, parameters_to_optimize, A_m_t0, D_ref, c_ref,
435     l_ref, verbose, menu, distribution_menu, opt_time_frame):
436     """ R2 = 1-SRS/SST; SSR = sum of residuals squared, SST = total
437     sum of squares"""
438     # https://www.ncl.ac.uk/webtemplate/ask-assets/external/mathss-
439     resources/statistics/regression-and-correlation/coefficient-of-
440     determination-r-squared.html
441
442     # Initialize internal parameters
443     _t_exp = t_exp
444     _F_exp = F_exp
445     _F_num_at_discrete_points = F_num_at_discrete_points
446
447     """
448     # Consider time frame
449     if t_start != 0:
450
451         # Calc position of t_start in t_exp
452         t_start_count = t_exp.index(t_start)
453
454         # Remove elements from list
455         _t_exp = _t_exp[t_start_count:]
456         _F_exp = _F_exp[t_start_count:]
457         _F_num_at_discrete_points = _F_num_at_discrete_points[
458             t_start_count:]
459
460     # Initialize and calc sum of squared residuals (SSR = sum of
461     # residuals squared)
462     _residuals_squared = []
463
464     for i in range(len(_F_exp)):
```

```
459     _residuals_squared += [(_F_exp[i]-_F_num_at_discrete_points[  
460     i])**2]  
461  
462     _sum_of_residuals_squared = sum(_residuals_squared)  
463  
464     # Initialize and calc sum of squared differences between F_exp  
465     # and the average of F_exp (SST = total sum of squares)  
466     _avg_F_exp = sum(_F_exp)/len(_F_exp)  
467     _squared_differences_between_F_exp_and_avg_of_F_exp = []  
468  
469     for j in range(len(_F_exp)):  
470         _squared_differences_between_F_exp_and_avg_of_F_exp += [(  
471             _F_exp[j]-_avg_F_exp)**2]  
472  
473     _sum_of_squared_differences_between_F_exp_and_avg_of_F_exp = sum  
474     (_squared_differences_between_F_exp_and_avg_of_F_exp) # = SST  
475  
476  
477     # Calc R2 = 1-SRS/SST  
478     _R_squared = 1.0 - (_sum_of_residuals_squared/  
479     _sum_of_squared_differences_between_F_exp_and_avg_of_F_exp)  
480  
481     # Save R2 to CSV  
482     date = str(datetime.date.today()) # convert date to str  
483  
484     if distribution_menu == 0:  
485         start_distribution = "constant_c0"  
486     elif distribution_menu == 1:  
487         start_distribution = "parabolic_c0"  
488  
489     if menu == 1:  
490  
491         [D, alpha, l_b, A_m_degrad, A_m_t0, const_degrad,  
492         degrad_break] = parameters_to_optimize  
493  
494         # Create path  
495         path_1 = str("results/01_simulation/"+date+"/"+  
496         start_distribution+"/_R_squared") # create path to .csv-file  
497  
498         # Create folder  
499         if not os.path.exists(path_1):  
500             os.makedirs(path_1)  
501  
502         # Create filename  
503         filename_1 = "/" + date + "__" + start_distribution + "__" +  
504         "_R_squared.csv"  
505  
506         # Write column name to .csv-file  
507         if not os.path.isfile(path_1 + filename_1):  
508             with open(path_1 + filename_1, 'w') as f:
```

```

502         writer = csv.writer(f, delimiter='\t')
503         writer.writerow(['R**2 = 1-SRS/SST', 'D [mm**2/s]', 
504 'alpha [mm**3/g]', 'l_b [mm]', 'A_m_degrad = x * A_m_t0 [-]', ' 
505 const_degrad [-]', 'degrad_break [-]'])
506
507     # Write results to .csv-file
508     with open(path_1 + filename_1, 'a') as f:
509         writer = csv.writer(f, delimiter='\t')
510         writer.writerow([_R_squared, D * D_ref, alpha / c_ref,
511 l_b * l_ref, A_m_degrad/A_m_t0, const_degrad, degrad_break])
512         f.close()
513
514 elif menu == 2:
515
516     if t_start == 0 and opt_time_frame == 'until_break':
517
518         [D, alpha, l_b] = parameters_to_optimize
519
520         # Create path
521         path_2a = str("results/02_optimization/" + date + "/" +
522 start_distribution + "/until_break") # create path to .csv-file
523
524         # Create folder
525         if not os.path.exists(path_2a):
526             os.makedirs(path_2a)
527
528         # Create filename
529         filename_2a = "/" + date + "__" + start_distribution + " "
530         __" + "R_squared_Optimization_until_break.csv"
531
532         # Write column name to .csv-file
533         if not os.path.isfile(path_2a + filename_2a):
534             with open(path_2a + filename_2a, 'w') as f:
535                 writer = csv.writer(f, delimiter='\t')
536                 writer.writerow(['Rš = 1-SRS/SST', 'D [mmš/s]', 
537 'alpha [mmš/g]', 'l_b [mm]'])
538
539         # Write results to .csv-file
540         with open(path_2a + filename_2a, 'a') as f:
541             writer = csv.writer(f, delimiter='\t')
542             writer.writerow([_R_squared, D*D_ref, alpha/c_ref,
543 l_b*l_ref])
544             f.close()
545
546     elif t_start == 0 and opt_time_frame == 'after_break':
547
548         A_m_degrad = parameters_to_optimize[0]
549
550         # Create path

```

```
545     path_2b = str("results/02_optimization/" + date + "/" +
546     start_distribution + "/after_break") # create path to .csv-file
547
548     # Create folder
549     if not os.path.exists(path_2b):
550         os.makedirs(path_2b)
551
552     # Create filename
553     filename_2b = "/" + date + "__" + start_distribution + "__"
554     + "R_squared_Optimization_after_break.csv"
555
556     # Write column name to .csv-file
557     if not os.path.isfile(path_2b + filename_2b):
558         with open(path_2b + filename_2b, 'w') as f:
559             writer = csv.writer(f, delimiter='\t')
560             writer.writerow(['R2 = 1-SRS/SST', 'Am_degrad [mm3]', 'Am_degrad/Am_t0 [-]'])
561
562     # Write results to .csv-file
563     with open(path_2b + filename_2b, 'a') as f:
564         writer = csv.writer(f, delimiter='\t')
565         writer.writerow([_R_squared, A_m_degrad*l_ref**2,
566         A_m_degrad/A_m_t0])
567         f.close()
568
569     # Print
570     if verbose != 0:
571
572         # Print R2
573         print('')
574         print(f' R2(week: {t_start}-{t_end}) = {_R_squared}')
575
576         if verbose == 2:
577
578             # Check R2 by calc R2 with sklearn.metrics r2_score
579             package
580             _r2 = r2_score(_F_exp, _F_num_at_discrete_points)
581             print(f' R2(week: {t_start}-{t_end}) = {_r2} (calculated by sklearn.metrics.r2_score package)')
582
583
584     return _R_squared
585
586
587
588 def print_residual_at_week_5(Fnum_at_discrete_points, F_exp,
589     verbose, opt_time_frame):
590
591     if opt_time_frame == 'until_break' and verbose != 0:
592
593         # Calc residuals at t=5
```

```
589     print(' residual (week: 5) =', F_exp[5]-
590           Fnum_at_discrete_points[5])
591     print("-----")
592     print('')
593
594
595 def print__convergence_rate(rate, nx, element_family, degree):
596
597     for i in range(len(nx)-1):
598         print('      nx=', nx[i], '--> nx=', nx[i+1])
599
600         for n in range(len(degree)):
601
602             print(element_family, degree[n], ': r =', rate[n][i])
603
604         print('-----')
605
606
607
608 def print_opt_parameters(parameters_to_optimize, const_degrad,
609                         A_m_t0, c_ref, D_ref, l_ref, opt_time_frame, verbose):
610
611     if opt_time_frame == 'until_break':
612
613         D_opt_units      = parameters_to_optimize[0] * D_ref
614         alpha_opt_units = parameters_to_optimize[1] / c_ref
615         l_b_opt_units   = parameters_to_optimize[2] * l_ref
616
617         print('')
618         print('-----')
619         print('# D_opt = ', D_opt_units, '[mm^2/s] ,       alpha_opt'
620               '=', alpha_opt_units, '[mm^3/g] ,       l_b_opt = ',
621               l_b_opt_units, '[mm^3/g]')
622         print('')
623         print('-----')
624
625     elif opt_time_frame == 'after_break':
626
627         A_m_degrad_opt_units = parameters_to_optimize[0] * l_ref**2
628         q_A_m = parameters_to_optimize[0]/A_m_t0
629
630         print('')
631         print('-----')
632         print('# A_m_degrad_opt = ', A_m_degrad_opt_units, '[mm^3/s]
633               mit const_degrad = ', const_degrad, "[-]")
634         print('# A_m_degrad_opt = ', q_A_m, '* A_m_t0')
```

```
631     print('
632 #-----')
633
634
635 def print_plot_all_results(results, menu, t_exp, F_exp, T,
636   kind_of_1d_interpolation, element_family, degree, nx, h, D,
637   alpha, l_b, const_degrad, degrad_break, A_m_t0, A_m_t0_units,
638   A_m_degrad_constant_units, A_m_degrad_parabolic_units,
639   opt_success, c_ref, D_ref, l_ref):
640
641   if menu == 1: # simulation mode was chosen
642
643       # Extract parameters
644       _t_num = results[0]
645       _F_num = results[1]
646       _distribution_menu = results[2]
647
648       # Define dummy variables to be able to use the same
649       # functions
650       DUMMY_opt_time_frame = 'after_break'
651
652       if _distribution_menu == 0:
653           DUMMY_parameters_to_optimize = [D, alpha, l_b,
654             A_m_degrad_constant_units, A_m_t0_units, const_degrad,
655             degrad_break]
656       elif _distribution_menu == 1:
657           DUMMY_parameters_to_optimize = [D, alpha, l_b,
658             A_m_degrad_parabolic_units, A_m_t0_units, const_degrad,
659             degrad_break]
660
661
662       # Post-processing
663       # t = 0-5
664       F_num_at_discrete_points_0_5 =
665       calc_F_num_at_discrete_time_steps(_t_num, _F_num, t_exp, 5.0,
666         kind_of_1d_interpolation)
667       t_exp_opt_0_5, F_exp_opt_0_5 =
668       adjust_t_exp_and_F_exp_to_optimization_time_frame(t_exp, F_exp,
669         5.0)
670       R_squared_0_1 = print__R_squared_for_given_time_frame(
671         F_num_at_discrete_points_0_5, F_exp_opt_0_5, t_exp_opt_0_5,
672         0.0, 1.0, 2,'until_break') # R2 initial burst (week:0-1)
673       R_squared_3_5 = print__R_squared_for_given_time_frame(
674         F_num_at_discrete_points_0_5, F_exp_opt_0_5, t_exp_opt_0_5,
675         3.0, 5.0, 2,'until_break') # R2 during stagnation (week:3-5)
676       # t = 0-15
677       F_num_at_discrete_points_0_15 =
678       calc_F_num_at_discrete_time_steps(_t_num, _F_num, t_exp, T,
679         kind_of_1d_interpolation)
680       R_squared_0_15 = print_save__R_squared(
```

```

F_num_at_discrete_points_0_15, F_exp, t_exp, 0.0, T,
DUMMY_parameters_to_optimize, A_m_t0, D_ref, c_ref, l_ref, 2,
menu, _distribution_menu, DUMMY_opt_time_frame)
662     plot_F_exp_vs_F_num(DUMMY_parameters_to_optimize, D_ref,
c_ref, l_ref, t_exp, _t_num, F_exp, _F_num, T, R_squared_0_15,
kind_of_1d_interpolation, 2, menu, opt_success,
_distribution_menu, DUMMY_opt_time_frame)

663
664
665     elif menu == 2: # inverse analysis
666
667         # Extract optimized parameters
668         _res_opt = results[0]
669
670         # Extract t_num & F_num for optimized parameters
671         _t_num = results[1]
672         _F_num = results[2]
673
674         # Extract various
675         _opt_time_frame = results[3]
676         _distribution_menu = results[4]
677
678         print('')
679         print('')
680         print('')
681         print('
#-----')
682         print('
#-----')
683         print('# S U C C E S S F U L L Y   O P T I M I Z E D')
684         print('
#-----')
685         print('
#-----')
686         print('
#-----')
687
688         # Post-processing
689         print_opt_parameters(_res_opt.x, const_degrad, A_m_t0, c_ref
, D_ref, l_ref, _opt_time_frame, 2)
690         F_num_at_discrete_points =
calc_F_num_at_discrete_time_steps(_t_num, _F_num, t_exp, T,
kind_of_1d_interpolation)
691         R_squared_0_T = print_save_R_squared(
F_num_at_discrete_points, F_exp, t_exp, 0, T, _res_opt.x, A_m_t0
, D_ref, c_ref, l_ref, 2, 'dummy', _distribution_menu,
_opt_time_frame) # menu = 'dummy' to avoid saving R(0,15) to R
(0,5) resp. R(5,15)
692         plot_F_exp_vs_F_num(_res_opt.x, D_ref, c_ref, l_ref, t_exp,
_t_num, F_exp, _F_num, T, R_squared_0_T,
kind_of_1d_interpolation, 2, menu, opt_success,
_distribution_menu, _opt_time_frame)

```

```
693
694
695 elif menu == 4: # convergence analysis
696
697     # Extract parameters
698     _nx = results[0]
699     _nt = results[1]
700     _L2_error_avg_list = results[2]
701     _rate = results[3]
702     _diff_F_num_total_avg = results[4]
703     _space_or_time = results[5]
704     _discretization_scheme = results[6]
705
706     if 1 <= _discretization_scheme <= 5:
707
708         plot__convergence_analysis_results(_nt, _nx,
709         _L2_error_avg_list, degree, _space_or_time)
710
711         if _space_or_time == 1:
712             # Print convergence rate
713             print__convergence_rate(_rate, _nx, element_family,
714             degree)
715
716     elif _discretization_scheme == 6 or _discretization_scheme
717     == 7:
718
719         plot__convergence_analysis_results(_nt, _nx,
720         _diff_F_num_total_avg, _space_or_time)
721
722 elif menu == 5:
723
724     # Extract parameters
725     rel_err_percent_list = results[0]
726     distribution_menu = results[1]
727
728     # Plot
729     plot__rel_err_m_d(nx, rel_err_percent_list, element_family,
730     degree, distribution_menu)
731     plt.show()
732
733
734 def print__t_dt_num_steps(t, dt, num_steps, T_explicit_Euler, menu,
735     verbose):
736
737     if menu == 1 or menu == 2:
```

```

738     if verbose == 0 and menu == 2 and t > T_explicit_Euler:
739         print(f'{num_steps}. CN-step: t = {t} (dt = {dt})')
740
741     if verbose == 1:
742
743         if t > T_explicit_Euler:
744
745             print(f'{num_steps}. CN-step: t = {t} (dt = {dt})')
746
747     elif verbose == 2:
748
749         if t > T_explicit_Euler:
750             print(f'{num_steps}. CN-step: t = {t} (dt = {dt})')
751
752         else:
753             print(f'{num_steps}. EULER-step: t = {t} (dt = {dt})')
754
755     ')
756
757
758
759 def plot__rel_err_m_d(nx, rel_err_percent_list, element_family,
760                      degree, distribution_menu):
761
762     # plots initialisieren
763     fig, ax = plt.subplots(1, 2)
764
765     if distribution_menu == 0:
766         plt.suptitle(f"CONSTANT distribution for u(t=0)", fontsize=15)
767     elif distribution_menu == 1:
768         plt.suptitle(f"PARABOLIC distribution for u(t=0)", fontsize=15)
769
770     # Achsenbezeichnung
771     ax[0].set_xlabel(r"$n \backslash \mathrm{s} [-]", loc="right")
772     ax[1].set_xlabel(r"$n \backslash \mathrm{s} [-]", loc="right")
773     ax[0].set_ylabel(r"$\backslash err_{m \backslash \mathrm{d}} [\%]", loc="top",
774                      rotation="horizontal")
775     ax[1].set_ylabel(r"$\backslash err_{m \backslash \mathrm{d}} [\%]", loc="top",
776                      rotation="horizontal")
777
778     # Plot
779     ax[0].plot(nx, rel_err_percent_list[0], color='orange', marker='x',
780                label=f'{element_family}{degree[0]}-Element')
781     ax[1].plot(nx, rel_err_percent_list[1], color='red', marker='x',
782                label=f'{element_family}{degree[1]}-Element')
783
784     # Show labels

```

```
781     ax[0].legend()  
782     ax[1].legend()
```