

# Analyse und Test von Methoden zur automatischen Reglersynthese

Bachelorarbeit in Electrical and Computer  
Engineering



Alex Krieg  
Herbstsemester 2025

**Dozent:** Prof. Dr. Lukas Ortmann  
**Betreuer:** Prof. Dr. Lukas Ortmann  
**Modul:** Regelungstechnik

19. Dez. 2025  
Version 1.0.1

## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Problemstellung	1
1.2 Aufgabenstellung	1
1.3 Aufbau des Dokuments	1
1.4 Verwendete Tools	1
1.5 Quellcode und Dokumentation	1
1.6 Methodik	2
<b>2 Recherche</b>	<b>4</b>
2.1 Recherchierte Methoden	4
2.2 Ergebnis	7
2.3 Kombination mehrerer Methoden	9
2.4 Auswahl der zu vertiefenden Methoden	9
<b>3 Systune am DC-Motor</b>	<b>11</b>
3.1 Regelkreis Definieren	11
3.2 Optimierungsziele definieren	12
3.3 Optimierung durchführen	12
3.4 Optimierung auswerten	13
3.5 Optimierten Regler am realen System testen	15
<b>4 Genetischer Algorithmus am DC-Motor</b>	<b>16</b>
4.1 System Definieren	16
4.2 Optimierungsziele definieren	17
4.3 Optimierung durchführen	20
4.4 Optimierung auswerten	21
4.5 Optimierten Regler am realen System testen	21
<b>5 Differential Evolution Algorithmus am DC-Motor</b>	<b>22</b>
5.1 Optimierung durchführen	22
5.2 Optimierung auswerten	23
5.3 Optimierten Regler am realen System testen	23
<b>6 Auswertung der Regleroptimierung am DC-Motor</b>	<b>24</b>
6.1 Synthetisierte Regler	24
6.2 Systemverhalten am realen Prozess	24
6.3 Lernverlauf	25
6.4 Frequenzanalyse der optimierten Regler	26
6.5 Frequenzanalyse der optimierten Systeme	27
6.6 Stabilitätsanalyse der optimierten Systeme	28
<b>7 Systune am Motor mit Schwungmasse</b>	<b>30</b>
7.1 Regelkreis Definieren	30
7.2 Optimierungsziele definieren	30
7.3 Regler extrahieren	31
7.4 Optimierung auswerten	31
7.5 Optimierten Regler am realen System testen	34
<b>8 GA am Motor mit Schwungmasse</b>	<b>35</b>
8.1 System Definieren	35
8.2 Optimierungsziele definieren	36
8.3 Optimierung durchführen	39
8.4 Optimierung auswerten	40
8.5 Optimierten Regler am realen System testen	40
<b>9 DE Algorithmus am Motor mit Schwungmasse</b>	<b>41</b>
9.1 Optimierung durchführen	41
9.2 Optimierung auswerten	42
9.3 Optimierten Regler am realen System testen	42
<b>10 Auswertung der Regleroptimierung am Motor mit Schwungmasse</b>	<b>43</b>
10.1 Synthetisierte Regler	43
10.2 Systemverhalten am realen Prozess	43
10.3 Lernverlauf	44
10.4 Frequenzanalyse der optimierten Regler	45
10.5 Frequenzanalyse der optimierten Systeme	46
10.6 Stabilitätsanalyse der optimierten Systeme	47
<b>11 Schlussteil</b>	<b>49</b>
11.1 Objektives Fazit	49
11.2 Persönliches Fazit	52
11.3 Weiterführende Arbeiten	57

---

11.4 Danksagung . . . . .	57
11.5 Erklärung zur Urheberschaft . . . . .	58
<b>A Anhang</b>	<b>59</b>
A.1 Formeln und Herleitungen . . . . .	59
A.2 Diverse Erklärungen . . . . .	62
A.3 Verzeichnisse . . . . .	84
A.4 Zeitplan . . . . .	90

# 1 | Einleitung

## 1.1 | Problemstellung

Für die **Reglersynthese** gibt es eine Vielzahl von Methoden. Diese unterscheiden sich zum einen durch ihre mathematische Herangehensweise und zum anderen durch das benötigte **Modellwissen** über den **Prozess**. Außerdem gibt es Unterschiede wie automatisiert die Reglersynthese abläuft und wie die Methode parametrisiert wird.

## 1.2 | Aufgabenstellung

Der Fokus dieser Arbeit soll auf der Parametrisierung der Methode liegen, also darauf, wie und in welcher Form der Methode kommuniziert wird, welches Verhalten der geschlossene Regelkreis haben soll. Diese Arbeit soll aufzeigen, welche Methoden zur automatischen Reglersynthese existieren und für welche Anwendungsfälle diese geeignet sind.

Die besten Methoden sollen implementiert und an verschiedenen Prozessen getestet werden. Zuletzt soll eine interaktive GUI erstellt werden mit der die unterschiedlichen Methoden auf einem Prozess angewendet werden können. Das Ziel der Arbeit ist es eine Aussage darüber zu treffen, welche Methoden, je nach Prozess und gewünschtem Verhalten, zur automatischen Reglersynthese geeignet sind, wie diese funktionieren und welche Vor- und Nachteile diese haben. Zur besseren Vermittlung des gewonnenen Wissens sollen die Methoden in einer GUI auf einem Beispielprozess implementiert werden.

## 1.3 | Aufbau des Dokuments

Das Dokument besteht aus 4 Teilen.

- Recherche zu verschiedenen Methoden für die automatische Reglersynthese
- Implementierung und Testen der ausgewählten Methoden an zwei Prozessen
  - DC-Motor
  - Motor-mit-Schwungmasse
- Schlussteil mit Fazit
- Anhang mit diversen theoretischen Erklärungen zu verschiedenen Themen die in der Arbeit vorkommen

### 1.3.1 | Abkürzungen

Bezeichnung	Beschreibung
DE	Differential Evolution
GA	Genetischer Algorithmus
MIMO	Multiple Input Multiple Output
NEAT	NeuroEvolution of Augmenting Topologies
NN	Neuronales Netzwerk
SISO	Single Input Single Output

### 1.3.2 | Randnotiz zu den getesteten Systemen

In dieser Arbeit werden zwei verschiedene **Prozesse** mit einem PID-Regler geregelt.

Mir ist durchaus bewusst, dass ein PID-Regler nicht zwingend die beste Wahl ist. Es geht in dieser Arbeit jedoch nicht darum, einen optimalen Regler für die Prozesse zu entwerfen, sondern die verschiedenen Methoden der automatischen Reglersynthese anzuwenden und zu vergleichen. Die Wahl eines nicht für alle Prozesse optimalen Reglers, kann als Challenge gesehen werden, ein möglichst gutes Ergebnis mit den gegebenen Mitteln zu erzielen.

## 1.4 | Verwendete Tools

In dieser Arbeit wurden folgende Tools verwendet:

- **LaTeX** [9] für die Dokumentation
- **MATLAB/Simulink** [10] für die Simulation der Systeme und die Verwendung der *systune* Methode
- **Visual Studio Community 2022** [15] für die C++ Implementierung der Optimierungsalgorithmen **GA** und **DE** und die GUI.
  - Verwendeter Compiler: **MSVC**
  - Build System: **CMake** [5]
  - GUI Framework: **Qt 5.15.2** [12]
  - **SFML: SFML 2.6.1** [13] für die Visualisierung der Simulation
  - **ImGui: ImGui 1.91.5** [7] als Grundlage für **ImPlot**
  - **ImPlot: ImPlot 0.16** [8] für die Plot Elemente in der GUI
- **Visual Studio Code** [14] für die Bearbeitung von LaTeX Dateien
- **Processing** [11] für die Erprobung und Visualisierung vom Differential Evolution Algorithmus in einer einfachen Umgebung
- **Git** [6] für die Versionskontrolle des Codes und der Dokumentation
- **ChatGPT, ClaudeAI und GitHub Copilot** [1] [4] [16] für die Unterstützung bei der Code Ergänzung und Hilfe bei der Erstellung von MATLAB-Skripte und LaTeX Makros.

## 1.5 | Quellcode und Dokumentation

Der Quellcode und die Dokumentation dieser Arbeit sind im GitHub Repository [2] verfügbar. Dort sind auch alle MATLAB-Skripte und Simulink-Modelle zu finden, die in dieser Arbeit verwendet wurden.

## 1.6 | Methodik

Durch eine Literaturrecherche sollen automatisierbare Optimierungsverfahren ermittelt werden. Anschliessend werden die gefundenen Methoden verglichen. MATLAB dient in diesem Teil der Arbeit dazu, die Methoden grob zu testen und zu erproben. Für den Vergleich und die daraus folgende Auswahl der zu vertiefenden Methoden werden verschiedene Aspekte betrachtet.

Es muss sich die Frage gestellt werden, wie ein guter Regler definiert ist. Jede Anwendung hat unterschiedliche Anforderungen an einen guten Regler und demnach kann nicht eindeutig bestimmt werden, wie das Verhalten eines guten Reglers aussieht. Es können jedoch einige Eigenschaften des gesamten [Systems](#) gemessen werden, um eine Aussage über die Qualität des Reglers zu treffen.

### 1.6.1 | Aspekte zur Bewertung eines Reglers

#### Stabilität

Ein stabiler Regelkreis ist im Normalfall eine Grundvoraussetzung an ein System. In sehr spezifischen Anwendungen kann jedoch auch ein instabiles Verhalten explizit gesucht sein. Im weiteren Verlauf dieser Arbeit wird jedoch nur auf stabile Regelkreise eingegangen.

- Der Regler muss in der Lage sein das System zu stabilisieren
- Der Regler muss in der Lage sein, ein System stabil zu halten

#### Zeitliches Verhalten

Das zeitliche Verhalten des Systems beschreibt, wie der Regler auf eine plötzliche Änderung des Sollwerts reagiert. Abhängig von der jeweiligen Anwendung kann gefordert sein, dass der Regler sehr schnell oder eher langsam auf Sollwertänderungen anspricht. Schnelle Regler haben jedoch häufig den Nachteil, dass sie weniger robust gegenüber Störungen und Modellabweichungen sind und stärker zu Schwingungen neigen.

- Wie schnell lässt sich die Regelstrecke auf den gewünschten Sollwert bringen?
- Wie verhält sich der Regler dabei? (Überschwingen?, Einschwingzeit?, etc.)

#### Robustheit

Die Robustheit beschreibt die Fähigkeit des Reglers, auch bei Störungen, Parameteränderungen oder Modellunsicherheiten ein stabiles und gewünschtes Systemverhalten sicherzustellen. Ein robuster Regler gewährleistet dabei, dass Leistungsanforderungen und Toleranzen trotz dieser Einflüsse eingehalten werden.

- Wie gut kann der Regler mit Störungen umgehen?
- Wie gut kann der Regler mit Modellabweichungen umgehen?
- Wie gut kann der Regler mit Änderungen eines realen Prozesses umgehen?  
Z.B. durch Verschleiss
- Wie gut kann der Regler mit den realen Nichtlinearitäten eines Prozesses umgehen?
- Betrachtung der Phasen- und Amplitudengangreserve als Indikator für die Robustheit

#### Stationäres Verhalten

Beim stationären Verhalten wird untersucht, wie sich das System verhält, nachdem eine ausreichend lange Zeit vergangen ist. Dies ist entscheidend, um festzustellen, ob das System den Sollwert tatsächlich erreicht oder ob dauerhaft eine bestimmte Abweichung bestehen bleibt.

- Wie genau wird der Sollwert im stationären Zustand erreicht/gehalten, wie klein wird der stationäre Fehler (Abweichung zum Sollwert)?

#### Komplexität des Reglers

Wie viele Parameter des Reglers müssen durch die Methode optimiert werden. Die Anzahl dieser Parameter, definiert die Dimension des Suchraums der Optimierung.

### 1.6.2 | Methodenspezifische Aspekte

Neben den oben genannten Aspekten, die zur Bewertung eines Reglers dienen können, werden für die Auswahl der zu vertiefenden Methoden noch weitere Aspekte betrachtet. Diese beziehen sich auf die Methode selbst und nicht auf den Regler.

#### Voraussetzungen der Systeme

Voraussetzungen die das System erfüllen muss, um von der jeweiligen Methode optimiert zu werden.

- Kann ein stabiles und instabiles System optimiert werden?
- Kann ein [SISO](#)- und [MIMO](#)-System optimiert werden?
- Kann nur ein lineares System optimiert werden, oder auch ein [nichtlineares System](#) oder sogar [zeitvariantes System](#)?

#### Die Schwierigkeit der Parametrierung

- Wie komplex ist die Parametrierung der Methode?
- Wie viele und welche Informationen muss der Anwender der Methode bereitstellen, um eine Optimierung durchzuführen?
- Lässt sich die Informationsbereitstellung einfach für den Anwender gestalten?
- Bestimmung der [Hyperparameter](#) der Methode und deren Einflussbereich.

#### Automatisierungsgrad

- Wie automatisiert läuft die Optimierung ab?
- Welche Tools/Technologien werden benötigt um die Methode anzuwenden?
- Wie viel Eingriff des Anwenders ist notwendig um eine Optimierung durchzuführen?

### 1.6.3 | Zusammenfassend

Insgesamt gibt es viele verschiedene Aspekte und Kriterien, die bei der Auswahl und Bewertung von Methoden zur Reglersynthese berücksichtigt werden müssen. Die oben genannten Punkte bieten eine gute Grundlage, um die Eignung einer Methode für ein bestimmtes System zu beurteilen. Die Auswahl der zu vertiefenden Methoden erfolgt demnach zusammenfassend anhand folgender Punkte:

- Mögliches Verhalten das durch den resultierenden Regler erreicht werden kann.
  - Stabilität
  - Zeitliches Verhalten
  - Stationäres Verhalten
  - Robustheit
- Voraussetzungen der Systeme, an denen die jeweilige Methode angewendet werden kann
  - stabile/instabile Systeme
  - SISO/MIMO Fähigkeit
  - lineare/nichtlineare Systeme & zeitinvariante/zeitvariante Systeme
  - Regler Strukturen, Anzahl der zu optimierenden Parameter
- Die Schwierigkeit der Parametrierung
  - Benutzerfreundlichkeit der Parametrisierung
  - Benötigtes Vorwissen des Anwenders bezüglich Regelungstechnik
- Automatisierungsgrad
  - Tools/Technologien
  - Eingriffe des Anwenders

Einige der besten Methoden, werden an verschiedenen Prozessen getestet und bewertet.

## 2 | Recherche

In der Recherche wird nach möglichen automatisierbaren Methoden oder Algorithmen für die Reglersynthese gesucht.

### 2.1 | Recherchierte Methoden

Nachfolgend sind die recherchierten Methoden zusammenfassend aufgeführt und mit Vor- und Nachteilen gekennzeichnet.

#### 2.1.1 | Lambda-Tuning

Lambda-Tuning [37] basiert auf dem Prinzip der **Pol-Nullstelle-Kürzung**. Die Methode ermöglicht es, mit Hilfe eines einzigen Parameters, das Verhalten des **Systems** einfach zu beeinflussen. Um die Methode anwenden zu können, muss die Regelstrecke eine spezifische Form aufweisen. Zudem muss das offene System bereits stabil sein, da die Pol-Nullstelle-Kürzung ein instabiles System nicht stabilisieren kann. Dies schränkt den praktischen Gebrauch dieser Methode auf wenige Anwendungsgebiete ein.

- + Verwendet nur einen Optimierungsparameter:  $\lambda$
- + Einfach automatisierbar
- + Benutzer muss nur einen Parameter einstellen. Dies kann mit einem GUI-Element wie z.B. einem Slider umgesetzt werden
- Die Regelstrecke muss stabil sein
- Die Regelstrecke muss eine spezifische Form aufweisen
- Die Regelstrecke muss linear und zeitinvariant sein
- Nicht geeignet für **MIMO**-Systeme
- Reglerstruktur ist auf PID-Regler beschränkt

#### 2.1.2 | Polplatzierung

Mit Hilfe der **Polplatzierung** lassen sich die Pole des geschlossenen Regelkreises frei platzieren. Mit der Möglichkeit Pole frei zu platzieren kann ein guter Regler ausgelegt werden. Die Schwierigkeit besteht jedoch darin, zu wissen wo die Pole platziert werden sollen. Mit wenig Erfahrung lassen sich die richtigen Polstellen nicht finden. Den besten Regler mit dieser Methode zu finden ist also sehr schwierig. Außerdem wird voller Zugriff auf den gesamten Zustandsvektor benötigt, was nicht für jedes System immer möglich ist, weshalb in solchen Fällen eine zusätzliche Ergänzung eines **Beobachters** benötigt wird.

- + Volle Kontrolle über die Pole des geschlossenen Regelkreises
- + Einfach automatisierbar
- + Kann auf instabile Systeme angewendet werden
- + Geeignet für **MIMO**-Systeme
- + Die Platzierung der Pole kann mit Hilfe eines GUI-Editors graphisch erfolgen
- Benötigt ein intuitives Verständnis für den Einfluss der Polpositionen auf das System
- Die Regelstrecke muss linear und zeitinvariant sein

#### 2.1.3 | Linear Quadratic Regulator (LQR)

Findet den optimalen Zustandsregler für die zu definierende, quadratische Kostenfunktion. Für die Kostenfunktion können die Zustandswerte, die Eingangssignale und das Kreuzprodukt der Zustände mit den Eingängen, gewichtet verwendet werden. Es wird voller Zugriff auf den gesamten Zustandsvektor benötigt, was nicht für jedes System immer möglich ist, weshalb in solchen Fällen eine zusätzliche Ergänzung eines Beobachters benötigt wird. Eine Schwierigkeit dieser Methode ist es, die passenden Gewichtungen der Kostenfunktion zu finden.

- + Volle Kontrolle über die Pole des geschlossenen Regelkreises
- + Einfach automatisierbar
- + Kann auf instabile Systeme angewendet werden
- + Geeignet für **MIMO**-Systeme
- System sollte in Regelungsnormalform vorliegen damit das bestimmen der Kostenmatrizen einfacher ist
- Bestimmung der Gewichtungsmatrizen ist nicht intuitiv
- Die Regelstrecke muss linear und zeitinvariant sein

#### 2.1.4 | Ziegler-Nichols

Mit Hilfe von Ziegler-Nichols und ähnlichen Varianten wie:

- Ziegler-Nichols mit Sprungantwort [38]
- Ziegler-Nichols mit Hilfe der Stabilitätsgrenze [34] [35]
- Chien-Hrones-Reswick [38]
- Latzel [38]

kann ein einfacher PID-Regler mit empirischen Daten entworfen werden. Die Methoden sind sehr einfach in der Anwendung, jedoch nicht optimal und in der Praxis auch nicht immer umsetzbar. Ein System kann eventuell nicht immer, mit einem Sprung angeregt werden, um die benötigten Messwerte zu erhalten. Für **MIMO**-Systeme sind diese Methoden ungeeignet, weil der Aufwand sehr gross wäre, für jede Übertragungsfunktion, von jedem Eingang zu jedem Ausgang, ein Regler mit diesen Methoden einzeln zu bestimmen. Auch wenn dieser Aufwand betrieben wird, wären die gegenseitigen Einflüsse der Eingänge nicht berücksichtigt. Das würde nur bei Systemen klappen die sich komplett in mehrere **SISO**-Systeme zerlegen lassen. Dann wäre das Problem auf mehrere **SISO**-Systeme aufgeteilt und könnten individuell betrachtet werden. Bei den meisten **MIMO**-Systemen sind die Zustände jedoch abhängig von mehreren Eingängen und Zuständen. Für instabile Systeme sind diese Methoden nicht anwendbar da die benötigten Daten nicht ermittelt werden können.

- + Benötigt kein Modell der Regelstrecke
- + Einfach automatisierbar und mit einem Modell besonders einfach automatisierbar
- + Benötigt keine komplexe Mathematik
- + GUI-Gestaltung ist sehr einfach da es nur wenige tabellarische Auswahlmöglichkeiten gibt
- Kann nicht auf instabile Systeme angewendet werden
- Kann nur für **minimalphasige Systeme** angewendet werden
- Ziegler-Nichols mit Hilfe der Stabilitätsgrenze kann ohne Modell am realen Regelstrecke kaum umgesetzt werden
- Die Methoden dienen nur für einen ersten Reglerentwurf und sind keineswegs optimal
- Nur Systeme der Form PT2 oder PT3 geeignet
- Bei schnellen Regelstrecken kaum umsetzbar
- Nicht geeignet für **MIMO**-Systeme
- Reglerstruktur ist auf PID-Regler beschränkt

### 2.1.5 | MATLAB *systune*

Mit *systune* [27] kann ein komplettes System, flexibel auf das gewünschte Verhalten optimiert werden. *Systune* verwendet dafür ein *genss*-Modell [19]. Es handelt sich dabei um ein verallgemeinertes Zustandsraum Modell, das zusätzlich noch Optimierungsparameter beinhaltet, welche durch MATLAB optimiert werden. Für die Optimierung werden Optimierungsziele [23] definiert, welche auf zwei Arten verwendet werden können.

- **Harte Ziele** [28] werden priorisiert bei der Optimierung
- **Weiche Ziele** [29] können vernachlässigt werden um die **harten Ziele** zu erreichen

MATLAB *systune* bietet als Ergebnis der Optimierung nicht nur die optimierten Parameter, sondern auch eine Auswertung der Optimierung für jedes Optimierungsziel.

- + Sehr flexibel
- + Einfach automatisierbar
- + Optimierung kann detailliert vorgenommen werden
- + Kann auf instabile Systeme angewendet werden
- + Geeignet für MIMO-Systeme
- + Kann beliebig viele Parameter optimieren
- Benötigt MATLAB
- Das Modell muss als *genss*-Modell [19] vorliegen
- GUI-Gestaltung zur Eingabe der Optimierungsziele ist komplex
- Reglerstruktur muss vorgegeben werden

### 2.1.6 | MATLAB *looptune*

Mit *looptune* [21] wird eine Feedbackschleife optimiert. *Looptune* verwendet für die Optimierung zum einen die Ziel-crossover Frequenz/Frequenzbereich und zum anderen können auch Optimierungsziele [23] definiert werden. MATLAB *looptune* bietet als Ergebnis der Optimierung nicht nur die optimierten Parameter, sondern auch eine Auswertung der Optimierung für jedes Optimierungsziel.

- + Kann auf instabile Systeme angewendet werden
- + Geeignet für MIMO-Systeme
- + Kann beliebig viele Parameter optimieren
- Benötigt MATLAB
- Das Modell muss als *genss*-Modell [19] vorliegen
- GUI-Gestaltung zur Eingabe der Optimierungsziele ist komplex
- Reglerstruktur muss vorgegeben werden

### 2.1.7 | Genetischer Algorithmus

Ein genetischer Algorithmus *GA* [39] ist eine Optimierungsme~~thode~~thode, basierend auf der natürlichen Evolution. Ein *GA* kann eine beliebige Anzahl an Parametern optimieren und benötigt dafür nur eine Bewertungsfunktion. Die Idee besteht darin, eine *Population* an Individuen mit zufälligen Parametern zu erstellen und diese einzeln mit Hilfe der Bewertungsfunktion zu bewerten. Die besten Individuen werden ausgewählt und miteinander gekreuzt und mutiert um neue Individuen zu erzeugen. Dieser Vorgang wird über mehrere Generationen wiederholt bis ein Abbruchkriterium erfüllt ist. Diese Methode kann verwendet werden um die Parameter eines PID-Reglers oder auch eines Reglers mit beliebiger Struktur zu optimieren [40]. *GA* ist grundsätzlich nur für *Maximierungsprobleme* ausgelegt, mit ein paar Tricks können aber auch *Minimierungsprobleme* gelöst werden.

- + Geeignet für MIMO-Systeme
- + Kann beliebig viele Parameter optimieren
- + Sehr flexibel bezüglich der Reglerstruktur
- + Kann auf nichtlineare Regelstrecken und Regler angewendet werden
- + Der Algorithmus allein kann ohne Modellwissen arbeiten
- + Gut automatisierbar
- + Ableitung der Bewertungsfunktion wird nicht benötigt
- Da die Optimierung auf reiner Simulation basiert, muss jedoch ein Modell der Regelstrecke vorliegen. Eine Optimierung am realen System wäre technisch zwar möglich, praktisch jedoch nicht umsetzbar.
- Kann nicht auf zeitvariante Regelstrecken angewendet werden
- Instabile Systeme können stabilisiert werden, jedoch ist dies nicht garantiert
- Benötigt eine gut definierte Bewertungsfunktion. Es kann sehr schwierig sein eine gute Bewertungsfunktion zu definieren:  
Gewünschtes Transientenverhalten, Überschwingen, Robustheitsanforderungen, etc. müssen während des Bewertens in eine Zahl zusammengefasst werden.
- Optimierung kann lange dauern je nach Anzahl Parameter, Komplexität der Regelstrecke und Population/Generationen Grösse
- Es kann nicht garantiert werden, einen optimalen Regler zu finden
- Reglerstruktur muss vorgegeben werden

## 2.1.8 | Neuroevolution

NEAT ist ein Ansatz, der evolutionäre Algorithmen mit NN kombiniert. Dabei werden neuronale Netzwerke als Individuen in einer Population betrachtet und durch genetische Algorithmen optimiert. Die Methode ist eine Erweiterung des klassischen GA und ermöglicht es, sowohl die Struktur als auch die Gewichtungen der Netzwerke zu optimieren. Der Anwender muss sich keine Reglerstruktur überlegen und vorgeben, der Algorithmus verändert die Struktur laufend um bessere Lösungen zu finden. Im Gegensatz zu einem vordefinierten NN, welches den Regler darstellen soll, wird durch diesen Algorithmus ein unnötiges aufblasen des Netzwerks vermieden. [41] Dies ist ein häufiges Problem bei der Verwendung von NN, einerseits ist zu Beginn nicht bekannt wie gross ein NN sein muss um die Aufgabe zu lösen, andererseits kann ein zu grosses NN dazu führen, zu komplex für die Implementierung auf dem Zielsystem zu sein. NEAT startet bei einer minimalen Struktur und wächst nur so weit wie nötig um die Aufgabe zu lösen. Eine Beispianwendung kann in der Literatur gefunden werden [43]. (Nicht peer-reviewed)

- + Geeignet für MIMO-Systeme
- + Kann beliebig viele Parameter optimieren
- + Sehr flexibel bezüglich der Reglerstruktur
- + Kann auf nichtlineare Regelstrecken angewendet werden
- + Der Algorithmus allein kann ohne Modellwissen arbeiten
- + Die Reglerstruktur muss nicht vorgegeben werden
- + Sehr gut automatisierbar
- + Ableitung der Bewertungsfunktion wird nicht benötigt
- Da die Optimierung auf reiner Simulation basiert, muss jedoch ein Modell der Regelstrecke vorliegen. Eine Optimierung am realen System wäre technisch zwar möglich, praktisch jedoch nicht umsetzbar.
- Kann nicht auf zeitvariante Regelstrecken angewendet werden
- Instabile Systeme können stabilisiert werden, jedoch ist dies nicht garantiert
- Benötigt eine gut definierte Bewertungsfunktion. Es kann sehr schwierig sein eine gute Bewertungsfunktion zu definieren:  
Gewünschtes Transientenverhalten, Überschwingen, Robustheitsanforderungen, etc. müssen während des Bewertens in eine Zahl zusammengefasst werden.
- Optimierung kann lange dauern je nach Anzahl Parameter, Komplexität der Regelstrecke und Population/Generationen Grösse
- Es kann nicht garantiert werden, einen optimalen Regler zu finden

## 2.1.9 | Differential Evolution

Der Differential Evolution Algorithmus DE [42] ist ein ähnlich wie der GA ein evolutionärer Algorithmus zur Optimierung von Parametern. Diese Methode hat wie der GA auch, den Vorteil, dass keine Ableitungen der Bewertungsfunktion benötigt wird. DE kann eine beliebige Anzahl an Parametern optimieren und benötigt dafür nur eine Bewertungsfunktion. Sowohl Maximierungs- als auch Minimierungsprobleme können mit dieser Methode gelöst werden. Ein Nachteil des DE ist jedoch, dass die Anfangswerte der Parameter aus Sicht der ganzen Population, den gesamten Suchraum aufspannen müssen. Ist dies nicht der Fall, kann das optimale Ergebnis unter Umständen nicht gefunden werden da der Suchraum nicht in alle Richtungen erkundet werden kann.

- + Geeignet für MIMO-Systeme
- + Kann beliebig viele Parameter optimieren
- + Sehr flexibel bezüglich der Reglerstruktur
- + Kann auf nichtlineare Regelstrecken und Regler angewendet werden
- + Der Algorithmus allein kann ohne Modellwissen arbeiten
- + Gut automatisierbar
- + Ableitung der Bewertungsfunktion wird nicht benötigt
- Da die Optimierung auf reiner Simulation basiert, muss jedoch ein Modell der Regelstrecke vorliegen. Eine Optimierung am realen System wäre technisch zwar möglich, praktisch jedoch nicht umsetzbar.
- Kann nicht auf zeitvariante Regelstrecken angewendet werden
- Instabile Systeme können stabilisiert werden, jedoch ist dies nicht garantiert
- Benötigt eine gut definierte Bewertungsfunktion. Es kann sehr schwierig sein eine gute Bewertungsfunktion zu definieren:  
Gewünschtes Transientenverhalten, Überschwingen, Robustheitsanforderungen, etc. müssen während des Bewertens in eine Zahl zusammengefasst werden.
- Es kann nicht garantiert werden, einen optimalen Regler zu finden
- Reglerstruktur muss vorgegeben werden

## 2.2 | Ergebnis

Wie im Kapitel [1.6 Methodik](#) bereits erwähnt, wird für die Ermittlung der zu vertiefenden Methoden folgende Punkte betrachtet.

- Mögliche Verhalten das durch den resultierenden Regler erreicht werden kann.
  - Stabilität
  - Zeitliches Verhalten
  - Stationäres Verhalten
  - Robustheit
- Voraussetzungen der Systeme, an denen die jeweilige Methode angewendet werden kann
  - stabile/instabile Systeme
  - SISO/MIMO Fähigkeit
  - lineare/nichtlineare Systeme & zeitinvariante/zeitvariante Systeme
  - Regler Strukturen, Anzahl der zu optimierenden Parameter
- Die Schwierigkeit der Parametrierung
  - Benutzerfreundlichkeit der Parametrisierung
  - Benötigtes Vorwissen des Anwenders bezüglich Regelungstechnik
- Automatisierungsgrad
  - Tools/Technologien
  - Eingriffe des Anwenders

### 2.2.1 | Bewertungstabelle

In der Tab. 2 ist die Bewertung der betrachteten Methoden in einer Tabelle dargestellt. Es ist jedoch schwierig eine Quantisierung der einzelnen Kriterien vorzunehmen, die Tabelle soll daher nur als grobe Orientierung dienen.

		Bewertungstabelle								
		Bewertungstabelle								
		Bewertungstabelle								
		Lambda-Tuning	Pol-Platzierung	LQR	Ziegler-Nichols	MATLAB systune	MATLAB looptune	Genetischen Algorithmen	Differential Evolution	Neuroevolution
		8	9	9	6	10	10	3	3	3
Reglerverhalten		Stabilität	8	9	9	6	10	10	3	3
		Zeitverhalten	8	7	9	6	10	10	3	3
		Stationär	10	10	10	10	10	10	3	3
		Robustheit	8	8	7	3	10	10	3	3
Systemvoraussetzung		Instabil möglich	✗	✓	✓	✗	✓	✓	✓	✓
		MIMO fähig	✗	✓	✓	✗	✓	✓	✓	✓
		Nichtlinear fähig	✗	✗	✗	✗	✗	✗	✓	✓
		Zeitvariant fähig	✗	✗	✗	✗	✗	✗	✓	✓
		Strukturoffenheit	1	1	1	3	10	8	10	10
Parametrierung		Benutzerfreundlichkeit	8	7	6	9	5	3	5	5
		Reglerstruktur Vorgabe erforderlich	✓	✗	✗	✗	✓	✓	✓	✗
		Flexibilität der Optimierungsvorgaben	5	1	6	3	10	10	10	10
Automatisierungsgrad		Benötigt MATLAB	✗	✗	✗	✗	✓	✓	✗	✗
		Eingriff des Anwenders	7	1	5	8	8	8	8	8

Tab. 2: Bewertungstabelle der betrachteten Methoden

## 2.3 | Kombination mehrerer Methoden

Es ist natürlich nicht vorgeschrieben nur eine Methode allein zu verwenden. Das Kombinieren von mehreren Methoden kann sehr viele Vorteile bringen.

### Beispiel 1: Ziegler-Nichols + genetischer Algorithmus für PID-Tuning

Ein klassischer Ansatz zur schnellen Parametrierung eines PID-Reglers ist die Ziegler-Nichols Methode. Diese Methode liefert eine erste Abschätzung der Reglerparameter basierend auf dem offenen Regelkreisverhalten. Allerdings sind die mit Ziegler-Nichols bestimmten Parameter oft nicht optimal für spezifische Anforderungen.

An dem Punkt kann ein [GA](#) ansetzen, um die Reglerparameter weiter auf die gewünschten Anforderungen zu optimieren. Weil Ziegler-Nichols für [MIMO](#) und instabile [Systeme](#) nicht geeignet ist, unterliegt auch die Kombination dieser beiden Methoden dieser Einschränkung.

### Beispiel 2: Ziegler-Nichols + diverse andere Optimierungsmethoden für PID-Tuning

Wie im vorherigen Beispiel beschrieben, kann Ziegler-Nichols eine gute erste Abschätzung der Reglerparameter liefern. Für die weitere Optimierung der Reglerparameter können aber auch andere Optimierungsmethoden verwendet werden:

- Differential Evolution [42]
- Gradient Descent
- Gradient Descent with Momentum

### Beispiel 3: Evolutionärer Algorithmus + MATLAB *systune* für komplexe Reglerstrukturen

Da für die Optimierung mit MATLAB *systune* [27] ein initiales Modell mit Reglerstruktur benötigt wird, kann eine Schwierigkeit darin bestehen, eine geeignete Reglerstruktur zu finden. Ein evolutionärer Algorithmus wie [NEAT](#) könnte verwendet werden, um verschiedene Reglerstrukturen zu generieren, welche anschließend mit MATLAB *systune* optimiert werden.

Die Optimierungsergebnisse von *systune* können in die Bewertungsfunktion des evolutionären Algorithmus einfließen, um die besten Reglerstrukturen zu identifizieren.

## 2.4 | Auswahl der zu vertiefenden Methoden

### 2.4.1 | MATLAB *systune*

Aus allen betrachteten Methoden sticht [2.1.5 MATLAB \*systune\*](#) heraus, es handelt sich dabei nicht um einen spezifischen Algorithmus für die [Reglersynthese](#) sondern um ein Tool aus der MATLAB Control System Toolbox [17]. Mit diesem Tool können verschiedene Reglerstrukturen und Systemmodelle optimiert werden. Die Flexibilität und die einfache Automatisierung machen dieses Tool zu einer guten Wahl für die weitere Vertiefung.

#### Voraussetzungen der Systeme

- Das System muss als *genss*-Modell [19] modelliert sein
- Für die Optimierung können beliebig viele Parameter optimiert werden
- Die zu optimierenden Parameter können überall im Regelkreis liegen
- Das System sollte in der Lage sein, die definierten Optimierungsziele zu erreichen
- Stabile als auch instabile Systeme können optimiert werden
- SISO- als auch MIMO-Systeme können optimiert werden
- Nur lineare- und Zeitinvariante Systeme können optimiert werden

#### Ermögliches Reglerverhalten

MATLAB *systune* [27] ermöglicht die Definition von verschiedenen Optimierungszielen [23], welche spezifisch auf das gewünschte Reglerverhalten abgestimmt werden können.

#### Die Schwierigkeit der Parametrisierung

Die Parametrisierung erfolgt über Optimierungsziele [23]. Einerseits können beliebig viele Optimierungsziele definiert werden, andererseits können diese auch noch in harte und weiche Ziele unterteilt werden. Dadurch kann die Optimierung sehr flexibel gestaltet werden. Jedes Optimierungsziel beinhaltet selbst noch weitere Parameter zur Konfiguration. Das macht die Parametrisierung etwas komplexer, jedoch auch sehr mächtig.

## Automatisierungsgrad

Um die Optimierung durchzuführen, muss der Anwender einige Informationen bereitstellen. Dazu gehören:

- Definition des *genss*-Modells [19], welches bereits den zu optimierenden Regler enthält
- Definition der Optimierungsziele [23] und deren Parametrisierung
- Bestimmung welche Optimierungsziele hart und welche weich sind
- (Optional) Definition von Startwerten für die zu optimierenden Parameter

Sobald diese Informationen bereitgestellt sind, kann der Optimierungsprozess gestartet werden. Das Tool übernimmt dann die komplette Optimierung und liefert ein *genss*-Modell [19] mit den optimierten Parametern zurück. Außerdem wird eine Auswertung der Optimierung für jedes Optimierungsziel bereitgestellt.

---

### 2.4.2 | Genetischer Algorithmus für PID-Tuning

#### Ermöglichtes Reglerverhalten

Benutzerdefinierte Bewertungsfunktionen ermöglichen die Optimierung des PID-Reglers auf sehr spezifische Anforderungen.

Die Theorie zum GA ist im Kapitel [A.2.5 Genetischer Algorithmus](#) beschrieben.

#### Voraussetzungen der Systeme

- Modell benötigt für den [GA](#)
- System kann [MIMO](#) sein
- Reglerstruktur ist beliebig

#### Die Schwierigkeit der Parametrisierung

Der Anwender muss eine Bewertungsfunktion definieren welche aus beliebigen Teilfunktionen bestehen kann.

#### Automatisierungsgrad

Mit bestehender Bewertungsfunktion und vordefiniertem Modell inklusive Reglerstruktur, kann die Optimierung gestartet werden. Die Optimierung ist ein iterativer Prozess welcher solange wiederholt wird, bis das Ergebnis zufriedenstellend ist.

---

### 2.4.3 | Differential Evolution für PID-Tuning

[DE](#) hat ziemlich ähnliche Eigenschaften wie der [GA](#). Der Unterschied besteht hauptsächlich darin, wie die Selektion, Kreuzung und Mutation durchgeführt werden.

Die Theorie zu DE ist im Kapitel [A.2.6 Differential Evolution Algorithmus](#) beschrieben.

Weil die beiden Algorithmen so ähnlich sind, wird hier nicht weiter auf die Eigenschaften eingegangen, da sie bereits im Abschnitt [2.4.2 Genetischer Algorithmus für PID-Tuning](#) beschrieben wurden.

## 3 | Systune am DC-Motor

Für *systune* wird MATLAB benötigt, nachfolgend wird Schritt für Schritt aufgezeigt, wie die Optimierung in MATLAB umgesetzt wird.

### 3.1 | Regelkreis Definieren

Der verwendete Regelkreis ist im Kapitel

**A.2.1 Regelkreis für Prozess: DC-Motor** genauer beschrieben. Da der Regelkreis für *systune* linear sein muss, werden die nichtlinearen Sättigungsblöcke vernachlässigt und erst beim Testen des optimierten Systems wieder berücksichtigt. Das komplette MATLAB-Skript kann im GitHub Repository [2] eingesehen werden.

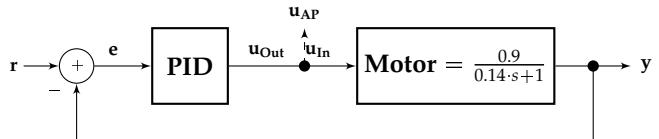


Abb. 1: Regelkreis für den DC-Motor

#### 3.1.1 | Motor definieren

```
1 T = 0.14; % Zeitkonstante
2 K1 = 0.9; % DC-Verstärkung
3 plantSys = tf(K1, [T, 1]);
4 plantSys.InputName = 'uIn';
5 plantSys.OutputName = 'y';
6 plantSys.Name = 'plantSys';
```

Code 1: Definition des DC-Motors

#### 3.1.2 | Tunable PID-Regler definieren

Mit der Funktion *tunablePID* aus der Control System Toolbox [17] kann ein PID-Regler erstellt werden, der sogenannte *tunable* Parameter besitzt. Diese Parameter können anschliessend mit *systune* optimiert werden.

```
1 controllerSysInitial = tunablePID('controllerSys', 'PID');
2 % Set initial values (optional)
3 controllerSysInitial.Kp.Value = 1;
4 controllerSysInitial.Ki.Value = 1;
5
6 % Name the controller inputs and outputs
7 controllerSysInitial.InputName = 'e';
8 controllerSysInitial.OutputName = 'uOut';
```

Code 2: Definition des PID-Reglers

#### 3.1.3 | Analysis-Point definieren

Ein Analysis-Point kann verwendet werden um während der Optimierung bestimmte Signale zu überwachen oder zu beeinflussen [27]. Dieser Analysis-Point kann später bei den **3.2 Optimierungsziele definieren** verwendet werden.

```
1 G_ap = AnalysisPoint('uAP');
2 G_ap.InputName = 'uOut';
3 G_ap.OutputName = 'uIn';
```

Code 3: Definition des Analysis-Points

#### 3.1.4 | Feedback Summation definieren

```
1 sum_block = sumblk('e = r - y');
```

Code 4: Definition des Feedback Summation Blocks

#### 3.1.5 | Alle Teilsysteme verbinden

```
1 % System inputs: 'r'
2 % System outputs: 'y', 'uIn', 'e'
3 sys0 = connect(plantSys, G_ap, controllerSysInitial, sum_block, {'r'}, {'y', 'uIn', 'e'});
4 sys0.Name = 'Untuned system';
```

Code 5: Definition des Gesamtsystems

*sys0* Abb. 1 ist nun das nicht optimierte **Gesamtsystem**, welches den DC-Motor, den PID-Regler, den Analysis-Point und das Feedback beinhaltet.

## 3.2 | Optimierungsziele definieren

Bei den Optimierungszielen können verschiedene Kriterien verwendet werden. In diesem Beispiel werden zwei Ziele definiert:

- **Step-Tracking:** Das System soll der Sprungantwort des Motors folgen.
- **Overshoot:** Das Überschwingen soll begrenzt werden.

Diese Ziele benötigen keinen Analysis-Point, deshalb wurde dieser nur zur Demonstration definiert. Es gibt aber andere Optimierungsziele, die auf einen Analysis-Point angewiesen sind.

### 3.2.1 | TuningGoal.StepTracking [26]

Das Step-Tracking Ziel sorgt dafür, dass das **Gesamtsystem** einer Sprungantwort folgt. In diesem Fall wird nur eine Zahl angegeben, welche vom Step-Tracking als PT1 System interpretiert wird:

$$H_{\text{ref}}(s) = \frac{\frac{1}{\tau}}{s + \frac{1}{\tau}} \quad \text{mit } \tau = 0.014 \quad (\text{Formel 3.2.1})$$

Das Referenzsystem  $H_{\text{ref}}(s)$  entspricht, abgesehen von der DC-Verstärkung, der Übertragungsfunktion des Motors. Damit wird ein möglichst schnelles Ansprechverhalten des geschlossenen Regelkreises angestrebt.

```
1 TR1 = TuningGoal.StepTracking('r', 'y', 0.014);
```

*Code 6: Definition des Step-Tracking Ziels*

### 3.2.2 | TuningGoal.Overshoot [24]

Das Overshoot Ziel begrenzt das Überschwingen des Systems. Ziel ist es, vom Referenzsignal  $r$  zur Ausgangsgröße  $y$  nicht mehr als 10% Überschwingen zuzulassen.

```
1 TR2 = TuningGoal.Overshoot('r', 'y', 10);
```

*Code 7: Definition des Overshoot Ziels*

### 3.2.3 | Optimierungsziele kombinieren

Die beiden definierten Ziele werden in zwei Vektoren gesammelt, es können beliebig viele Ziele definiert werden. Die Ziele werden in harte und weiche Ziele unterteilt. Harte Ziele müssen zwingend erfüllt werden, während weiche Ziele nur angestrebt werden.

```
1 targetSoftGoals = [TR1]; % Step Tracking
2 targetHardGoals = [TR2]; % Overshoot
```

*Code 8: Kombinieren der Optimierungsziele*

## 3.3 | Optimierung durchführen

Mit dem Befehl **systune** wird die Optimierung durchgeführt [27]. Als Eingabe werden das nicht optimierte System **sys0** sowie die definierten Optimierungsziele übergeben. Als Ausgabe wird das optimierte System **sysTuned** sowie die Optimierungsergebnisse der weichen *fSoft* und harten *fHard* Ziele zurückgegeben.

```
1 [sysTuned, fSoft, fHard] = systune(sys0, targetSoftGoals, targetHardGoals);
2 sysTuned.Name = 'Tuned system';
```

*Code 9: Optimierung durchführen*

### 3.3.1 | Regler extrahieren

Nach der Optimierung kann der optimierte Regler aus dem Gesamtsystem extrahiert werden.

```
1 controllerSysTuned = getBlockValue(sysTuned, 'controllerSys');
2 disp(controllerSysTuned);
```

*Code 10: Regler extrahieren*

```
1 pid with properties:
2
3     Kp: 11.1112
4     Ki: 79.3650
5     Kd: -1.8759e-04
6     Tf: 1.9757
7     ...
...
```

*Code 11: Optimierter Regler*

## 3.4 | Optimierung auswerten

Systune liefert für jedes Optimierungsziel einen Wert zurück, der angibt, wie gut das jeweilige Ziel erfüllt wurde.

```

1 % Display final objective value
2 fprintf('Final soft objective: %.4f (< 1 is good)\n', fSoft);
3 fprintf('Final hard objective: %.4f (< 1 is good)\n', fHard);

```

Code 12: TuningGoals auswerten

Das erste Soft-Ziel (Step-Tracking) wurde sehr gut erfüllt, da der Wert nahe bei 0 liegt. Das zweite Hard-Ziel (Overshoot) wurde ebenfalls erfüllt, da der Wert unter 1 liegt.

```

1 Final soft objective: 0.0000 (< 1 is good)
2 Final hard objective: 0.6893 (< 1 is good)

```

Code 13: TuningGoals Auswertung

Es können auch noch mehr Analysen durchgeführt werden, um die Optimierung zu bewerten.

```

1 % Create closed-loop systems
2 CL_initial = feedback(controllerSysInitial*plantSys, 1);
3 CL_tuned = feedback(controllerSysTuned*plantSys, 1);

4

5 % Get step response info
6 info_initial = stepinfo(CL_initial);
7 info_tuned = stepinfo(CL_tuned);

8

9 fprintf('\n==== Performance Comparison ===\n');
10 fprintf('          Initial      Tuned\n');
11 fprintf('Rise Time:    %.3f s    %.3f s\n', info_initial.RiseTime, info_tuned.RiseTime);
12 fprintf('Settling Time: %.3f s    %.3f s\n', info_initial.SettlingTime, info_tuned.SettlingTime);
13 fprintf('Overshoot:    %.2f %    %.2f %\n', info_initial.Overshoot, info_tuned.Overshoot);

14

15 % Check margins
16 [Gm_i, Pm_i] = margin(plantSys*controllerSysInitial);
17 [Gm_t, Pm_t] = margin(plantSys*controllerSysTuned);

18

19 fprintf('\nGain Margin:    %.2f dB    %.2f dB\n', 20*log10(Gm_i), 20*log10(Gm_t));
20 fprintf('Phase Margin:   %.2f °    %.2f °\n', Pm_i, Pm_t);

```

Code 14: Optimierung analysieren

```

1 %==== Performance Comparison ===
2 %          Initial      Tuned
3 %Rise Time:    3.370 s    0.031 s
4 %Settling Time: 6.662 s    0.055 s
5 %Overshoot:    0.00 %    0.00 %
6 %
7 %Gain Margin:    Inf dB    Inf dB
8 %Phase Margin:   136.74 °   90.00 °

```

Code 15: Optimierungsanalyse Ausgabe

### 3.4.1 | Step-Tracking Ergebnis visualisiert

```
1 viewGoal(targetSoftGoals(1), sysTuned);
```

Code 16: Step-Tracking Diagramm

Das Diagramm in Abb. 2 zeigt die Sprungantwort des optimierten Systems. Die Kurve mit dem Namen *Desired* entspricht der gewünschten Sprungantwort, die durch das Step-Tracking Ziel definiert wurde:  $H_{ref}(s) = \frac{1}{\tau \cdot s + 1}$  mit  $\tau = 0.014$ . In blau ist die Sprungantwort des optimierten Systems dargestellt. Diese deckt sich sehr gut mit der gewünschten Sprungantwort, was zeigt, dass das Step-Tracking Ziel sehr gut erfüllt wurde.

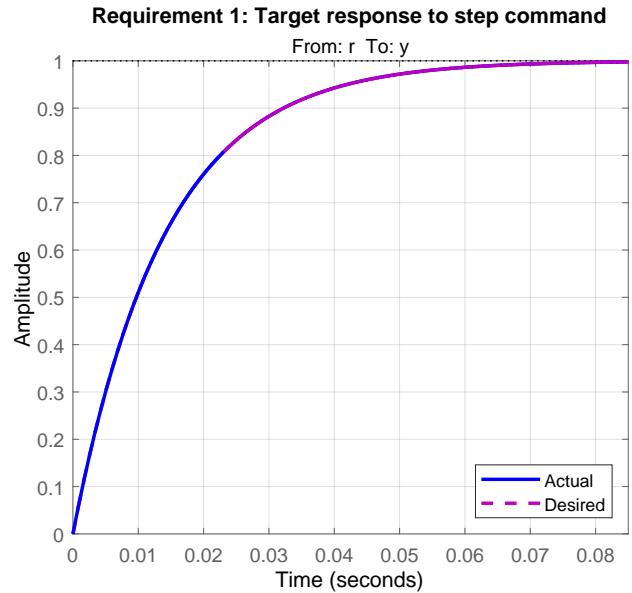


Abb. 2: Step-Tracking Optimierungsergebnis mit systune am DC-Motor

### 3.4.2 | Overshoot Ergebnis visualisiert

```
1 viewGoal(targetHardGoals(1), sysTuned);
```

Code 17: Overshoot Diagramm

Das Diagramm in Abb. 3 zeigt den Frequenzgang des optimierten geschlossenen Systems. Der orangefarbene Bereich stellt den Teil der Verstärkung dar, welcher grösser als die erlaubten 10% Überschwingen ist. Die blaue Kurve zeigt die Verstärkung des optimierten Systems, diese ist im gesamten Frequenzbereich unterhalb der 10% Grenze, was zeigt, dass das Overshoot Ziel erfolgreich erfüllt wurde.

Mehr Informationen darüber, wie die Optimierungsergebnisse visualisiert werden, ist in der MATLAB Dokumentation zu den Tuning-Goals beschrieben. [31]

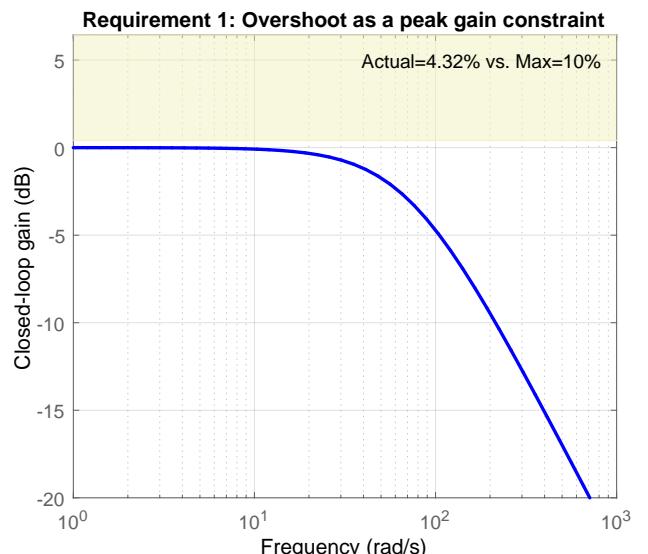


Abb. 3: Overshoot Optimierungsergebnis mit systune am DC-Motor

### 3.5 | Optimierten Regler am realen System testen

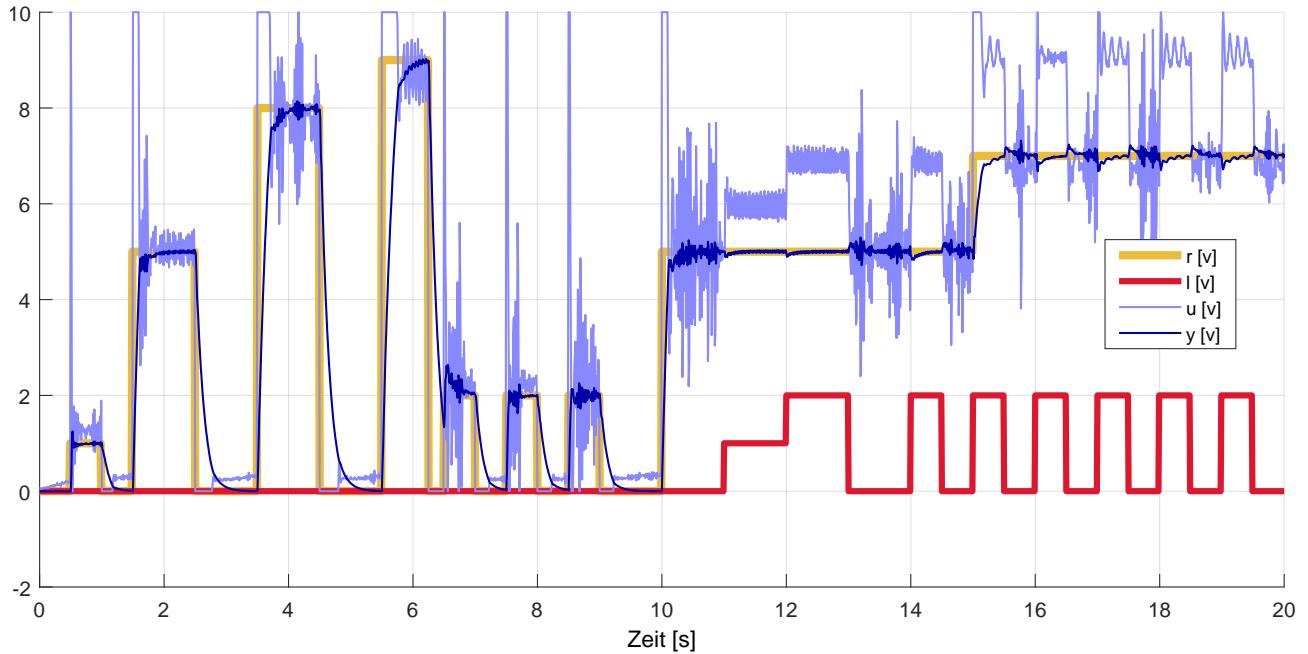


Abb. 4: Antwort auf Stimulation des optimierten Reglers am DC-Motor, mit dem durch systune optimierten PID-Regler

Für den Test am Realen Prozess sind im Simulink Modell unterschiedliche Sprünge vorbereitet. Das Signal  $r$  definiert eine Spannung die proportional für eine gewünschte Drehzahl steht. Die Last  $l$  steuert zwei Wirbelstrombremsen an, wobei ein Wert von 1 bedeutet, dass nur eine Bremse aktiv ist und ein Wert von 2 bedeutet, dass beide Bremsen aktiv sind. Auch die Sättigungen sind wieder eingebaut.

Da der PID-Regler am Ausgang einen Sättigungsblock besitzt, entsteht das Problem des Integrator Windups. Die *Clamping* Anti-Windup Methode wird verwendet, da diese keinen weiteren Tuning-Parameter erzeugt. *Systune* kann leider keinen Anti-Windup Tuning-Parameter optimieren.

Mit der Clamping Methode des Simulink PID-Blocks entsteht ein ungewöhnliches Verhalten am Ausgang des Reglers, wenn  $r$  auf 0 wechselt. Dies ist im Abb. 4 z.B. bei der Zeit um 3s zu erkennen. Mehr zu diesem Verhalten und wie man dies verhindern kann, ist in folgendem Kapitel beschrieben:

#### A.2.4 Simulinks PID Clamping Anti-Windup Problematik

Was deutlich auffällig ist, dass der Regler-Ausgang  $u$  sehr starkes Rauschen aufweist. Dieses Rauschen wird an den Motor weitergegeben und ist deshalb auch in der Drehzahl  $y$  zu erkennen.

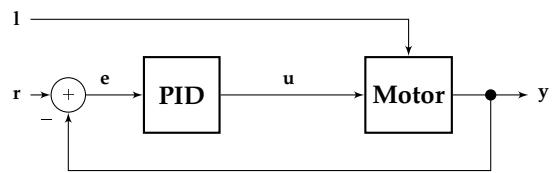


Abb. 5: Regelkreis des DC-Motors

## 4 | Genetischer Algorithmus am DC-Motor

Weil der genetische Algorithmus kein MATLAB voraussetzt, ist diese Methode auch ohne MATLAB implementiert worden. Das hat den Vorteil, dass ein System auch ohne MATLAB Lizenz optimiert werden kann.

Die Funktionsweise des **GA** ist im Kapitel [A.2.5 Genetischer Algorithmus](#) beschrieben.

### 4.1 | System Definieren

In der C++ Anwendung werden die Teilsysteme objektorientiert implementiert und anschliessend als ein gesamtheitliches Objekt zusammengeführt und verbunden. Der Kernteil der Implementierungslogik spielt sich dabei in den Update-Funktionen der jeweiligen Teilsysteme ab. Aufgrund des Codes-Umfangs werden deshalb in den nachfolgenden Unterkapiteln nur die Update-Funktionen gezeigt.

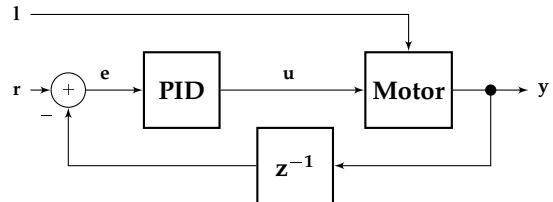


Abb. 6: Regelkreis des DC-Motors

#### 4.1.1 | Motor definieren

Die Implementierung im C++ Code erfolgt natürlich diskretisiert. Wie genau die Diskretisierte Implementierung aussieht wird nicht weiter erläutert und kann im Code eingesehen werden. Als Grundlage für die Implementierung dient die [Formel 4.1.1](#).

Die dazugehörige C++ Implementierung ist im folgenden Codeausschnitt ersichtlich:

[30 Update Funktion des DC-Motors im Code](#)

$$y(t) = \frac{1}{T} \cdot \int (K_1 \cdot u(t) - y(t) \cdot (1 + K_2 \cdot K_3 \cdot l(t))) dt \quad (\text{Formel 4.1.1})$$

Herleitung siehe [Anhang A.1.1](#)

#### 4.1.2 | PID-Regler definieren

Der PID-Regler wurde bereits im Kapitel [A.2.3 Reglerstruktur: Erweiterter PID-Regler](#) beschrieben und auch die C++ Implementierung wird dort aufgeführt. Deshalb wird hier nicht weiter darauf eingegangen.

#### 4.1.3 | Alle Teilsysteme verbinden

```

1 // Update Funktion des Gesamtsystems im C++ Code. class TestSystem
2 void update(double deltaTime) override
3 {
4     double y = m_dcMotorSystem.getAngularVelocity();      // y(t) * z^{-1}, Letzter Ausgang des Motors
5     m_errorValue = m_referenceValue - y;                  // e(t) = r(t) - y(t)
6
7     m_pidController.setInput(m_errorValue);
8     m_pidController.update(deltaTime);
9     m_pidOutputValue = m_pidController.getOutput();        // u(t)
10
11    m_dcMotorSystem.setInputs(m_pidOutputValue, m_disturbanceValue);
12    m_dcMotorSystem.update(deltaTime);
13 }
```

Code 18: Update-Funktion des Gesamtsystems im Code

## 4.2 | Optimierungsziele definieren

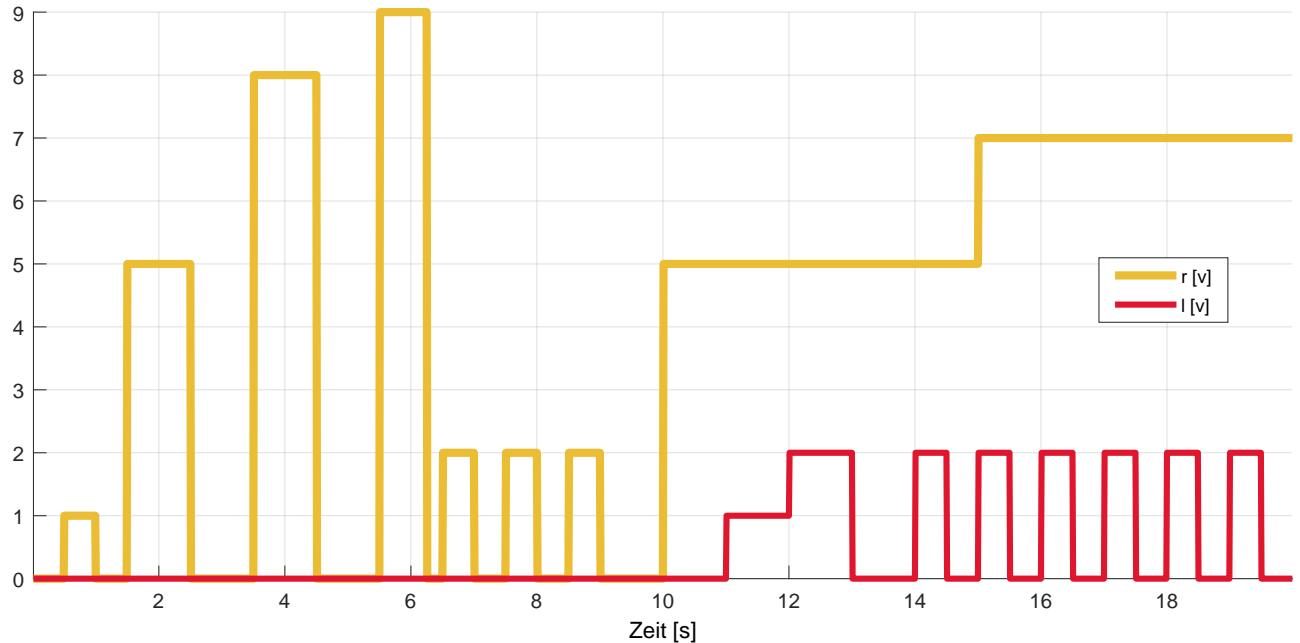


Abb. 7: Anregungssignale für die Systemsimulation

Um die Bewertungsfunktion zu definieren wird eine zeitliche Simulation des Systems benötigt. Dazu wird ein vordefiniertes Anregungssignal verwendet. Das Anregungssignal besteht aus  $r$  (Ziel-Drehzahl) und  $l$  (Last/Störgröße) und ist so gewählt, dass verschiedene Aspekte des Regelverhaltens bewertet werden können.

Einerseits ermöglichen die unterschiedlichen Sprünge im Ziel-Drehzahl  $r$  eine Beurteilung über das Verhalten des Reglers auf den Prozess, der eine nichtlineare DC-Verstärkung hat. Andererseits wird durch die Nutzung der Wirbelstrombremsen  $l$ , die Einwirkung von Störgrößen mitberücksichtigt. Mit diesem Signal ist es dem **GA** möglich, den Regler auf verschiedene Situationen zu optimieren.

Symbol	Beschreibung	Wert	Einheit
$T_s$	Abtastzeit	0.01	s
$t_{\text{start}}$	Startzeit	0	s
$t_{\text{end}}$	Endzeit	20	s
$u_{\text{satHigh}}$	Obere Sättigungsgrenze der Stellgrösse	10	V
$u_{\text{satLow}}$	Untere Sättigungsgrenze der Stellgrösse	0	V
$K$	Anzahl Zeitschritte in der Simulation	2000	
$D$	Anzahl der Optimierungs-Parameter	4	
$\vec{T}_i$	Individuum $i$ in der Population	$\vec{T}_i \in \mathbb{R}^D$	
$g(\vec{T}_i)$	Fehler-Minimierungs-Funktion.	$g(\vec{T}_i) \in \mathbb{R}$	
$a_o$	Gewichtungsfaktor für das Optimierungsziel $o$	$a_o \in \mathbb{R}^{+0}$	
$O$	Anzahl Optimierungsziele	3	
$g_o(\vec{T}_i, k)$	Optimierungsfunktion für das Ziel $o$ zum Zeitpunkt $k$	$g_o(\vec{T}_i, k) \in \mathbb{R}$	

Tab. 3: Symbole für die Simulation am DC-Motor

Für die Abtastzeit  $T_s$  wurde ein realistischer Wert von 10ms gewählt.

Die Anzahl Zeitschritte  $K$  wird aus der Abtastzeit  $T_s$  und dem Simulationszeitraum  $[t_{\text{start}}, t_{\text{end}}]$  berechnet.

$$K = \frac{t_{\text{end}} - t_{\text{start}}}{T_s} \quad (\text{Formel 4.2.1})$$

Der **GA** ist für **Maximierungsprobleme** ausgelegt. Für diese Anwendung wäre es aber sinnvoller die Optimierungsziele als **Minimierungsprobleme** zu formulieren. Es gibt jedoch eine Möglichkeit zur Umrechnung von **Minimierungsproblemen** in Maximierungsprobleme. Dies ist im Kapitel **A.2.5.5 Vorgehen bei der Umwandlung  $g(\vec{T}_i) \rightarrow f(\vec{T}_i)$**  beschrieben. Dank dieser Umrechnung können die Optimierungsziele als Minimierungsproblem formuliert werden.

Die Fehler-Minimierungs-Funktion  $g(\vec{T}_i)$  wird aus mehreren Teilsummen zusammengesetzt. Die Teilsummen bewerten verschiedene Aspekte des Regelverhaltens. Allgemein lässt sich die Fehler-Minimierungs-Funktion  $g(\vec{T}_i)$  wie in der [Formel 4.2.2](#) darstellen.

$$g(\vec{T}_i) = \frac{1}{K} \cdot \sum_{k=0}^{K-1} \left( \sum_{o=0}^{O-1} (a_o \cdot g_o(\vec{T}_i, k)) \right) \quad (\text{Formel 4.2.2})$$

- $\frac{1}{K}$  dient zur Normierung der Fehlerbewertung auf die Anzahl der Zeitschritte.
- $\sum_{k=0}^{K-1}$  summiert die Fehlerbewertung über alle Zeitschritte der Simulation.
- $\sum_{o=0}^{O-1} (a_o \cdot g_o(\vec{T}_i, k))$  summiert die gewichteten Optimierungsziele  $g_o(\vec{T}_i, k)$  für den Zeitschritt  $k$ .

#### 4.2.1 | $g_0(\vec{I}_i, k)$ : Absoluter Regelungsfehler mit Sättigungsbedingung

$g_0(\vec{I}_i, k)$  zielt darauf ab, den Absoluten Regelungsfehler zu minimieren. An dieser Stelle könnte berechtigterweise die Frage auftreten, warum nicht das Quadratische Fehlerintegral verwendet wird? Tests haben gezeigt, dass es grundsätzlich egal ist ob das absolute oder das quadratische Fehlerintegral als Optimierungsziel verwendet wird, der Unterschied liegt in der Anwenderfreundlichkeit. Für die Optimierung mit den verschiedenen Zielen müssen passende Gewichtungen gewählt werden damit das gewünschte Verhalten erreicht wird. Bei der Verwendung des quadratischen Fehlerintegrals ist es schwieriger passende Gewichtungen zu finden als beim absoluten Fehlerintegral. Mit dem absoluten Fehlerintegral konvergiert der genetische Algorithmus in einem grösseren Bereich des Gewichtes für das Optimierungsziel des Fehlerintegrals.

$$\begin{aligned} ignoreError = & \left( u(\vec{I}_i, k) \geq u_{\text{sat+}} \wedge r(k) > y(\vec{I}_i, k) \right) \vee \\ & \left( u(\vec{I}_i, k) \leq u_{\text{sat-}} \wedge r(k) < y(\vec{I}_i, k) \right) \end{aligned} \quad (\text{Formel 4.2.3})$$

Die Bedingung [Formel 4.2.3](#) dient als Fairness-Ausgleich bei der Bestrafung des Fehlers. Im Fall, wo der Regler in die Sättigung läuft, weil eine grosse Änderung der Referenz vorliegt, wird der Fehler nicht weiter aufaddiert, da der Regler in diesen Fällen nicht dazu beitragen kann, dass die Fläche des Fehlers kleiner wird. Der Motor gibt bereits alles was er kann. Durch diese Bedingung konzentriert sich der die Fehlerbewertung auf die Zeiträume in denen der Regler den Fehler tatsächlich beeinflussen kann.

$$g_0(\vec{I}_i, k) = \begin{cases} 0 & \text{wenn } ignoreError = 1 \\ |e(\vec{I}_i, k)| & \text{sonst} \end{cases} \quad (\text{Formel 4.2.4})$$

#### 4.2.2 | $g_1(\vec{I}_i, k)$ : Stellwertänderungsrate

$g_1(\vec{I}_i, k)$  bewertet grosse Stellwertänderungen als schlecht. Ohne dieses Optimierungsziel tendiert der [GA](#) dazu, den Regler sehr aggressiv zu machen, was sich negativ auf das Stellverhalten auswirkt. Zur Schonung des Motors wird deshalb mit diesem Optimierungsziel versucht, grosse Stellwertänderungen zu vermeiden. Dank diesem Ziel wird auch automatisch die Rauschunterdrückung des Reglers gefördert.

$$g_1(\vec{I}_i, k) = \frac{|u(\vec{I}_i, k) - u(\vec{I}_i, k-1)|}{T_s} \quad (\text{Formel 4.2.5})$$

#### 4.2.3 | $g_2(\vec{I}_i, k)$ : Positives Überschwingen unterdrücken

$g_2(\vec{I}_i, k)$  fokussiert sich auf das Überschwingen des Systems in die positive Richtung. Es wird dabei nur die erste Überschwingung bewertet nach einer Referenzänderung, weil es schwierig zu definieren ist, ab wann Signal als nicht mehr schwingend zu betrachten ist. Das Kriterium für das Überschwingen ist in [Abb. 8](#) als Fläche zwischen der Referenz  $r$  und der Systemantwort  $y$  dargestellt. Diese Fläche wird noch normiert mit  $u_{\text{sat+}}$ .

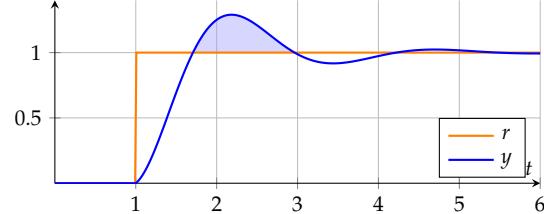


Abb. 8: Kriterium für das Überschwingen in positive Richtung

#### 4.2.4 | $g(\vec{I}_i)$ : Gesamte Fehlerbewertung

Die Teilbewertungen werden jeweils mit einem Gewichtungsfaktor  $a_0$  skaliert. Diese müssen vom Anwender passend gewählt werden, um das gewünschte Verhalten zu erzielen.

$$g(\vec{I}_i) = \frac{1}{K} \cdot \sum_{k=0}^{K-1} \left( a_0 \cdot g_0(\vec{I}_i, k) + a_1 \cdot g_1(\vec{I}_i, k) + a_2 \cdot g_2(\vec{I}_i, k) \right) \quad (\text{Formel 4.2.6})$$

## 4.3 | Optimierung durchführen

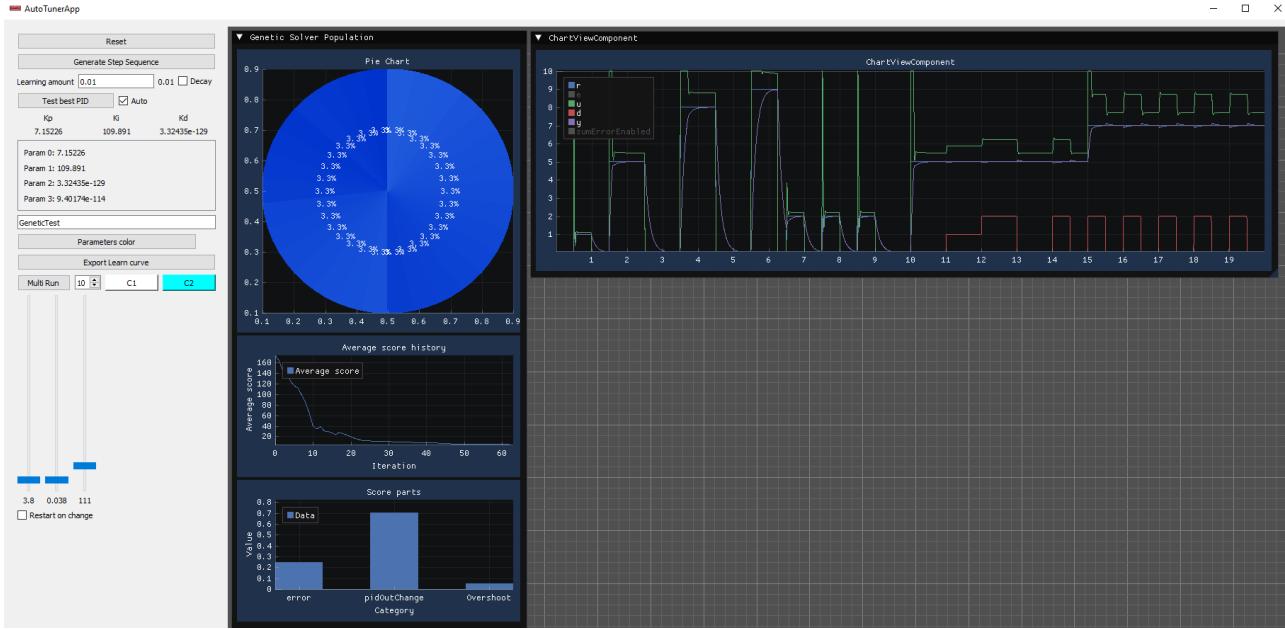


Abb. 9: Software der Systemsimulation für die Optimierung

Die Optimierung wird 10-mal mit dem **GA** durchgeführt, weil der GA aufgrund seiner stochastischen Natur nicht immer zum gleichen Ergebnis führt. Bei der Optimierung wurden die Einstellungen aus der [Tab. 4](#) verwendet.

Für den **Hyperparameter** der Mutationsstärke  $M(k)$  wird ein abklingender Wert verwendet, um zu Beginn der Optimierung eine größere Diversität in der Population zu ermöglichen. Am Ende der Optimierung soll die Population jedoch weniger divers sein, um eine bessere **Konvergenz** zu ermöglichen.

Die Wahl der PID Ausgangs-Sättigung  $u_{\text{satHigh}}$  und  $u_{\text{satLow}}$  ist aufgrund der realen Spannungsbegrenzung des DC-Motors notwendig. Die PID-Integrator-Sättigung  $I_{\text{SatHigh}}$  und  $I_{\text{SatLow}}$  hingegen sind, wie viele andere dieser Hyperparameter, willkürlich gewählt und können auch andere Werte annehmen. Mehr zur Bestimmung der Hyperparameter ist im Kapitel [11.1.5 Iteratives Tuning der Methoden](#) beschrieben.

Parameter	Beschreibung	Wert
$D$	Anzahl der einzelnen Optimierungsdurchläufe	10
$N$	Anzahl Individuen in der Population	30
$G$	Anzahl Generationen	5000
$C$	Mutations-Wahrscheinlichkeit	0.05
$M(k)$	Mutationsstärke	$1 \cdot 0.999^k$
Anti-Windup	Verwendete Anti-Windup Methode	Clamping
<b>Optimierungsziele</b>		
$a_0$	Absoluter Regelungsfehler	3.8
$a_1$	Stellwertänderungsrate	0.038
$a_2$	Positives Überschwingen	111
<b>Konstante Systemparameter</b>		
$u_{\text{satHigh}}$	PID Ausgangs-Sättigung oberer Wert	10
$u_{\text{satLow}}$	PID Ausgangs-Sättigung unterer Wert	0
$I_{\text{SatHigh}}$	PID-Integrator-Sättigung oberer Wert	10
$I_{\text{SatLow}}$	PID-Integrator-Sättigung unterer Wert	-10
<b>PID-Startparameter</b>		
$K_p$	Proportionalitätsfaktor	$0 + \text{randG}(-10, 10) \cdot *$
$K_i$	Integrationsfaktor	$0 + \text{randG}(-10, 10) \cdot *$
$K_d$	Ableitungsfaktor	$0 + \text{randG}(-10, 10) \cdot *$
$K_n$	Filterkonstante	$0 + \text{randG}(-10, 10) \cdot *$

Tab. 4: Hyper- & Tuningparameter für Optimierung mit dem genetischen Algorithmus

\* $\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

## 4.4 | Optimierung auswerten

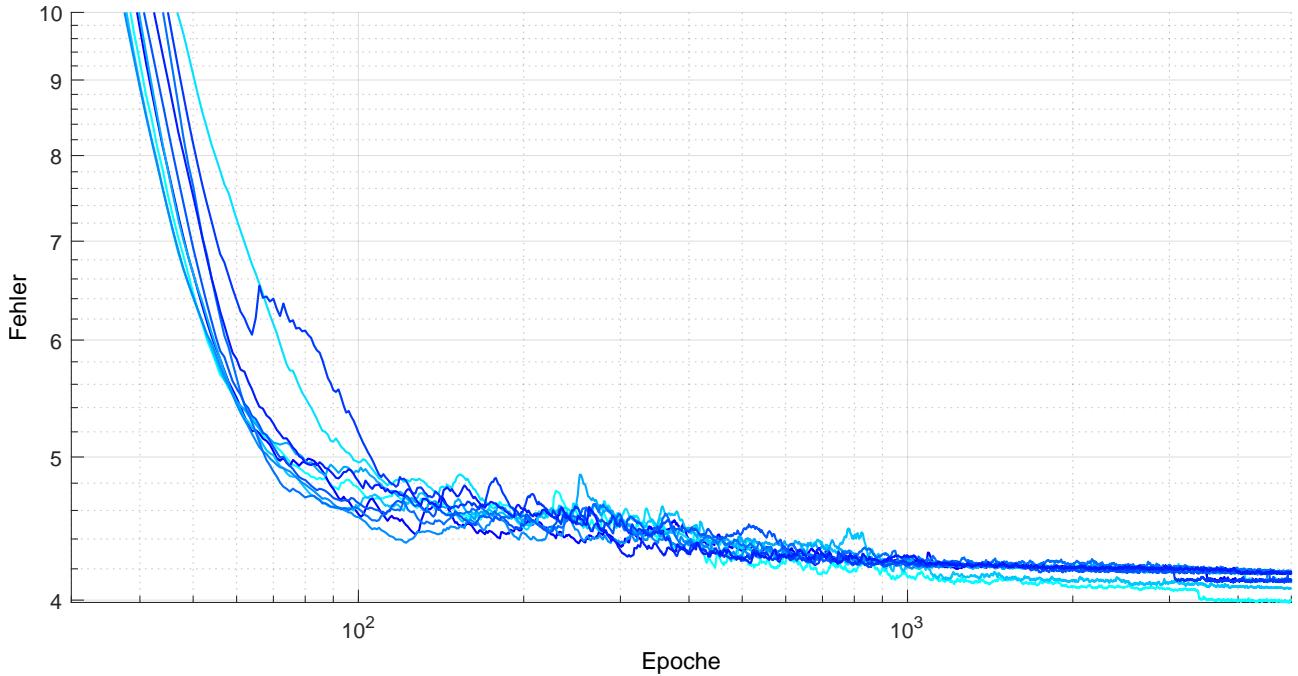


Abb. 10: Lernverlauf der Optimierung mit dem genetischen Algorithmus bei 10 Durchläufen

Die Abb. 10 zeigt den Lernverlauf der 10 Optimierungs durchläufe mit dem GA. Ohne den abklingenden Mutations-Parameter  $M(k)$  würde die Lernkurve eher linear verlaufen.

Die hohe Mutationsstärke zu Beginn der Optimierung sorgt dafür, dass sich die Population schneller in Richtung eines guten Lösungsbereichs bewegt. Am Ende der Optimierung ist die Mutationsstärke klein, was eine feinere Suche im nach einer guten Lösung ermöglicht.

## 4.5 | Optimierte Regler am realen System testen

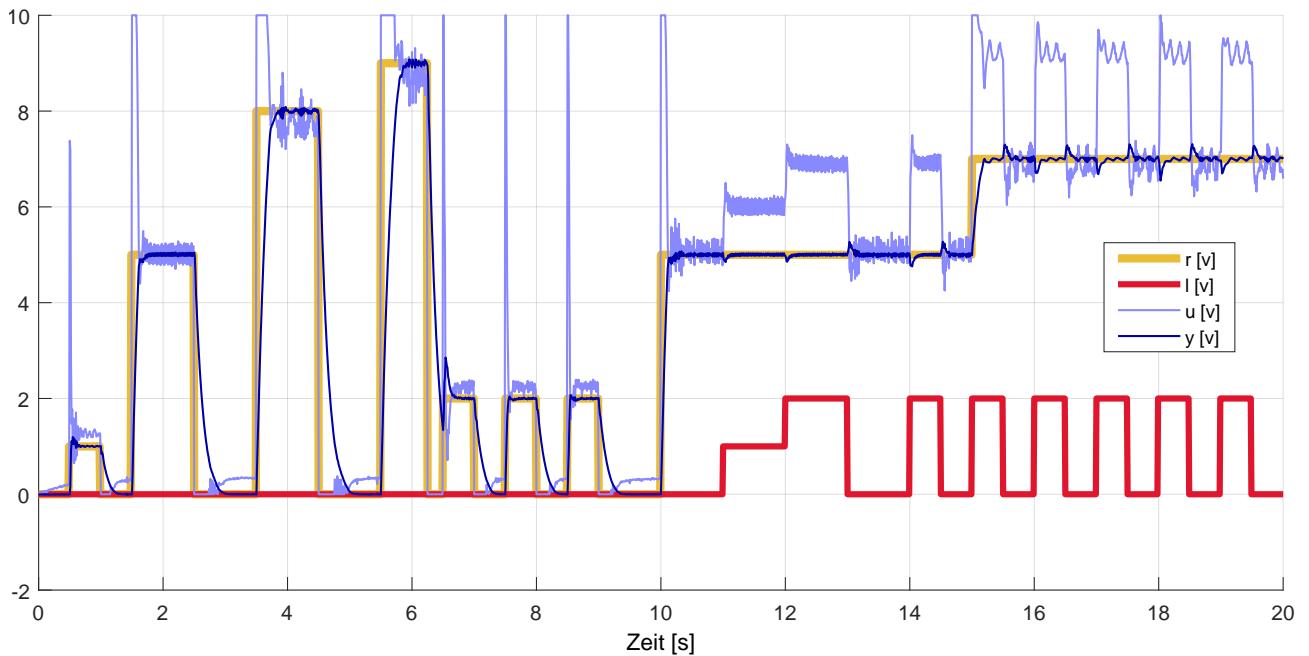


Abb. 11: Antwort auf Stimulation des optimierten Reglers am DC-Motor, mit dem durch GA optimierten PID-Regler

# 5 | Differential Evolution Algorithmus am DC-Motor

Da der DE Algorithmus dem GA sehr ähnlich ist, lässt sich der [Solver](#) im C++ Projekt sehr einfach austauschen. Deshalb kann der größte Teil der Implementierung vom Test des GA wieder verwendet werden und wird hier nicht nochmals im Detail beschrieben.

Im Gegensatz zum GA wird beim DE kein abklingender [Hyperparameter](#) für die Mutationsstärke  $M(k)$  verwendet, da der Algorithmus von sich aus kleinere Mutationen durchführt, während die Population konvergiert. Die Funktionsweise des DE Algorithmus ist im Kapitel [A.2.6 Differential Evolution Algorithmus](#) beschrieben.

## 5.1 | Optimierung durchführen

Die Software führt die Optimierung 10-mal durch, weil der DE aufgrund der zufälligen Startwerte nicht immer zum gleichen Ergebnis führt. Bei der Optimierung wurden die Einstellungen aus der [Tab. 5](#) verwendet.

Wie bereits im Beispiel mit dem GA [Abb. 9](#) werden diverse Hyperparameter willkürlich gewählt. Mehr zur Bestimmung der Hyperparameter ist im Kapitel [11.1.5 Iteratives Tuning der Methoden](#) beschrieben.

Parameter	Beschreibung	Wert
$D$	Anzahl der einzelnen Optimierungsdurchläufe	10
$N$	Anzahl Individuen in der Population	30
$G$	Anzahl Generationen	5000
$C$	Crossover-Wahrscheinlichkeit	0.7
$M$	Mutationsstärke	0.5
Anti-Windup	Verwendete Anti-Windup Methode	Clamping
<b>Optimierungsziele</b>		
$a_1$	Absoluter Regelungsfehler	3.8
$a_2$	Stellwertänderungsrate	0.038
$a_3$	Positives Überschwingen	111
<b>Konstante Systemparameter</b>		
$u_{\text{satHigh}}$	PID Ausgangs Sättigung oberer Wert	10
$u_{\text{satLow}}$	PID Ausgangs Sättigung unterer Wert	0
$I_{\text{satHigh}}$	PID Integrator Sättigung oberer Wert	10
$I_{\text{satLow}}$	PID Integrator Sättigung unterer Wert	-10
<b>PID-Startparameter</b>		
$K_p$	Proportionalitätsfaktor	$0 + \text{randG}(-10, 10) \ast$
$K_i$	Integrationsfaktor	$0 + \text{randG}(-10, 10) \ast$
$K_d$	Ableitungsfaktor	$0 + \text{randG}(-10, 10) \ast$
$K_n$	Filterkonstante	$0 + \text{randG}(-10, 10) \ast$

*Tab. 5: Hyper- & Tuningparameter für Optimierung mit dem Differential Evolution Algorithmus*

\* $\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

## 5.2 | Optimierung auswerten

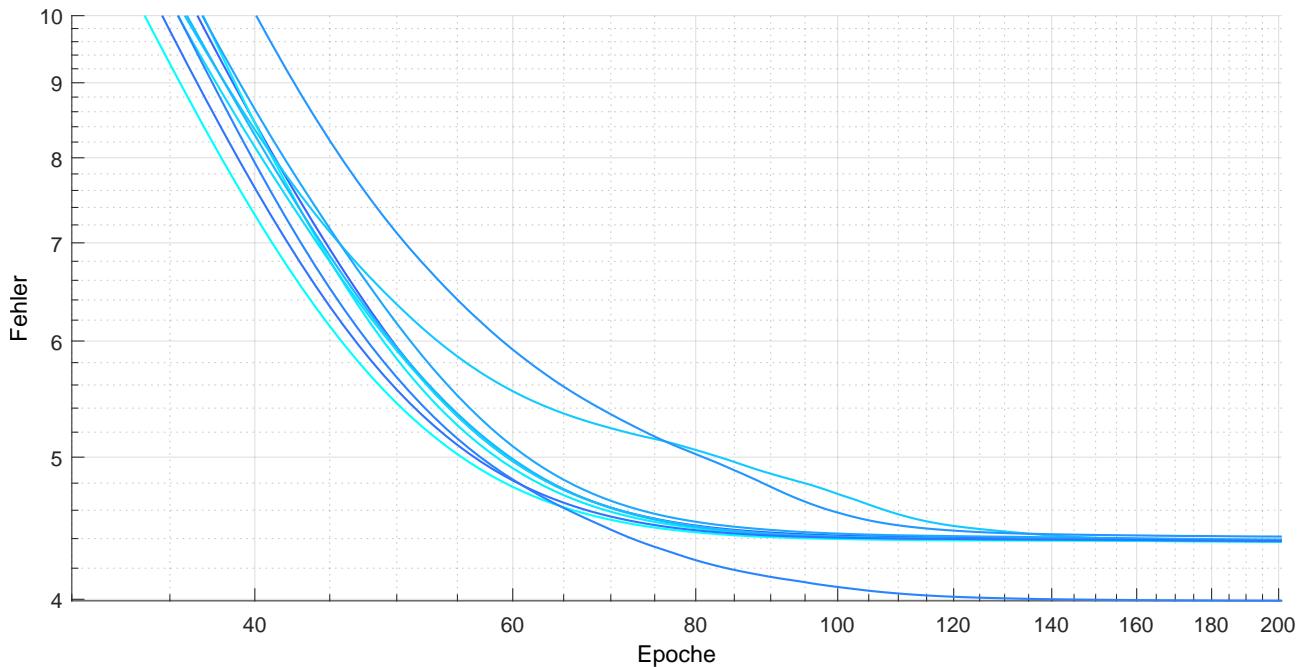


Abb. 12: Lernverlauf der Optimierung mit dem Differential Evolution Algorithmus bei 10 Durchläufen

Sehr auffällig ist, dass der DE Algorithmus im Vergleich zum GA schöner konvergiert und kein stochastisches Verhalten zeigt. Weiter fällt auf, dass von den 10 durchgeführten Optimierungen nur 1 eine noch bessere Lösung gefunden hat. Man kann gut sehen, dass wenn sich die Population auf eine gute Lösung einigt, sie nicht mehr aus dem lokalen Minimum herausfindet und somit nicht immer die beste Lösung findet.

## 5.3 | Optimierten Regler am realen System testen

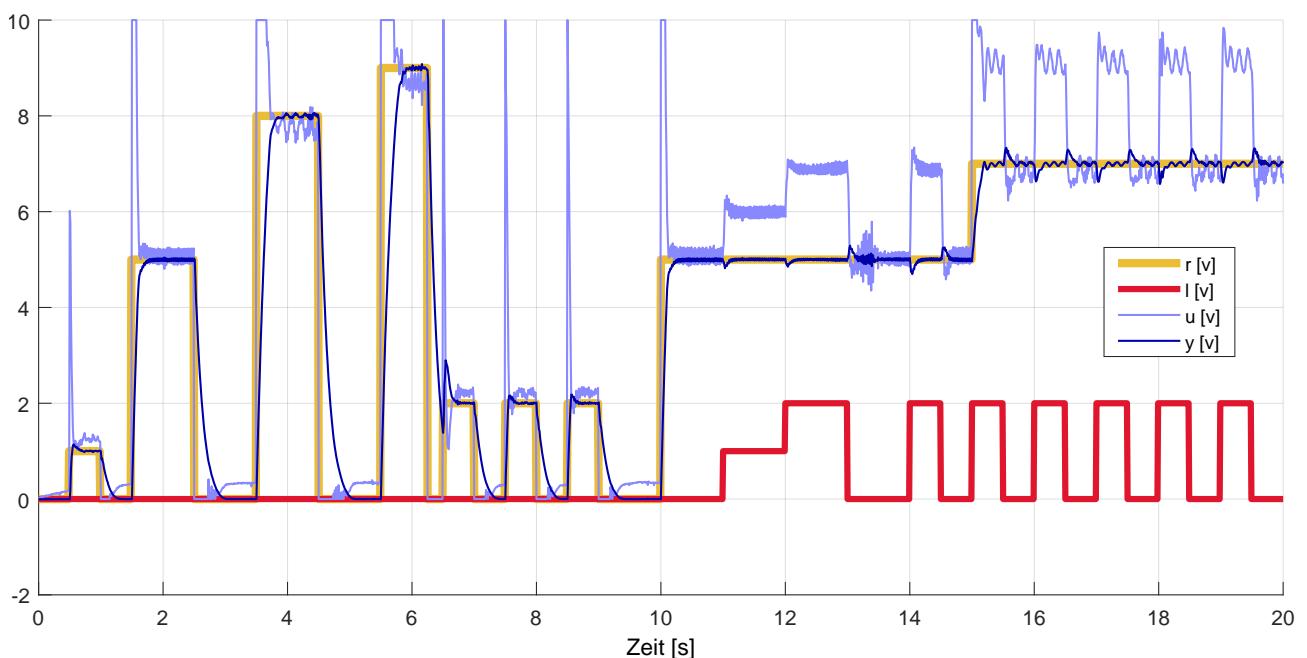


Abb. 13: Antwort auf Stimulation des optimierten Reglers am DC-Motor, mit dem durch DE optimierten PID-Regler

# 6 | Auswertung der Regleroptimierung am DC-Motor

## 6.1 | Synthetisierte Regler

Parameter	Wert
$K_p$	11.1112
$K_i$	79.365
$K_d$	-1.8759e-4
$K_n$	0.50616
<b>Konstante</b>	<b>Wert</b>
$u_{satHigh}$	10
$u_{satLow}$	0
$I_{SatHigh}$	10
$I_{SatLow}$	-10
Anti-Windup	Clamping

Tab. 6: Reglerparameter des mit *systune* optimierten Reglers

Parameter	Wert
$K_p$	6.62
$K_i$	109
$K_d$	0
$K_n$	0
<b>Konstante</b>	<b>Wert</b>
$u_{satHigh}$	10
$u_{satLow}$	0
$I_{SatHigh}$	10
$I_{SatLow}$	-10
Anti-Windup	Clamping

Tab. 7: Reglerparameter des mit *GA* optimierten Reglers

Parameter	Wert
$K_p$	5.7
$K_i$	84.5
$K_d$	-0.488
$K_n$	0.625
<b>Konstante</b>	<b>Wert</b>
$u_{satHigh}$	10
$u_{satLow}$	0
$I_{SatHigh}$	10
$I_{SatLow}$	-10
Anti-Windup	Clamping

Tab. 8: Reglerparameter des mit *DE* optimierten Reglers

## 6.2 | Systemverhalten am realen Prozess

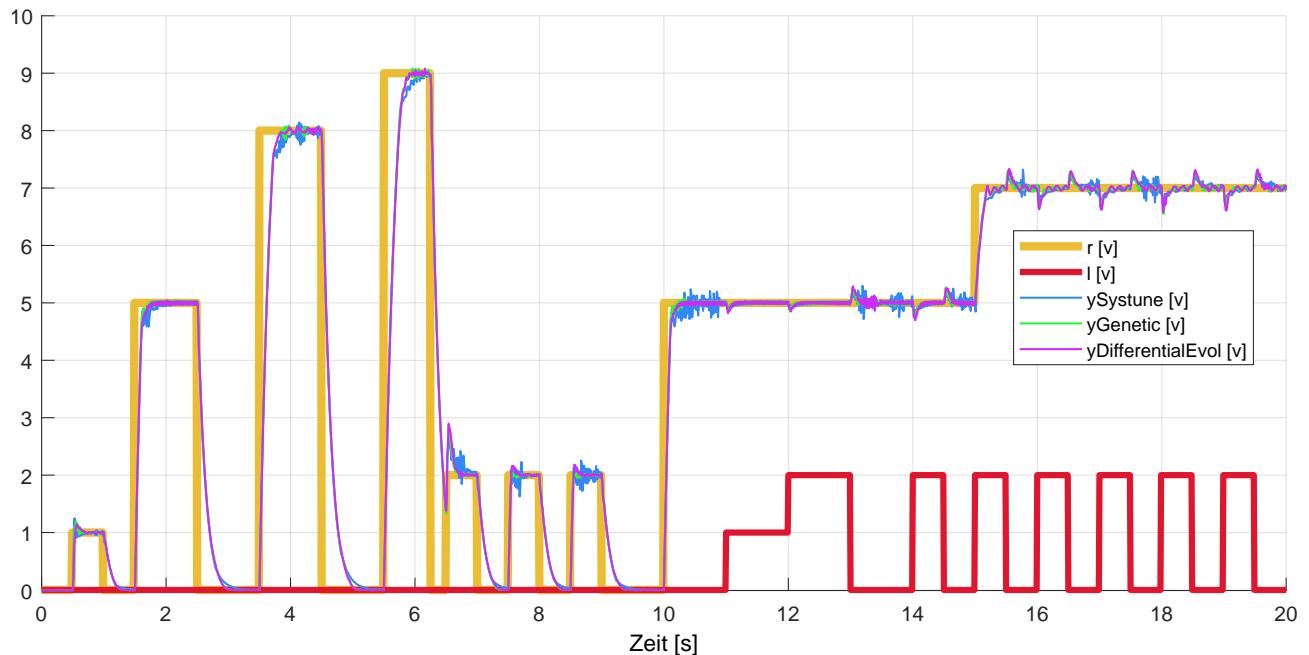


Abb. 14: Kombinierte Antwort auf Stimulation der optimierten Regler am DC-Motor

In Abb. 14 ist die Antwort auf die Stimulationssignale der drei Systeme dargestellt. Das Signal  $r$  ist die Ziel-Drehzahl und  $l$  stellt die Störgröße dar.

### Systune Regler

Der mit *systune* optimierte Regler zeigt ein stärkeres Rauschverhalten als die anderen beiden Regler.

### Genetisch optimierter Regler

Der mit dem *GA* optimierte Regler zeigt ein schnelleres Verhalten als der *systune* Regler

### Differential Evolution optimierter Regler

Der mit dem *DE* optimierte Regler zeigt ein ziemlich ähnliches Verhalten wie der Genetisch optimierte Regler. Er hat jedoch ein etwas höheres Überschwingen. Die Rauschunterdrückung ist bei diesem Regler am besten.

## 6.3 | Lernverlauf

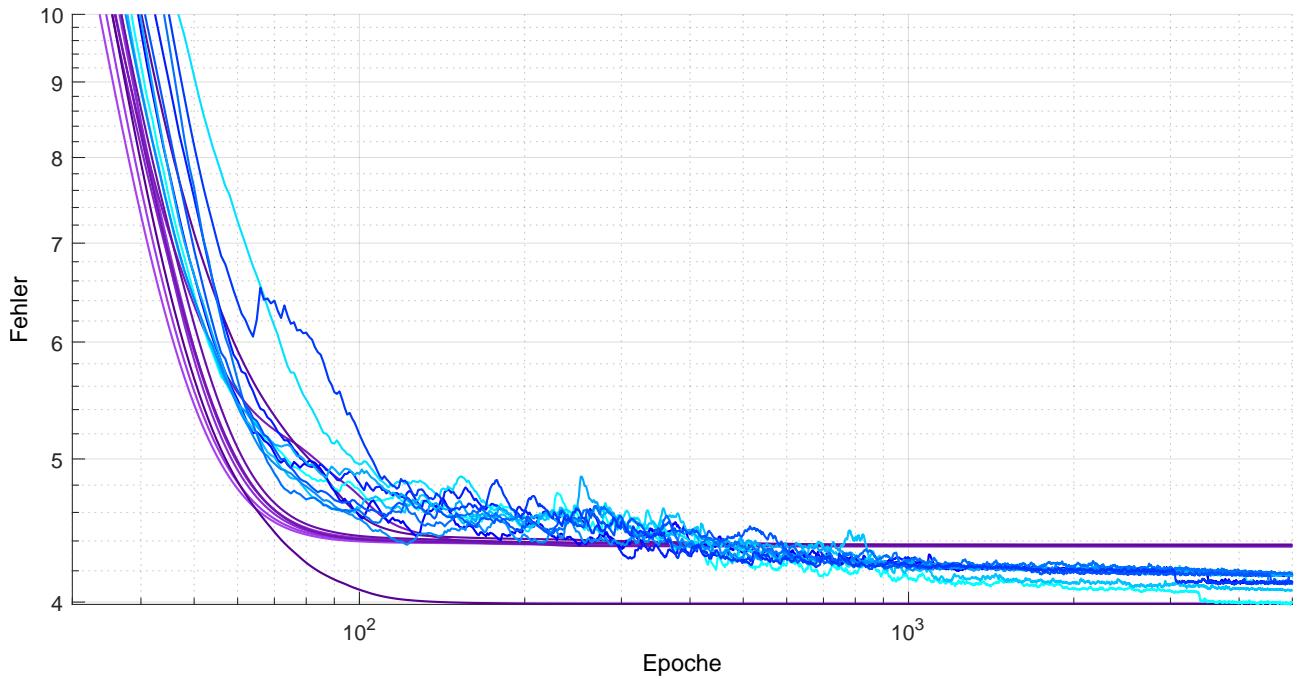


Abb. 15: Lernverlauf der kombinierten Optimierungsmethoden (GA + DE)

In Abb. 15 ist der Lernverlauf der beiden Methoden **GA** und **DE** übereinandergelegt dargestellt. Der Vergleich zeigt, beide Methoden sind in der Lage das gleiche Optimum zu finden. Der DE konvergiert dabei etwas schneller als der GA, hat aber das Problem, dass in dem Test 9 von 10 Durchläufen nicht das bessere Optimum gefunden wurde.

Der Lernverlauf ist leider für den *systune* Regler nicht verfügbar, damit dennoch ein Vergleich möglich ist wendet man die vom *systune* berechneten Reglerparameter in der Simulationssoftware an und misst den Fehler mit der gleichen Bewertungsfunktion wie bei den anderen beiden Methoden. *Systune* erreicht mit den in [Tab. 6 Reglerparameter des mit \*systune\* optimierten Reglers](#) angegebenen Parametern einen Fehler von 5.06624.

Der GA hat deutlich länger um das bislang beste Ergebnis des DE zu erreichen, schafft es aber in allen 10 Durchläufen eine bessere Lösung zu finden als der DE in 9 von 10 Durchläufen.

*Systune* hat mit 5.06624 einen höheren Fehler als beide anderen Methoden erreicht. Dies liegt sehr wahrscheinlich daran, dass die Optimierungskriterien von *systune* nicht exakt mit der Bewertungsfunktion der Simulationssoftware übereinstimmen.

Eine weitere Vermutung ist, dass die Methoden GA und DE eine bessere Lösung finden, weil sie auch an nichtlinearen Systemen angewendet werden und sich dadurch besser auf die nichtlinearen Effekte, wie die vorhandenen Sättigungen und den Anti-Windup, anpassen können.

## 6.4 | Frequenzanalyse der optimierten Regler

Im Abb. 16 sind die Bode-Diagramme der drei optimierten Regler dargestellt. Diese zeigen die Frequenzgänge der Regler ohne die Regelstrecke. Es ist zu erkennen, dass die Regler die Phase auf  $0^\circ$  anheben und die Verstärkung für kleine Frequenzen sehr hoch ist. Alle drei Regler zeigen kein striktes Tiefpassverhalten. Frequenzen ab 10rad/s werden nicht weiter gedämpft.

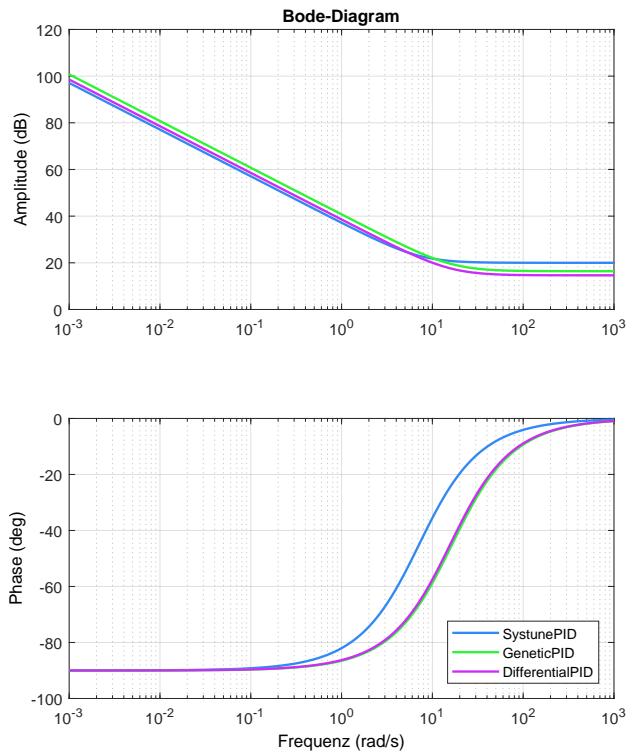
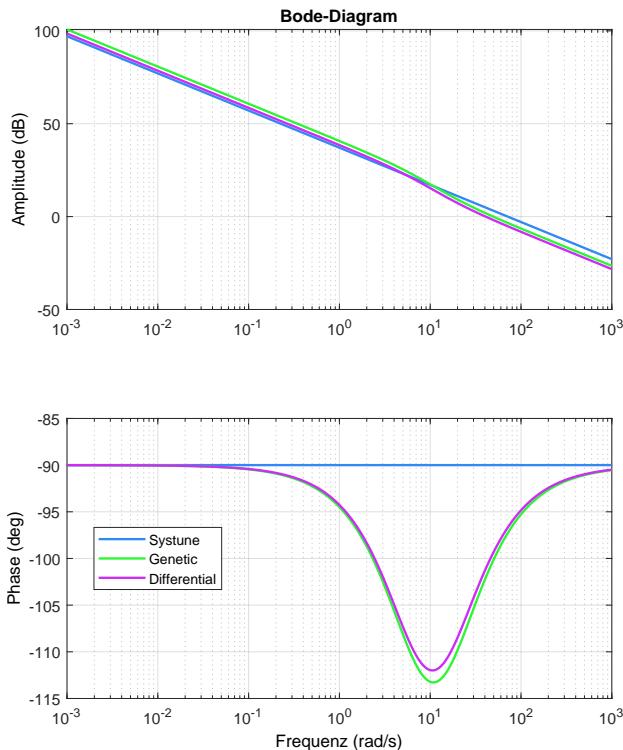


Abb. 16: Bode-Diagramm der optimierten Regler

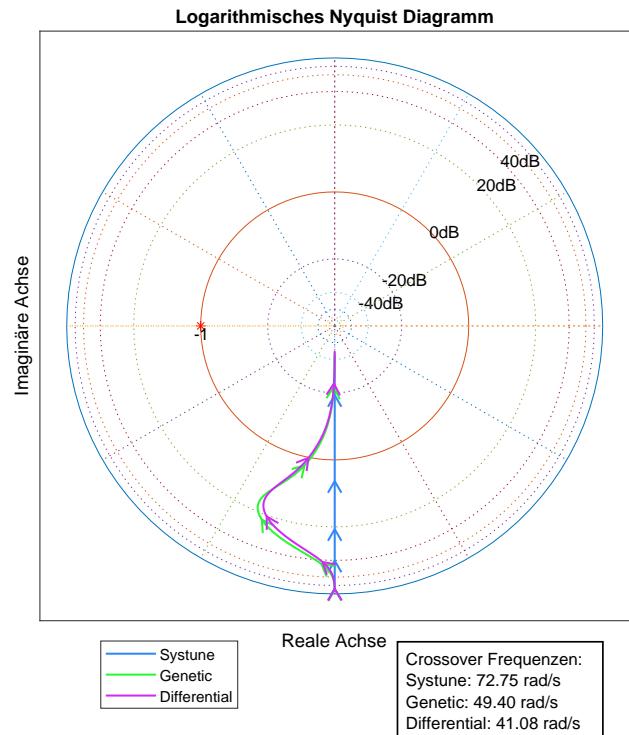
## 6.5 | Frequenzanalyse der optimierten Systeme



Das Bode-Diagramm in Abb. 17 und das Nyquist-Diagramm in Abb. 18 zeigen, jeweils die Frequenzgänge der optimierten Vorwärts Pfade der drei Systeme:

- $PID_{systune} \cdot MotorSystem$
- $PID_{genetic} \cdot MotorSystem$
- $PID_{differential} \cdot MotorSystem$

Im Bode-Diagramm fällt auf, dass der Regler von *systune* nicht zusätzliche Phase klaut. Die beste Rauschunterdrückung bietet jedoch der Regler von Differential Evolution.



Phasenreserve	Wert
<i>Systune</i>	90.00 °
Genetisch	80.83 °
Differential Evolution	79.97 °
Verstärkungsreserve	
<i>Systune</i>	Inf dB
Genetisch	Inf dB
Differential Evolution	Inf dB

Tab. 9: Reserven optimierten Systeme

## 6.6 | Stabilitätsanalyse der optimierten Systeme

### 6.6.1 | Sensitivität

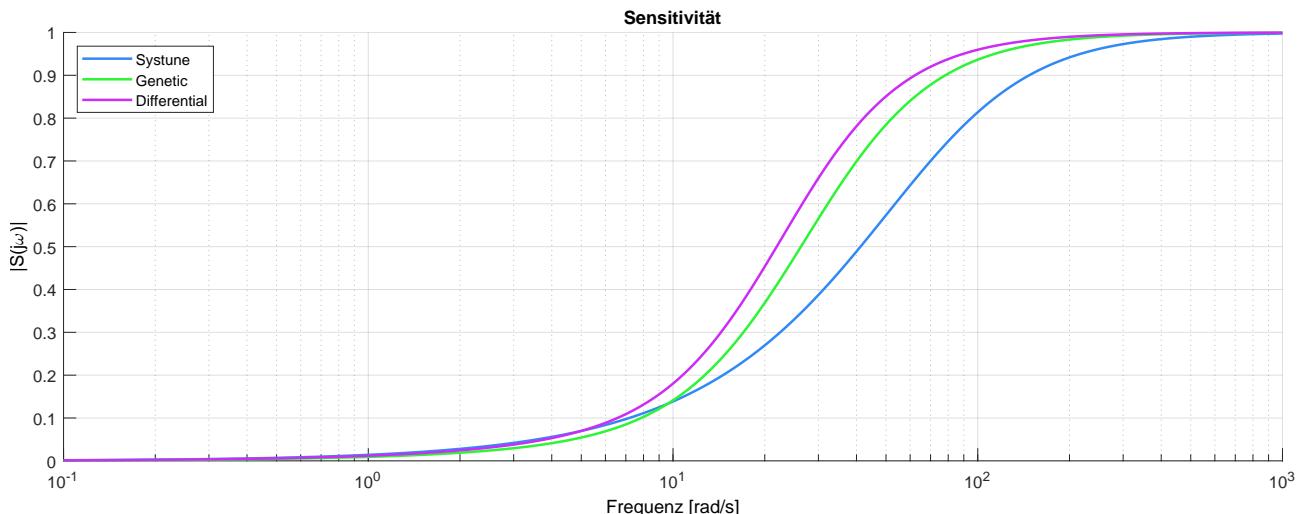


Abb. 19: Sensitivität der optimierten Systeme

Das Sensitivitäts-Diagramm in Abb. 19 zeigt, für alle drei Systeme einen ähnlichen Verlauf. Keine Überhöhung, was ein gutes Stabilitätsverhalten andeutet. Nicht auf den ersten Blick ersichtlich ist jedoch, dass die Systeme sehr empfindlich auf Totzeiten reagieren. Dies ist in Abschnitt 6.6.3 Totzeit- & Verstärkungsreserve Diagramme der optimierten Systeme dargestellt.

### 6.6.2 | Phasen- & Verstärkungsreserve Diagramme der optimierten Systeme

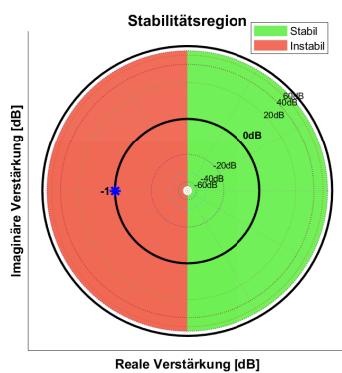


Abb. 20: Phasen- & Verstärkungsreserve des mit systune optimierten Systems

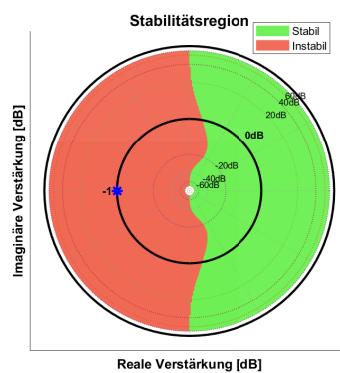


Abb. 21: Phasen- & Verstärkungsreserve des mit Genetic optimierten Systems

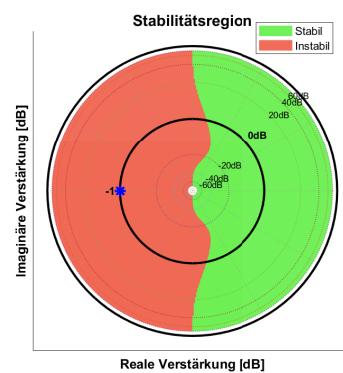


Abb. 22: Phasen- & Verstärkungsreserve des mit Differential Evolution optimierten Systems

Die drei Diagramme in Abb. 20, Abb. 21 und Abb. 22 zeigen die detaillierten Stabilitätsreserven als Kombination von Verstärkung und konstanter Phasenverschiebung. Wie diese Diagramme zu lesen sind, ist in folgendem Kapitel erklärt:

#### A.2.7 Phasen- & Verstärkungsreserve Diagramm

Das System welches mit *systune* optimiert wurde, zeigt eine größere Fläche indem das System stabil bleibt. Dies deutet darauf hin, dass dieses System robuster gegenüber Verstärkungsänderungen und Phasenverschiebungen ist.

### 6.6.3 | Totzeit- & Verstärkungsreserve Diagramme der optimierten Systeme

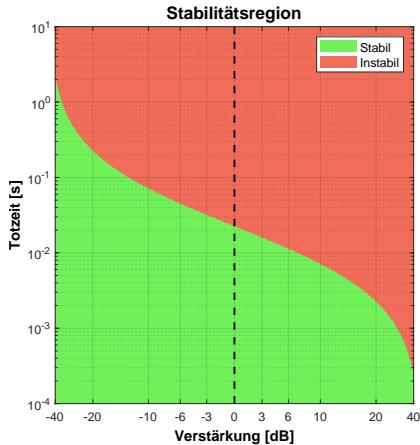


Abb. 23: Totzeit- & Verstärkungsreserve des mit *systune* optimierten Systems

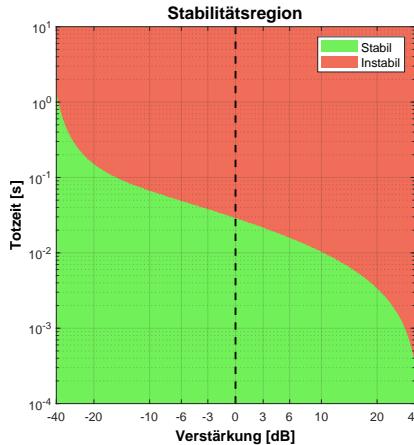


Abb. 24: Totzeit- & Verstärkungsreserve des mit *Genetic* optimierten Systems

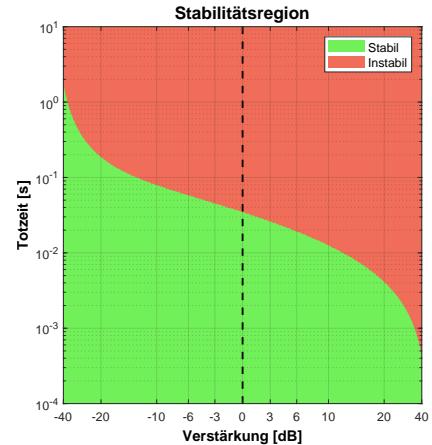


Abb. 25: Totzeit- & Verstärkungsreserve des mit *Differential Evolution* optimierten Systems

Die drei Diagramme in Abb. 23, Abb. 24 und Abb. 25 zeigen die detaillierten Stabilitätsreserven aus der Kombination von Verstärkung und Totzeit. Wie diese Diagramme zu lesen sind, ist in folgendem Kapitel erklärt:

#### A.2.8 Totzeit- & Verstärkungsreserve Diagramm

Auch wenn die Verstärkungsreserven und Phasenreserven der drei Systeme scheinbar sehr gut sind, reagieren die Systeme auf Kombinationen von Totzeit und Verstärkungsänderungen sehr empfindlich. Bereits eine kleine Totzeit von ca. 4ms führt bei allen drei Systemen zu Instabilität. Die Systeme verhalten sich also nicht sehr robust.

Besonders das mit *systune* optimierte System, welches in den obigen Analysen doch die besten Ergebnisse, was Robustheit anbelangt, gezeigt hat, reagiert auf Totzeiten am empfindlichsten und wird bereits bei einer Totzeit von ca. 2.5ms instabil.

## 7 | Systune am Motor mit Schwungmasse

Wie bereits im Beispiel 3 [Systune am DC-Motor](#) beschrieben, wird hier der gleiche Ablauf, aber diesmal für das Motor-mit-Schwungmasse-System, durchgeführt. Deshalb wird darauf verzichtet jeden Schritt in Detail erneut zu beschreiben. Das komplette MATLAB-Skript kann im GitHub Repository [\[2\]](#) eingesehen werden.

### 7.1 | Regelkreis Definieren

Der verwendete Regelkreis ist im Kapitel [A.2.2 Regelkreis für Prozess: Motor-mit-Schwungmasse](#) genauer beschrieben. Da der Regelkreis für *systune* linear sein muss, werden die nichtlinearen Sättigungsblöcke vernachlässigt und erst beim Testen des optimierten Systems wieder berücksichtigt.

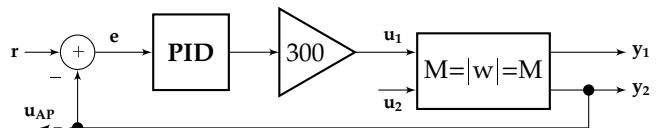


Abb. 26: Regelkreis des Motors mit Schwungmasse

### 7.2 | Optimierungsziele definieren

Bei den optimierungszielen können verschiedene Kriterien verwendet werden. In diesem Beispiel werden drei Ziele definiert:

- **Step-Tracking:** Das System soll der Sprungantwort eines PT2-Systems folgen.
- **Overshoot:** Das Überschwingen soll begrenzt werden.
- **Phasen- und Verstärkungsreserve:** Die Stabilitätsreserven sollen eingehalten werden.

#### 7.2.1 | TuningGoal.StepTracking [26]

Das Step-Tracking Ziel sorgt dafür, dass das System einer vordefinierten Sprungantwort folgt. Dafür wird ein PT2-System als Referenz verwendet. In diesem Beispiel wird eine Zeitkonstante von  $\tau = 0.2$  und eine Dämpfung von  $D = 1$  verwendet. Die Wahl dieser Parameter entspricht approximativ den Parametern aus der Impulsantwort welche bei der Systemidentifikation in Kapitel [A.2.2 Regelkreis für Prozess: Motor-mit-Schwungmasse](#) ermittelt wurde.

$$H_{\text{ref}}(s) = \frac{1}{\tau^2 \cdot s + 2 \cdot D \cdot \tau \cdot s + 1} \quad \text{mit } \tau = 0.2, D = 1$$

(Formel 7.2.1)

```
1 pt2T = 0.2;
2 pt2D = 1;
3 pt2Step = tf([1], [pt2T*pt2T 2*pt2D*pt2T 1]);
4 TR1 = TuningGoal.StepTracking('r', 'y2', pt2Step);
```

Code 19: Definition des Step-Tracking Ziels

#### 7.2.2 | TuningGoal.Overshoot [24]

Das Overshoot Ziel begrenzt das Überschwingen des Systems. Ziel ist es, vom Referenzsignal  $r$  zur Ausgangsgröße  $y2$  nicht mehr als 10% Überschwingen zuzulassen.

```
1 TR2 = TuningGoal.Overshoot('r', 'y2', 10);
```

Code 20: Definition des Overshoot Ziels

#### 7.2.3 | TuningGoal.Margins [22]

Das Margins-Ziel sorgt dafür, dass die Stabilitätsreserven eingehalten werden. In diesem Beispiel sollen eine Phasenreserve von mindestens  $45^\circ$  und eine Verstärkungsreserve von mindestens 6dB eingehalten werden.

Der angegebene Analysis-Point 'uAP' steht für den Teil des Feedbacks, welcher für die Stabilitätsanalyse aufgetrennt wird um die Messung der Phasen- und Verstärkungsreserven zu ermöglichen.

```
1 TR4 = TuningGoal.Margins('uAP', 6, 45);
```

Code 21: Definition des Margins Ziels

#### 7.2.4 | Optimierungsziele kombinieren

```
1 targetSoftGoals = [TR1 TR2]; % Step-Tracking und Overshoot
2 targetHardGoals = [TR4]; % Phasen- und Verstärkungsreserven
```

Code 22: Kombinieren der Optimierungsziele

## 7.3 | Regler extrahieren

Nach der Optimierung kann der optimierte Regler aus dem Gesamtsystem extrahiert werden.

```
1 controllerSysTuned = getBlockValue(sysTuned, 'controllerSys');
2 disp(controllerSysTuned);
```

*Code 23: Regler extrahieren*

```
1 pid with properties:
2
3     Kp: -0.4108
4     Ki: -4.6926e-08
5     Kd: -0.7332
6     Tf: 0.0802
7     ...
8
```

*Code 24: Optimierter Regler*

## 7.4 | Optimierung auswerten

Die Ziele wurden gut erreicht, da alle unter 1 liegen.

```
1 Final soft objective: 0.3065 (< 1 is good)
2 Final soft objective: 0.6893 (< 1 is good)
3 Final hard objective: 0.6066 (< 1 is good)
4
5 %==== Performance Comparison ===
6 %           Initial   Tuned
7 %Rise Time:      NaN s   0.660 s
8 %Settling Time:  NaN s   1.180 s
9 %Overshoot:     NaN %   0.31 %
10 %
11 %Gain Margin:   Na dB   19.83 dB
12 %Phase Margin: <0 °    75.71 °
```

*Code 25: TuningGoals Auswertung*

### 7.4.1 | Step-Tracking Ergebnis visualisiert

```
1 viewGoal(targetSoftGoals(1), sysTuned);
```

Code 26: Step-Tracking Diagramm

Das Diagramm in Abb. 27 zeigt die Sprungantwort des optimierten Systems. Die Kurve mit dem Namen *Desired* entspricht der gewünschten Sprungantwort, die durch das Step-Tracking Ziel definiert wurde:  $H_{ref}(s) = \frac{1}{\tau^2 \cdot s + 2 \cdot D \cdot \tau \cdot s + 1}$  mit  $\tau = 0.2, D = 1$ . In blau ist die Sprungantwort des optimierten Systems dargestellt. Diese deckt sich hauptsächlich kurz nach dem Start sehr gut mit der gewünschten Sprungantwort.

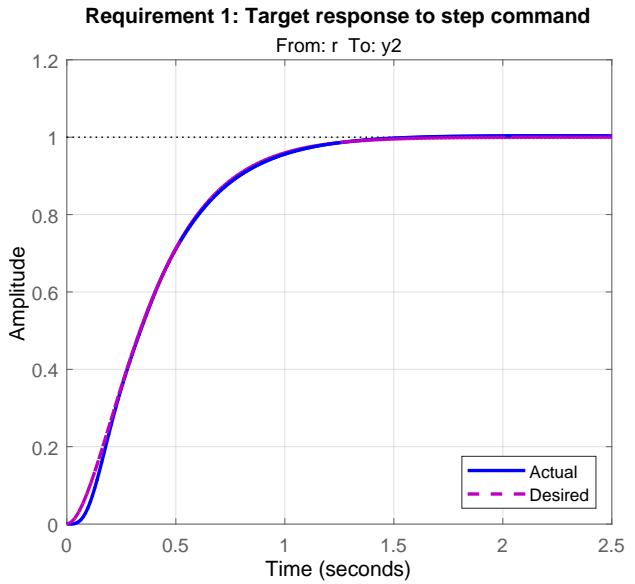


Abb. 27: Step-Tracking Optimierungsergebnis mit systune am Motor mit Schwungmasse

### 7.4.2 | Overshoot Ergebnis visualisiert

```
1 viewGoal(targetSoftGoals(2), sysTuned);
```

Code 27: Overshoot Diagramm

Das Diagramm in Abb. 28 zeigt den Frequenzgang des optimierten geschlossenen Systems. Der orange Bereich stellt den Tiel der Verstärkung dar, welcher grösser als die erlaubten 10% Überschwingen ist. Die blaue Kurve zeigt die Verstärkung des optimierten Systems, diese ist im gesamten Frequenzbereich unterhalb der 10% Grenze, was zeigt, dass das Overshoot Ziel erfolgreich erfüllt wurde.

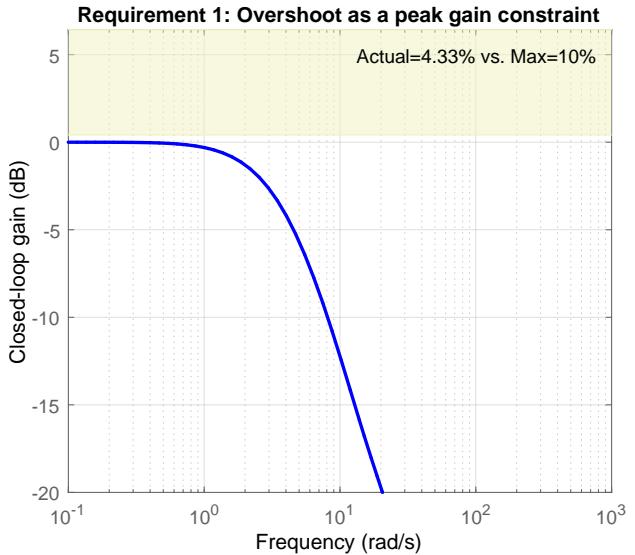


Abb. 28: Overshoot Optimierungsergebnis mit systune am Motor mit Schwungmasse

### 7.4.3 | Phasen- und Verstärkungsreserven Ergebnis visualisiert

```
1 viewGoal(targetHardGoals(1), sysTuned);
```

Code 28: Phasen- und Verstärkungsreserven Diagramm

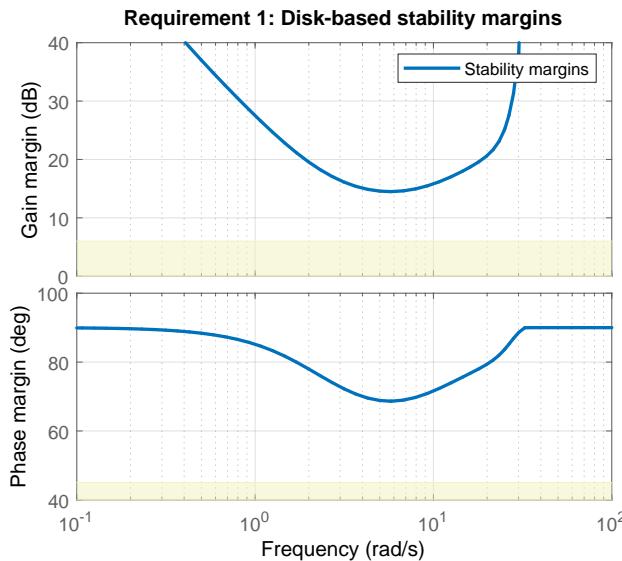


Abb. 29: Phasen- und Verstärkungsreserven Optimierungsergebnis mit systune am Motor mit Schwungmasse

Das Diagramm in Abb. 29 zeigt die Verstärkungs- und Phasenreserven des optimierten geschlossenen Systems. Die blaue Kurve zeigt jeweils den Verstärkungsreserven (oben) und die Phasenreserve (unten) des optimierten Systems. Beide Kurven liegen über den geforderten Mindestreserven (oranger Bereich), was zeigt, dass das Margins-Ziel erfolgreich erfüllt wurde.

Mehr Informationen darüber, wie die Optimierungsergebnisse visualisiert werden, ist in der MATLAB Dokumentation zu den Tuning-Goals beschrieben. [31]

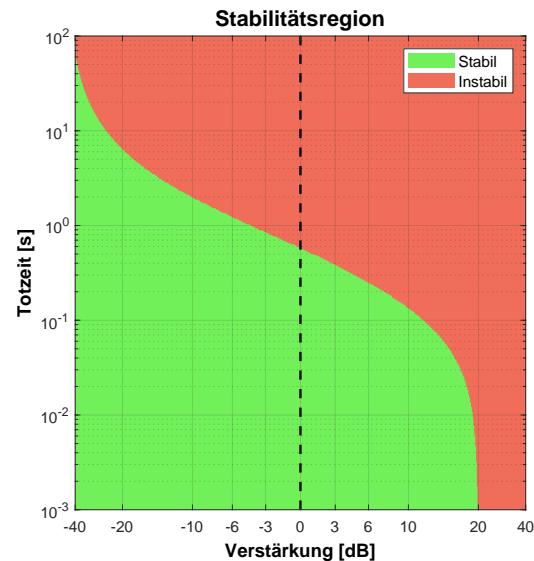


Abb. 30: Totzeit & Verstärkungsreserve Diagramm des optimierten Systems

Das von MATLAB generierte Diagramm kann verwirren und ist nicht einfach zu interpretieren. Eine bessere Darstellung über die resultierten Phasen- und Verstärkungsreserven ist in Abb. 30 zu sehen. Wie diese Abbildung interpretiert wird, ist in folgendem Kapitel erklärt:

#### A.2.8 Totzeit- & Verstärkungsreserve Diagramm

Das Diagramm zeigt, dass das System über eine grosse Phasenreserve verfügt. Das zeigt sich durch die Toleranz der möglichen Totzeiten mit bis zu 0.6s bei einer Verstärkung von 1. Außerdem besitzt das System eine gute Verstärkungsreserve mit bis zu 20dB (Verstärkungsfaktor = 10).

## 7.5 | Optimierten Regler am realen System testen

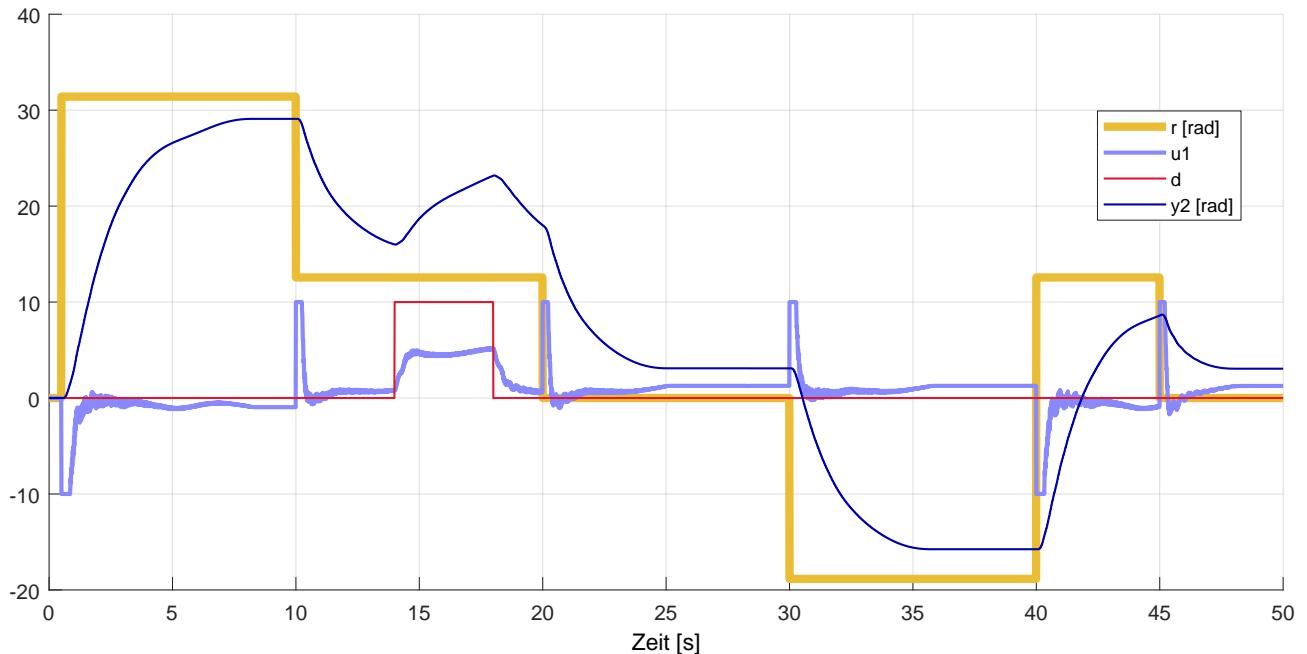


Abb. 31: Antwort auf Stimulation des optimierten Reglers am Motor mit Schwungmasse, mit dem durch systune optimierten PID-Regler

Für den Test am realen Prozess sind im Simulink Modell unterschiedliche Sollwerte vorbereitet. Auch die Sättigungen sind wieder eingebaut. Da der PID-Regler am Ausgang einen Sättigungsblock besitzt, entsteht das Problem des Integrator Windups. Die Clamping Anti-Windup Methode wird verwendet, da diese keinen weiteren Tuning-Parameter erzeugt. *Systune* kann leider keinen Anti-Windup Tuning-Parameter optimieren.

Der Regler verstärkt an seinem Ausgang  $u_1$  das Messrauschen des Encoders, welches zu starken Stromschwankungen am Motoreingang führen. Es ist zu erkennen, dass der Ist-Winkel  $y_2$ , sobald die Störgrösse dazu geschaltet wird, davon driftet. Der Regler besitzt deshalb keine guten Eigenschaften um Störgrößen zu kompensieren.

## 8 | GA am Motor mit Schwungmasse

Weil der [GA](#) kein MATLAB voraussetzt, ist diese Methode auch ohne MATLAB implementiert worden. Das hat den Vorteil, dass ein System auch ohne MATLAB Lizenz optimiert werden kann.

Die Funktionsweise des GA ist im Kapitel [A.2.5 Genetischer Algorithmus](#) beschrieben.

### 8.1 | System Definieren

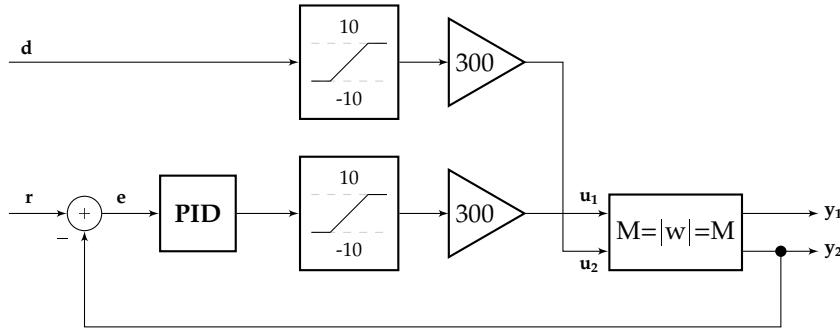


Abb. 32: Regelkreis Motor mit Schwungmasse

In der C++ Anwendung werden die Teilsysteme objektorientiert implementiert und anschliessend als ein gesamtheitliches Objekt zusammengeführt und verbunden. Der Kernteil der Implementierungslogik spielt sich dabei in den Update-Funktionen der jeweiligen Teilsysteme ab. Aufgrund des Codes-Umfangs werden deshalb in den nachfolgenden Unterkapiteln nur die Update-Funktionen gezeigt.

#### 8.1.1 | Motor definieren

Die Implementierung im C++ Code ist auf der Grundlage der Modellidentifikation des Systems im Kapitel [A.2.2 Regelkreis für Prozess: Motor-mit-Schwungmasse](#) durchgeführt worden. Die dazugehörige C++ Implementierung ist unter [31 Update Funktion des Motors Mit Schwungmasse im Code](#) zu finden.

#### 8.1.2 | PID-Regler definieren

Der PID-Regler wurde bereits im Kapitel [A.2.3 Reglerstruktur: Erweiterter PID-Regler](#) beschrieben und auch die C++ Implementierung wird dort aufgeführt. Deshalb wird hier nicht weiter darauf eingegangen.

#### 8.1.3 | Alle Teilsysteme verbinden

```

1 // Update Funktion des Gesamtsystems im C++ Code. class TestSystem
2 void update(double deltaTime) override
3 {
4     double y = m_motorWithMass.getOutputs()[1]; // y(t)
5     double measurementNoise = AutoTuner::Solver::getRandomDouble(-1, 1) * 0.1;
6     m_errorValue = m_referenceValue - y + measurementNoise; // e(t) = r(t) - y(t)
7
8     m_pidController.setInput(m_errorValue);
9
10    m_pidController.update(deltaTime);
11    m_pidOutputValue = m_pidController.getOutput();
12    m_motorWithMass.setInputSignal(0, m_pidOutputValue);
13    m_motorWithMass.setInputs(1, m_disturbanceValue);
14    m_motorWithMass.update(deltaTime);
15 }

```

Code 29: Update-Funktion des Gesamtsystems im Code

Wie im Code ersichtlich ist, wird dem System ein zusätzliches Messrauschen hinzugefügt. Dieses begünstigt die Eigenschaft der Rauschunterdrückung des Reglers durch die Bewertungsteilfunktion  $g_1(\vec{T_i})$  (Stellwertänderungsrate).

## 8.2 | Optimierungsziele definieren

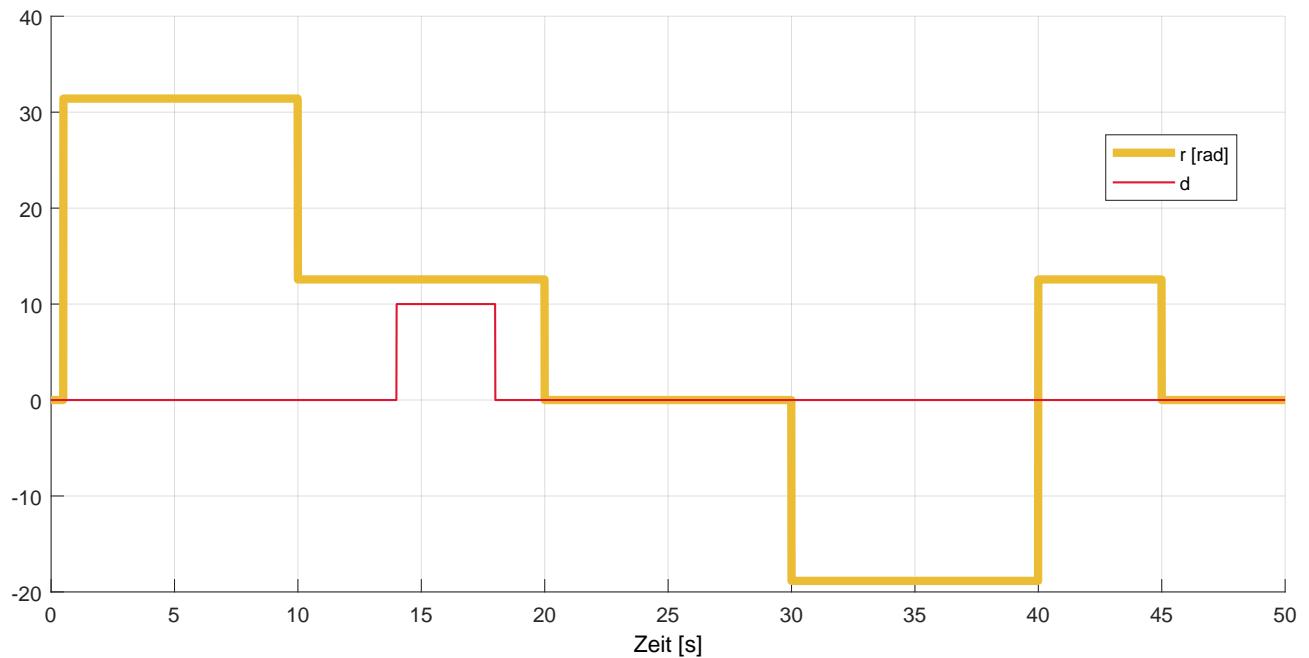


Abb. 33: Anregungssignale für die Systemsimulation

Um die Bewertungsfunktion zu definieren wird eine zeitliche Simulation des Systems benötigt. Dazu wird ein vordefiniertes Anregungssignal verwendet. Das Anregungssignal besteht aus  $r$  (Ziel-Winkel) und  $d$  (Störungsmoment).

Symbol	Beschreibung	Wert	Einheit
$T_s$	Abtastzeit	0.01	s
$t_{\text{start}}$	Startzeit	0	s
$t_{\text{end}}$	Endzeit	50	s
$u_{\text{satHigh}}$	Obere Sättigungsgrenze der Stellgrösse	10	
$u_{\text{satLow}}$	Untere Sättigungsgrenze der Stellgrösse	-10	
$K$	Anzahl Zeitschritte in der Simulation	5000	
$D$	Anzahl der Optimierungs-Parameter	4	
$\vec{T}_i$	Individuum $i$ in der Population	$\vec{T}_i \in \mathbb{R}^D$	
$g(\vec{T}_i)$	Fehler-Minimierungs-Funktion.	$g(\vec{T}_i) \in \mathbb{R}$	
$a_o$	Gewichtungsfaktor für das Optimierungsziel $o$	$a_o \in \mathbb{R}^{+0}$	
$O$	Anzahl Optimierungsziele	5	
$g_o(\vec{T}_i, k)$	Optimierungsfunktion für das Ziel $o$ zum Zeitpunkt $k$	$g_o(\vec{T}_i, k) \in \mathbb{R}$	

Tab. 10: Symbole für die Simulation am Motor mit Schwungmasse

Für die Abtastzeit  $T_s$  wurde ein realistischer Wert von 10ms gewählt.

Die Anzahl Zeitschritte  $K$  wird aus der Abtastzeit  $T_s$  und dem Simulationszeitraum  $[t_{\text{start}}, t_{\text{end}}]$  berechnet.

$$K = \frac{t_{\text{end}} - t_{\text{start}}}{T_s} \quad (\text{Formel 8.2.1})$$

Der **GA** ist für **Maximierungsprobleme** ausgelegt. Für diese Anwendung wäre es aber sinnvoller die Optimierungsziele als **Minimierungsprobleme** zu formulieren. Es gibt jedoch eine Möglichkeit zur Umrechnung von **Minimierungsproblemen** in Maximierungsprobleme. Dies ist im Kapitel **A.2.5.5 Vorgehen bei der Umwandlung  $g(\vec{T}_i) \rightarrow f(\vec{T}_i)$**  beschrieben. Dank dieser Umrechnung können die Optimierungsziele als Minimierungsproblem formuliert werden.

Die Fehler-Minimierungs-Funktion  $g(\vec{T}_i)$  wird aus mehreren Teilsummen zusammengesetzt. Die Teilsummen bewerten verschiedene Aspekte des Regelverhaltens. Allgemein lässt sich die Fehler-Minimierungs-Funktion  $g(\vec{T}_i)$  wie in der **Formel 8.2.2** darstellen.

$$g(\vec{T}_i) = \frac{1}{K} \cdot \sum_{k=0}^{K-1} \left( \sum_{o=0}^{O-3} (a_o \cdot g_o(\vec{T}_i, k)) \right) + a_3 \cdot g_3(\vec{T}_i) + a_4 \cdot g_4(\vec{T}_i) \quad (\text{Formel 8.2.2})$$

- $\frac{1}{K}$  dient zur Normierung der Fehlerbewertung auf die Anzahl der Zeitschritte.
- $\sum_{k=0}^{K-1}$  summiert die Fehlerbewertung über alle Zeitschritte der Simulation.
- $\sum_{o=0}^{O-3} (a_o \cdot g_o(\vec{T}_i, k))$  summiert die gewichteten Optimierungsziele  $g_o(\vec{T}_i, k)$  für den Zeitschritt  $k$ .
- $a_3 \cdot g_3(\vec{T}_i)$  bewertet die Verstärkungsreserve des Systems.
- $a_4 \cdot g_4(\vec{T}_i)$  bewertet die Phasenreserve des Systems.

$g_3(\vec{T}_i)$  und  $g_4(\vec{T}_i)$  werden nicht über die Zeitschritte während der Simulation summiert, da sie nicht abhängig von der Laufzeit der Simulation sind, sondern approximiert im Frequenzbereich berechnet werden.

Folgende Optimierungsziele wurden bereits im Kapitel [4.2 Optimierungsziele definieren](#) im Zusammenhang mit dem DC-Motor beschrieben und werden hier deshalb nicht nochmals erklärt:

- $g_0(\vec{I}_i, k)$ : Absoluter Regelungsfehler aber ohne Sättigungsbedingung
- $g_1(\vec{I}_i, k)$ : Stellwertänderungsrate
- $g_2(\vec{I}_i, k)$ : Positives Überschwingen unterdrücken

### 8.2.1 | $g_3(\vec{I}_i)$ : Verstärkungsreserve

$g_3(\vec{I}_i)$  bewertet die quadratische Abweichung der Verstärkungsreserve des Systems von dem gewünschten Wert von 2.

Diese Bewertung wurde für den Test deaktiviert ( $a_3 = 0$ ), da die Optimierung dem System von sich aus eine weitaus bessere Verstärkungsreserve verpasst als gefordert und so wie die Bewertung formuliert ist, würde der [Solver](#) versuchen, die Verstärkungsreserve wieder zu verschlechtern.

### 8.2.2 | $g_4(\vec{I}_i)$ : Phasenreserve

$g_4(\vec{I}_i)$  bewertet die quadratische Abweichung der Phasenreserve des Systems von dem gewünschten Wert von  $90^\circ$ . Die Bewertung wird erst ab Epoche 100 durchgeführt, damit der Algorithmus zuerst eine grobe Lösung finden kann, bevor die Stabilitätskriterien berücksichtigt werden. Die Frequenzanalyse wird mit dem nicht linearisierten System im Zeitbereich durchgeführt. Zu Beginn der Simulation ist das System noch instabil, was zu einer falsch berechneten Frequenzanalyse führt und somit ist die Frequenzanalyse erst ab Epoche 100 sinnvoll.

Es wurde  $90^\circ$  gewählt, weil damit versucht wird, möglichst nah an  $90^\circ$  zu kommen. *Systune* hat gezeigt, dass es möglich ist, nah an  $90^\circ$  zu kommen, ohne das System zu träge zu machen.

### 8.2.3 | $g(\vec{I}_i)$ : Gesamte Fehlerbewertung

Die Teilbewertungen werden jeweils mit einem Gewichtungsfaktor  $a_0$  skaliert. Diese müssen vom Anwender passend gewählt werden, um das gewünschte Verhalten zu erzielen. Es ist zu beachten, dass  $g_4(\vec{I}_i)$  nicht über die Simulationszeit summiert wird, sondern nur einmal pro Individuum berechnet wird, weil sich der Frequenzgang im Verlauf einer Simulationsdurchführung nicht ändert.

$$g(\vec{I}_i) = \frac{1}{K} \cdot \sum_{k=0}^{K-1} (a_0 \cdot g_0(\vec{I}_i, k) + a_1 \cdot g_1(\vec{I}_i, k) + a_2 \cdot g_2(\vec{I}_i, k)) + a_4 \cdot g_4(\vec{I}_i)$$

(Formel 8.2.3)

## 8.3 | Optimierung durchführen

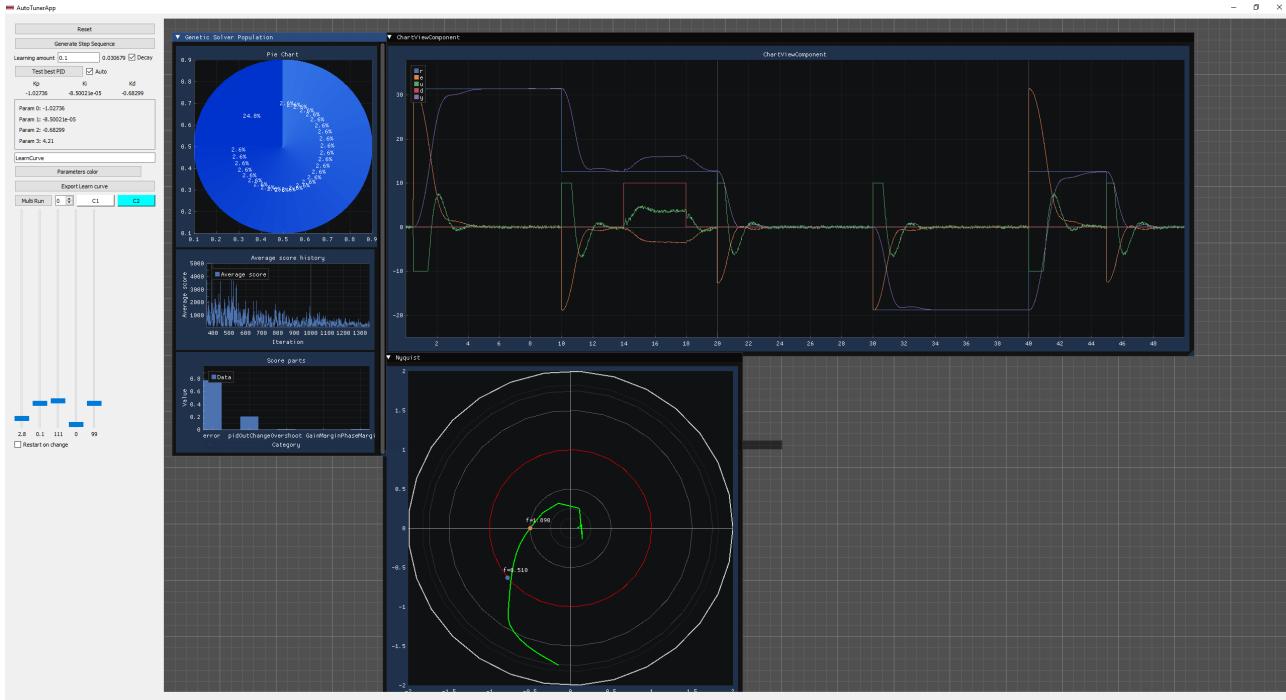


Abb. 34: Software der Systemsimulation für die Optimierung

Wie beim Beispiel mit dem DC-Motor Abb. 9 wird die Optimierung 10-mal durchgeführt und bestimmte Hyperparameter werden willkürlich gewählt.

Mehr zur Bestimmung der Hyperparameter ist im Kapitel [11.1.5 Iteratives Tuning der Methoden](#) beschrieben.

Parameter	Beschreibung	Wert
$D$	Anzahl der einzelnen Optimierungsdurchläufe	10
$N$	Anzahl Individuen in der Population	30
$G$	Anzahl Generationen	1000
$C$	Mutations-Wahrscheinlichkeit	0.05
$M(k)$	Mutationsstärke	$1 \cdot 0.99^k$
Anti-Windup	Verwendete Anti-Windup Methode	Clamping
<b>Optimierungsziele</b>		
$a_0$	Absoluter Regelungsfehler	2.8
$a_1$	Stellwertänderungsrate	0.1
$a_2$	Positives Überschwingen	111
$a_3$	Verstärkungsreserve (6dB)	0
$a_4$	Phasenreserve (90°)	99 Ab Epoche 100
<b>Konstante Systemparameter</b>		
$u_{\text{satHigh}}$	PID Ausgangs-Sättigung oberer Wert	10
$u_{\text{satLow}}$	PID Ausgangs-Sättigung unterer Wert	-10
$I_{\text{SatHigh}}$	PID-Integrator-Sättigung oberer Wert	100000000
$I_{\text{SatLow}}$	PID-Integrator-Sättigung unterer Wert	-100000000
<b>PID-Startparameter</b>		
$K_p$	Proportionalitätsfaktor	$0 + \text{randG}(-1,1) *$
$K_i$	Integrationsfaktor	$0 + \text{randG}(-1,1) *$
$K_d$	Ableitungsfaktor	$0 + \text{randG}(-1,1) *$
$K_n$	Filterkonstante	$0 + \text{randG}(-1,1) *$

Tab. 11: Hyper- & Tuningparameter für Optimierung mit dem genetischen Algorithmus

\* $\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

## 8.4 | Optimierung auswerten

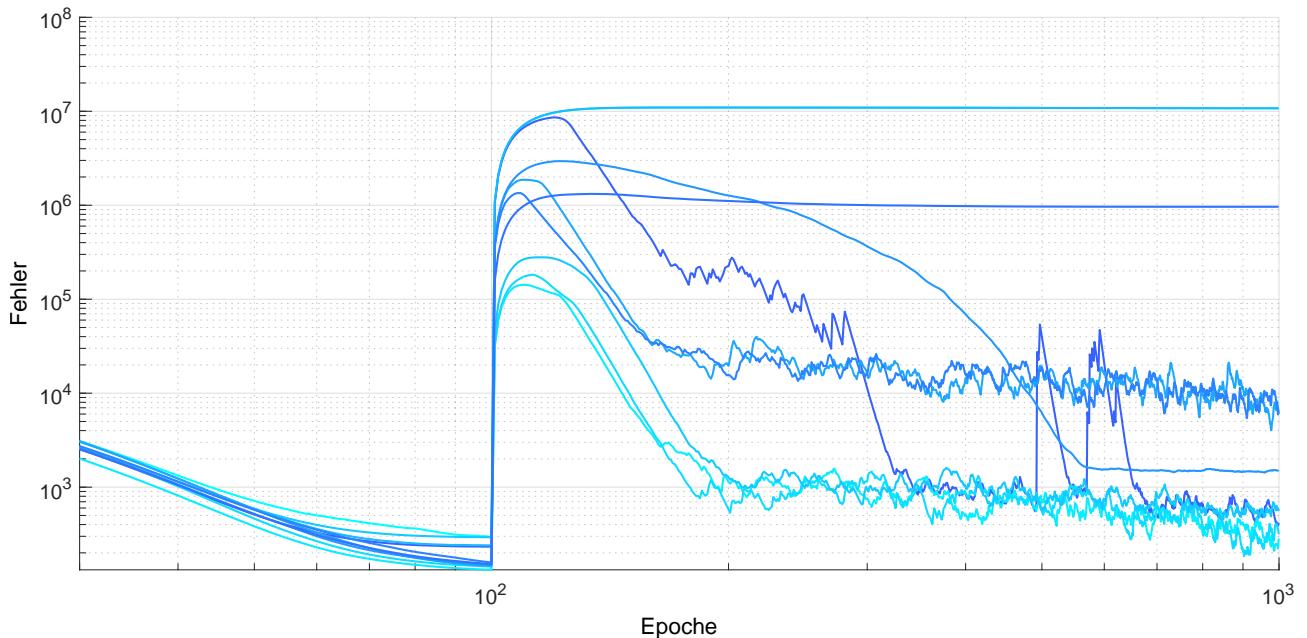


Abb. 35: Lernverlauf der Optimierung mit dem genetischen Algorithmus bei 10 Durchläufen

Bis die Bewertungsfunktion der Phasen- und Verstärkungsreserve aktiviert werden, konvergiert der GA sehr zuverlässig. Der Sprung bei Epoche 100 ist zu erwarten, da dort die Gewichtung der Phasenreserve in der Bewertungsfunktion aktiviert wird. Ein Grossteil der Durchläufe finden ein schlechtes Ergebnis, besonders die Phasen- und Verstärkungsreserve-Kriterien bringen die Optimierung oft aus dem Gleichgewicht.

Die Bewertungen der Phasen- und Verstärkungsreserve wird erst nach 100 Epochen aktiviert, da die Frequenzanalyse des im Code implementierten, zeitlichen und nichtlinearen Systems bei nicht optimalen Reglerparametern falsche und sprunghafte Ergebnisse liefert. Dies führt zu einem Chaos in der Bewertungsfunktion und kann die Optimierung stark behindern. Nach ca. 100 Epochen sind die Reglerparameter meistens so gut, dass die Frequenzanalyse auch im Zeitbereich verlässliche Ergebnisse liefert und deshalb in die Bewertungsfunktion aufgenommen werden kann.

## 8.5 | Optimierten Regler am realen System testen

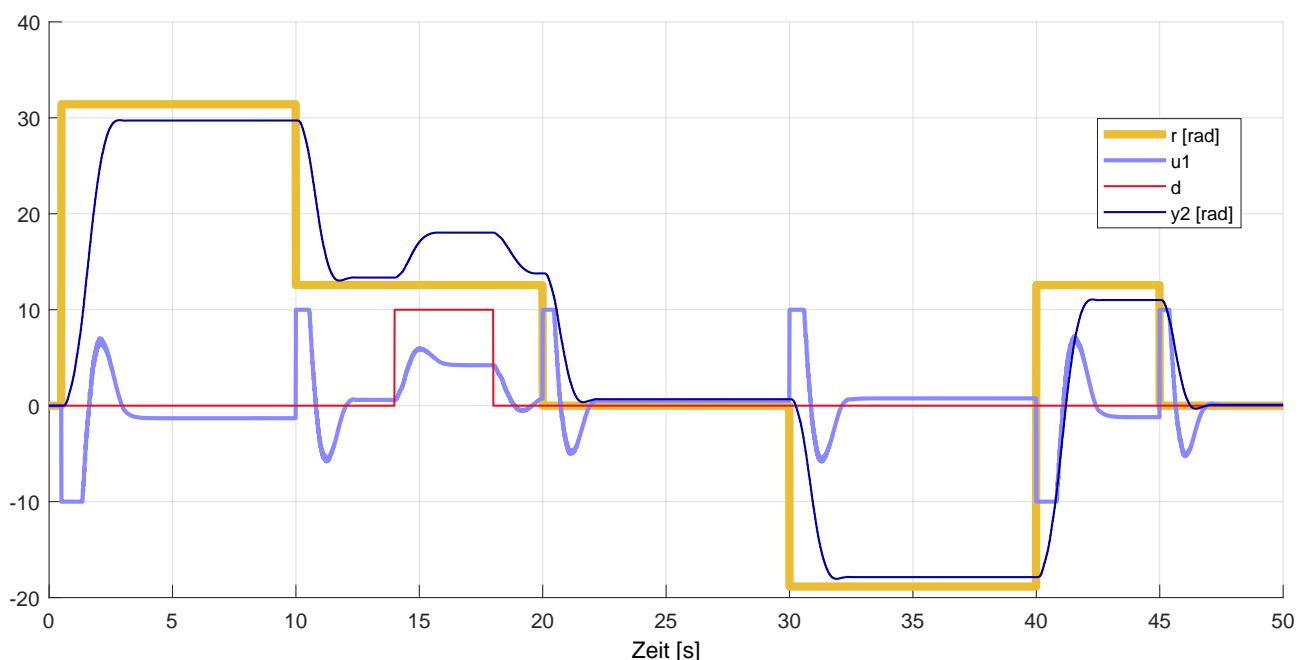


Abb. 36: Antwort auf Stimulation des optimierten Reglers am Motor mit Schwungmasse, mit dem durch GA optimierten PID-Regler

## 9 | DE Algorithmus am Motor mit Schwungmasse

Im Gegensatz zum [GA](#) wird beim [DE](#) kein abklingender [Hyperparameter](#) für die Mutationsstärke  $M(k)$  verwendet, da der Algorithmus von sich aus kleinere Mutationen durchführt, während die Population konvergiert.

Die Funktionsweise des DE Algorithmus ist im Kapitel [A.2.6 Differential Evolution Algorithmus](#) beschrieben.

### 9.1 | Optimierung durchführen

Die Software führt die Optimierung 10-mal durch, weil der [DE](#) aufgrund der zufälligen Startwerte nicht immer zum gleichen Ergebnis führt. Bei der Optimierung wurden die Einstellungen aus der [Tab. 12](#) verwendet.

Parameter	Beschreibung	Wert
$D$	Anzahl der einzelnen Optimierungsdurchläufe	10
$N$	Anzahl Individuen in der Population	30
$G$	Anzahl Generationen	1000
$C$	Crossover-Wahrscheinlichkeit	0.7
$M$	Mutationsstärke	0.5
Anti-Windup	Verwendete Anti-Windup Methode	Clamping
<b>Optimierungsziele</b>		
$a_1$	Absoluter Regelungsfehler	2.8
$a_2$	Stellwertänderungsrate	0.1
$a_3$	Positives Überschwingen	111
$a_4$	Verstärkungsreserve (6dB)	0
$a_5$	Phasenreserve (90°)	99 Ab Epoche 100
<b>Konstante Systemparameter</b>		
$u_{\text{satHigh}}$	PID Ausgangs Sättigung oberer Wert	10
$u_{\text{satLow}}$	PID Ausgangs Sättigung unterer Wert	-10
$I_{\text{SatHigh}}$	PID Integrator Sättigung oberer Wert	100000000
$I_{\text{SatLow}}$	PID Integrator Sättigung unterer Wert	-100000000
<b>PID-Startparameter</b>		
$K_p$	Proportionalitätsfaktor	$0 + \text{randG}(-1, 1) *$
$K_i$	Integrationsfaktor	$0 + \text{randG}(-1, 1) *$
$K_d$	Ableitungsfaktor	$0 + \text{randG}(-1, 1) *$
$K_n$	Filterkonstante	$0 + \text{randG}(-1, 1) *$

*Tab. 12: Hyper- & Tuningparameter für Optimierung mit dem Differential Evolution Algorithmus*

\* $\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

## 9.2 | Optimierung auswerten

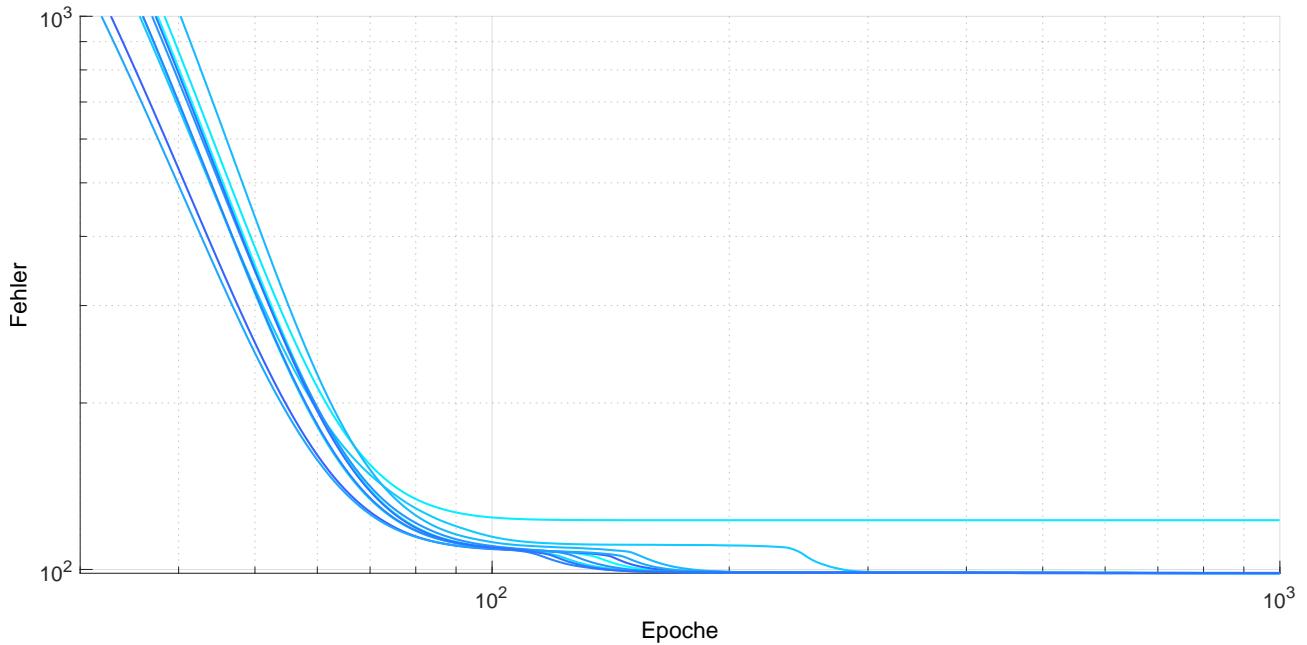


Abb. 37: Lernverlauf der Optimierung mit dem Differential Evolution Algorithmus bei 10 Durchläufen

Sehr auffällig ist, dass der **DE** Algorithmus im Vergleich zum **GA** schöner konvergiert und kein stochastisches Verhalten zeigt. Abgesehen von einem Durchlauf, finden alle anderen Durchläufe nahezu die gleiche Lösung.

## 9.3 | Optimierte Regler am realen System testen

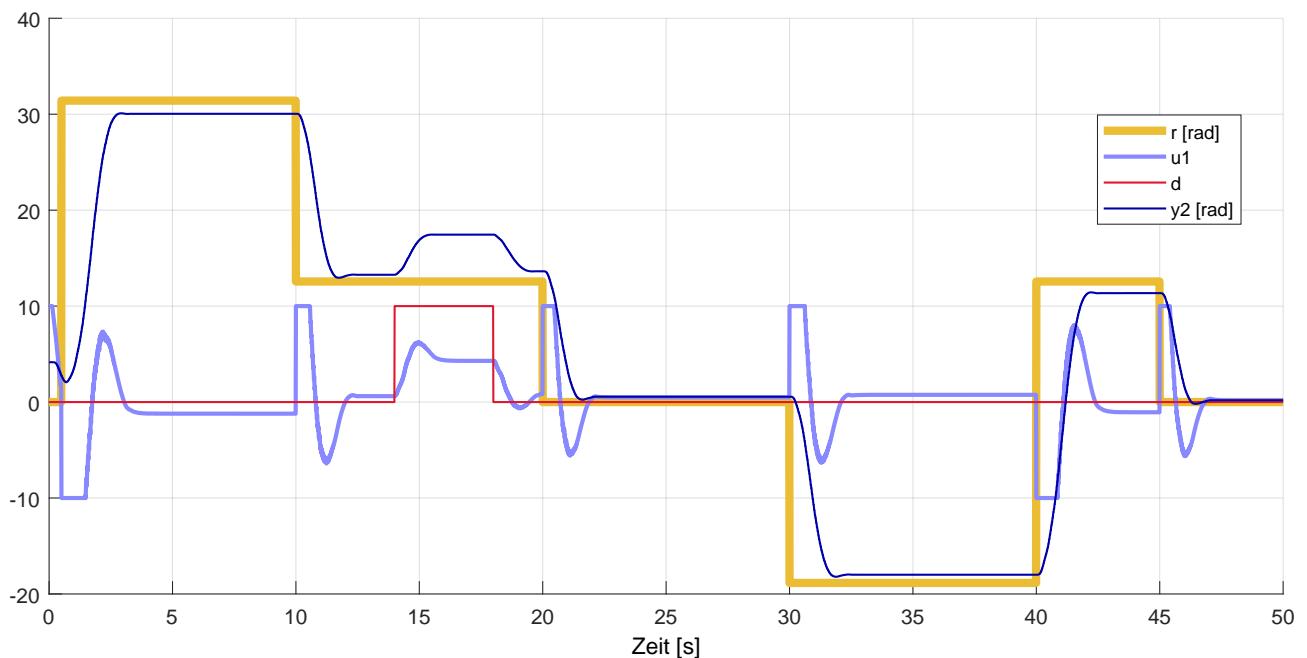


Abb. 38: Antwort auf Stimulation des optimierten Reglers am Motor mit Schwungmasse, mit dem durch **DE** optimierten PID-Regler

# 10 | Auswertung der Regleroptimierung am Motor mit Schwungmasse

## 10.1 | Synthetisierte Regler

Parameter	Wert
$K_p$	-0.411
$K_i$	-4.69e-8
$K_d$	-0.733
$K_n$	12.453
Konstante	Wert
$u_{satHigh}$	10
$u_{satLow}$	-10
$I_{SatHigh}$	100000000
$I_{SatLow}$	-100000000
Anti-Windup	Clamping

Tab. 13: Reglerparameter des mit *systune* optimierten Reglers

Parameter	Wert
$K_p$	-0.771
$K_i$	-5.8e-5
$K_d$	-0.605
$K_n$	4.065
Konstante	Wert
$u_{satHigh}$	10
$u_{satLow}$	-10
$I_{SatHigh}$	100000000
$I_{SatLow}$	-100000000
Anti-Windup	Clamping

Tab. 14: Reglerparameter des mit *GA* optimierten Reglers

Parameter	Wert
$K_p$	-0.88
$K_i$	-9.48e-5
$K_d$	-0.635
$K_n$	4.39
Konstante	Wert
$u_{satHigh}$	10
$u_{satLow}$	-10
$I_{SatHigh}$	100000000
$I_{SatLow}$	-100000000
Anti-Windup	Clamping

Tab. 15: Reglerparameter des mit *DE* optimierten Reglers

## 10.2 | Systemverhalten am realen Prozess

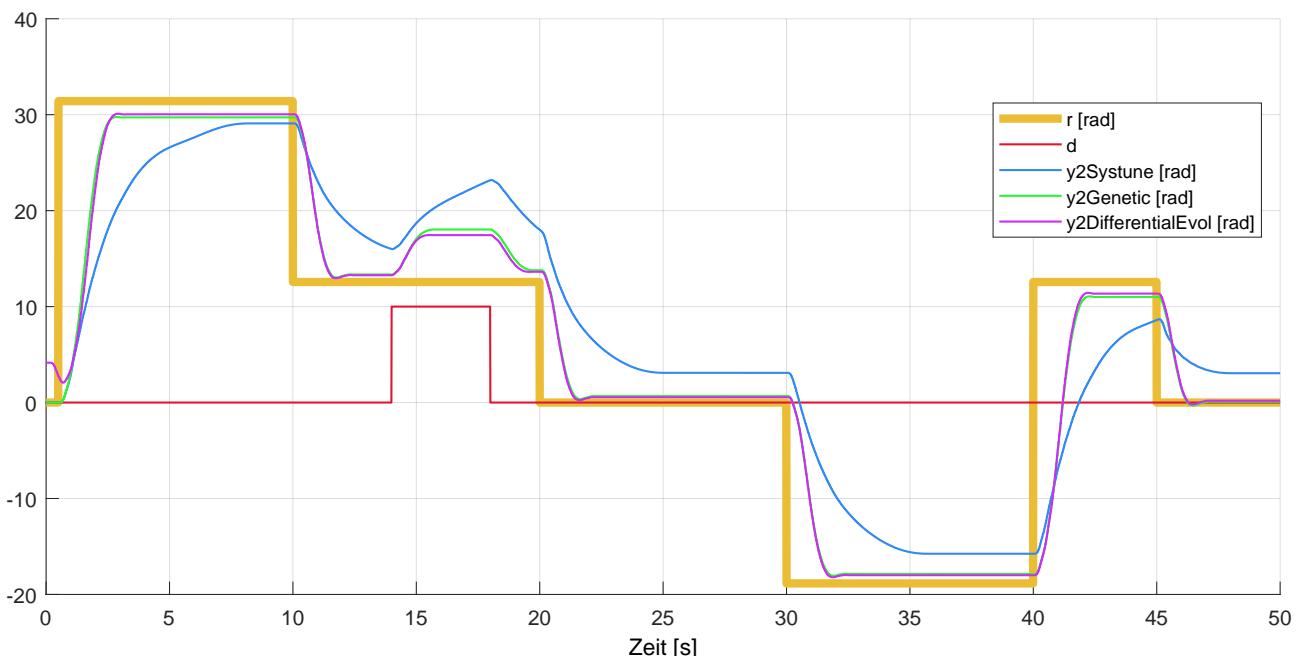


Abb. 39: Kombinierte Antwort auf Stimulation der optimierten Regler am Motor mit Schwungmasse

In Abb. 39 ist die Antwort auf die Stimulationssignale der drei Systeme dargestellt. Am realen System haben alle drei Regler mit den realen Bedingungen zu kämpfen. Die Reibungen im realen Prozess sorgen dafür, dass sich der Motor erst ab einem Wert von  $\pm 400$  (normierter Wert:  $\pm 1.33$ ) beginnt zu drehen und wenn das Motor Ansteuerungssignal auf 200 (normierter Wert:  $\pm 0.67$ ) absinkt bleibt der Motor stehen.

### Systune Regler

Der mit *systune* optimierte Regler regelt das System deutlich langsamer als die beiden anderen Regler.

### Genetisch optimierter Regler

Der mit *GA* optimierte Regler zeigt ein schnelleres Verhalten als der *systune* Regler.

Da diese Effekte im Modell nicht berücksichtigt wurden, haben die Optimierungsmethoden den I-Anteil des Reglers nicht dementsprechend hoch genug gewählt, um diese Effekte zu kompensieren. Deshalb ist bei allen drei Reglern ein stationärer Fehler zu beobachten.

### Differential Evolution optimierter Regler

Der mit *DE* optimierte Regler zeigt ein ziemlich ähnliches Verhalten wie der mit *GA* optimierte Regler. Er verhält sich minimal besser als der *GA* Regler.

## 10.3 | Lernverlauf

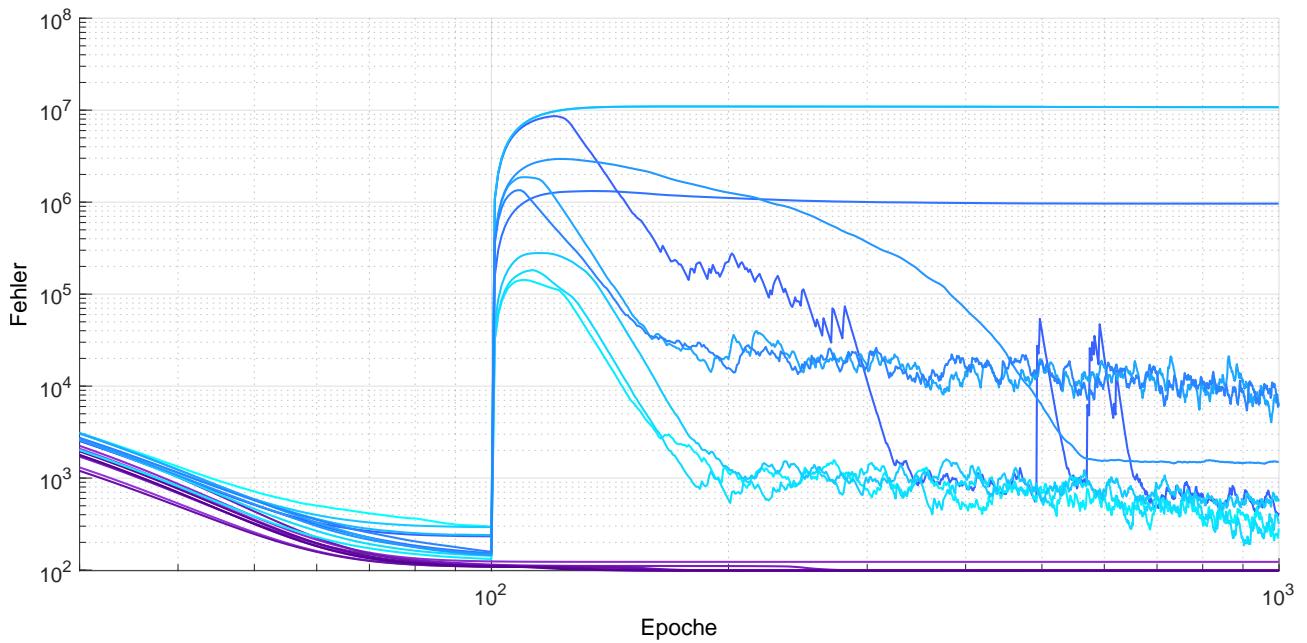


Abb. 40: Lernverlauf der kombinierten Optimierungsmethoden (GA + DE)

In Abb. 40 ist der Lernverlauf der beide Methoden **GA** und **DE** übereinandergelegt dargestellt. Die Anzahl der Lernepochen ist auf 1000 gesetzt im Vergleich zum Beispiel mit dem DC-Motor mit 5000 Epochen. Der Grund liegt darin, dass die Frequenzanalyse sehr viel Rechenzeit beansprucht und die Durchläufe deshalb sehr lange dauern können ohne sehr viel Mehrwert zu bieten.

Weil der GA zu Beginn der Optimierung die Verstärkungs- und Phasenreserve noch nicht richtig berechnen kann, wird diese Bewertung erst ab Epoche 100 berücksichtigt. Dadurch entsteht der Sprung im Lernverlauf des GA. Warum der DE diesen Sprung nicht hat, ist auf Seite 53: 'Dynamisches anpassen der Bewertungsfunktion' erklärt.

Der Lernverlauf ist leider für den *systune* Regler nicht verfügbar, damit dennoch ein Vergleich möglich ist wendet man die vom *systune* berechneten Reglerparameter in der Simulationssoftware an und misst den Fehler mit der gleichen Bewertungsfunktion wie bei den anderen beiden Methoden. *Systune* erreicht mit den in Tab. 13 angegebenen Parametern einen Fehler von 252.651.

*Systune* hat mit 252.651 einen Fehler erreicht, der grösser als die Fehler von **DE** und grösstenteils kleiner oder gleich den Fehlern von **GA** ist. Auch bei diesem System haben unterschiedliche Optimierungsziele einen wesentlichen Einfluss auf den erzielten Fehler von *systune*. Der Vergleich des erreichten Fehlers mit dem GA optimierten Regler ist aufgrund des starken Sprunges im Lernverlauf sowieso schwierig.

## 10.4 | Frequenzanalyse der optimierten Regler

Im Abb. 41 sind die Bode-Diagramme der drei optimierten Regler dargestellt. Diese Zeigen die Frequenzgänge der Regler ohne die Regelstrecke. Der mit *systune* optimierte Regler unterscheidet sich deutlich von den beiden anderen Reglern. *Systune* hat für tiefe Frequenzen eine starke Dämpfung realisiert und eine konstante Verstärkung von ca. 20dB für hohe Frequenzen.

Der *systune* Regler wird also Rauschen am Eingang mit einer Verstärkung von Faktor 10 am Ausgang weitergeben. Die beiden anderen Regler zeigen ebenfalls eine Rauschverstärkung, jedoch mit einem Faktor von ca. 3.2 deutlich geringer. Der Regler vom *GA* zeigt die kleinste Rauschverstärkung unter den drei Reglern.

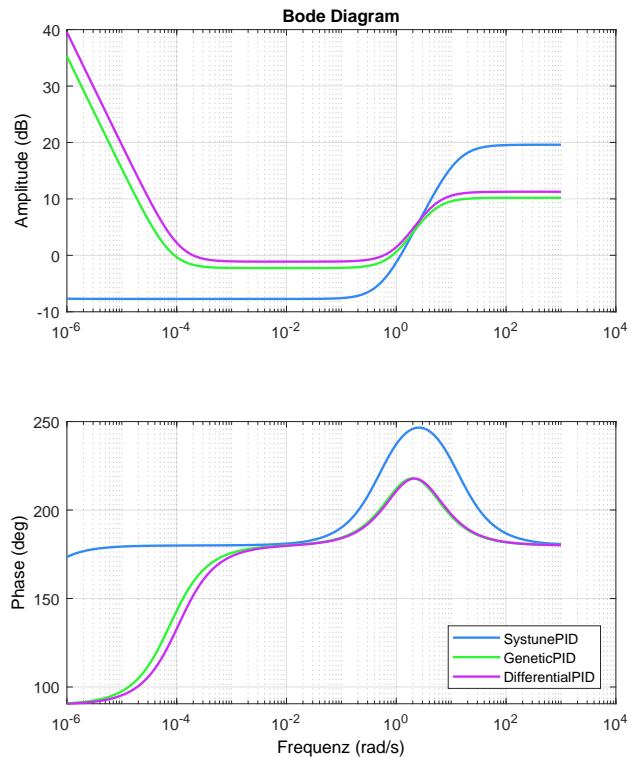


Abb. 41: Bode-Diagramm der optimierten Regler

## 10.5 | Frequenzanalyse der optimierten Systeme

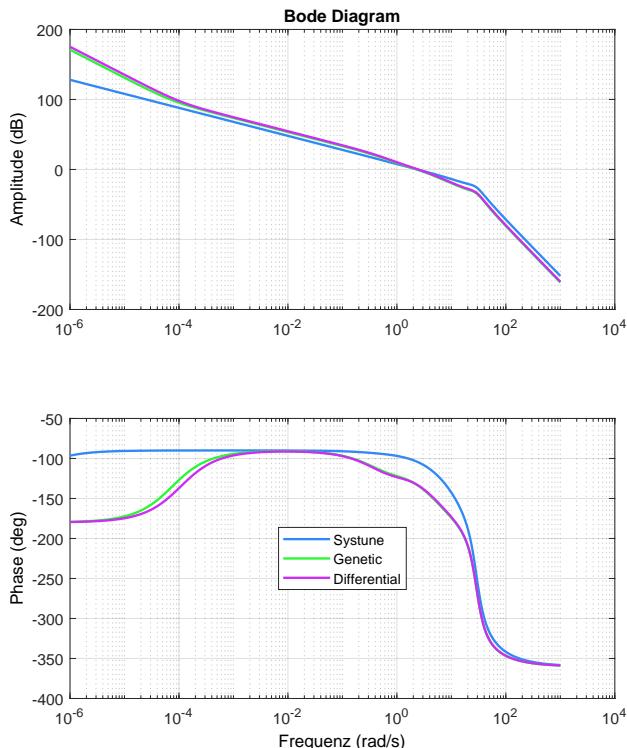


Abb. 42: Bode-Diagramm der optimierten Vorwärts Pfade der einzelnen Systeme

Das Bode-Diagramm in Abb. 42 und das Nyquist-Diagramm in Abb. 43 zeigen, jeweils die Frequenzgänge der optimierten Vorwärts Pfade der drei Systeme:

- $PID_{systune} \cdot MotorSystem$
- $PID_{genetic} \cdot MotorSystem$
- $PID_{differential} \cdot MotorSystem$

Im Nyquist-Diagramm fällt auf, dass der Regler von *systune* seinem System deutlich mehr Phasenreserve bietet. Die Verstärkungsreserve ist bei den Reglern von GA und DE etwas höher als bei *systune*, wobei aber auch *systune* mit 19.83dB eine sehr gute Verstärkungsreserve erreicht hat.

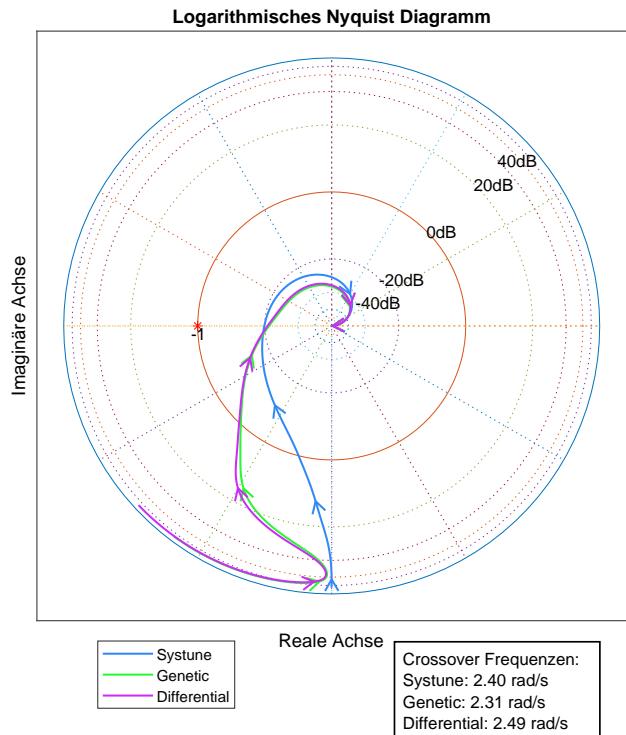


Abb. 43: Logarithmisches Nyquist-Diagramm der optimierten Vorwärts Pfade der einzelnen Systeme [32]

Phasenreserve	Wert
<i>Systune</i>	75.71 °
Genetisch	47.30 °
Differential Evolution	46.30 °
Verstärkungsreserve	
<i>Systune</i>	19.83 dB
Genetisch	21.92 dB
Differential Evolution	21.25 dB

Tab. 16: Reserven der optimierten Systeme

## 10.6 | Stabilitätsanalyse der optimierten Systeme

### 10.6.1 | Sensitivität

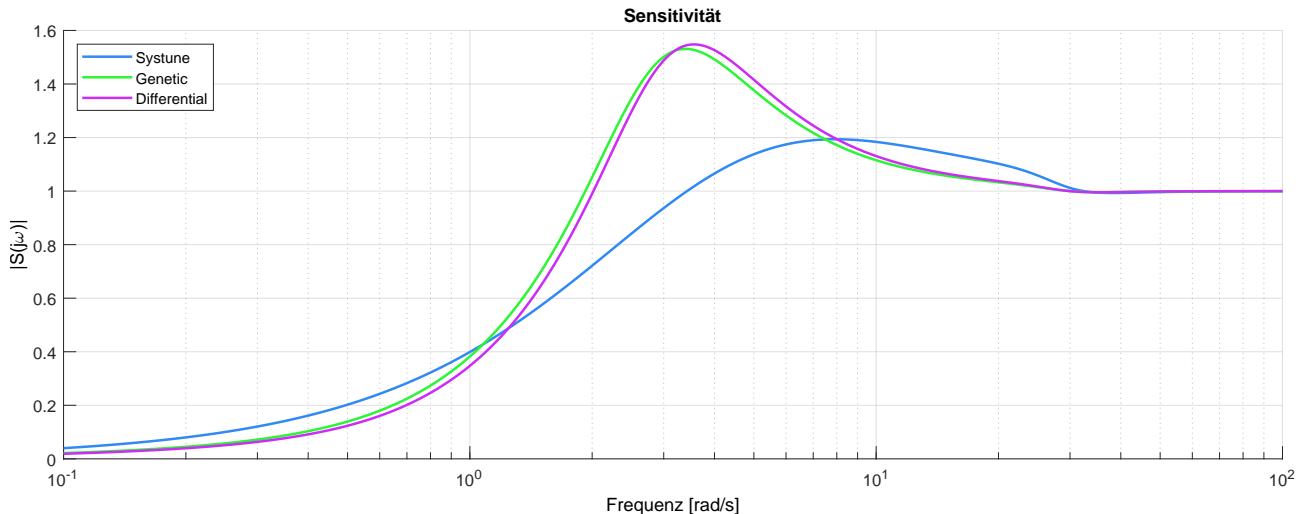


Abb. 44: Sensitivität der optimierten Systeme

Das Sensitivitäts-Diagramm in Abb. 44 zeigt, für die mit GA und DE optimierten Systeme einen ähnlichen Verlauf. Beide haben eine stärkere Überhöhung im Vergleich zum mit *systune* optimierten System. Insgesamt sind aber alle drei robuster gegenüber Totzeiten im Vergleich zum System mit dem DC-Motor.

### 10.6.2 | Phasen- & Verstärkungsreserve Diagramme der optimierten Systeme

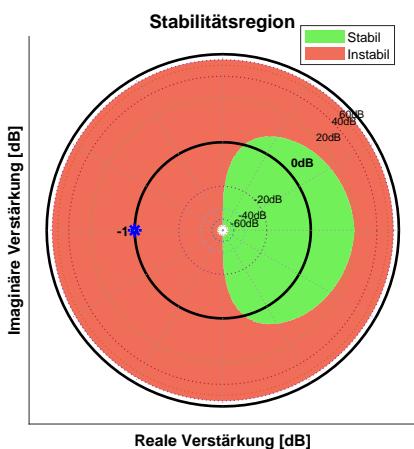


Abb. 45: Phasen- & Verstärkungsreserve des mit *systune* optimierten Systems

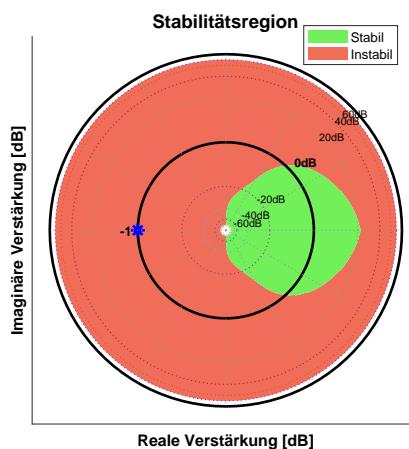


Abb. 46: Phasen- & Verstärkungsreserve des mit *Genetic* optimierten Systems

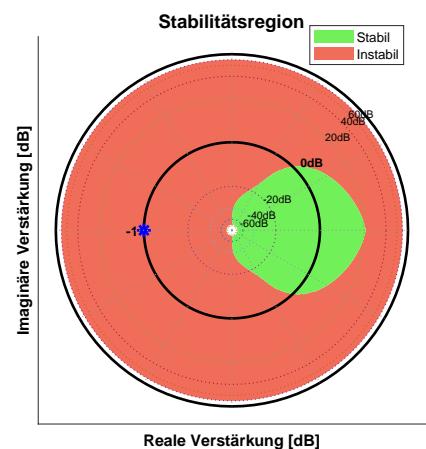


Abb. 47: Phasen- & Verstärkungsreserve des mit *Differential Evolution* optimierten Systems

Die drei Diagramme in Abb. 45, Abb. 46 und Abb. 47 zeigen die detaillierten Stabilitätsreserven als Kombination von Verstärkung und konstanter Phasenverschiebung. Wie diese Diagramme zu lesen sind, ist in folgendem Kapitel erklärt:

#### A.2.7 Phasen- & Verstärkungsreserve Diagramm

Die Stabile Region ist beim Regler von *systune* am besten ausgebildet. Dieser bietet viel Spielraum für Phasenänderungen auch mit zusätzlicher Verstärkung. Die Regler von *Genetic* und *Differential Evolution* bieten eine ähnliche Stabile Region, welche nicht ganz so gross ist wie die von *systune*.

### 10.6.3 | Totzeit- & Verstärkungsreserve Diagramme der optimierten Systeme

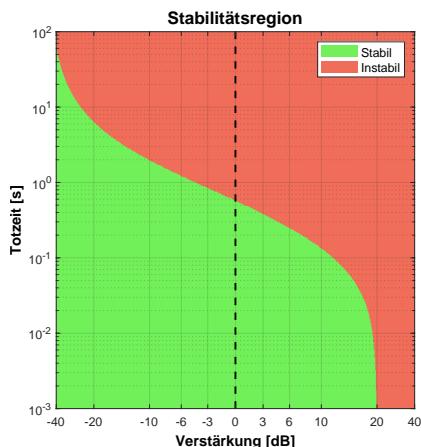


Abb. 48: Totzeit- & Verstärkungsreserve des mit  
systeme optimierten Systems

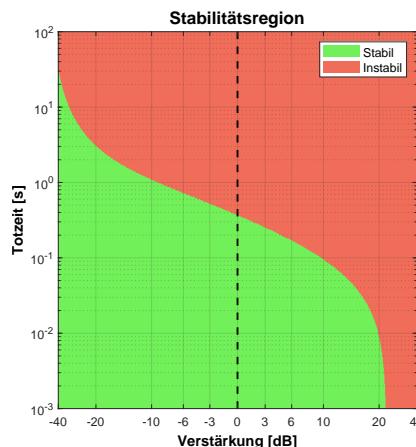


Abb. 49: Totzeit- & Verstärkungsreserve des mit  
Genetic optimierten Systems

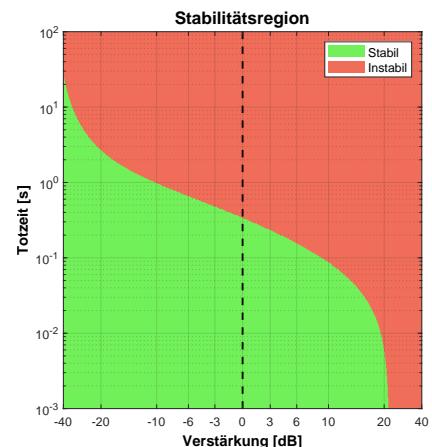


Abb. 50: Totzeit- & Verstärkungsreserve des mit  
Differential Evolution optimierten Systems

Die drei Diagramme in Abb. 48, Abb. 49 und Abb. 50 zeigen die detaillierten Stabilitätsreserven aus der Kombination von Verstärkung und Totzeit. Wie diese Diagramme zu lesen sind, ist in folgendem Kapitel erklärt:

#### A.2.8 Totzeit- & Verstärkungsreserve Diagramm

Auch wenn die Verstärkungsreserve nicht beliebig gross ist, verhalten sich alle drei Systeme relativ robust gegenüber Kombinationen von Totzeit und Verstärkungsänderungen. Bei einer Verstärkung von 1, bleiben alle drei Systeme noch mit einer Totzeit von ca. 300ms stabil.

# 11 | Schlussteil

## 11.1 | Objektives Fazit

### 11.1.1 | Systune

- + Mit *systune* lassen sich Systeme schnell und mit wenig Code optimieren.
- + Die Tuning-Goals lassen sich einfach kombinieren.
- + *Systune* bietet viele vordefinierte Tuning-Goals für unterschiedliche Kriterien an.
- + *Systune* ist in MATLAB integriert, wodurch der Grossteil der Komplexität nicht sichtbar ist.
- + *Systune* übernimmt für die Optimierung die benötigten Simulationen im Zeit- und Frequenzbereich automatisch.
- + Erfordert keine Programmierkenntnisse um eine Optimierungsumgebung zu implementieren.
- Die Parametrisierung der Tuning-Goals ist nicht immer einfach. Teilweise mangelt es an einer guten Dokumentation.
- Die Tuning-Goals lassen sich nur in zwei Kategorien einteilen: hart und weich. Es gibt keine Möglichkeit die Priorität der Ziele zu gewichten.
- Zu wenige vordefinierte Tuning-Goals für spezifische Kriterien.
- Da nur lineare Systeme optimiert werden können, lassen sich keine Sättigungen oder Anti-Windup-Parameter optimieren. Anti-Windup-Parameter werden von MATLABs tunablePID [30] auch nicht angeboten.
- Die Diagramme, welche die Optimierungsergebnisse der Tuning-Goals zeigen, sind nicht immer verständlich und teilweise nicht dokumentiert. Höhere Fachkenntnisse sind deshalb notwendig um diese zu verstehen.

Die Optimierungsziele von MATLAB sind nicht ganz zufriedenstellend. Es gibt zwar vordefinierte Ziele, die für einige Kriterien verwendet werden können, aber es bleibt noch viel Potenzial für weitere Ziele. Z.B Ziele für Rauschunterdrückung zwischen zwei Punkten im System fehlen. Die Parametrierung der Optimierungsziele ist teilweise etwas komplex, und die Beschreibung in der Dokumentation ist nicht immer sehr hilfreich.

Um ein zufriedenstellendes Systemverhalten zu erreichen, müssen die Optimierungsziele mehrfach angepasst, neue hinzugefügt oder entfernt werden. Teilweise führt das hinzufügen eines neuen Ziels dazu, dass ein schlechteres Ergebnis dadurch erzielt wird, auch wenn die Ziele in einem realistischen Verhältnis zueinanderstehen.

### 11.1.2 | Differential Evolution Algorithmus

- + Auch durch das simple Optimierungsziel: Minimierung des aufsummierten, absoluten Fehlers, lässt sich schon ein sehr guter Regler finden.
- + Sehr flexibel, es können beliebige Systeme optimiert werden, auch nichtlineare (nur zeitinvariante).
- + Die Optimierungsziele sind frei definierbar.
- + Der Algorithmus ist relativ einfach zu verstehen und zu implementieren.
- + Konvergiert sehr schnell.
- Findet nicht immer eine gute Lösung.
- Für die Optimierung muss mehr Code geschrieben werden oder eine Toolbox gefunden werden, die einem viel Arbeit abnimmt.
- Die Definition der [Fitnessfunktion](#) ist nicht immer einfach, da alle Optimierungsziele in einer einzigen Formel sind.
- Die Gewichtung der unterschiedlichen Kriterien in der Fitnessfunktion ist nicht immer einfach zu bestimmen und erfordert mehrere Anläufe.
- Erfordert höhere Programmierkenntnisse um die Optimierungsumgebung zu implementieren.

[DE](#) hat sehr ähnliche Vor- und Nachteile wie der [GA](#). Der Algorithmus konvergiert jedoch schneller und hat einen schöneren Konvergenzverlauf. DE garantiert, dass die [Population](#) nicht schlechter wird mit jeder Generation, unterliegt aber auch oft dem Problem zu früh in einem lokalen Minimum stecken zu bleiben.

DE ist in der Lage aus lokalen Minima herauszukommen, wenn die Streuung der Population nicht zu klein wird. Für den DE müssen die Optimierungsumgebung und die Datenerhebung selbst implementiert werden, wie beim GA.

### 11.1.3 | Genetischer Algorithmus

- + Auch durch das simple Optimierungsziel: Minimierung des aufsummierten, absoluten Fehlers, lässt sich schon ein sehr guter Regler finden.
- + Sehr flexibel, es können beliebige Systeme optimiert werden, auch nichtlineare (nur zeitinvariante).
- + Die Optimierungsziele sind frei definierbar.
- + Der Algorithmus ist ziemlich einfach zu verstehen und zu implementieren.
- Findet nicht immer eine gute Lösung.
- Für die Optimierung muss mehr Code geschrieben werden oder eine Toolbox gefunden werden, die einem viel neue Codierung abnimmt.
- Die Definition der **Fitnessfunktion** ist nicht immer einfach, da alle Optimierungsziele in einer einzigen Formel sind.
- Die Gewichtung der unterschiedlichen Kriterien in der Fitnessfunktion, ist nicht immer einfach zu bestimmen und erfordert mehrere Anläufe.
- Erfordert höhere Programmierkenntnisse um die Optimierungsumgebung zu implementieren.

Anders als mit *systune* muss die Optimierungsumgebung selbst implementiert werden. In dieser Arbeit wurde auch der GA selbst implementiert, obwohl MATLAB eine fertige Implementierung anbietet [18]. Die Optimierungsziele müssen programmiert werden, was für sehr spezifische Anforderungen aufwendig sein kann.

Eine Schwierigkeit bei der Verwendung des GA liegt einerseits in der Definition der Fitnessfunktion, welche bei **Minimierungsproblemen** eine spezifische Umwandlung erfordert, nur ein Vorzeichenwechsel funktioniert nicht. Andererseits müssen die Optimierungsziele als Formel ausgedrückt werden die in einem einzigen Wert resultieren. Die Problematik besteht darin, die richtigen Formeln für die unterschiedlichen Kriterien zu erstellen und diese auch noch in ein gutes Verhältnis zueinander zu setzen. (Gewichtung der Bewertungskriterien) Deshalb benötigt auch diese Methode mehrere Anläufe beim Erstellen der Fitnessfunktion bis ein zufriedenstellendes Systemverhalten erzielt wird.

Der GA hat den Nachteil, dass sich die zu optimierenden Parameter nur schwer einschränken lassen. In spezifischen Fällen ist im Vorhinein bekannt, dass gewisse Bereiche von Parametern nicht möglich sind. Diese Bereiche lassen sich im GA nur schwer ausschliessen, da der Algorithmus zufällig neue Individuen generiert. Eine Möglichkeit wäre, die Fitness dieser Individuen sehr schlecht zu bewerten.

Ein weiterer Nachteil des GA besteht darin, dass der Algorithmus am besten auf gleich normierte Parameter anwendbar ist. Befinden sich bestimmte Parameter in einem viel grösseren Werte-Bereich als andere, kann es sein dass die Mutations-Stärke für einige Parameter zu klein ist und für andere zu gross. Bei gleich normierten Parametern entfällt dieses Problem. Eine Lösung hierfür wäre, entweder die Parameter normiert dem Algorithmus zu übergeben, oder den Algorithmus so anzupassen, dass die Mutations-Stärke für jeden Parameter individuell skaliert ist. Es gibt verschiedenste Strategien für die individuelle Mutation eines Gens. (Stichwort: *self-adaptive mutation*)

### 11.1.4 | Welche ist die beste Methode?

Welche Methode besser geeignet ist, lässt sich nicht allgemein sagen und hängt stark vom jeweiligen Anwendungsfall ab. Die Tests der drei Methoden an nur zwei Prozessen sind nicht ausreichend um eine allgemeingültige Aussage treffen zu können. Für eine fundierte Aussage wären deutlich mehr Tests an sehr unterschiedlichen Prozessen notwendig.

Mit *systune* lässt sich schneller ein Regler entwerfen und testen als mit den anderen beiden Methoden. Allerdings wird es schwierig mit *systune* den Regler auf allfällig folgende weitere Anforderungen zu verfeinern, da sich solche Anforderungen nicht, oder nur ungenügend, in den vorhandenen Tuning-Goals abbilden lassen.

Mit stetigem verkleinern der Mutations-Stärke kann die Geschwindigkeit der **Konvergenz** erhöht werden, wodurch der GA vergleichbar schnell wie DE konvergiert. Sein stochastisches Verhalten hilft dabei, nicht zu schnell in lokalen Minima stecken zu bleiben.

Damit die Optimierung überhaupt durchgeführt werden kann, müssen für die Fitnessfunktion die benötigten Daten ermittelt werden, diese können einerseits aus einer Simulation des Systems im Zeitbereich stammen, andererseits können auch frequenzabhängige Messungen durchgeführt werden. Diese Datenerhebung muss allenfalls selbst implementiert werden und kann je nach Anwendungsfall und Anforderung an das Reglerverhalten beliebig komplex werden. Dafür sind weitere Programmierkenntnisse notwendig im Bereich der numerischen Simulation von Systemen und der Frequenzanalyse notwendig. Natürlich lässt sich die Optimierung mit MATLAB und Simulink durchführen wodurch die Probleme der manuellen Implementierung einer Simulation und Frequenzanalyse entfallen, aber auch dann kann noch viel Aufwand bei der Implementierung der Optimierungsumgebung entstehen. Das sind Aspekte welche nichts mit dem **GA** an sich zu tun haben, aber trotzdem notwendig sind um den Algorithmus anwenden zu können.

Für GA und DE ist es einfacher weitere Anforderungen in der Fitnessfunktion zu berücksichtigen. Ein Kriterium kann frei definiert und gewichtet werden und bietet somit von einfach bis beliebig komplex viele Möglichkeiten. Die Flexibilität und die Möglichkeit beliebige nichtlineare Systeme zu optimieren, bietet außerdem den grossen Vorteil, dass sich der Regler besser auf reale Bedingungen abstimmen lässt.

### 11.1.5 | Iteratives Tuning der Methoden

Alle drei Methoden erfordern ein iteratives Vorgehen um ein gutes Ergebnis zu erhalten.

#### Systune

Bei *systune* müssen die Tuning-Goals angepasst werden, bis ein zufriedenstellendes Ergebnis erzielt wird. Verschiedene Kombinationen von Zielen und deren Parameter müssen ausprobiert werden, denn nicht immer führt das Hinzufügen eines neuen Ziels zu einem besseren Ergebnis.

#### Genetischer Algorithmus und Differential Evolution

Bei *GA* und *DE* muss die Fitnessfunktion angepasst werden, bis ein zufriedenstellendes Ergebnis erzielt wird. Außerdem müssen für eine Simulation im Zeitbereich die Stimulationssignale so gewählt werden, dass die gewünschten Kriterien auch tatsächlich in die Fitness der Population mit einfließen. Die Hyperparameter des Algorithmus müssen teilweise angepasst werden. In der Regel genügen die Standardeinstellungen für einen Grossteil der Anwendungen aus aber in spezifischen Fällen kann es notwendig sein, diese anzupassen um bessere Ergebnisse zu erzielen.

#### 11.1.5.1 | Bestimmung der Hyperparameter bei *GA* und *DE*

Für die Bestimmung der Hyperparameter wurde viel ausprobiert. Gewisse Parameter haben grösseren Einfluss auf den Optimierungsprozess und andere weniger. Das finden der optimalen Hyperparameter ist in sich selbst ein Optimierungsproblem.

##### Populationsgrösse

Die gewählte Populationsgrösse ist abhängig von der Komplexität des Problems, also wie viele Parameter optimiert werden müssen. Je komplexer das Problem, desto grösser sollte die Population gewählt werden, um den Suchraum besser abdecken zu können. Zu gross sollte die Population aber auch nicht gewählt werden, da dies die Rechenzeit unnötig in die Höhe treibt und ab einer bestimmten Grösse kaum noch Vorteile bringt.

##### Anzahl Generationen

Dieser Parameter definiert, wie oft die Population aktualisiert wird. Je komplexer das Problem, desto mehr Generationen werden benötigt. Dieser Parameter wird als Abbruchbedingung der Optimierung verwendet und kann je nach Implementierung auch weggelassen werden, um stattdessen ein anderes Abbruchkriterium zu verwenden, wie z.B. ein Unterschreiten des Fehlers unter einen bestimmten Wert.

##### Mutations-Wahrscheinlichkeit (nur *GA*)

Die Mutations-Wahrscheinlichkeit definiert, die Wahrscheinlichkeit, dass ein Genom eines Individuums mutiert wird. Ein Genom ist eine einzelne Variable des Individuums, also z.B. ein Reglerparameter. Je höher die Mutations-Wahrscheinlichkeit, desto grösser die Diversität in der Population. Eine zu hohe Mutations-Wahrscheinlichkeit kann jedoch dazu führen, dass die Optimierung nicht konvergiert. Dies kann mit einer radioaktiven Umgebung verglichen werden, in der sich die Individuen ständig verändern und somit langfristig keine auf das Problem abgestimmte Population entstehen kann. Je mehr Parameter ein Individuum hat, desto kleiner sollte die Mutations-Wahrscheinlichkeit gewählt werden, um nicht zu viele Mutationen pro Generation zu erzeugen. Ein zu kleiner Wert schadet nur der Zeit, die für die Konvergenz benötigt wird. Ein zu grosser Wert kann jedoch verhindern, dass eine gute Lösung gefunden wird.

Der implementierte PID-Regler hat viele Einstellmöglichkeiten um unterschiedliche Kombinationen zu testen. Schlussendlich wurden jedoch nur die einfachsten Einstellungen verwendet. *Systune* kann keinen Anti-Windup optimieren, weshalb sich die verwendete Anti-Windup-Strategie auf *Clamping* [A.2.3.2](#) beschränkt. Diese Methode benötigt keinen zusätzlichen Parameter und ist im Simulink PID-Regler-Block bereits integriert.

Bei *GA* und *DE* wurde der PID-Regler direkt mit dieser Anti-Windup-Strategie optimiert. Auch die Anti-Windup-Strategie *Back-Calculation* [A.2.3.2](#), bei der ein zusätzlicher Parameter optimiert werden muss, wurde ausprobiert. Dies hat auch zu guten Ergebnissen geführt, jedoch wurde schlussendlich auf die einfachere *Clamping* Methode zurückgegriffen, um den Vergleich mit *systune* fairer zu gestalten.

Auch die Filterkonstante für den D-Anteil ist optional und lässt sich im Simulink PID-Regler-Block aktivieren oder deaktivieren. Das gleiche gilt für die Implementierung in der C++ Simulation. Da *systune* aber die Filterkonstante immer mit optimiert, wurde dies auch bei *GA* und *DE* so gehandhabt.

##### Mutations-Stärke

Die Mutations-Stärke definiert, wie stark eine Mutation ausfällt. Je grösser die Mutations-Stärke, desto grösser werden die Veränderungen an den zu mutierenden Genen. Eine grosse Mutations-Stärke kann dabei helfen, aus lokalen Minima herauszukommen, jedoch kann eine zu grosse Mutations-Stärke auch dazu führen, dass die Optimierung nicht konvergiert. Im Fall vom *GA* wurde die Mutations-Stärke im Verlauf der Optimierung verkleinert um die Vorteile von grossen und kleinen Mutationen zu kombinieren. Für *DE* muss die Mutations-Stärke nicht angepasst werden, da diese automatisch durch die abnehmende Distanz zweier Individuen im Verlauf der Optimierung abnimmt.

##### Konstante Systemparameter

Systemparameter der Regelstrecke welche nicht optimiert wurden, sind von Hand festgelegt worden. Die Sättigung des I-Anteils des PID-Reglers ist ein Beispiel dafür. In beiden getesteten Systemen wird ein gutes Ergebnis erzielt, sowohl mit einer Sättigung als auch ohne. Deshalb ist eine genaue Bestimmung dieser Parameter nicht kritisch.

##### Startwerte gesuchten Parameter

Die Startwerte der zu optimierenden Parameter können einen grossen Einfluss auf die Konvergenzgeschwindigkeit haben, bzw. generell darauf, ob überhaupt eine gute Lösung gefunden wird.

In dieser Arbeit wurde eine zufällige Initialisierung der Parameter in der Umgebung um den Nullpunkt verwendet. Die Wahl der Grösse der Startumgebung ist abhängig von der ungefähren Größenordnung der zu optimierenden Parameter. Es lohnt sich, mit der Grösse des Startbereichs zu experimentieren.

Bei bestimmten Problem stehen eventuell schon gute Startwerte zur Verfügung oder es gibt bekannte Bereiche, in denen sich gute Lösungen befinden. In solchen Fällen sollten diese Informationen genutzt werden, um die Startwerte entsprechend zu wählen.

## 11.2 | Persönliches Fazit

Eigentlich war geplant den **GA** und den **DE** Algorithmus in Kombination mit der Ziegler-Nichols Methode zu verwenden. Die gewählten Prozesse lassen dies jedoch nicht zu, da sie entweder instabil sind oder nur sehr kleine Zeitkonstanten haben.

Die Optimierung, ohne die durch Ziegler-Nichols bestimmten Startwerte, hat jedoch an beiden Prozessen gut funktioniert und deshalb bietet Ziegler-Nichols keinen Mehrwert für diese Optimierungsmethoden. Ausserdem wäre Ziegler-Nichols nur für PID-Regler anwendbar, während die Optimierungsmethoden beliebige Reglerstrukturen optimieren können.

Es konnten leider nicht alle geplanten Aufgaben umgesetzt werden. Eine Demo Applikation zur Visualisierung der Optimierungsprozesse ist nicht genug überarbeitet um sie mit dieser Arbeit abgeben zu können. Mein Ziel ist es jedoch, auch nach der Abgabe der Arbeit die Applikation noch aufzuräumen und auf GitHub zu veröffentlichen [2].

### 11.2.2 | Genetischer Algorithmus

Der genetische Algorithmus ist nicht allzu kompliziert zu verstehen und lässt sich relativ einfach implementieren. Es gibt sogar eine fertige Implementierung in MATLAB [18], aber diese habe ich nicht angeschaut. Die eigene Implementierung hat es mir ermöglicht, einen kompletten Einblick in den Optimierungsprozess zu bekommen.

### 11.2.4 | Welches ist die beste Methode?

Ich persönlich bevorzuge den **GA** und den **DE**-Algorithmus, da sie bessere Möglichkeiten für spezifische Optimierungsziele bieten.

Ich glaube durchaus, dass *systune* in der Lage wäre mit den geeigneten Optimierungszielen ein besseres Ergebnis zu liefern als das was ich es in der Arbeit erzielt habe, aber dennoch sehe ich den grösseren Vorteil darin ein nichtlineares System optimieren zu können.

Sättigungen sind da ein gutes Beispiel, welche bei realen Systemen immer vorhanden sind und das Systemverhalten stark beeinflussen können. Beispielsweise lassen sich stabile Regler für Systeme mit Sättigung umsetzen welche im linearisierten Fall ohne Sättigung instabil wären. Wird der Regler jedoch auf das linearisierte System optimiert, so ist das System zwar stabil, verschenkt aber im realen Prozess mögliche Leistungspotentiale.

### 11.2.1 | Systune

Die Optimierung mit *systune* hat aus meiner Sicht nicht so gut funktioniert wie ich das erhofft hatte. Ich glaube, dass mit tieferem Wissen über *systune* und die Tuning-Goals, ein besseres Ergebnis erzielt werden könnte. Eine bessere Dokumentation von Seite Mathworks wäre sehr hilfreich gewesen.

Teilweise habe ich mittels Testen von neuen Tuning-Goals festgestellt, dass die Optimierung völlig versagt. Ohne Rückmeldung dazu, was genau an dem neu hinzugefügten Tuning-Goal falsch ist, ist es schwierig zu verstehen, warum die Optimierung damit nicht funktioniert. Manchmal hatten auch bestimmte Tuning-Goals Einflüsse auf Dynamiken des Systems, welche ich nicht erwartet hätte oder nicht in diesem Ausmass. Viele Tuning-Goals musste ich deshalb wieder aus der Optimierung entfernen, weil ich es nicht geschafft habe diese korrekt zu konfigurieren.

### 11.2.3 | Differential Evolution Algorithmus

Der Differential Evolution Algorithmus ist ein wenig schwieriger zu verstehen als der genetische Algorithmus, aber die Implementierung ist auch hier relativ einfach. Im Gegensatz zum genetischen Algorithmus konnte ich **DE** vor der Arbeit noch nicht aber für zukünftige Projekte, welche einen Optimierungsalgorithmus benötigen, werde ich **DE** definitiv ausprobieren.

Sättigungen im System in Kombination mit Integratoren im Regler führen zum Windup-Problem, welches leider mit *systune* nicht mitberücksichtigt werden kann. Mit dem genetischen- oder dem Differential Evolution Algorithmus können unterschiedliche Anti-Windup-Methoden sowie deren Parameter in die Optimierung einzbezogen werden, was auch dafür sorgt, dass die Nebeneffekte der Anti-Windup-Methoden berücksichtigt werden.

## 11.2.5 | Zu spät erkannte Fehler

### Falsche Normierung der Fehler-Minimierungs-Funktion

Während der Überarbeitung der Arbeit ist mir aufgefallen, dass die [Formel 4.2.2 Optimierungsziele definieren](#) fälschlicherweise wie folgt definiert wurde:

$$g(I_i) = \frac{1}{t_{\text{end}} - t_{\text{start}}} \sum_{k=0}^{K-1} \left( \sum_{o=0}^{O-1} (a_o \cdot g_o(k)) \right)$$

Der Fehler liegt in dem Skalierungsfaktor  $\frac{1}{t_{\text{end}} - t_{\text{start}}}$ , dieser müsste eigentlich  $\frac{1}{K}$  sein, also die Anzahl der Zeitschritte. Da für jeden Simulationsschritt die jeweiligen Teilsummen addiert werden, muss der [Fehler](#) durch die Anzahl der Zeitschritte normiert werden und nicht durch die Simulationszeit.

Das gleiche Problem tritt beim zweiten getesteten Prozess auf.

[Formel 8.2.2:](#)

$$g(I_i) = \frac{1}{t_{\text{end}} - t_{\text{start}}} \sum_{k=0}^{K-1} \left( \sum_{o=0}^{O-3} (a_o \cdot g_o(k)) \right) + a_3 \cdot g_3(I_i) + a_4 \cdot g_4(I_i)$$

Beide Fehler sind in der Dokumentation korrigiert und haben auch keinen Einfluss auf die Resultate der Arbeit. Die einzige Inkonsistenz die dadurch entsteht, ist dass die gemessenen Lernkurven falsch skaliert sind.

Ich empfehle eine Normierung der Bewertungsfunktion, aber es ist theoretisch nicht zwingend notwendig, solange die Anzahl der Zeitschritte in der Simulation pro Trainings-Iteration für jedes Individuum gleich bleibt. Notwendig ist eine Normierung dann, wenn die Individuen mit unterschiedlichen Simulationszeiten oder Abtastzeiten bewertet werden, da in diesem Fall z.B. für Individuum A, 10-mal die Regelabweichung integriert wird und für Individuum B nur 5 mal. Das würde dafür sorgen, dass Individuum A einen grösseren Fehlerwert erhält als Individuum B und somit beim Selektionsprozess benachteiligt wird.

### Dynamisches anpassen der Bewertungsfunktion

Im Beispiel des Motors mit Schwungmasse wird die Teilbewertung  $g_4(I_i)$  erst ab Epoche 100 in die Gesamtbewertung einbezogen. Dies hat zu starke Schwankungen in der Lernkurve geführt, wie in [Abb. 35 Lernverlauf der Optimierung mit dem genetischen Algorithmus bei 10 Durchläufen](#) zu sehen ist.

Im Test mit dem [DE](#) ist jedoch kein solcher Sprung in der Lernkurve zu erkennen. Der Grund dafür liegt in der Natur des [DE](#) Algorithmus. Der [DE](#) bevorzugt immer das bessere Individuum in der Population, gegenüber dem jeweils neu getesteten Individuum. Sobald die Teilbewertung  $g_4(I_i)$  in die Gesamtbewertung einbezogen wird, wird der Durchschnittsfehler der Population sofort schlechter. Der [DE](#) sorgt jedoch dafür, dass deshalb alle neuen Individuen verworfen werden weil sie schlechter sind als die bisherigen Individuen. Dadurch sieht es so aus als würde [DE](#) die hinzugefügte Teilbewertung sehr gut verkratzen. Dies ist jedoch nicht der Fall.

Im Gegensatz zum [GA](#) ist [DE](#) abhängig von den [Fehlerwerten](#) der alten Population. Deshalb darf eine solche Änderung der Bewertungsfunktion nicht vorgenommen werden, ausser es wird garantiert, dass durch die Anpassung der Bewertungsfunktion, der durchschnittliche Fehlerwert der Population verbessert wird.

Der [GA](#) hingegen ist nicht von diesem Problem betroffen, da jede Generation komplett individuell betrachtet wird und grosse Sprünge in den durchschnittlichen Fehler der Population keine Auswirkungen auf die Selektion der Individuen haben.

Dieses Problem ändert jedoch nicht grossartig das erzielte Resultat, da der [DE](#) zum Zeitpunkt als die Teilbewertung hinzugefügt wird, sowieso schon so gut ist, dass die Fehlerwerte die durch die Teilbewertung  $g_4(I_i)$  hinzugefügt werden, im Vergleich zu den anderen Teilbewertungen sehr klein sind.

## 11.2.6 | Herausforderungen & Learnings

### Die erste Arbeit mit LaTeX

Dies ist meine erste Arbeit, die ich mit LaTeX verfasst habe. Schon seit längerer Zeit hatte ich mir vorgenommen, LaTeX zu erlernen, jedoch war die Hemmschwelle dafür stets sehr hoch. Obwohl mir die Wahl des Werkzeugs freigestanden hätte, habe ich mich bewusst dafür entschieden, diese Arbeit in LaTeX zu schreiben. Es bot sich dadurch eine gute Gelegenheit, mich intensiv mit LaTeX auseinanderzusetzen.

Die Arbeit mit LaTeX hat mich teilweise viel Geduld gekostet, dennoch bin ich froh, diesen Schritt gegangen zu sein. Besonders störend empfand ich die Fehlersuche: Tritt ein Fehler auf, wird häufig nicht eindeutig angezeigt, wo dieser tatsächlich liegt. Stattdessen meldet der Compiler Probleme an anderen Stellen, was die Fehlersuche erschwert.

Aus programmietechnischer Sicht sehe ich ebenfalls Verbesserungspotenzial. Meiner Meinung nach könnte LaTeX deutlich mächtiger sein, wenn es syntaktisch und strukturell besser aufgebaut wäre.

### Implementierung der Systeme und Optimierungsalgorithmen in C++

Für die Implementierung von GA und DE habe ich mich für C++ entschieden, da ich diese Sprache gut beherrsche und damit im Vergleich zu anderen Sprachen schneller Ergebnisse erzielen kann. Als Grundlage für die C++-Applikation diente meine eigene, einfache Game-Engine [3], die es mir ermöglichte, mich rasch auf die wesentlichen Aspekte der Implementierung zu konzentrieren. Zudem ließen sich grafische Visualisierungen mit dieser Basis unkompliziert realisieren.

### Implementierung von unterschiedlichen diskreten Integrationsverfahren

Für die diskrete Simulation der Systeme habe ich mich entschieden, verschiedene Integrationsverfahren zu implementieren. Dies ermöglicht es, problemlos zwischen den unterschiedlichen Verfahren zu wechseln, um die Genaugkeit der Simulation gezielt zu optimieren.

### Alternative Darstellung des Nyquist-Diagramms

Um das Nyquist-Diagramm darzustellen, habe ich mich gegen die klassische Darstellung von MATLAB entschieden, da ich sie nicht besonders übersichtlich finde. Stattdessen habe ich das Nyquist-Diagramm mit einem externen MATLAB-Skript [32] in einer logarithmischen Darstellung geplottet. Um mehrere Systeme innerhalb desselben Diagramms darstellen zu können, waren einige Anpassungen am Skript erforderlich. Die resultierende Darstellung bietet meiner Ansicht nach eine übersichtlichere und verständlichere Visualisierung. Das Skript ist nicht perfekt aber für die Zwecke dieser Arbeit ausreichend.

### MATLAB Programmierung für die Regleroptimierung und Erstellung von Grafiken

Vor Beginn dieser Arbeit hatte ich nur wenig Erfahrung mit der Programmierung in MATLAB. Für die automatisierte Erstellung der Grafiken erwies sich MATLAB jedoch als sehr hilfreich. Um den Aufwand zu reduzieren, habe ich ein Skript entwickelt, das per Knopfdruck nahezu alle Grafiken der Arbeit erzeugt und automatisch im vorgesehenen Verzeichnis speichert.

### Auffrischung der Z-Transformation

Für die diskrete Simulation der Systeme musste ich mich zum ersten Mal seit langer Zeit wieder mit der Z-Transformation befassen. Das Erreichen eines lauffähigen Ergebnisses erwies sich dabei als sehr motivierend und steigerte die Freude an der Thematik.

### Modellbildung des Prozesses: Motor mit Schwungmasse

Bei der Modellbildung des Prozesses konnte nicht auf die dokumentierten Modellbeschreibungen zurückgegriffen werden, da diese ein anderes Systemverhalten als der reale Prozess abbilden. Daher wurde das Modell des Prozesses mithilfe einer Systemidentifikation erstellt. A.2.2 Mir kam die Idee, den GA in diesem Zusammenhang zu nutzen, anstatt mich lange mit manueller Parametrierung aufzuhalten. Um den GA für die Modellidentifikation zu nutzen, musste lediglich das strukturelle Modellgerüst definiert werden, während die Parameteroptimierung automatisch durch den Algorithmus erfolgte.

### Alternative Darstellung der Stabilitätsgrenzen

Bei der Recherche zur Interpretation des Resultatdiagramms vom Optimierungskriterium **TuningGoal.Margins** [22], bin ich auf ein Video von Brian Douglas gestossen [33]. In diesem Video wird eine alternative Darstellung der Stabilitätsgrenzen eines Systems erläutert. Die dort gezeigte Visualisierung wurde jedoch nicht vollständig übernommen, da sie einen Teil des möglichen Stabilitätsbereichs ausblendet. Stattdessen konzentrierte ich mich auf den Teil des Videos, in dem er zeigte, wie man für jeden Punkt berechnet, ob das System stabil ist oder nicht. Auf dieser Grundlage entstand der

#### A.2.7 Phasen- & Verstärkungsreserve Diagramm

Nach einer Diskussion mit Prof. Dr. Lukas Ortmann wurde deutlich, dass diese Darstellung in der praktischen Anwendung nur eingeschränkt hilfreich ist. Sie setzt voraus, dass die Phasenverschiebung über den gesamten Frequenzbereich konstant ist. In realen Systemen entstehen Phasenverschiebungen jedoch unter anderem auch durch Totzeiten, welche dann in frequenzabhängigen Phasenverschiebungen enden. Aus diesem Grund wurde ein weiteres Diagramm entwickelt, das den Einfluss von Verstärkung und Totzeit auf die Stabilität des Systems veranschaulicht.

#### A.2.8 Totzeit- & Verstärkungsreserve Diagramm

## Stabilitätsanalysen

Im Verlauf der Arbeit sind immer wieder neue Möglichkeiten zur Stabilitätsanalyse aufgetaucht.

- A.2.7 Phasen- & Verstärkungsreserve Diagramm,**
- A.2.8 Totzeit- & Verstärkungsreserve Diagramm**

und das Sensitivitäts-Diagramm wurden zu einem Zeitpunkt hinzugefügt, an dem ich schon alle Tests und Messungen abgeschlossen hatte. Die zusätzlichen Stabilitätsanalysen haben aber gezeigt, dass z.B. die optimierten Systeme für den DC-Motor zwar gute Verstärkungs- und Phasenreserven haben, aber sehr empfindlich auf Totzeiten reagieren. Für ein robusteres Systemverhalten wäre es notwendig gewesen, die Tuning-Goals und die Fitnessfunktionen dementsprechend anzupassen.

## Irrtum bei der Kombination von Ziegler-Nichols mit Optimierungsalgorithmen

In **2.3 Kombination mehrerer Methoden** habe ich beschrieben, dass die Kombination von Ziegler-Nichols mit Optimierungsalgorithmen gut sein könnte. Wie bereits erwähnt war geplant eine solche Kombination zu testen. Dies hat sich aber leider ziemlich schnell als nicht umsetzbar herausgestellt. Selbst wenn der Prozess die Anforderungen für die Verwendung von Ziegler-Nichols erfüllt, haben die durch Ziegler-Nichols bestimmten Startwerte keinen Mehrwert für die Optimierung mit **GA** oder **DE** gebracht. Im Gegenteil, der Prozess: DC-Motor hat eine sehr kleine Totzeit ( $T_u$ ), wodurch Ziegler-Nichols zu sehr aggressiven Startwerten führt, die den Optimierungsprozess eher behindern als unterstützen. Durch die Verwendung von Ziegler-Nichols werden die möglichen Regler automatisch auf PID-Regler beschränkt, wodurch die Flexibilität der Optimierungsalgorithmen stark eingeschränkt wird. Ich sehe deshalb den die Kombination von Ziegler-Nichols mit anderen Optimierungsalgorithmen nicht weiter als nützlich an.

## Dynamisches anpassen der Bewertungsfunktion für DE

Wie bereits auf Seite **53: 'Dynamisches anpassen der Bewertungsfunktion'** beschrieben, ist es heikel, die Bewertungsfunktion für **DE** während des Optimierungsprozesses zu verändern.

Mir ist es während den Tests bereits aufgefallen, dass die Lernkurve des **DE** ein ungewöhnliches Verhalten zeigt, da auch dort ein kleiner Anstieg im durchschnittlichen Fehlerwert der Population erkennbar sein müsste. Ich konnte den Fehler jedoch auch nach langer Suche nicht finden und habe das Problem deshalb ignoriert, schliesslich war das Ergebnis trotzdem mehr als zufriedenstellend.

Mir ist die Lösung dieses Problems erst in der letzten Woche vor der Abgabe der Arbeit eingefallen. Deshalb blieb mir leider die Zeit nicht, um die Tests mit dem **DE** zu wiederholen.

## Performance-Optimierung der C++ Implementierung

Die Performance der C++ Implementierung war während der Tests bis auf den Zeitpunkt, als die approximierten Frequenzgang-Berechnungen hinzugefügt wurden, immer zufriedenstellend. Die Berechnung des Frequenzgangs für die Verstärkungs- und Phasenreserve ist jedoch sehr rechenintensiv, da diese am nichtlinearen Modell im Zeitbereich durchgeführt werden. Die Berechnung des Frequenzgangs wurde in C++ implementiert, um weiterhin davon zu profitieren, unabhängig von MATLAB zu sein. Der Nachteil war jedoch, dass die Optimierungsdauer dadurch stark angestiegen ist. Ich habe mich deshalb dazu entschieden, den **GA** und **DE** so zu optimieren, dass die einzelnen Simulationen der Individuen dank Multithreading parallel ausgeführt werden können. Die Population wird dabei gleichmäßig auf die verfügbaren Prozessorkerne verteilt, wodurch die Optimierungsdauer deutlich reduziert wird.

Trotz dieser Optimierung wäre noch viel Potential zur Performance-Verbesserung vorhanden, da ich keinen Fokus auf die Effizienz der Implementierung gelegt habe. Dies wäre bei einer weiterführenden, vertieften Entwicklung dieser Infrastruktur eine Überlegung wert.

### Verfeinerung der Simulink-Modelle

Im Verlauf der Arbeit habe ich die Simulink-Modelle der Prozesse immer wieder verfeinert, um ein realistischeres Verhalten zu erzielen. Mein Ziel war es, so viel wie möglich mit diesen Modellen zu testen um nicht zu viele Experimente am realen Prozess durchführen zu müssen.

Einige dieser Verfeinerungen haben nicht so viel Einfluss auf das Systemverhalten gehabt, was aber nicht weiter schlimm ist, da es mir trotzdem Spass gemacht hat, die Modelle zu verbessern. Beispielsweise habe ich die nichtlineare DC-Verstärkung des Motors modelliert, das hat ziemlich gut funktioniert.

Für die Modellierung der Wirbelstrombremse hingegen habe ich kaum Aufwand betrieben und habe einfach einen Faktor für die Drehzahlabhängigkeit des Bremsmoments erfunden, da sich dieser sowieso, mit der auf die Bremse gegebene Spannung, kompensieren lassen würde.

### Frequenzgang-Approximationen im Zeitbereich

Für die Berechnung der Verstärkungs- und Phasenreserven im Optimierungsprozess musste ich eine Methode finden, um den Frequenzgang des Systems im Zeitbereich zu approximieren. Ich habe mich dabei für die Methode entschieden, bei der für eine Reihe von Frequenzen Sinus-Signale in das System eingespeist werden und die Amplitude sowie die Phase des Ausgangssignals nach einer vordefinierten Einschwingzeit gemessen werden.

Diese Methode hat aber nur in den Fällen gut funktioniert, als das System bereits stabil war und die nichtlinearen Effekte (z.B. Sättigungen) nur noch eine geringe Rolle gespielt haben. Das war überhaupt der Grund, warum die Bewertung der Verstärkungs- und Phasenreserven im Test mit dem Motor mit Schwungmasse, erst ab Epoche 100 in die Gesamtbewertung einbezogen wurde. Davor hat die Frequenzgang-Approximation zu unbrauchbaren Ergebnissen geführt, welche die Optimierung stark gestört haben.

Zudem muss bei der Nutzung der berechneten Phase auf die Winkel-Wrap-Around Problematik geachtet werden. Die Quadratische Abweichung der Phase ist nur dann sinnvoll, wenn sich die Phase des Systems in der Nähe der Zielphase befindet. Eine Phase von  $270^\circ$  ist z.B. weit weg von  $-90^\circ$ , obwohl sie mit Winkel-Wrap-Around auf demselben Punkt liegen.

Für eine Verbesserung müsste die Frequenzgang-Approximation sorgfältiger implementiert werden, und bei der Verwendung der berechneten Phase müsste der Winkel-Wrap-Around Acht genommen werden.

**Alex Krieg**

### Rauschunterdrückung der Regler

Bei den ersten Tests der Regler auf dem realen Prozess sind mir, vor allem bei den mit *systune* optimierten Reglern, ein starkes Rauschverhalten am Ausgang des Reglers aufgefallen. Es war teilweise so stark, dass dies sogar durch die Motoren hörbar wurde. Aufgrund dessen habe ich für den DC-Motor eine Rauschmessung durchgeführt und ein möglichst ähnliches Rauschsignal im Simulink-Modell für die modellbasierten Tests hinzugefügt.

Das hat es mir ermöglicht bereits vorgängig zu erkennen wie stark sich auch am realen Prozess das Rauschverhalten auswirken wird. Aufgrund dessen konnten die Tuning-Goals und die Fitness-Funktionen so angepasst werden, um ein besseres Rauschverhalten der Regler zu erzielen.

Für den Prozess mit der Schwungmasse habe ich keine Rauschmessung durchgeführt, habe aber dennoch ein Rauschsignal in das modellbasierte Testmodell hinzugefügt, um auch bereits vor den Tests am realen Prozess ein Rauschverhalten der Regler zu erkennen. Für die Optimierung der **GA** und **DE** konnte ich ein kleines Rauschsignal bereits mit in die Simulation einbauen, wodurch die Optimierung automatisch dazu veranlasst wurde, das Rauschverhalten der Regler zu gossen werden zu lassen.

Ich habe daraus gelernt, dass es sich lohnt, bereits für die modellbasierten Tests das Rauschen des realen Prozesses zu berücksichtigen. Alternativ können natürlich auch Filter helfen, das Rauschsignal der Sensoren zu reduzieren. In der Praxis wäre dies natürlich die bevorzugte Lösung.

### Der weltbeste Taschenrechner (MATLAB) & Support von Mathworks

MATLAB ist ein sehr mächtiges Werkzeug für viele Anwendungsbereiche. Dennoch bin ich während der Arbeit immer wieder an Grenzen gestossen wie bei der Auswahl der Tuning-Goals in *systune*, oder bei der Interpretation der Resultate von *systune*. Auch die Dokumentation von MATLAB ist nicht immer vollständig oder fehlerfrei.

Apropos Fehler: Bezüglich der in [A.2.4](#) beschriebenen Problematik mit der Simulink PID-Regler Anti-Windup-Logik, habe ich den Support von Mathworks kontaktiert und das Problem genau beschrieben, eine Lösung vorgeschlagen und ein Beispiel mit der implementierten Lösung bereitgestellt. Der Technische Support hat mir bestätigt, dass die Dokumentation tatsächlich nicht mit der Implementierung des Simulink Blocks übereinstimmt und hat das Problem an die Entwickler weitergeleitet. Ich sollte benachrichtigt werden, wenn das Problem behoben ist.

Wie einst ein sehr geschätzter Professor in einem Praktikum zu mir sagte:

*Warum nutzt du den Windows Taschenrechner, wenn du Zugriff zum weltbesten Taschenrechner hast??!*

Jetzt weiss ich weshalb

*Auch der weltbeste Taschenrechner kann Fehler haben. ;)*

## 11.3 | Weiterführende Arbeiten

### 11.3.1 | Vergleich der Methoden

Es war mir nicht möglich, die gleichen Optimierungskriterien für alle Methoden zu definieren, weil die Umsetzung der Tuning-Goals in MATLAB nicht offen gelegt wird und mir daher nicht bekannt ist. Um einen fairen Vergleich der Methoden zu ermöglichen, wäre es jedoch wichtig, alle Methoden mit den gleichen Optimierungskriterien zu testen. Für eine weiterführende Arbeit könnte dies untersucht und gegebenenfalls umgesetzt werden.

### 11.3.3 | Vertiefung der Stabilitätsanalyse

In dieser Arbeit wurden bestimmte Stabilitätsanalysen erst in einem späten Stadium durchgeführt. Für eine weiterführende Arbeit sollten diverse Stabilitätskriterien bereits im Entwurf der Tuning-Goals und der Fitnessfunktionen verstärkt berücksichtigt werden.

### 11.3.2 | Frequenzabhängige Optimierung

In dieser Arbeit wurden die Methoden **GA** und **DE** hauptsächlich im Zeitbereich angewendet. Die Berechnung der Verstärkungs- und Phasenreserve wurden ebenfalls im Zeitbereich approximativ durchgeführt. MATLAB eignet sich für Frequenzabhängige Analysen und Simulationen besser als die eigene Implementierung. Deshalb könnte in einer weiterführenden Arbeit vertiefter auf die unterschiedlichen, frequenzabhängigen Optimierungskriterien eingegangen werden.

### 11.3.4 | Weitere Prozesse testen

In dieser Arbeit wurden die Methoden nur an zwei Prozessen getestet. Für eine weiterführende Arbeit, könnten die Methoden an weiteren Prozessen getestet werden, die auch erschwerte Bedingungen aufweisen um die Leistungsfähigkeit der Methoden unter härteren Bedingungen zu untersuchen.

## 11.4 | Danksagung

Ich bedanke mich bei allen Personen, die mich in den letzten 16 Wochen auf meinem Weg unterstützt haben. Ein besonderer Dank geht an:

- **Prof. Dr. Lukas Ortmann**

Ich habe es sehr geschätzt, gute Diskussionen mit dir zu führen und gemeinsam über diverse Probleme den Kopf zu zerbrechen. In einigen Momenten habe ich mich ein wenig wie im Film 'Oppenheimer (2023)' in den Szenen vor der Wandtafel gefühlt. Auch spät in der Nacht konnte ich noch voller Erstaunen Antworten auf meine Fragen erhalten. Für meine teilweise sehr langen Nachrichten möchte ich mich entschuldigen, im Nachhinein habe ich bemerkt, dass so lange Nachrichten vermutlich nicht immer durchgelesen werden, was ich verstehen kann. Oft hat es nur schon geholfen meine Gedanken zu ordnen, indem ich dir die Probleme mit meinem Monolog beschrieben habe.

- **Bruno Vollenweider**

Vielen Dank für die Unterstützung bei der Inbetriebnahme und dem Erklären der Laborsysteme. Deine Unterstützung hat mir sehr geholfen effizient voran zu kommen. Auch in Zeiten, in denen es nicht selbstverständlich war, eine Antwort zu erhalten, musste ich nie lange warten.

Zusätzlich möchte ich mich bei allen Mitstudenten bedanken, die mich während des Studiums moralisch unterstützt haben.

## 11.5 | Erklärung zur Urheberschaft

**Erklärung**

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Benutzung anderer als, der angegebenen Hilfsmittel erstellt haben; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

**Sprachmodelle als Unterstützung**

KI-Software-Tools wie ChatGPT, GitHub-Copilot oder Claude AI wurden als Unterstützung bei der Erstellung diverser Codeabschnitte im MATLAB, C++ und LaTeX Code verwendet.

Ort	Datum
Rapperswil	19. Dez. 2025

**Unterschrift**

Alex Krieg

# A | Anhang

## A.1 | Formeln und Herleitungen

### A.1.1 | Herleitung von Formel 4.1.1

Als Ausgangslage dient das Modell aus dem Skript [36] Kapitel 3.3, Gleichung (34):

$$T \cdot \dot{y}(t) + y(t) = K_1 \cdot u(t) - K_2 \cdot l(t)$$

Parameter	Beschreibung
$K_1$	Substitution mehrerer Motor spezifischer Konstanten
$K_2$	Substitution mehrerer Motor spezifischer Konstanten
$t$	Zeit
$T$	Zeitkonstante des Motors
$u(t)$	Motor Eingangsspannung
$y(t)$	Motor Ausgangsdrehzahl als Spannung abgebildet

Tab. 17: Symbole für die Herleitung des DC-Motor Modells

Als Modifikation wird die Last  $l$  mit dem Ausgang  $y$  multipliziert, da das Lastmoment der Wirbelstrombremse abhängig von der Drehzahl des Motors ist. Dabei wird der Faktor  $K_3$  als Proportionalitätsfaktor eingeführt.  $K_3$  gibt an, wie stark die Last vom Ausgang abhängt. Somit ergibt sich die rechts dargestellte modifizierte Differential Gleichung.

$$T \cdot \dot{y}(t) + y(t) = K_1 \cdot u(t) - K_2 \cdot l(t) \cdot K_3 \cdot y(t)$$

Umstellen nach  $\dot{y}(t)$

$$\dot{y}(t) = \frac{1}{T} (K_1 \cdot u(t) - K_2 \cdot l(t) \cdot K_3 \cdot y(t) - y(t))$$

Durch die Integration erhält man  $y(t)$ . Diese Gleichung lässt sich in code implementieren.

$$y(t) = \frac{1}{T} \cdot \int (K_1 \cdot u(t) - y(t) \cdot (1 + K_2 \cdot K_3 \cdot l(t))) dt$$

Da die Implementierung diskretisiert erfolgt, muss die Formel noch diskretisiert werden. Dies wird in diesem Teil nicht weiter beschrieben, da die Integration im code auf unterschiedliche Arten implementiert worden ist. (Backward Euler, Forward Euler und Bilinear Transformation) Die folgende Formel dient deshalb nur zur Verdeutlichung des Prinzips der Implementierung im code. Für jede Integrationsmethode muss die Transformation entsprechend durchgeführt werden. Dies kann im C++ Code eingesehen werden. [2]

$$y(t) = \frac{1}{T} \cdot \int (K_1 \cdot u(t) - y(t) \cdot (1 + K_2 \cdot K_3 \cdot l(t))) dt$$

## A.1.2 | Herleitung von Formel A.2.6

In Simulink wird der D-Anteil der Übertragungsfunktion des PID-Reglers mit Filter definiert als:

$$\frac{U_{d2}(z)}{E(z)} = Kd \cdot \frac{Kn}{1 + Kn \cdot Ts \cdot \frac{1}{z-1}}$$

Parameter	Beschreibung
$Kn$	Filterkonstante
$Ts$	Abtastzeit
$Kd$	Verstärkung des D-Anteils im PID-Regler
$z$	Komplexe Variable in der $z$ -Transformation
$k$	Diskreter Zeitpunkt
$U_{d2}(z)$	Ausgang 2 für den D-Anteil im PID-Regler
$E(z)$	Eingangssignal des PID-Reglers

Tab. 18: Symbole für die Herleitung des D-Anteils mit Filter

Ziel ist es jetzt die diskrete Differenzengleichung zu bestimmen, welche dann im C++ Code umgesetzt werden kann.

Dazu wird die Gleichung umgestellt um die Brüche in  $z$  zu eliminieren  
Doppelbruch auflösen: Erweitern mit  $(z - 1)$

$$\frac{U_{d2}(z)}{E(z)} = Kd \cdot \frac{Kn \cdot (z - 1)}{(z - 1) + Kn \cdot Ts}$$

Gleichung umstellen: Mit  $E(z)$  und  $((z - 1) + Kn \cdot Ts)$  multiplizieren

$$U_{d2}(z) \cdot (z - 1 + Kn \cdot Ts) = Kd \cdot Kn \cdot (z - 1) \cdot E(z)$$

Ausmultiplizieren, damit  $U_{d2}(z)$  und  $E(z)$  mit den jeweiligen  $z$ -Termen alleine stehen

$$U_{d2}(z) \cdot z + U_{d2}(z) \cdot (Kn \cdot Ts - 1) = Kd \cdot Kn \cdot (z \cdot E(z) - E(z))$$

Da eine Multiplikation mit  $z$  im Zeitbereich einer Verschiebung um eine Abtastperiode entspricht und nicht in die Zukunft geschaut werden kann, wird die Gleichung noch mit  $z^{-1}$  multipliziert damit die Transformation in den diskreten Zeitbereich auch gleich die richtige zeitliche Verschiebung beinhaltet

$$U_{d2}(z) + U_{d2}(z) \cdot z^{-1} \cdot (Kn \cdot Ts - 1) = Kd \cdot Kn \cdot (E(z) - E(z) \cdot z^{-1})$$

Umstellen nach  $U_{d2}(z)$

$$U_{d2}(z) = Kd \cdot Kn \cdot (E(z) - E(z) \cdot z^{-1}) - U_{d2}(z) \cdot z^{-1} \cdot (Kn \cdot Ts - 1)$$

Transformation in den diskreten Zeitbereich

$$u_{d2}[k] = Kd \cdot Kn \cdot (e[k] - e[k - 1]) - u_{d2}[k - 1] \cdot (Kn \cdot Ts - 1)$$

### A.1.3 | Herleitung von Formel A.2.5

Im Simulink wird der D-Anteil der Übertragungsfunktion des PID-Reglers ohne Filter definiert als:

$$\frac{U_{d1}(z)}{E(z)} = Kd \cdot \frac{1}{Ts} \cdot \frac{z-1}{z}$$

Parameter	Beschreibung
$Ts$	Abtastzeit
$Kd$	Verstärkung des D-Anteils im PID-Regler
$z$	Komplexe Variable in der z-Transformation
$U_{d1}(z)$	Ausgang 1 für den D-Anteil im PID-Regler
$E(z)$	Eingangssignal des PID-Reglers

Tab. 19: Symbole für die Herleitung des D-Anteils ohne Filter

Ziel ist es jetzt die Differenzengleichung zu bestimmen, welche dann im C++ Code umgesetzt werden kann.

Dazu wird die Gleichung umgestellt um die Brüche in  $z$  zu eliminieren

$$z \cdot U_{d1}(z) = Kd \cdot \frac{1}{Ts} \cdot (z-1) \cdot E(z)$$

Da eine Multiplikation mit  $z$  im Zeitbereich einer Verschiebung um eine Abtastperiode entspricht und nicht in die Zukunft geschaut werden kann, wird die Gleichung noch mit  $z^{-1}$  multipliziert damit die Transformation in den diskreten Zeitbereich auch gleich die richtige Verschiebung beinhaltet

$$U_{d1}(z) = \frac{Kd}{Ts} \cdot (1 - z^{-1}) \cdot E(z)$$

Klammer auflösen

$$U_{d1}(z) = \frac{Kd}{Ts} \cdot E(z) - \frac{Kd}{Ts} \cdot z^{-1} \cdot E(z)$$

Transformation in den diskreten Zeitbereich, wobei  $k$  der aktuelle Zeitpunkt und  $k-1$  der vorherige Zeitpunkt ist

$$u_{d1}[k] = \frac{Kd}{Ts} \cdot e[k] - \frac{Kd}{Ts} \cdot e[k-1]$$

Zusammengefasst ergibt sich die Differenzengleichung

$$u_{d1}[k] = Kd \cdot \frac{e[k] - e[k-1]}{Ts}$$

## A.2 | Diverse Erklärungen

### A.2.1 | Regelkreis für Prozess: DC-Motor

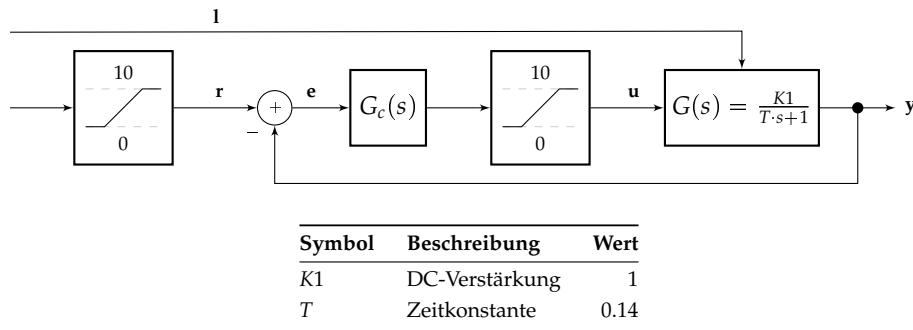


Abb. 51: Regelkreis für Prozess: DC-Motor

Das Beispielsystem ist in Abb. 51 dargestellt. Es handelt sich dabei um einen einfachen Regelkreis mit einem DC-Motor als Regelstrecke. Der Motor wird mit einer Stellgröße  $u$  (Spannung) angesteuert und die Drehgeschwindigkeit  $y$  wird als Spannung gemessen und zurückgeführt. Der Motor wird so gehandhabt als wäre es nicht möglich, ihn in entgegengesetzte Richtung zu betreiben, daher kommt die untere Sättigung der Stellgröße  $u$  bei 0V zum Tragen. Als Regler wird ein PID-Regler verwendet. Der Aufbau des PID-Reglers ist in Abb. 61 dargestellt. Die Last  $l$  treibt eine Wirbelstrombremse an, die eine bremsende Wirkung auf den Motor hat. Das System wird im Skript von Regelungstechnik 1-2 [36] genauer beschrieben. Im Gegensatz zu dem im Skript definierten  $K_1 = 0.91$  wurde hier  $K_1 = 1$  verwendet, da der im Skript verwendete Sprung von 4V für die Systemidentifikation nur für Sprünge von 4V geeignet ist. Dies liegt daran, dass die DC-Verstärkung des Motors sehr stark nichtlinear ist. Die Wahl des korrekten  $K_1$  ist deshalb nicht sehr kritisch für die Optimierung des Reglers.

Der Regelkreis verfügt über zwei Eingänge:

- Sollwert  $r$ : Gewünschte Drehgeschwindigkeit des Motors
  - Störgröße  $l$ : Externe Störung die auf den Motor wirkt.
- Für die Last wird eine Wirbelstrombremse verwendet.

#### A.2.1.1 | Implementierung im C++ Code

```

1 // Update Funktion des DC-Motors im C++ Code. class DCMotorSystem
2 void update(double deltaTime) override
3 {
4     // Variablen Aliasen für bessere Lesbarkeit in der Formel
5     double& y = m_outputAngularVelocity;
6     double& u = m_inputVoltage;
7     double& l = m_disturbance;
8     double& k1 = m_k1;
9     double& k2 = m_k2;
10    double& k3 = m_k3;
11    double& invT = m_invTimeConstant;
12
13    // Signal welches integriert wird
14    // invT = 1 / T wird bereits vor der Integration dazu multipliziert, anders als in der Formel oben dargestellt
15    double preIntSignal = (k1 * u - y * (1.0 + k2 * k3 * l)) * invT;
16
17    switch (getIntegrationSolver())
18    {
19        case IntegrationSolver::ForwardEuler:
20        {
21            m_integratorOutput = getIntegrated_forwardEuler(m_lastPreIntegratorSignal, m_integratorOutput, deltaTime);
22            break;
23        }
24        case IntegrationSolver::BackwardEuler:
25        {
26            m_integratorOutput = getIntegrated_backwardEuler(preIntSignal, m_integratorOutput, deltaTime);
27            break;
28        }
29        case IntegrationSolver::Bilinear:
30        {
31            m_integratorOutput = getIntegrated_Bilinear(m_lastPreIntegratorSignal, preIntSignal, m_integratorOutput, deltaTime);
32            break;
33        }
34    }
35    m_outputAngularVelocity = m_integratorOutput;
36    m_lastPreIntegratorSignal = preIntSignal;
37 }
```

Code 30: Update Funktion des DC-Motors im Code

Das Modell des DC-Motors ist als PT1-Glied modelliert. Die Übertragungsfunktion der Regelstrecke lautet:

$$G(s) = \frac{K_1}{T \cdot s + 1} \quad (\text{Formel A.2.1})$$

Die Wirbelstrombremse wird im Simulink-Modell abhängig von der Drehzahl des Motors modelliert. Da dies ein nichtlineares Verhalten abbildet und die Last bei der Optimierung von *systune* nicht berücksichtigt wird, genügt die Übertragungsfunktion in Formel A.2.1 als Modell für die Optimierung.

Bei den Optimierungen mit dem **GA** und **DE** ist es jedoch möglich, ein nichtlineares Modell zu verwenden, weshalb dort die Last berücksichtigt wird.

### A.2.1.2 | Implementierung des DC-Gains im Simulink Modell

Im Simulink-Modell ist der nichtlineare DC-Gain des Motors so modelliert, dass das Modell dem realen Motor möglichst nahe kommt. Dies ist eigentlich für die Optimierung nicht relevant und diente nur dazu, die Regler im Simulink-Modell vorgängig zu testen bevor sie am realen Prozess ausprobiert wurden. Wie die nichtlineare DC-Verstärkung modelliert ist, wird nachfolgend erklärt.

Eine Systemidentifikation mit Sprüngen von 0V - 10V in 1V Schritten wurde durchgeführt, um unterschiedliche DC-Verstärkungen an den unterschiedlichen Arbeitspunkten zu ermitteln.

Die ermittelten DC-Verstärkungen waren linear mit Offset in Abhängigkeit der Eingangsspannung  $u$ . Die Geradengleichung wurde mit Hilfe von Linearer Regression ermittelt. Die Geradengleichung lautet:

$$f_2(x) = m \cdot x - q \quad (\text{Formel A.2.2})$$

Symbol	Wert
$m$	1.07
$q$	-0.43

Abb. 52: Parameter der Geradengleichung

Im Simulink-Modell wird für  $x$  der Ausgangswert des PT1-Glieds eingesetzt. Das ist möglich, weil für das PT1-Glied eine Verstärkung von 1 verwendet wird, obwohl sich das  $x$  in der Geradengleichung eigentlich auf das Eingangssignal bezieht.

Da der Offset von -0.43 dazu führen würde, dass der Motor auch bei einer Eingangsspannung von 0V eine negative Drehzahl annehmen würde, muss ein Teil der Geradengleichung durch ein Polynom abgerundet werden. Der Wechsel der beiden Funktionen wird bei  $x = 1$  vorgenommen.

Um dieses Polynom zu finden wurde ein Gleichungssystem aufgestellt:

$$f_1(x) = a \cdot x^2 + b \cdot x \quad (\text{Formel A.2.3})$$

$$\begin{cases} m = f_1(1) \\ f_1(x) = f_2(x) \end{cases} \quad (\text{Formel A.2.4})$$

Mit diesem Gleichungssystem werden die Koeffizienten  $a$  und  $b$  des Polynoms bestimmt. Die Lösung des Gleichungssystems ist:

Symbol	Wert
$a$	0.43
$b$	0.21

Abb. 53: Parameter des Polynoms

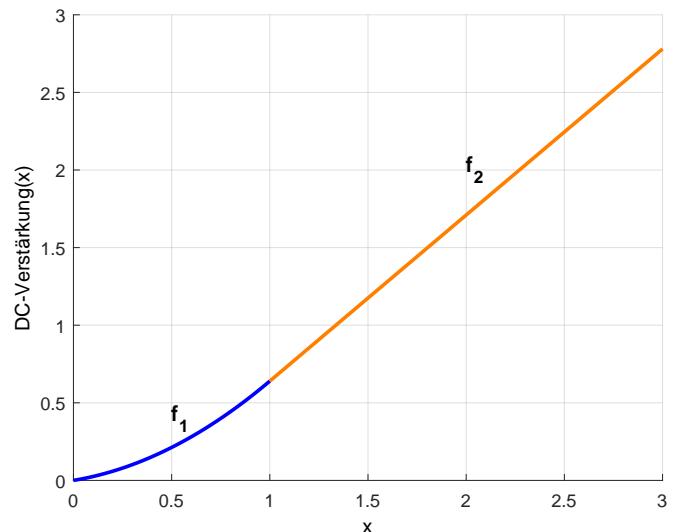


Abb. 54: Annäherung der DC-Verstärkung des Motors im Simulink-Modell

## A.2.2 | Regelkreis für Prozess: Motor-mit-Schwungmasse

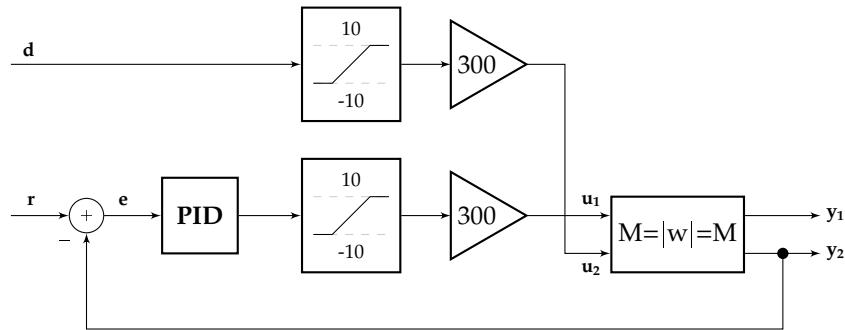


Abb. 55: Regelkreis des Prozesses Motor mit Schwungmasse

Der Prozess besteht aus zwei Motoren, welche auf einer federgekoppelten Welle sitzen. Auf den Achsen der Motoren befindet sich jeweils eine Schwungmasse, welche die Trägheit des Prozesses erhöht. Beide Motoren besitzen Encoder, welche die Winkelposition der Motorachsen messen. Der Prozess ist darauf ausgelegt, mit dem Motor1 ( $u_1$ ) einen bestimmten Winkel anzufahren. Der Motor am Eingang 2 ( $u_2$ ) dient zur Störung des Prozesses. Die Motoren sind unterschiedlich und verhalten sich deshalb nicht ganz gleich. Der Prozess beinhaltet somit zwei Eingänge und zwei Ausgänge. Für die Modellbildung werden folgende Schritte durchgeführt.

- Impulsmessungen an beiden Eingängen und Ausgängen
- Definition eines Modellgerüsts mit fehlenden Parametern
- Finden der Modellparameter mit Hilfe des genetischen Algorithmus
- Validierung des Modells
- Umwandlung des Modells in den Zustandsraum
- Verbesserung des Modells im Arbeitsbereich
- Normierung der Eingänge auf einen Eingangsbereich von  $[-10, 10]$  im C++ Code

### A.2.2.1 | Impulsmessungen

Eine Sprungantwort ist für dieses Prozess nicht geeignet für die Systemidentifikation, da der Encoder-Wert bereits das zweite Integral der Stellgröße darstellt und deshalb ein Sprung am Eingang zu einem linearen Anstieg am Ausgang führen würde. Deshalb werden kurze Impulse am Eingang des Prozesses angelegt und die Antwort des Prozesses aufgezeichnet. Abb. 56 zeigt die angelegten Impulse am den beiden Eingängen des Prozesses. Für die Identifizierung des Modells wurden diese Impulse mit unterschiedlichen Amplituden verwendet:

[500, 1000, 1500, 2000, 2500, 3000]

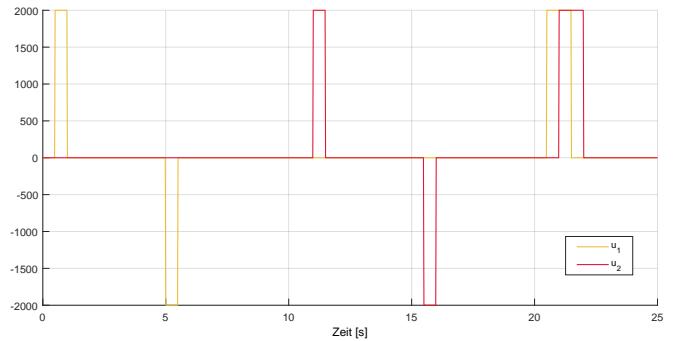


Abb. 56: Eingansimpulse 2000 des Motor-mit-Schwungmasse-Prozesses

Abb. 57 zeigt die Antwort des Prozesses auf die angelegten Impulse. Die Encoder-Werte sind in Radianen dargestellt und verlaufen jeweils mit gegensätzlichem Vorzeichen.

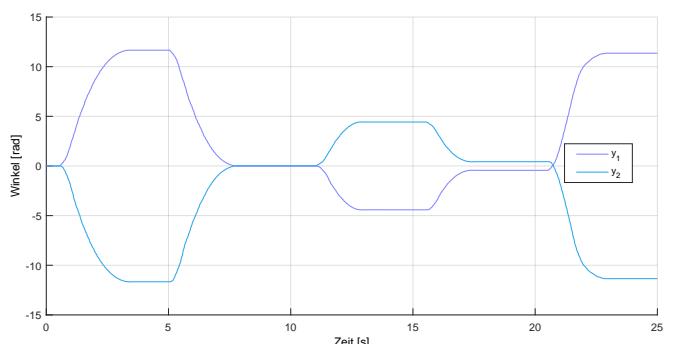


Abb. 57: Antworten des Motor-mit-Schwungmasse-Prozesses auf die Eingangsimpulse 2000

### A.2.2.2 | Modellgerüst

Der Modellentwurf basiert darauf, nur die Struktur des Prozesses festlegen zu müssen und die fehlenden Parameter mit Hilfe des genetischen Algorithmus zu finden. Prinzipiell ist das der gleiche Vorgang wie beim Optimieren eines Reglers mit dem genetischen Algorithmus. Die im Abb. 58 dargestellte Struktur entsteht aus folgenden Überlegungen:

- Die Eingangsgrößen werden mit den Vorfaktoren  $k_1$  und  $k_4$  skaliert, um die unterschiedlichen Motorstärken zu berücksichtigen.
- Beide Motoren verhalten sich wie PT1-Glieder. → Je ein Integator mit den Vorfaktoren  $k_2, k_5$  und Feedback Verbindungen.
- Die Umrechnung der Winkelgeschwindigkeit in die Winkelposition erfolgt über einen weiteren Integator mit den Vorfaktoren  $k_3, k_6$ .
- Die Ausgänge sind die Winkelpositionen der beiden Motoren.
- Die Kopplung der beiden Motoren erfolgt über eine Schwungmasse, welche ein Drehmoment proportional zur Winkelabweichung erzeugt. Weil die Encoder-Werte ein entgegengesetztes Vorzeichen haben, werden diese Beiden Encoder-Werte addiert. (Summation zwischen den Ausgängen) Die über  $k_7$  skalierte Winkelabweichung wird auf beide Motoren als Kraft mit einander entgegengesetzten Vorzeichen zurückgeführt.

Die Größen *Drehmoment* und *Winkelgeschwindigkeit* sind in diesem Modell nicht explizit dargestellt, sondern werden nur implizit über die Integatoren modelliert. Die tatsächlichen physikalischen Größen können beliebig skaliert sein im schlussendlichen Modell. Da diese Zustände jedoch am echten Prozess nicht gemessen werden, ist auch in diesem Modell der Zugang zu diesen Zuständen nicht vorhanden. Deshalb ist es egal wie diese Zustände skaliert sind.

### A.2.2.3 | Modellparameter finden

Um die fehlenden Modellparameter  $k_1$  bis  $k_7$  zu finden, wird der genetische Algorithmus verwendet. Als **Fitnessfunktion** wird der mittlere quadratische Fehler (MSE) zwischen den gemessenen Ausgangsgrößen und den modellierten Ausgangsgrößen verwendet. Als Referenz wird das gemessene Prozessverhalten mit der Impulsamplitude von 2000 verwendet, weil sich dieser Wert circa in der Mitte befindet und somit das nichtlineare Verhalten des Prozesses gut mittelt.

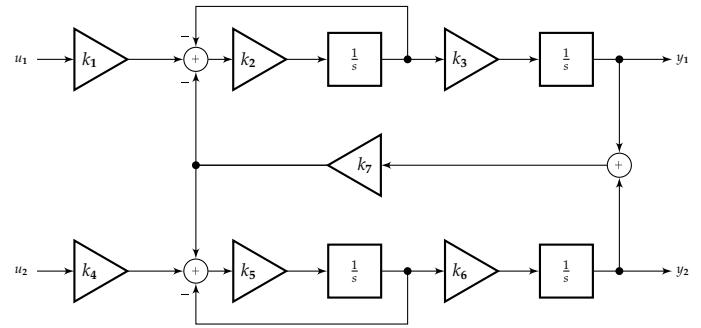


Abb. 58: Blockdiagramm der Regelstrecke Motor-mit-Schwungmasse

Der genetische Algorithmus sucht die Parameterkombination, welche den MSE minimiert. Die Initialwerte der Parameter wurden von Hand abgeschätzt und als Startpunkt für den Algorithmus verwendet um die Chance zu erhöhen, dass der Algorithmus konvergiert.

#### A.2.2.4 | Modellvalidierung

In Abb. 59 sind die Impulsantworten des realen Prozesses mit den Impulsantworten des modellierten und nach den Modellparametern optimierten Prozesses, verglichen. Die gestrichelten Linien stellen die Referenzantworten aus den echten Messungen dar. Die durchgezogenen Linien stellen das Antwortverhalten des modellierten Prozesses dar.

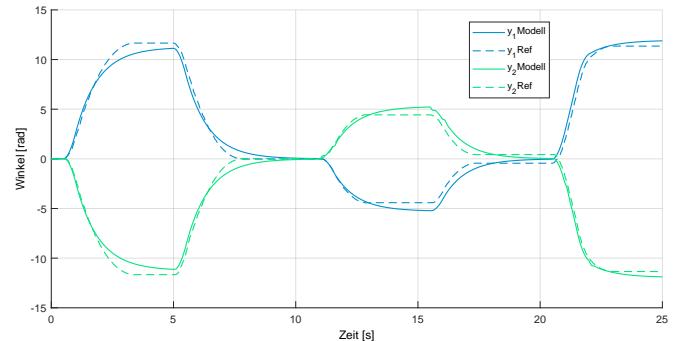


Abb. 59: Vergleich der Impulsantworten des realen Prozesses mit dem modellierten Prozess

Die Abb. 60 zeigt den Vergleich des Modells mit allen gemessenen Impulsantworten. Es ist ersichtlich, dass sich die DC-Verstärkung des realen Prozesses stark nichtlinear verhält. Es ist außerdem ersichtlich, dass sich der reale Prozess, bei kleinen Eingangsgrößen, nicht gleich verhält, je nach dem in welche Richtung der Eingang angesteuert wird. Die Impulse zum Zeitpunkt  $t = 1$ s sind positiv und die Impulse zum Zeitpunkt  $t = 5$ s sind negativ und haben die gleiche Fläche. Das bedeutet, dass der Prozess eigentlich wieder an die Ursprungposition zurückkehren müsste. Dies ist jedoch bei kleinen Eingangssignalen nicht der Fall.

Das Modell ist nicht so gut wie erhofft und um ein besseres Modell zu erhalten müsste die Modellstruktur angepasst werden und weitere Modellparameter hinzugefügt werden. Für die Verwendung in der Reglersynthese ist dieses Modell jedoch ausreichend.

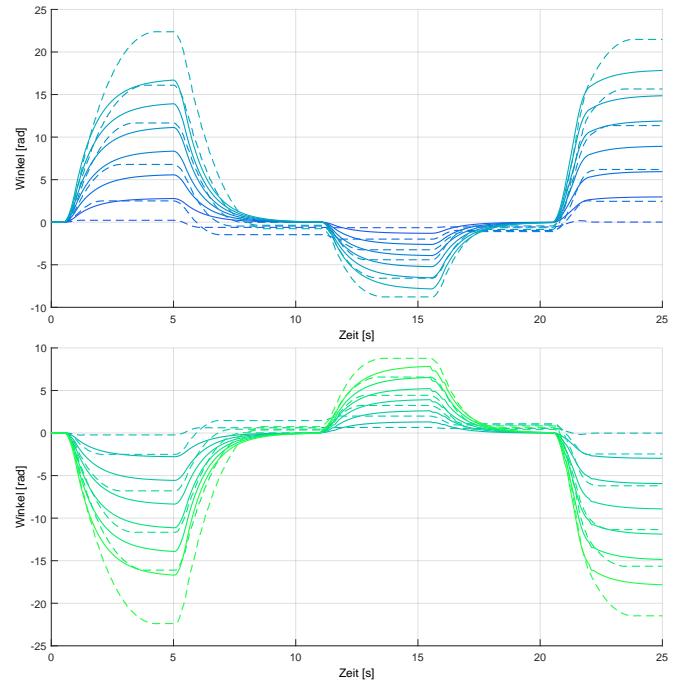


Abb. 60: Stark nichtlineare DC-Verstärkung des realen Prozesses im Vergleich zum modellierten Prozess

#### A.2.2.5 | Zustandsraummodell

Das fertige Simulink Modell kann mit dem Befehl `linmod` [20] in ein Zustandsraummodell umgewandelt werden. Das Zustandsraummodell kann anschliessend im Simulink und im MATLAB für die Reglersynthese verwendet werden.

#### A.2.2.6 | Verbesserung des Modells im Arbeitsbereich

Das Modell verhält sich sehr unterschiedlich im Vergleich zum realen Prozess. Das hat sich im Verlauf der Tests ergeben, deshalb wurde das Modell verfeinert um im verwendeten Arbeitsbereich besser zu funktionieren.

Dafür wurde der reale Prozess im closed loop Betrieb mit einem PID-Regler betrieben. Der PID-Regler wurde mit dem alten Modell entworfen und anschliessend auf den realen Prozess angewendet. Das Verhalten am realen Prozess mit dem PID-Regler wurde aufgezeichnet.

#### A.2.2.7 | Normierung der Eingänge

Die Eingänge des Modells werden auf einen Bereich von  $[-10, 10]$  normiert. Der Hauptgrund dafür ist, die dadurch verbesserte Darstellung in den Diagrammen.

Anschliessend wurde das Modell wieder mit Hilfe des genetischen Algorithmus verfeinert. Dabei wurde natürlich auch das geschlossene System mit dem gleichen PID-Regler verwendet. Das Ergebnis war ein Modell, welches sich im verwendeten Arbeitsbereich deutlich besser verhält.

Die Normierung wird durch die vorgeschalteten Verstärkungen von 300 erreicht. Dadurch wird der Eingangsbereich von  $[-10, 10]$  auf  $[-3000, 3000]$  skaliert, was dem Arbeitsbereich des realen Prozesses entspricht.

### A.2.2.8 | Implementierung im C++ Code

```
1 // Update Funktion des Modells im C++ Code. class DCMotorWithMassSystem
2 void update(double deltaTime) override
3 {
4     /*
5      * m_parameters = {
6      *     0.0637851,
7      *     -0.0232653,
8      *     189.219,
9      *     -0.207233,
10     *     18.8808,
11     *     -0.809018,
12     *     -0.228862 };
13 */
14 double input1Scaled = m_inputs[0] * 300 * m_parameters[0];
15 double input2Scaled = m_inputs[1] * 300 * m_parameters[1];
16
17 double angleDifferenceTorque = (m_disk1.angle + m_disk2.angle) * m_parameters[2];
18
19 double torque1 = (input1Scaled - m_disk1.angularVelocity - angleDifferenceTorque) * m_parameters[3];
20 double torque2 = (input2Scaled - m_disk2.angularVelocity + angleDifferenceTorque) * m_parameters[4];
21
22 double lastAngularVelocity1 = m_disk1.angularVelocity * m_parameters[5];
23 double lastAngularVelocity2 = m_disk2.angularVelocity * m_parameters[6];
24 m_disk1.angularVelocity = TimeBasedSystem::getIntegrated_Bilinear(m_disk1.lastPreIntegrationTorque, torque1, m_disk1.
25     angularVelocity, deltaTime);
26 m_disk2.angularVelocity = TimeBasedSystem::getIntegrated_Bilinear(m_disk2.lastPreIntegrationTorque, torque2, m_disk2.
27     angularVelocity, deltaTime);
28
29 m_disk1.angle = TimeBasedSystem::getIntegrated_Bilinear(lastAngularVelocity1, m_disk1.angularVelocity * m_parameters[5],
30     m_disk1.angle, deltaTime);
31 m_disk2.angle = TimeBasedSystem::getIntegrated_Bilinear(lastAngularVelocity2, m_disk2.angularVelocity * m_parameters[6],
32     m_disk2.angle, deltaTime);
33 m_outputs[0] = m_disk1.angle;
34 m_outputs[1] = m_disk2.angle;
35
36 m_disk1.lastPreIntegrationTorque = torque1;
37 m_disk2.lastPreIntegrationTorque = torque2;
38 }
```

Code 31: Update Funktion des Motors Mit Schwungmasse im Code

### A.2.3 | Reglerstruktur: Erweiterter PID-Regler

Für die Tests der unterschiedlichen Methoden wird auf ein Simulink Modell mit der in [Abb. 51 Regelkreis für Prozess: DC-Motor](#) dargestellten Regelstrecke zurückgegriffen. Der darin verwendete PID-Regler Block beinhaltet nicht nur die üblich bekannten  $K_i$ ,  $K_p$  und  $K_d$  Anteile, sondern auch noch diverse weitere Einstellmöglichkeiten und Parameter. Einige dieser zusätzlichen PID-Regler Erweiterungen sind in [Abb. 61 Aufbau des Erweiterten PID-Reglers](#) dargestellt. Für die Tests werden die unterschiedlichen Einstellungen so gut es geht berücksichtigt. Der Grund warum gleich mehrere Einstellungsmöglichkeiten getestet werden ist, dass einige Einstellungen zu einer grösseren Anzahl an Parametern führen, welche optimiert werden müssen.

Dies ermöglicht es, ein besseres Ergebnis zu erzielen und gleichzeitig zeigt es auch die Probleme der getesteten Methoden besser auf. Der PID-Regler wird im diskreten Zeitbereich angewendet um Faire Bedingungen der unterschiedlichen Methoden zu gewährleisten. Deshalb sind die dargestellten Übertragungsfunktionen auch im Z-Transformationsbereich angegeben.

Der PID-Block von Simulink hat jedoch für den Anti-Windup *Clamping* Mechanismus ein falsches Verhalten, dies ist in den Testresultaten ersichtlich. Mehr dazu kann im [A.2.4 Simulinks PID Clamping Anti-Windup Problematik](#) nachgelesen werden.

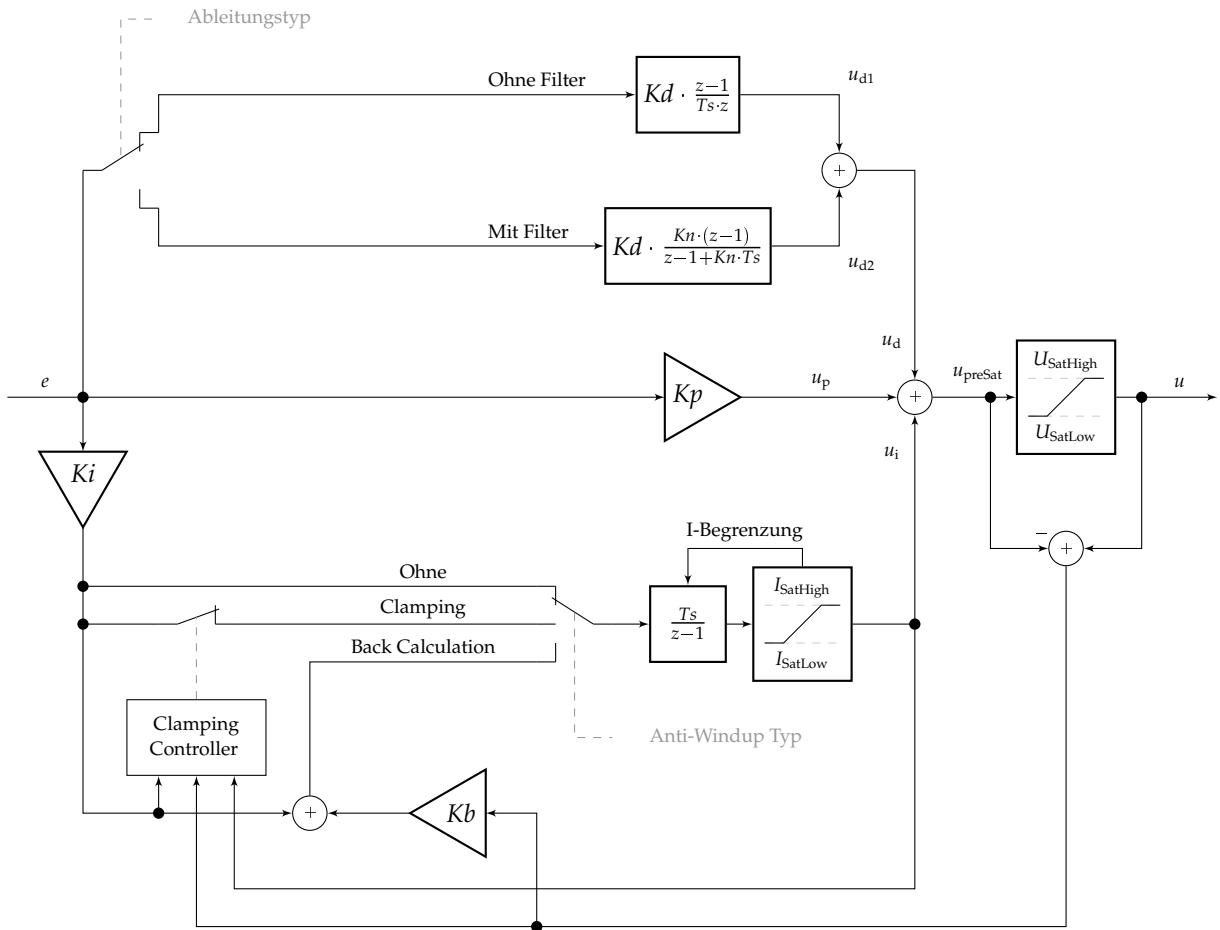


Abb. 61: Aufbau des Erweiterten PID-Reglers

Aus allgemeiner Sicht auf den PID-Regler wird bei der Erweiterung für den Ausgang  $u$  eine Sättigung eingeführt. Eine Sättigung an dieser Stelle ist sinnvoll, da in realen Systemen die Stellgrösse meist physikalischen Grenzen unterliegt und die Sättigung somit das reale Verhalten besser abbildet.

### A.2.3.1 | P-Anteil

Der Proportionalanteil verhält sich wie in einem klassischen PID-Regler.

### A.2.3.2 | I-Anteil

Aufgrund der Sättigung am Ausgang des PID-Reglers kann es zu einem sogenannten *Integral Windup* kommen. Der Integralanteil verfügt über 3 Anti-Windup Einstellungen:

- Kein Anti-Windup
- Back-Calculation
- Clamping

**Back-Calculation** Bei der Back-Calculation Methode wird die Differenz zwischen dem gesättigten Ausgang und dem ungesättigten Ausgang berechnet. Diese Differenz wird mit einem einstellbaren Faktor  $K_b$  multipliziert und vom zu integrierenden Signal subtrahiert. Dies bewirkt, dass der Integrator Eingang sinkt, wenn der Ausgang gesättigt ist und somit der Integralanteil nicht beliebig weiter ansteigen soll.  $K_b$  ist dabei ein zusätzlicher zu optimierender Parameter.

**Clamping** Bei der Clamping Methode wird der Integrator Eingang komplett auf null gesetzt, wenn  $(\text{sgn}(e \cdot K_i) = \text{sgn}(u_i)) \wedge (u - u_{\text{preSat}} \neq 0)$ , wobei  $\text{sgn}(x)$  der Signum-Operator ist. Diese Logik ist in [Abb. 61 Aufbau des Erweiterten PID-Reglers](#) als *Clamping Controller* dargestellt.

Neben dem Anti-Windup Mechanismus kann der Integrator auch noch mit einer Sättigung versehen werden, welche über die Parameter  $I_{\text{SatLow}}$  und  $I_{\text{SatHigh}}$  eingestellt wird. Die dargestellte *I-Begrenzung*'s Rückführung symbolisiert, dass der interne Zustandswert durch die Sättigung überschrieben wird. Der Sättigungsblock bezeichnet also die Sättigung des internen Integrator Zustandswertes und nicht die Sättigung am Ausgang des Integrators. Wenn die Sättigung des Integrators aktiv ist, können die Parameter  $I_{\text{SatLow}}$  und  $I_{\text{SatHigh}}$  ebenfalls optimiert werden. Für die Optimierung werden jedoch nicht beide Parameter individuell optimiert. Es wird stattdessen nur ein Wert positiv trainiert für die obere Sättigung  $I_{\text{SatHigh}}$ . Der untere Sättigungswert  $I_{\text{SatLow}}$  wird dann automatisch als negativer Wert von  $I_{\text{SatHigh}}$  gesetzt. Dadurch wird die Sättigung symmetrisch um Null gehalten und die Anzahl der zu optimierenden Parameter reduziert.

### A.2.3.3 | D-Anteil

Der in Simulink zur Verfügung stehende PID-Regler Block bietet für den diskreten D-Anteil zwei unterschiedliche Implementierungen an.

Ohne Filterung wird der D-Anteil implementiert wie in [Formel A.2.5](#) dargestellt.

$$u_{d1}[k] = K_d \cdot \frac{e[k] - e[k-1]}{T_s} \quad (\text{Formel A.2.5})$$

Herleitung siehe [Anhang A.1.3](#)

Mit Filterung wird der D-Anteil implementiert wie in [Formel A.2.6](#) dargestellt. Bei der Verwendung des D-Anteils mit Filterung kommt der zusätzliche Parameter  $K_n$  hinzu, welcher ebenfalls optimiert werden muss.

$$u_{d2}[k] = K_d \cdot K_n \cdot (e[k] - e[k-1]) - u_{d2}[k-1] \cdot (K_n \cdot T_s - 1) \quad (\text{Formel A.2.6})$$

Herleitung siehe [Anhang A.1.2](#)

### A.2.3.4 | Implementierung im C++ Code

Bei der Anti-Windup Methode *Clamping* wird im Code nicht die gleiche Implementierung wie in Simulink verwendet. Mehr Informationen dazu sind im Kapitel [A.2.4 Simulinks PID Clamping Anti-Windup Problematik](#) zu finden. Nachfolgend wird lediglich die Update Funktion des PID-Reglers im C++ Code dargestellt.

```
1 // Update Funktion des PID-Reglers im C++ Code. class PID
2 void PID::update(double deltaTime)
3 {
4     double proportional = m_kp * m_inputValue;
5     double derivative = 0;
6
7     // D-Anteil berechnen
8     switch (getDerivativeType())
9     {
10        case DerivativeType::Unfiltered:
11        {
12            switch (getDifferentiationSolver())
13            {
14                case DifferentiationSolver::BackwardEuler:
15                {
16                    derivative = TimeBasedSystem::getDifferentiated_backwardEuler(m_lastInputValue, m_inputValue, deltaTime) *
17                    m_kd;
18                    break;
19                }
20                break;
21            }
22        }
23        case DerivativeType::Filtered:
24        {
25            derivative = m_kd * m_kn * (m_inputValue - m_lastInputValue) - m_lastDerivativeValue * (m_kn * deltaTime - 1);
26            break;
27        }
28
29        double toIntegrateSignal = m_inputValue * m_ki;
30        double toIntegrateLastSignal = m_lastInputValue * m_ki;
31
32        // Anti-Windup Methode anwenden
33        switch (getAntiWindupMethod())
34        {
35            case AntiWindupMethod::None:
36            {
37                break;
38            }
39            case AntiWindupMethod::BackCalculation:
40            {
41                break;
42            }
43            case AntiWindupMethod::Clamping:
44            {
45                break;
46            }
47        }
48    }
49}
```

```

34     {
35         case AntiWindupMethod::None:
36             break;
37         case AntiWindupMethod::Clamping:
38     {
39             bool integralSignEqual = (m_integral > 0 && toIntegrateSignal > 0) || (m_integral < 0 && toIntegrateSignal < 0);
40             if ((m_outputPositiveSaturated || m_outputNegativeSaturated) && integralSignEqual)
41             {
42                 // Integration überspringen, wenn der Regler-Ausgang gesättigt ist und
43                 // das Integrator-Eingang das gleiche Vorzeichen wie der Integrator-Ausgang hat
44                 goto afterIntegration;
45             }
46             break;
47         }
48         case AntiWindupMethod::BackCalculation:
49     {
50             double antiWindupSignal = (m_outputValue - m_outputValueBeforeSaturation) * m_backCalculationConstant;
51             toIntegrateSignal += antiWindupSignal;
52             toIntegrateLastSignal += antiWindupSignal;
53             break;
54         }
55     }
56
57     // Integration durchführen
58     switch (getIntegrationSolver())
59     {
60         case IntegrationSolver::ForwardEuler:
61     {
62             m_integral = TimeBasedSystem::getIntegrated_forwardEuler(toIntegrateSignal, m_integral, deltaTime);
63             break;
64         }
65         case IntegrationSolver::BackwardEuler:
66     {
67             m_integral = TimeBasedSystem::getIntegrated_backwardEuler(toIntegrateLastSignal, m_integral, deltaTime);
68             break;
69         }
70         case IntegrationSolver::Bilinear:
71     {
72             m_integral = TimeBasedSystem::getIntegrated_Bilinear(toIntegrateLastSignal, toIntegrateSignal, m_integral, deltaTime
73         );
74             break;
75         }
76     }
77     afterIntegration: // Goto Sprungziel für Clamping Anti-Windup Methode
78
79     // Integrator Sättigung anwenden
80     if (m_integral < -m_iSaturationLimit)
81     {
82         m_integral = -m_iSaturationLimit;
83     }
84     else if (m_integral > m_iSaturationLimit)
85     {
86         m_integral = m_iSaturationLimit;
87     }
88
89     // Regler-Ausgang vor Sättigung berechnen
90     m_outputValueBeforeSaturation = proportional + m_integral + derivative;
91
92     // Regler-Ausgangssättigung anwenden
93     m_outputPositiveSaturated = false;
94     m_outputNegativeSaturated = false;
95     if (m_outputValueBeforeSaturation < m_outputSaturationLimitLower)
96     {
97         m_outputValue = m_outputSaturationLimitLower;
98         m_outputNegativeSaturated = true;
99     }
100    else if (m_outputValueBeforeSaturation > m_outputSaturationLimitUpper)
101    {
102        m_outputValue = m_outputSaturationLimitUpper;
103        m_outputPositiveSaturated = true;
104    }
105    else
106    {
107        m_outputValue = m_outputValueBeforeSaturation;
108    }
109
110    m_lastInputValue = m_inputValue;
111    m_lastDerivativeValue = derivative;
112 }

```

Code 32: Update Funktion des PID-Reglers im Code

### A.2.3.5 | Zusammenfassung

Der verwendete PID-Regler bietet die Möglichkeit unterschiedliche Einstellungen zu testen und zu vergleichen. In der Tabelle sind die möglichen Optimierungsparameter des PID-Reglers zusammengefasst.

Parameter	Beschreibung	Parameter vorhanden, wenn:
$K_p$	Proportionalverstärkung	immer
$K_i$	Integralverstärkung	immer
$K_d$	Differentialverstärkung	immer
$K_n$	D-Anteil Filterkonstante	Ableitungstyp = Mit Filter
$U_{SatHigh}$	Obere Ausgangssättigung	Ausgangssättigung aktiviert
$U_{SatLow}$	Untere Ausgangssättigung	Ausgangssättigung aktiviert
$K_b$	Anti-Windup Verstärkung	Anti-Windup Typ = Back-Calculation
$I_{SatHigh}/I_{SatLow}$	$\pm$ Integrator Sättigung	aktiviert

Tab. 20: Übersicht der PID-Regler Parameter

## A.2.4 | Simulinks PID Clamping Anti-Windup Problematik

In Simulink wird die Clamping Anti-Windup Methode etwas UNDers umgesetzt als in dieser Arbeit beschrieben. In der Dokumentation für den PID-Regler Block [25] wird die Clamping Logik wie folgt definiert:

### clamping

Integration stops when the sum of the block components exceeds the output limits  $U_{IND}$  and the integrator output  $U_{IND}$  block input have the same sign. Integration resumes when the sum of the block components exceeds the output limits  $U_{IND}$  and the integrator output  $U_{IND}$  block input have opposite sign. Clamping is sometimes referred to as conditional integration.

Clamping can be useful for plants with relatively small dead times, but can yield a poor transient response for large dead times.

Wie sich herausgestellt hat, ist die Tatsächliche Implementierung nicht so wie in der Dokumentation beschrieben. Bei bestimmten  $K_p$ ,  $K_i$  und  $K_d$  Kombinationen kann es teilweise zu unerwünschtem Verhalten am Ausgang des PID-Reglers kommen.

Die Abb. 62 zeigt das Verhalten des PID-Reglers beim ausschalten des Motors. In orange ist der Regler-Ausgang  $u$  dargestellt, welcher die Spannung am Motor repräsentiert. In blau ist die Drehgeschwindigkeit  $y$  des Motors dargestellt. Beim ausschalten des Motors (Referenz auf 0 setzen) sollte der Regler-Ausgang  $u$  auf 0 gehen und bleiben. Es ist jedoch zu erkennen, dass der Regler plötzlich beginnt kleine Rippel zu erzeugen. Diese Rippel führen dazu, dass der Motor immer wieder kurz anläuft und wieder stoppt, was zu einem unerwünschten Verhalten führt.

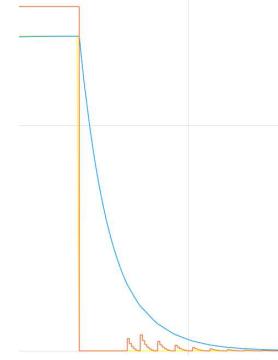


Abb. 62: Motor ausschalten mit Simulink PID-Regler und Clamping Anti-Windup  
 $K_p = 5, K_i = 35, K_d = 0$

### A.2.4.1 | Clamping Logik

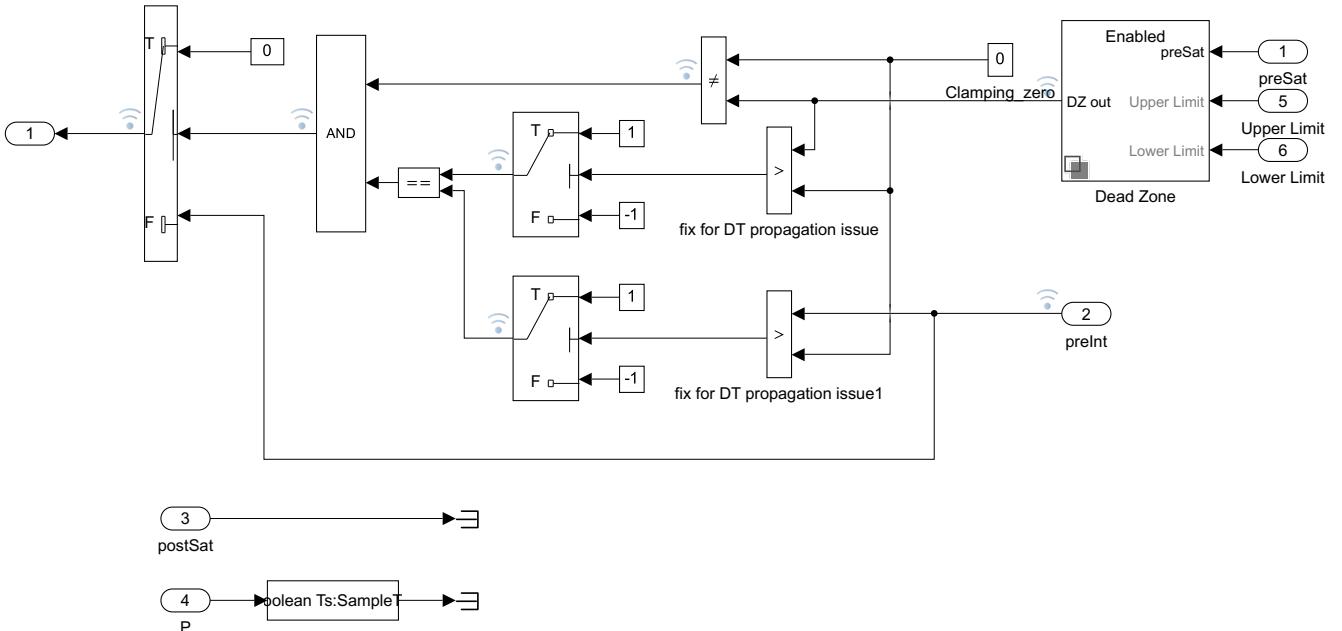


Abb. 63: Clamping Logik im Simulink PID-Regler [25]

Die Eingänge dieses Blockes sind:

- $preSat$ : Welches dem Signal  $u_{preSat}$  aus der Abb. 61 Aufbau des Erweiterten PID-Reglers entspricht.
- $UpperLimit$ : Ist das obere Sättigungs-Limit am Ausgang des PID-Reglers.  $UpperLimit = 10$
- $LowerLimit$ : Ist das untere Sättigungs-Limit am Ausgang des PID-Reglers.  $LowerLimit = 0$
- $preInt$ : Ist das Signal vor dem Integrator, also  $e \cdot K_i$ .

Der Ausgang links mit dem Namen 1 ist der effektive Integrator Eingang nach der Clamping Logik.

## Logikbeschreibung

Die Eingangssignale *preSat*, *UpperLimit* und *LowerLimit* werden im Block *Dead Zone* verarbeitet.

Der Ausgang *Clamping\_zero* (Signal-Name) der Dead Zone ist negativ, wenn sich der PID-Regler in der negativen Sättigung befindet. Er ist positiv, wenn sich der PID-Regler in der positiven Sättigung befindet. Wenn *Clamping\_zero*  $\neq 0$ , dann ist der erste Eingang am *UND - Gatter* aktiv. Der zweite Eingang am *UND - Gatter* ist aktiv, wenn *preInt* und *Clamping\_zero* das gleiche Vorzeichen haben. Wenn der Ausgang am *UND - Gatter* aktiv ist, wird der Integrator Eingang auf 0 gesetzt.

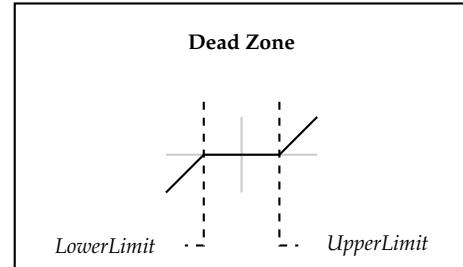


Abb. 64: Dead Zone Charakteristik

## Problematik bei dieser Logik

Das Vergleichen des Vorzeichens ist grundsätzlich richtig, aber *preSat* ist nicht das richtige Signal dafür. Was in Abb. 62 passiert ist folgendes:

- Zeitpunkt wo *r* auf 0 gesetzt wird:
  - Der Integrator hat einen Wert von ca. 2 gespeichert.
  - Der Proportionalteil schlägt in den negativen Bereich mit einem Wert von  $Ki \cdot e = 5 \cdot -2 = -10$  aus.
  - Somit ist *preSat*  $= -10 + 2 + 0 = -8$ .
  - Da *preSat*  $< LowerLimit$  folgt  $\rightarrow Clamping\_zero < 0$ .
  - $PreInt = Ki \cdot e = 35 \cdot -2 = -70$
  - $sgn(PreInt) = sgn(Clamping\_zero) \rightarrow$  Integrator Eingang = 0
  - Der Integrator bleibt auf 2 stehen.
- Der Ausgang des PID-Reglers ist in der unteren Sättigung bei 0, deshalb dreht der Motor nur noch aus.
- Sobald der Term  $Kp \cdot e + Ui > 0$  wird, befindet sich das Signal *preSat* nicht mehr im gesättigten Bereich. Was dann passiert ist folgendes:
  - $Kp \cdot e + Ui = 0 \rightarrow e = \frac{-Ui}{Kp} = \frac{-2}{5} = -0.4$
  - $Clamping\_zero = 0 \rightarrow$  Der erste Eingang am *UND - Gatter* ist nicht mehr aktiv.
  - Da der erste Eingang am *UND - Gatter* nicht mehr aktiv ist, wird der Integrator Eingang wieder auf *preInt* gesetzt.
  - Der Integrator beginnt nun wieder abzointegrieren.
- Der Integrator integriert schneller ab als der Motor abremst, was dazu führt, dass  $Kp \cdot e + Ui < 0$  wird und der PID-Regler wieder in die negative Sättigung geht.
- Dadurch wird *Clamping\_zero* wieder negativ und der Integrator Eingang wieder auf 0 gesetzt.
- Dieses Verhalten wiederholt sich, was zu den Rippeln am Ausgang des PID-Reglers führt.

Das Verhalten ist unten in Abb. 65 dargestellt.

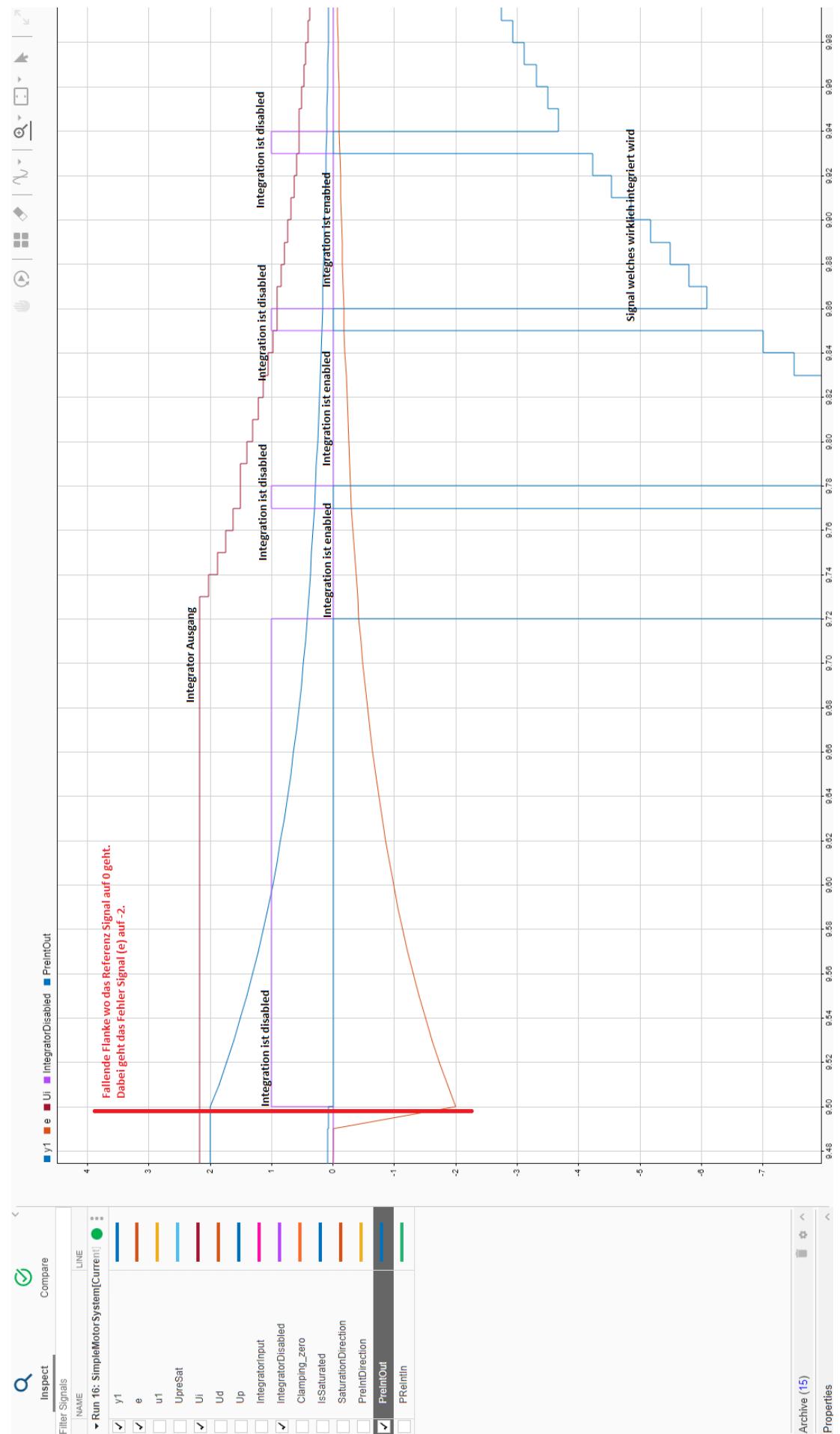


Abb. 65: Interne Signale des PID-Reglers während der Entstehung der Rippel

## Lösung des Problems

 **Info:**

Die folgende Implementierung ist in der C++ PID-Implementierung umgesetzt.

Wenn man die Implementierung so umsetzt, wie sie in der Dokumentation [A.2.4](#) beschrieben ist, dann funktioniert die Clamping Logik wie erwartet.

When the pid is in saturation, positive or negative AND the sign of the integrator input is the same as the integrator output sign then the output of the AND gate is '1'

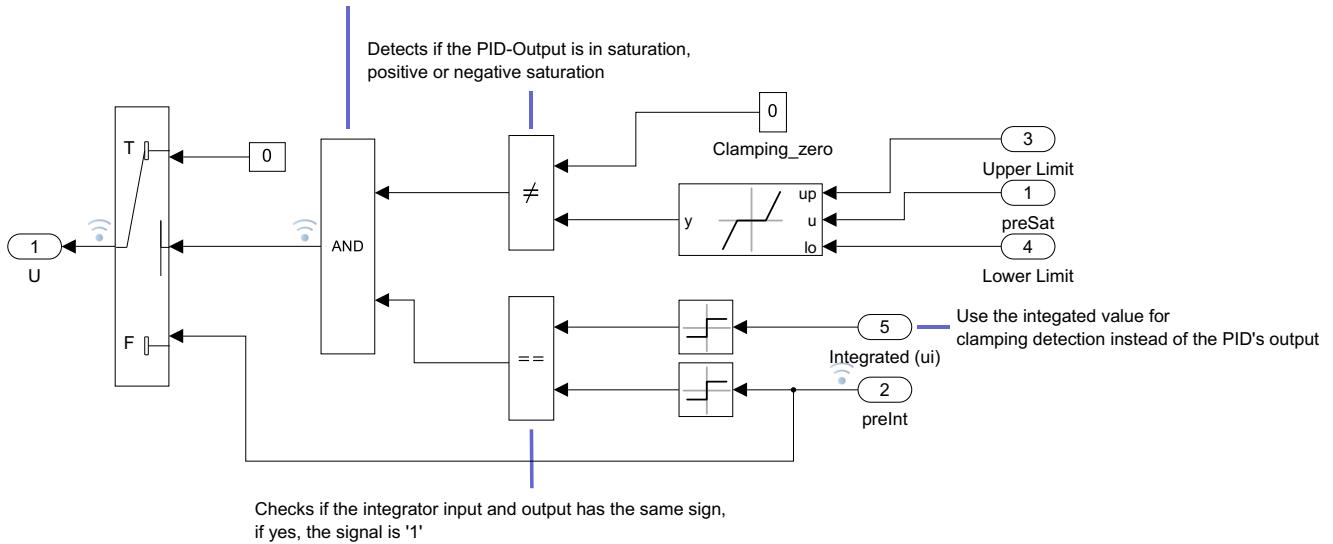
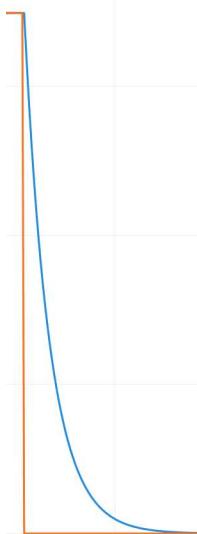


Abb. 66: Verbesserte clamping Logik im Simulink PID-Regler

Wie man in Abb. 67 sehen kann, sind die Rippel am Ausgang des PID-Reglers verschwunden. Der PID-Ausgang (orange) bleibt auf 0, wie erwartet. Leider lässt sich dieser Fix nicht in den Simulink PID-Regler Block integrieren, deshalb kommt dieser Effekt dennoch an einigen Stellen der Dokumentation vor.



**Abb. 67:** Motor ausschalten mit Simulink PID-Regler und verbesserter Clamping Anti-Windup  
 $K_p = 5, K_i = 35, K_d = 0$

## A.2.5 | Genetischer Algorithmus

### A.2.5.1 | Idee

Der genetische Algorithmus (GA) [39] ist ein evolutionärer Optimierungsalgorithmus, der von den Prinzipien der natürlichen Selektion und Genetik inspiriert ist. Er wird verwendet, um optimale oder nahezu optimale Lösungen für komplexe Probleme zu finden, indem er eine Population von möglichen Lösungen (Individuen) iterativ verbessert.

Ein Individuum setzt sich aus einer Menge von Genen zusammen, die zusammen ein Chromosom bilden. Die Gene repräsentieren die Parameter oder Variablen des zu optimierenden Problems.

### A.2.5.2 | Grundprinzip

Der Algorithmus beginnt mit einer zufällig generierten Population von Individuen. Die Anzahl der Individuen in der Population wird durch die Populationsgrösse  $N$  bestimmt und bleibt während des gesamten Optimierungsprozesses konstant. Jedes Individuum  $\vec{T}_i$  wird anhand einer **Fitnessfunktion** bewertet, die misst, wie gut es das Optimierungsziel erfüllt. Basierend auf den **Fitness**werten werden Individuen ausgewählt, um sich zu paaren und Nachkommen zu erzeugen. Die Paarung erfolgt durch genetische Operationen wie Kreuzung (Crossover) und Mutation. Die Kreuzung kombiniert die Gene zweier Elternindividuen, um neue Nachkommen zu erzeugen, während die Mutation zufällige Änderungen an den Genen eines Individuums vornimmt, um genetische Vielfalt zu gewährleisten. Nach der Erzeugung einer neuen Generation von Individuen wird die Fitness jedes Nachkommens bewertet. Der Prozess der Bewertung, Auswahl, Kreuzung und Mutation wird über mehrere Generationen hinweg wiederholt, bis eine Abbruchbedingung erfüllt ist, wie z.B. eine maximale Anzahl von Generationen oder eine zufriedenstellende Fitness erreicht wird.

### A.2.5.3 | Ablauf

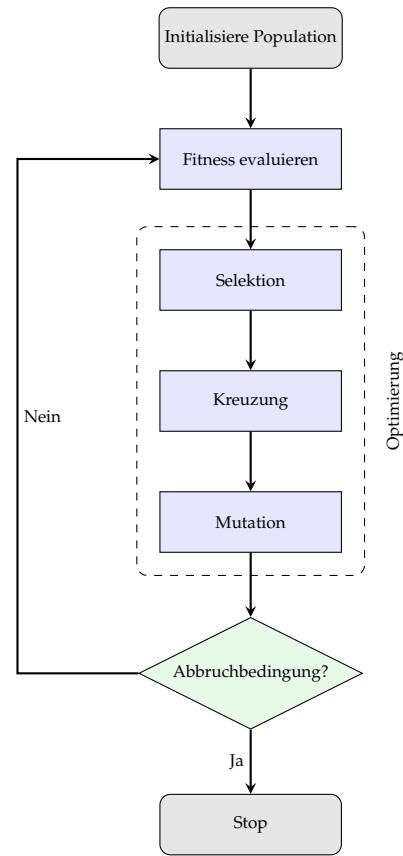


Abb. 68: Flussdiagramm des genetischen Algorithmus

#### Initialisierung

Zu Beginn wird eine Population von  $N$  Lösungsvektoren zufällig im Suchraum verteilt. Jeder Vektor repräsentiert eine potenzielle Lösung des Optimierungsproblems und besteht aus  $D$  Parametern, wobei  $D$  die Dimensionalität des Problems darstellt. Die Anfangswerte werden typischerweise gleichverteilt innerhalb der definierten Grenzen des Suchraums gewählt.

#### Simulation und Bewertung

Für jedes Individuum in der Population wird eine Simulation mit den Parametern des Individuums durchgeführt. Während der Simulation wird das Verhalten des Individuums beobachtet und bewertet. Am Ende des Simulationsdurchlaufs hat jedes Individuum einen Fitnesswert, welcher angibt, wie gut das Individuum das Optimierungsziel erfüllt hat. Die Fitnessfunktion ist dabei spezifisch für das zu optimierende Problem und muss entsprechend definiert werden. Bei der Definition der Fitnessfunktion muss darauf geachtet werden, dass diese immer einen positiven Wert zurückgibt und je höher der Wert, desto besser die Lösung ist. Die Begründung dazu ist in [A.2.5.5 Warum die Fitnessfunktion einen positiven Wert generieren muss](#) zu finden. Der GA ist nicht zum minimieren einer Funktion ausgelegt, sondern zum maximieren. Im Abschnitt [A.2.5.5 Verschiebung und Invertierung der Fitnesswerte](#) wird erklärt, der Algorithmus dennoch mit Minimierungsproblemen umgehen kann.

### A.2.5.4 | Optimierung

Nachfolgend werden die einzelnen Schritte des Optimierungsprozesses, wie sie in [Abb. 68](#) dargestellt sind, erläutert.

Für die Optimierung werden diverse Symbole verwendet, welche in der [Tab. 21](#) zusammengefasst sind.

Symbol	Beschreibung
$N$	Anzahl Individuen in der Population
$D$	Anzahl Gene pro Individuum / Anzahl Parameter
$\vec{T}_i$	Individuum $i$ in der Population $\vec{T}_i \in \mathbb{R}^D$
$f(\vec{T}_i)$	Maximierungs-Fitness-Funktion. $f(\vec{T}_i) \in \mathbb{R}^+$
$P(\vec{T}_i)$	Selektionswahrscheinlichkeit des Individuums $i$ . $P(\vec{T}_i) \in [0, 1]$
$k_{ij}$	Gen $j$ des Individuums $i$
$\alpha$	Mutationsstärke (Skalierungsfaktor für die Normalverteilung). $\alpha \in \mathbb{R}^+$
$\beta$	Mutationswahrscheinlichkeit. $\beta \in [0, 1]$

Tab. 21: Symbole für die nachfolgenden Erklärungen des genetischen Algorithmus

## Selektion

Es gibt unterschiedliche Selektionsmethoden. In dieser Arbeit wird die *Roulette Wheel Selection* Methode angewendet. Für die Selektion wird die *Fitness*  $f(\vec{I}_i)$  jedes Individuums in der Population verwendet. Je höher die Fitness, desto grösser ist die Wahrscheinlichkeit, dass das Individuum für die Paarung ausgewählt wird. In Abb. 69 ist ein Beispiel, wie man sich die Selektion vorstellen kann. Die Fläche jedes Sektors im Kreisdiagramm repräsentiert die Fitness eines Individuums. Es werden nun zwei zufällige Punkte im Kreisdiagramm ausgewählt, welche nicht im selben Sektor liegen dürfen. Die Individuen, in denen die Punkte liegen, werden als Eltern für die Kreuzung ausgewählt.

$$P(\vec{I}_i) = \frac{f(\vec{I}_i)}{\sum_{n=0}^{N-1} f(\vec{I}_n)} \quad (\text{Formel A.2.7})$$

Wobei  $P(\vec{I}_i)$  die Selektionswahrscheinlichkeit des Individuums  $\vec{I}_i$  ist,  $f(\vec{I}_i)$  die Fitness des Individuums  $\vec{I}_i$  ist und  $N$  die Gesamtanzahl der Individuen in der Population ist.

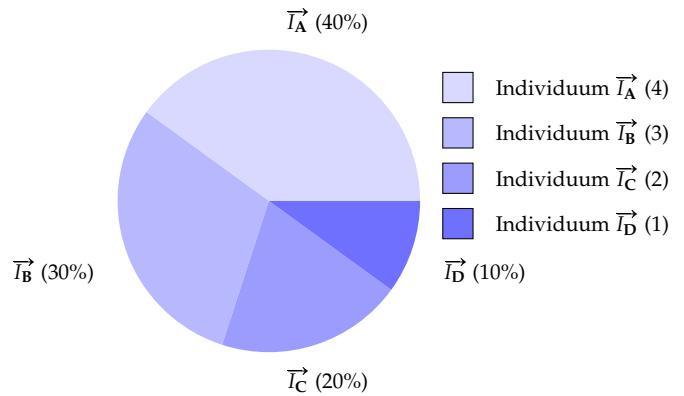


Abb. 69: Selektion von Individuen basierend auf ihrer Fitness

Zwei selektierte Eltern werden in Abb. 70 dargestellt. Dabei repräsentieren die Kästchen die Gene der Individuen, welche die zu optimierenden Parameter  $k_{ij}$  darstellen.

Elternteil A:	$k_{A0} = 0.5$	$k_{A1} = 0.1$	$k_{A2} = 0.01$
---------------	----------------	----------------	-----------------

Elternteil B:	$k_{B0} = 0.2$	$k_{B1} = 0.2$	$k_{B2} = 0.05$
---------------	----------------	----------------	-----------------

Abb. 70: Selektierte Eltern für die Kreuzung

## Kreuzung

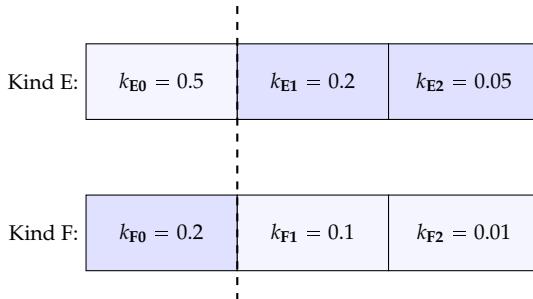


Abb. 71: Kreuzung (Crossover) zweier Eltern zur Erzeugung von Nachkommen

Bei der Kreuzung wird das Genom der beiden Eltern an einer zufälligen Stelle geteilt. In Abb. 71 ist dies durch die gestrichelte Linie dargestellt. Die beiden Teile der Gene werden dann vertauscht, um zwei neue Nachkommen zu erzeugen. Dies ermöglicht die Kombination von genetischem Material beider Eltern.

## Mutation

Kind E:	$k_{E0} = 0.51$	$k_{E1} = 0.2$	$k_{E2} = 0.05$
---------	-----------------	----------------	-----------------

Kind F:	$k_{F0} = 0.2$	$k_{F1} = 0.09$	$k_{F2} = 0.01$
---------	----------------	-----------------	-----------------

Abb. 72: Mutation von Genen in den Nachkommen

Bei der Mutation werden an den Nachkommen zufällige Änderungen an einzelnen Genen vorgenommen. In Abb. 72 sind die mutierten Gene rot hervorgehoben. Wie viele Gene mutieren und wie stark sie verändert werden, hängt von der Mutationschance und der Mutationsrate ab.

$$k_{ij\_neu} = \begin{cases} k_{ij} + \alpha \cdot \text{randN}(-1, 1) & \text{wenn } \text{randG}(0, 1) < \beta \\ k_{ij} & \text{sonst} \end{cases} \quad (\text{Formel A.2.8})$$

$\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

$\text{randN}(a, b)$  ist eine normalverteilte Zufallsvariable im Intervall  $[a, b]$

## Ersetzung

Nach der Bewertung der neuen Generation von Individuen, werden die Individuen der alten Generation durch die neuen Individuen ersetzt. Dies geschieht in der Regel vollständig, wobei die gesamte alte Population durch die neue Population ersetzt wird.

## Abbruchbedingung

Der Optimierungsprozess wird wiederholt, bis eine Abbruchbedingung erfüllt ist. Dies kann eine vordefinierte Anzahl von Generationen sein, eine bestimmte Fitness-Schwelle, die erreicht werden muss, oder eine Kombination aus beiden. Sobald die Abbruchbedingung erfüllt ist, wird das beste Individuum in der Population als die optimale Lösung des Problems betrachtet.

### A.2.5.5 | Einfluss der Startbedingungen und Hyperparameter

Massgeblich für den Erfolg eines guten Resultates sind folgende Parameter beteiligt:

- Populationsgrösse, also die Anzahl der Individuen
- Streuung der Gene in der Startpopulation, also wie unterschiedlich die Individuen zu Beginn sind
- Mutationschance, also wie häufig Gene zufällig verändert werden
- Mutationsrate, also wie stark die Gene bei einer Mutation verändert werden
- Anzahl Generationen

#### Populationsgrösse

Wie in der Natur auch, ist eine grössere Population vorteilhaft um eine grössere genetische Vielfalt zu gewährleisten. Dies erhöht die Wahrscheinlichkeit, dass gute Lösungen gefunden werden.

Allerdings steigt mit der Populationsgrösse auch der Rechenaufwand pro Generation, da mehr Individuen bewertet werden müssen.

#### Streuung der Startpopulation

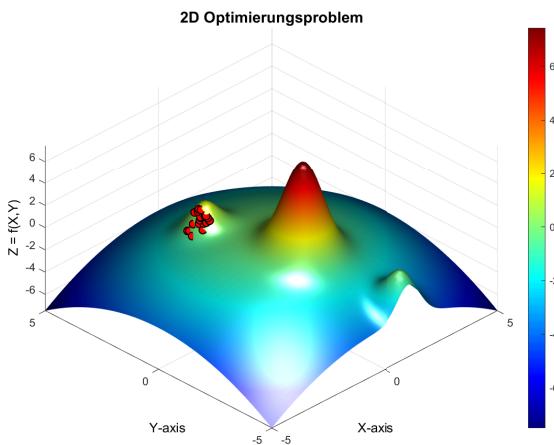


Abb. 73: Geringe Streuung der Startpopulation in einem 2D Problemraum

Die Streuung der Startpopulation beeinflusst die genetische Vielfalt der Individuen zu Beginn des Optimierungsprozesses. Eine kleine Streuung, wie in Abb. 73 dargestellt, kann dazu führen, dass der Algorithmus in lokalen Optima stecken bleibt, da die Individuen zu ähnlich sind und somit nicht genügend Vielfalt vorhanden ist, um den Suchraum effektiv zu erkunden.

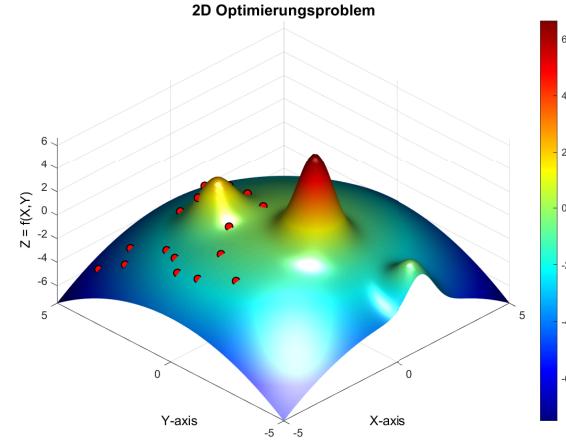


Abb. 74: Hohe Streuung der Startpopulation in einem 2D Problemraum

Eine hohe Streuung, wie in Abb. 74 dargestellt, ermöglicht es dem Algorithmus, verschiedene Bereiche des Suchraums zu erkunden. Dies erhöht die Chancen, globale Optima zu finden, da die Individuen unterschiedliche Lösungen repräsentieren.

Über den Verlauf der Optimierung hinweg, tendiert die Streuung der Population dazu abzunehmen. Wie gross die Streuung der Population langfristig ist, hängt von der Mutationschance und der Mutationsrate ab.

#### Mutationschance $\beta$

Die Mutationschance bestimmt, wie häufig Gene zufällig verändert werden. Während des Mutationsschrittes wird für jedes Gen, also für jeden Optimierungsparameter eines Individuums, entschieden, ob es mutiert oder nicht. Eine höhere Mutationschance führt zu einer grösseren genetischen Vielfalt in der Population. Es ist jedoch nicht ratsam die Mutationschance zu hoch zu wählen, da dies den Suchprozess zu zufällig machen kann. Zu viele Mutationen auf einmal sind kontraproduktiv da jeder Parameter Einfluss auf eine gute Lösung hat und zu viele Veränderungen auf einmal, die Chance erhöht eine bereits gute Lösung zu verschlechtern.

#### Mutationsrate $\alpha$

Die Mutationsrate bestimmt, wie stark die Gene bei einer Mutation verändert werden. Ein Gen wird bei der Mutation immer um einen zufälligen Wert verändert, der durch die Mutationsrate skaliert wird. Eine höhere Mutationsrate führt zu grösseren Veränderungen der Gene, was die genetische Vielfalt erhöht. Allerdings kann eine zu hohe Mutationsrate dazu führen, dass gute Lösungen zerstört werden, da die Veränderungen zu drastisch sind. Die Mutationsrate wird optimalerweise im Verlauf der Optimierung verringert, um zu Beginn eine breite Suche zu ermöglichen und später die Suche zu verfeinern.

## Warum die Fitnessfunktion einen positiven Wert generieren muss

Bei der Selektion der Individuen für die Paarung wird die Fitness jedes Individuums verwendet. Die Wahrscheinlichkeit, dass ein Individuum ausgewählt wird, ist proportional zu seiner Fitness und ist definiert als:

$$P(\vec{I}_i) = \frac{f(\vec{I}_i)}{\sum_{j=0}^{N-1} f(\vec{I}_j)} \quad (\text{Formel A.2.9})$$

Wobei  $P(\vec{I}_i)$  die Selektionswahrscheinlichkeit des Individuums  $\vec{I}_i$  ist,  $f(\vec{I}_i)$  die Fitness des Individuums  $\vec{I}_i$  ist und  $N$  die Gesamtanzahl der Individuen in der Population ist.

Mit negativen Fitnesswerten lassen sich keine Wahrscheinlichkeiten berechnen, da nach dieser Formel die Wahrscheinlichkeit, selektiert zu werden auch negativ werden könnte, was keinen Sinn ergibt. Daher muss die Fitnessfunktion so definiert werden, dass sie immer einen positiven Wert zurückgibt.

Es ist außerdem auch schwierig ein **Minimierungsproblem** mit diesem Selektionsverfahren zu lösen, da eine geringere Fitness zu einer höheren Selektionswahrscheinlichkeit führen müsste.

Es gibt Methoden wie man ein Minimierungsproblem zu einem **Maximierungsproblem** umwandeln kann.

## Vorzeichen wechseln

Durch das wechseln des Vorzeichens der Bewertungsfunktion, also  $f(\vec{x}) = -g(\vec{x})$ , wird aus einem Minimierungsproblem ein Maximierungsproblem.

## Problematik

Dabei können jedoch negative Fitnesswerte entstehen, wenn die Bewertungsfunktion  $g(\vec{x})$  positive Werte annimmt.

## Vorzeichen wechseln und verschieben

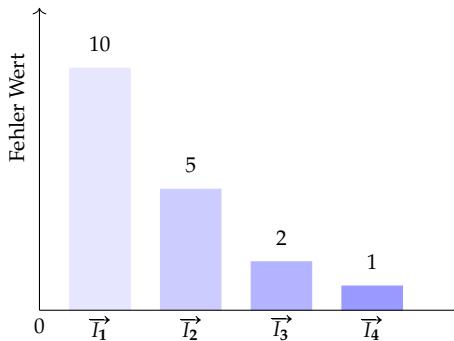


Abb. 75: Originale Fehlerwerte der Individuen

Um sicherzustellen, dass die Fitnessfunktion immer positive Werte zurückgibt, kann die Bewertungsfunktion um einen konstanten Wert verschoben werden.

$$f(\vec{x}) = -g(\vec{x}) + C \quad (\text{Formel A.2.10})$$

## Problematik

Mit dem benötigten Offset  $C$  werden die Verhältnisse der Fitnesswerte zueinander verändert. Wird ein zu grosses  $C$  gewählt, so werden die Unterschiede zwischen den Fitnesswerten immer kleiner. Dadurch erhalten alle Individuen eine ähnliche Selektionswahrscheinlichkeit, was sich negativ auf die Effizienz des **GA** auswirkt.

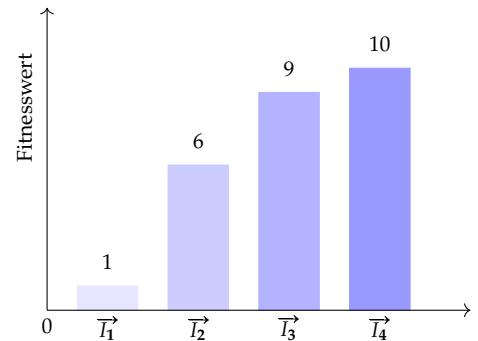


Abb. 76: Vorzeichenwechsel + Verschiebung

Offset  $C$  ist beispielsweise als 11 gewählt worden, da ansonsten Individuum A auf 0 kommen würde. → Somit hätte Individuum A keine Chance selektiert zu werden, was nicht fair ist.

$$\lim_{C \rightarrow \infty} P(\vec{I}_i) = \lim_{C \rightarrow \infty} \frac{C - f(\vec{I}_i)}{\sum_{j=0}^{N-1} (C - f(\vec{I}_j))} \quad \forall i \in [0, N-1] \quad (\text{Formel A.2.11})$$

$$= \lim_{C \rightarrow \infty} \frac{C}{\sum_{j=0}^{N-1} C} \quad (\text{Formel A.2.12})$$

$$= \lim_{C \rightarrow \infty} \frac{C}{N \cdot C} = \frac{1}{N} \quad (\text{Formel A.2.13})$$

## Verschiebung und Invertierung der Fitnesswerte

### Info:

Die folgende Methode ist in der C++ Implementierung umgesetzt und wird angewendet, wenn die Bewertungsfunktion als Minimierungsproblem definiert ist.

Eine weitere Möglichkeit besteht darin, alle Fitnesswerte um den niedrigsten Fitnesswert in den positiven Bereich zu verschieben und anschliessend zu invertieren.

Symbol	Beschreibung
$g(\vec{I}_i)$	Fehler-Minimierungs-Funktion. $g(\vec{I}_i) \in \mathbb{R}$
$\delta$	Konstanter Wert zur Verschiebung der Fitnesswerte. $\delta = \varepsilon, \varepsilon > 0 \wedge \varepsilon \rightarrow 0$

Abb. 77: Symbole für die Fitnessfunktion Transformation

Bei der Anwendung des **GA** muss die Fehler-Minimierungs-Funktion  $g(\vec{I}_i)$  gegeben sein. Damit diese vom **GA** verwendet werden kann, muss sie in eine Maximierungs-Fitness-Funktion  $f(\vec{I}_i)$  umgewandelt werden. Bei der Umwandlung müssen jedoch einige Kompromisse in Kauf genommen werden:

- Die Verhältnisse der Fitnesswerte zueinander werden verändert.
- Das schlechteste Individuum erhält eine Fitness von 0 und hat somit keine Chance selektiert zu werden.
- Bei sehr ähnlichen Fehlerwerten steigt der Einfluss von  $\delta$  auf die Wahrscheinlichkeiten der einzelnen Individuen.

### Vorgehen bei der Umwandlung $g(\vec{I}_i) \rightarrow f(\vec{I}_i)$

Als erstes wird der niedrigste Fehlerwert bestimmt:

$$g_{\min} = \min_{i \in [0, N-1]} (g(\vec{I}_i)) = -2 \quad (\text{Formel A.2.14})$$

Wenn dieser negativ ist, werden alle Fehlerwerte um den Betrag des niedrigsten Wertes verschoben:

$$g_{\text{shift}}(\vec{I}_i) = \begin{cases} g(\vec{I}_i) - g_{\min} & \text{if } (g_{\min} < 0) \\ g(\vec{I}_i) & \text{sonst} \end{cases} \quad (\text{Formel A.2.15})$$

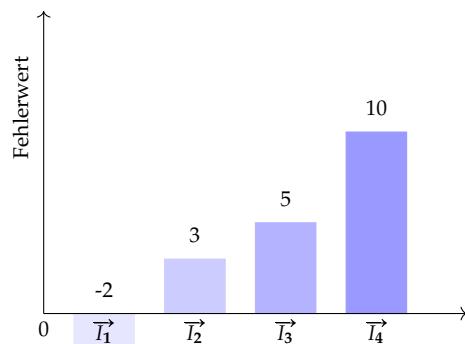


Abb. 78: Fehlerwerte der Individuen  $g(\vec{I}_i)$

Für den nächsten Schritt wird der maximale Fehlerwert benötigt:

$$g_{\text{shift\_max}} = \max_{i \in [0, N-1]} (g_{\text{shift}}(\vec{I}_i)) = 12 \quad (\text{Formel A.2.16})$$

Nun werden die Fehlerwerte am maximalen Fehlerwert invertiert, um eine Maximierungs-Fitness-Funktion zu erhalten.

$$g_{\text{invert}}(\vec{I}_i) = g_{\text{shift\_max}} - g_{\text{shift}}(\vec{I}_i) \quad (\text{Formel A.2.17})$$

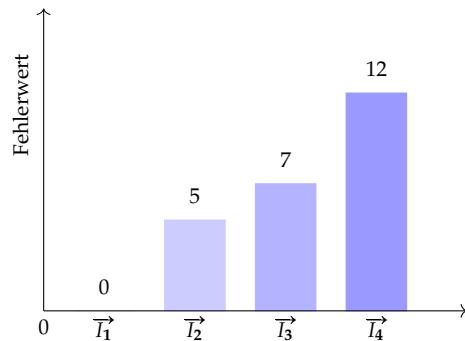


Abb. 79: Fehlerwerte der Individuen  $g_{\text{shift}}(\vec{I}_i)$

Im Fall, bei dem alle Individuen nahezu den gleichen Fehlerwert haben, würde  $f(\vec{I}_i)$  für alle Individuen nahe bei 0 liegen. Es sollte verhindert werden, dass alle Fitnesswerte den Wert 0 annehmen, da der Algorithmus damit nicht gut umgehen kann. Deshalb wird ein sehr kleiner positiver Wert  $\delta$  zu allen Fitnesswerten addiert.

$$f(\vec{I}_i) = g_{\text{invert}}(\vec{I}_i) + \delta \quad (\text{Formel A.2.18})$$

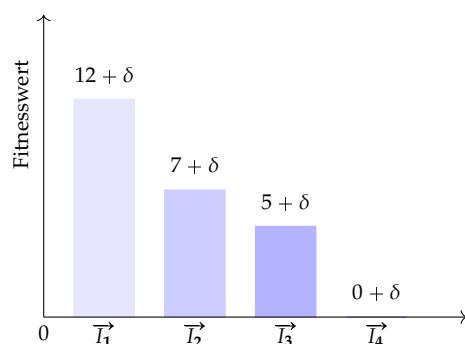


Abb. 80: Fitness der Individuen  $f(\vec{I}_i)$

## A.2.6 | Differential Evolution Algorithmus

### A.2.6.1 | Idee

Differential Evolution (DE) [42] ist ein evolutionärer Optimierungsalgorithmus. Eine Population von Lösungskandidaten entwickelt sich über mehrere Generationen hinweg, wobei bessere Lösungen eine höhere Überlebenschance haben. Der entscheidende Unterschied zu anderen evolutionären Algorithmen liegt in der Art und Weise, wie neue Kandidaten erzeugt werden. Differential Evolution nutzt die Differenz zwischen zwei zufällig ausgewählten Populationsmitgliedern, um Mutationen zu erzeugen. Diese Differenzvektoren werden skaliert und zu einem dritten Vektor addiert, wodurch neue Suchrichtungen im Lösungsraum entstehen. Dieser Mechanismus ermöglicht es dem Algorithmus, sich selbst an die Topologie des Optimierungsproblems anzupassen: In Regionen mit grosser Streuung der Population werden grössere Schritte unternommen, während in konvergenten Bereichen feinere Anpassungen erfolgen. Die Eleganz von Differential Evolution liegt in seiner Einfachheit und Robustheit. Mit nur wenigen Kontrollparametern kann der Algorithmus auf eine Vielzahl kontinuierlicher Optimierungsprobleme angewendet werden, von der Parameterschätzung über technische Designprobleme bis hin zu komplexen multidimensionalen Optimierungsaufgaben. Die Methode benötigt keine Ableitungsinformationen der Zielfunktion und kommt mit nichtlinearen und nicht-differenzierbaren Problemen gut zurecht.

### A.2.6.2 | Ablauf

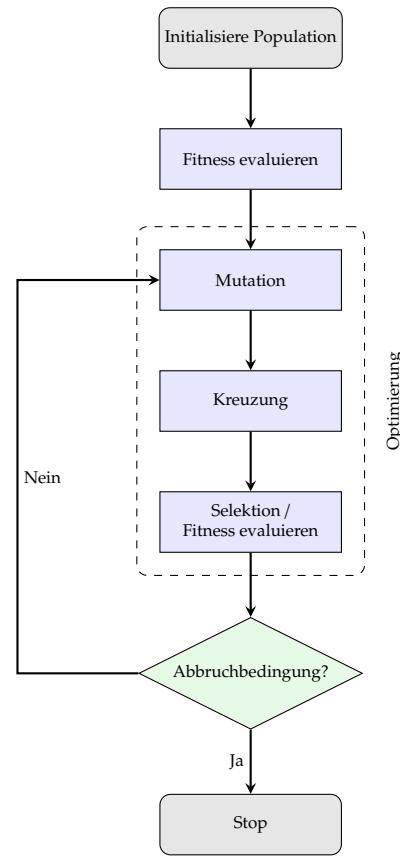


Abb. 81: Flussdiagramm des Differential Evolution Algorithmus

#### Initialisierung

Zu Beginn wird eine Population von  $N$  Lösungsvektoren zufällig im Suchraum verteilt. Jeder Vektor repräsentiert eine potenzielle Lösung des Optimierungsproblems und besteht aus  $D$  Parametern, wobei  $D$  die Dimensionalität des Problems darstellt. Die Anfangswerte werden typischerweise gleichverteilt innerhalb der definierten Grenzen des Suchraums gewählt.

#### Simulation und Bewertung

Für jedes Individuum in der Population wird eine Simulation mit den Parametern des Individuums durchgeführt. Während der Simulation wird das Verhalten des Individuums beobachtet und bewertet. Am Ende des Simulationsdurchlaufs hat jedes Individuum einen Fitness Wert, welcher angibt, wie gut das Individuum das Optimierungsziel erfüllt hat. Die **Fitnessfunktion** ist dabei spezifisch für das zu optimierende Problem und muss entsprechend definiert werden. Im Gegensatz zum **GA** kann der **DE** Algorithmus problemlos mit **Minimierungsproblemen** und **Maximierungsproblemen** umgehen.

### A.2.6.3 | Optimierung

Nachfolgend werden die einzelnen Schritte des Optimierungsprozesses, wie sie in Abb. 81 dargestellt sind, erläutert.

Für die Optimierung werden diverse Symbole verwendet, welche in der Tab. 22 zusammengefasst sind.

Symbol	Beschreibung
$N$	Anzahl Individuen in der Population
$D$	Anzahl Gene pro Individuum / Anzahl Parameter
$\vec{I}_i$	Individuum $i$ in der Population $\vec{I}_i \in \mathbb{R}^D$
$f(\vec{I}_i)$	Minimierungs-Fehler-Funktion. $f(\vec{I}_i) \in \mathbb{R}$
$k_{ij}$	Gen $j$ des Individuums $i$
$\alpha$	Mutationsstärke (Skalierungsfaktor für die Normalverteilung). $\alpha \in \mathbb{R}^+$
$\beta$	Kreuzungswahrscheinlichkeit. $\beta \in [0, 1]$

Tab. 22: Symbole für die nachfolgenden Erklärungen des Differential Evolution Algorithmus

### Mutation

Für jedes Individuum  $\vec{I}_i$  wird ein Mutationsvektor  $\vec{M}_i$  erzeugt. Für die Mutation werden zufällig drei verschiedene Vektoren aus der Population ausgewählt ( $\vec{I}_a$ ,  $\vec{I}_b$ ,  $\vec{I}_c$ ). Der Mutationsvektor wird berechnet, indem die gewichtete Differenz zwischen zwei dieser Vektoren zum dritten addiert wird. Der Skalierungsfaktor  $\alpha$  (typischerweise zwischen 0.5 und 1.0) kontrolliert die Amplifikation der Differenzvektoren und beeinflusst damit die Schrittweite der Suche.

$$\vec{M}_i = \vec{I}_c + \alpha \cdot (\vec{I}_b - \vec{I}_a) \quad (\text{Formel A.2.19})$$

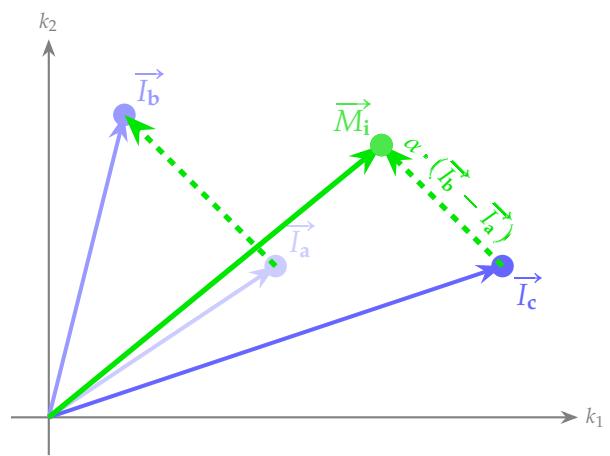


Abb. 82: Grafische Darstellung des Mutationsvektors  $\vec{M}_i$

### Kreuzung

Der Mutationsvektor  $\vec{M}_i$  wird mit dem Individuum Vektor  $\vec{I}_i$  kombiniert, um einen Testvektor  $\vec{T}_i$  zu erzeugen. Dabei wird für jedes Element im Vektor mit der Kreuzungswahrscheinlichkeit  $\beta$  (typischerweise zwischen 0.8 und 1.0) entschieden, ob der Wert vom Mutationsvektor  $\vec{M}_i$  oder vom Individuum Vektor  $\vec{I}_i$  übernommen wird. Dieser Schritt erhält erfolgreiche Merkmale aus der bestehenden Population, während gleichzeitig neue Variationen eingeführt werden.

$$k_{ij\_neu} = \begin{cases} m_{ij} & \text{wenn } \text{randG}(0,1) \leq \beta \\ k_{ij} & \text{sonst} \end{cases} \quad (\text{Formel A.2.20})$$

$\text{randG}(a, b)$  ist eine gleichverteilte Zufallsvariable im Intervall  $[a, b]$

### Selektion

Der Testvektor  $\vec{T}_i$  wird anhand der Zielfunktion bewertet und mit dem ursprünglichen Individuum  $\vec{I}_i$  verglichen. Bei Minimierungsproblemen ersetzt der Testvektor das Individuum  $\vec{I}_i$  nur dann, wenn er einen besseren (kleineren) Funktionswert aufweist.  $\vec{I}_{ineu} = \vec{T}_i$  Andernfalls bleibt der ursprüngliche Vektor in der Population erhalten.  $\vec{I}_{ineu} = \vec{I}_i$ . Dieses gierige Selektionsschema garantiert, dass die Population nie schlechter wird.

$$\vec{I}_{ineu} = \begin{cases} \vec{T}_i & \text{wenn } f(\vec{T}_i) < f(\vec{I}_i) \\ \vec{I}_i & \text{sonst} \end{cases} \quad (\text{Formel A.2.21})$$

### Abbruchbedingung

Der Optimierungsprozess wird wiederholt, bis eine Abbruchbedingung erfüllt ist. Dies kann eine vordefinierte Anzahl von Generationen sein, eine bestimmte Fitness-Schwelle, die erreicht werden muss, oder eine Kombination aus beiden. Sobald die Abbruchbedingung erfüllt ist, wird das beste Individuum in der Population als die optimale Lösung des Problems betrachtet.

## A.2.7 | Phasen- & Verstärkungsreserve Diagramm

Das Phasen- & Verstärkungsreserve Diagramm [33] in Abb. 84 zeigt die Verstärkungs- und Phasenreserven des optimierten geschlossenen Systems in logarithmischer Darstellung. Der Rote Bereich stellt die instabilen Verstärkungs- und Phasen- Paare dar. Der Grüne Bereich stellt die stabilen Verstärkungs- und Phasen- Paare dar. Das Diagramm gibt an, wie viel Verstärkung und Phase dem open loop System  $G$  hinzugefügt werden kann, bis das closed loop System instabil wird.

Ermittelt wird das Diagramm, indem das zu testende System  $G$  mit einem komplexen Faktor  $K$  multipliziert wird und anschliessend geprüft wird, ob sich ein Pol des geschlossenen Systems in der rechten Halbebene befindet.

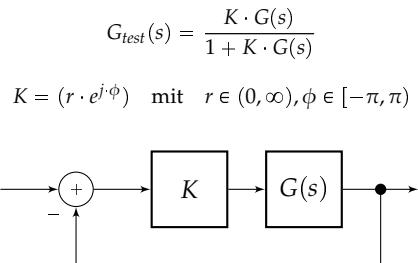


Abb. 83: Testsystem für Phasen- & Verstärkungsreserve Diagramm

Für jeden möglichen Wert von  $K$  wird die Stabilität des geschlossenen Systems überprüft. Wenn es stabil ist, wird der Punkt  $K$  in der komplexen Ebene mit grüner Farbe markiert, andernfalls mit rot. Weil es natürlich nicht möglich ist alle möglichen Werte von  $K$  zu testen, wird der Bereich in einem Raster getestet. Damit ein möglichst grosser Bereich abgedeckt und dargestellt werden kann, wird die komplexe Ebene für beide Achsen logarithmisch skaliert.

## A.2.8 | Totzeit- & Verstärkungsreserve Diagramm

Das Totzeit- & Verstärkungsreserve Diagramm in Abb. 86 zeigt die Verstärkungs- und Totzeitreserven des optimierten geschlossenen Systems. Der Rote Bereich stellt die instabilen Verstärkungs- und Totzeit- Paare dar. Der Grüne Bereich stellt die stabilen Verstärkungs- und Totzeit- Paare dar. Das Diagramm gibt an, wie viel Verstärkung und Totzeit dem open loop System  $G$  hinzugefügt werden kann, bis das closed loop System instabil wird.

Ermittelt wird das Diagramm, indem das zu testende System  $G$  mit einem komplexen Faktor  $K$  multipliziert wird und anschliessend geprüft wird, ob sich ein Pol des geschlossenen Systems in der rechten Halbebene befindet.

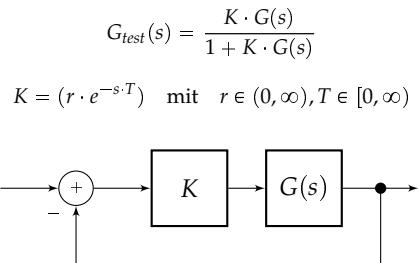


Abb. 85: Testsystem für Totzeit- & Verstärkungsreserve Diagramm

Für jeden möglichen Wert von  $K$  wird die Stabilität des geschlossenen Systems überprüft. Wenn es stabil ist, wird der Punkt  $K$  im Diagramm mit grüner Farbe markiert, andernfalls mit rot. Weil es natürlich nicht möglich ist alle möglichen Werte von  $K$  zu testen, wird der Bereich in einem Raster getestet. Damit ein möglichst grosser Bereich abgedeckt und dargestellt werden kann, wird die Verstärkung in dB dargestellt.

### A.2.8.1 | Wofür ist dieses Diagramm nun nützlich?

Wird der Vorwärtspfad des Regelkreises mit diesem Diagramm analysiert, lässt sich ablesen, wie viel Verstärkung und Totzeit das System verträgt.

In der Praxis bietet besonders dieses Diagramm mit der Totzeitinformation einen grossen Vorteil gegenüber vielen anderen Robustheitsanalysen. Eine Totzeit schleicht sich in realen Systemen sehr schnell ein und kann sich je nach System stark auf die Stabilität auswirken.

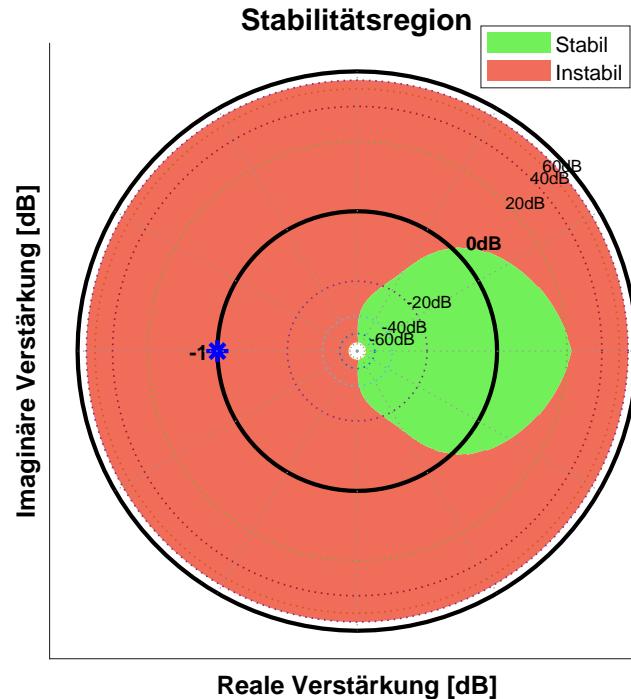


Abb. 84: Phasen- & Verstärkungsreserve Diagramm für System  $G$

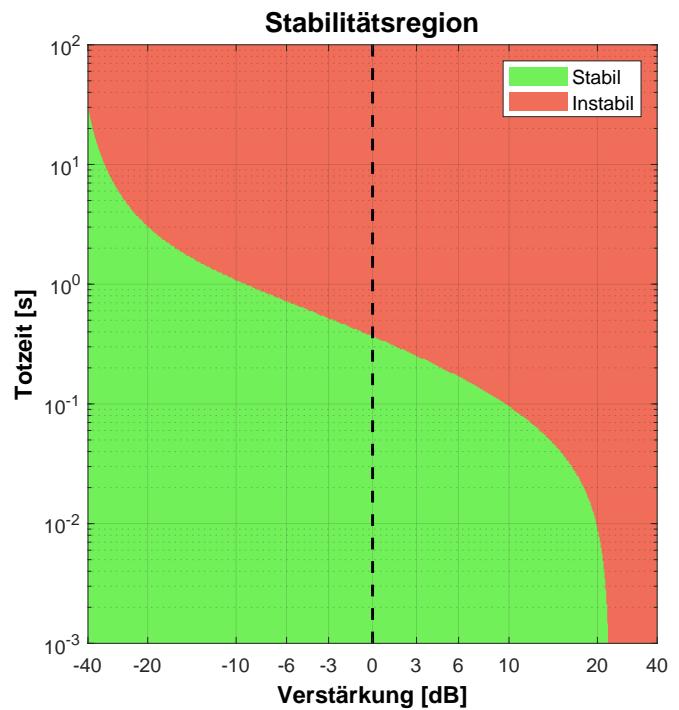


Abb. 86: Totzeit- & Verstärkungsreserve Diagramm für System  $G$

Dieses Diagramm erleichtert es, im Designprozess bereits frühzeitig zu erkennen, ob die realen Zeitverzögerungen zu Stabilitätsproblemen führen können oder nicht. Gegebenenfalls kann die benötigte Totzeitreserve als Anforderung in den Optimierungsprozess mit einbezogen werden.

## A.3 | Verzeichnisse

### A.3.1 | Begriffserklärung

#### Beobachter

Eine möglichst realistische Kopie des [Systems](#), welche mit Hilfe der gemessenen Ein- und Ausgangssignale den internen Zustand des Systems schätzt. Dies ist notwendig, wenn nicht alle Zustandsgrößen des Systems gemessen werden können.

#### Fehler

Der Fehler ist ein Mass für die Qualität oder Eignung einer Lösung in einem Optimierungsproblem. In der Regel wird die Fehlerfunktion so gestaltet, dass sie niedrige Werte bessere Lösungen darstellen. Synonyme dieses Begriffs sind:

- Fitness
- Score

#### Fitness

Die Fitness ist ein Mass für die Qualität oder Eignung einer Lösung in einem Optimierungsproblem. In der Regel wird die Fitnessfunktion so gestaltet, dass höhere Werte bessere Lösungen darstellen. Im Kontext dieser Arbeit wird die Fitnessfunktion jedoch als Minimierungsproblem formuliert, sodass niedrigere Werte für bessere Lösungen stehen. Synonyme dieses Begriffs sind:

- Fehler
- Score

#### Fitnessfunktion

Eine Funktion, die verwendet wird, um die Qualität oder Eignung einer Lösung in einem Optimierungsproblem zu bewerten. In der Regel wird die Fitnessfunktion so gestaltet, dass sie höhere Werte für bessere Lösungen liefert. Im Kontext dieser Arbeit wird die Fitnessfunktion jedoch als Minimierungsproblem formuliert, sodass niedrigere Werte für bessere Lösungen stehen. Die Fitnessfunktion wird als Sammelbegriff für die Fehler-Minimierungs-Funktion und die Maximierungs-Fitness-Funktion verwendet.

#### Gesamtsystem

Das Gesamtsystem bezeichnet in der Regelungstechnik die Kombination aus dem zu regelnden [Prozess](#) und dem Regler, der den Prozess steuert. Das Gesamtsystem umfasst alle Komponenten, die an der Regelung beteiligt sind, einschließlich Sensoren, Aktoren und Rückkopplungsschleifen.

#### Hyperparameter

Parameter, die nicht direkt im Optimierungsprozess angepasst werden, sondern vorab festgelegt werden. Diese Parameter beeinflussen das Verhalten des Optimierungsprozesses selbst und können Einfluss auf die Qualität der Lösung haben.

#### Individuum

In der Evolutionären Optimierung bezeichnet ein Individuum eine einzelne Lösung innerhalb einer Population. Ein Individuum wird durch einen Satz von Parametern (Genotyp) definiert, die das Verhalten oder die Eigenschaften der Lösung bestimmen. Die Qualität eines Individuums wird durch die Fitnessfunktion bewertet.

#### Integral Windup

Ein Phänomen bei PID-Reglern, bei dem der I-Anteil aufgrund von Sättigungen am Ausgang des Reglers übermäßig ansteigt, was zu einer Verzögerung oder Überschwingung im Regelverhalten führt. Der Integrator im Regler weiss nicht, dass der Ausgang gesättigt ist und somit der Aktor nicht mehr steuern kann. Dadurch kann es sein, dass der Fehler nie auf Null sinkt wodurch der Integrator immer weiter ansteigt. Das Problem dabei ist, wenn der Fehler plötzlich in die andere Richtung kippt, weil z.B. die Referenz verändert wird, wird der Regler am Ausgang immer noch den gesättigten Wert liefern bis der Integrator wieder entladen ist. Dies kann zu grossen Überschwingen und langen Einstellzeiten führen. Anti-Windup Mechanismen verhindern dieses Verhalten durch verschiedene Lösungsansätze.

#### Konvergenz

Der Prozess, bei dem eine Folge von Werten oder eine iterative Methode sich einem festen Punkt oder einer Lösung annähert. Im Kontext dieser Arbeit bezieht sich Konvergenz auf den Prozess, bei dem ein Optimierungsalgorithmus schrittweise bessere Lösungen findet, bis er eine optimale oder zufriedenstellende Lösung erreicht.

#### LTI

Abkürzung für "Linear Time-Invariant" (Linear Zeitinvariant). Ein LTI-System ist ein System, das sowohl linear als auch zeitinvariant ist. Linearität bedeutet, dass das System die Prinzipien der Superposition erfüllt, während Zeitinvarianz bedeutet, dass die Systemparameter sich nicht mit der Zeit ändern. LTI-Systeme sind in der Regelungstechnik und Signalverarbeitung weit verbreitet, da sie mathematisch gut handhabbar sind und viele nützliche Eigenschaften besitzen.

(Dies ist nur eine Übersichtsbeschreibung und entspricht nicht der genauen Definition von LTI-Systemen)

#### Maximierungsproblem

Ein mathematisches Problem, bei dem das Ziel darin besteht, die Parameter einer Funktion zu finden, welche zu dem grösstmöglichen Resultat der Funktion führen.

$$\max_{x \in \mathcal{X}} f(x) \quad (\text{Formel A.3.1})$$

Wobei:

- $x \in \mathbb{R}^n$  sind die Parameter (Entscheidungsvariablen)
- $\mathcal{X} \subseteq \mathbb{R}^n$  ist die zulässige Menge (Beschränkungsmenge)
- $f : \mathcal{X} \rightarrow \mathbb{R}$  ist die Zielfunktion
- $x^* \in \mathcal{X}$  ist die optimale Lösung

So dass:

$$f(x^*) \geq f(x), \quad \forall x \in \mathcal{X} \quad (\text{Formel A.3.2})$$

#### Minimalphasiges System

Ein System ist minimalphasig, wenn alle Nullstellen des [Systems](#) in der linken Halbebene der S-Ebene liegen. Ein nicht minimalphasiges System kann daran erkannt werden, dass die Sprungantwort zu Beginn des Sprunges in die entgegengesetzte Richtung des Sprunges verläuft.

## Minimierungsproblem

Ein mathematisches Problem, bei dem das Ziel darin besteht, die Parameter einer Funktion zu finden, welche zu dem kleinsten möglichen Resultat der Funktion führen.

$$\min_{x \in X} f(x) \quad (\text{Formel A.3.3})$$

Wobei:

- $x \in \mathbb{R}^n$  sind die Parameter (Entscheidungsvariablen)
- $X \subseteq \mathbb{R}^n$  ist die zulässige Menge (Beschränkungsmenge)
- $f : X \rightarrow \mathbb{R}$  ist die Zielfunktion
- $x^* \in X$  ist die optimale Lösung

So dass:

$$f(x^*) \leq f(x), \quad \forall x \in X \quad (\text{Formel A.3.4})$$

## Modell

Eine mathematische Darstellung eines realen [Systems](#) oder Prozesses, die dessen Verhalten und Eigenschaften beschreibt. Modelle sind immer vereinfachte Darstellungen der Realität und können unterschiedliche Komplexitätsgrade aufweisen. Modelle werden verwendet, um das Verhalten von Systemen zu analysieren, vorherzusagen und zu steuern. In der Regelungstechnik können Modelle in verschiedenen Formen vorliegen, wie z.B. Differentialgleichungen, Übertragungsfunktionen oder Zustandsraumdarstellungen.

## Modellwissen

Das Verständnis und die Kenntnisse über das Verhalten und die Eigenschaften eines [Systems](#), die zur Erstellung eines genauen [Modells](#) des Systems verwendet werden können. Modellwissen umfasst Informationen über die physikalischen Gesetze, die das System steuern, sowie über die Parameter und Dynamiken des Systems.

## Nichtlineares System

Ein System, bei welchem die Beziehung zwischen Ein- und Ausgang nicht linear im mathematischen Sinne ist. Nichtlinearitäten entstehen beispielsweise durch: Sättigungen, Hysteresen, Trigonometrische Funktionen, Totzeiten, etc. Da jedes reale System nichtlineares Verhalten aufweist, wird in der Praxis das [Modell](#) des Systems durch eine linearisierte Näherung beschrieben. Der Arbeitspunkt der Linearisierung ist dabei Anwendungsspezifisch zu wählen.

## Pol-Nullstelle-Kürzung

Methode bei welcher versucht wird ein Pol durch eine Nullstelle aufzuheben. In der Theorie verschwindet dadurch der Pol an dieser Position und kann durch einen Pol an einer anderen Stelle ersetzt werden. In der Praxis lässt sich diese Methode nur Näherungsweise umsetzen, da die exakte Position des Poles nie genau getroffen werden kann und sich der Pol auch verschieben kann mit der Zeit.

## Polplatzierung

Methode bei welcher die vorhandenen Pole der geschlossenen Regelstrecke, mit Hilfe der Zustandsrückführung, gezielt in der S-Ebene platziert werden können um das gewünschte Systemverhalten zu erreichen.

## Population

In der Evolutionären Optimierung bezeichnet die Population eine Menge von Individuen (Lösungen), die gleichzeitig optimiert werden. Jedes Individuum repräsentiert eine mögliche Lösung des Optimierungsproblems. Die Population entwickelt sich über Generationen hinweg durch Auswahl, Kreuzung und Mutation.

## Prozess

Ein technisches System oder eine Anlage, die physikalische, chemische oder biologische Vorgänge durchführt. In der Regel wird ein Prozess durch Eingangsgrößen beeinflusst und liefert Ausgangsgrößen als Resultat. In der Regelungstechnik wird ein Prozess oft als das zu regelnde System betrachtet, das durch einen Regler gesteuert wird, um ein gewünschtes Verhalten zu erreichen.

## Reglersynthese

Der Prozess der Entwicklung und Gestaltung eines Reglers, der die gewünschten Leistungsanforderungen für ein gegebenes System erfüllt.

## $\text{sgn}(x)$

Der Signum-Operator gibt das Vorzeichen einer Zahl zurück.

$$\text{sgn}(x) = \begin{cases} 1 & \text{wenn } x \geq 0 \\ -1 & \text{wenn } x < 0 \end{cases} \quad (\text{Formel A.3.5})$$

## Solver

Ein Werkzeug oder ein Algorithmus, das mathematische Probleme löst, entweder indem es eine exakte Lösung findet oder eine annähernde Lösung durch iterative Berechnungen liefert

## System

Ein System bezeichnet in der Regelungstechnik ein technisches Gebilde, das Eingangsgrößen in Ausgangsgrößen umwandelt. In den meisten Fällen wird ein System als ein [LTI](#) System modelliert.

## Zeitvariantes System

Ein System, bei welchem die Beziehung zwischen Ein- und Ausgang nicht konstant ist, sondern sich über die Zeit verändert. Alterung des realen [Systems](#) führt zur Veränderung des Systems. Z.B. mehr Reibung durch altes Schmiermittel oder Verschleiss von Teilen. In der Regel werden solche Systeme durch zeitkonstante [Modelle](#) angenähert.

### A.3.2 | Quellenverzeichnis

- [1] Open AI. *ChatGPT*. Large language model for code generation. 2025. URL: <https://chatgpt.com/> (besucht am 2025-12-13).
- [2] Krieg Alex. *Automatische Reglersynthese*. GitHub repository for the project. 2025. URL: [https://github.com/KROIA/Automatische\\_Reglersynthese](https://github.com/KROIA/Automatische_Reglersynthese) (besucht am 2025-12-14).
- [3] Krieg Alex. *QSFML EditorWidget*. C++ library. 2025. URL: [https://github.com/KROIA/QSFML\\_EditorWidget](https://github.com/KROIA/QSFML_EditorWidget) (besucht am 2025-12-13).
- [4] Anthropic. *Claude*. Large language model for code generation. 2025. URL: <https://claude.ai> (besucht am 2025-12-13).
- [5] Multiple Authors. *CMake*. Cross-platform build system. 2025. URL: <https://cmake.org/> (besucht am 2025-12-13).
- [6] Multiple Authors. *Git*. Distributed version control. 2025. URL: <https://git-scm.com/> (besucht am 2025-12-13).
- [7] Multiple Authors. *ImGui*. Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies (C++ library). 2025. URL: <https://github.com/ocornut/imgui> (besucht am 2025-12-15).
- [8] Multiple Authors. *ImGui*. Immediate Mode Plotting (C++ library). 2025. URL: <https://github.com/epezent/implot> (besucht am 2025-12-15).
- [9] Multiple Authors. *LaTeX Project*. Document preparation system. 2025. URL: <https://www.latex-project.org/> (besucht am 2025-12-13).
- [10] Multiple Authors. *Matlab*. Programming and numeric computing platform. 2025. URL: <https://www.mathworks.com/products/matlab.html> (besucht am 2025-12-13).
- [11] Multiple Authors. *Processing*. Flexible software sketchbook. 2025. URL: <https://processing.org/> (besucht am 2025-12-13).
- [12] Multiple Authors. *Qt Framework*. C++ framework for cross-platform application development. 2025. URL: <https://www.qt.io/development/qt-framework> (besucht am 2025-12-13).
- [13] Multiple Authors. *SFML*. Simple and Fast Multimedia Library (C++ library). 2025. URL: <https://www.sfm1-dev.org/> (besucht am 2025-12-15).
- [14] Multiple Authors. *Visual Studio Code*. Source-code editor. 2025. URL: <https://code.visualstudio.com/> (besucht am 2025-12-13).
- [15] Multiple Authors. *Visual Studio Community 2022*. Integrated development environment (IDE). 2025. URL: <https://visualstudio.microsoft.com/de/> (besucht am 2025-12-13).
- [16] GitHub. *GitHub Copilot*. Large language model for code generation. 2025. URL: <https://github.com/features/copilot> (besucht am 2025-12-13).
- [17] Inc The MathWorks. *Matlab Help Center: Control System Toolbox*. API documentation. 2025. URL: [https://ch.mathworks.com/help/control/index.html?s\\_tid=hc\\_product\\_card](https://ch.mathworks.com/help/control/index.html?s_tid=hc_product_card) (besucht am 2025-10-24).
- [18] Inc The MathWorks. *Matlab Help Center: Genetic Algorithm*. API documentation. 2025. URL: <https://ch.mathworks.com/help/gads/genetic-algorithm.html> (besucht am 2025-12-10).
- [19] Inc The MathWorks. *Matlab Help Center: genss*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/genss.html> (besucht am 2025-10-17).
- [20] Inc The MathWorks. *Matlab Help Center: linmod (Simulink)*. API documentation. 2025. URL: [https://ch.mathworks.com/help/simulink/slref/linmod.html?s\\_tid=srchtitle\\_support\\_results\\_1\\_linmod](https://ch.mathworks.com/help/simulink/slref/linmod.html?s_tid=srchtitle_support_results_1_linmod) (besucht am 2025-12-04).
- [21] Inc The MathWorks. *Matlab Help Center: looptune*. API documentation. 2025. URL: <https://www.mathworks.com/help/control/ref/dynamicsystem.looptune.html> (besucht am 2025-10-17).
- [22] Inc The MathWorks. *Matlab Help Center: Margins Goal*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/tuninggoal.margins.html> (besucht am 2025-12-07).
- [23] Inc The MathWorks. *Matlab Help Center: Optimierungsziele*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/tuning-goals-1.html> (besucht am 2025-10-23).
- [24] Inc The MathWorks. *Matlab Help Center: Overshoot Goal*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/tuninggoal.overshoot.html> (besucht am 2025-11-22).
- [25] Inc The MathWorks. *Matlab Help Center: PID Controller (Simulink)*. API documentation. 2025. URL: <https://ch.mathworks.com/help/simulink/slref/pidcontroller.html> (besucht am 2025-11-21).
- [26] Inc The MathWorks. *Matlab Help Center: Step Tracking Goal*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ug/step-tracking-goal.html> (besucht am 2025-11-22).
- [27] Inc The MathWorks. *Matlab Help Center: systune*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/dynamicsystem.systune.html> (besucht am 2025-10-17).
- [28] Inc The MathWorks. *Matlab Help Center: systune*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/dynamicsystem.systune.html#btjp1ym-1-HardReqs> (besucht am 2025-10-24).
- [29] Inc The MathWorks. *Matlab Help Center: systune*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/dynamicsystem.systune.html#btjp1ym-1-SoftReqs> (besucht am 2025-10-24).
- [30] Inc The MathWorks. *Matlab Help Center: tunablePID*. API documentation. 2025. URL: <https://ch.mathworks.com/help/control/ref/tunablepid.html> (besucht am 2025-12-12).
- [31] Inc The MathWorks. *Matlab Help Center: Visualize Tuning Goals*. API documentation. 2025. URL: <https://ch.mathworks.com/help/ug/visualize-tuning-goals.html> (besucht am 2025-11-22).
- [32] Luca Ballotta. *Complete Nyquist plot with logarithmic-scale real and imaginary parts of frequency response. Imaginary poles are efficiently handled*. GitHub repository. 2023. URL: <https://github.com/lucaballotta/NyquistLog> (besucht am 2025-11-29).
- [33] Brian Douglas. *Understanding Disk Margin | Robust Control, Part 2*. YouTube video. 2023. URL: <https://youtu.be/XazdN6eZF80?si=Hp-svlk5-Z5nZ00b> (besucht am 2025-11-29).
- [34] Aleksander Haber. *Clear Explanation of the Ziegler-Nichols PID Control Tuning Method (Second Method) with MATLAB codes + Model Assisted Tuning*. 2023. URL: <https://aleksandarhaber.com/model-assisted-ziegler-nichols-pid-control-tuning-method/> (besucht am 2025-10-16).
- [35] Aleksander Haber. *Clear Explanation of Ziegler-Nichols Method for PID Controller Tuning - With/without Plant Mode*. YouTube video. 2023. URL: <https://youtu.be/YYxkS1iFdV?si=QKK1kc-PUCkqVALs> (besucht am 2025-10-16).
- [36] Prof. Dr. Markus Kottmann. *Regelungstechnik 1 & 2 (SG E)*. PDF document. Skript, OST - Ostschweizer Fachhochschule. 2023.
- [37] Massimiliano Veronesi und Antonio Visioli. „On the selection of lambda in lambda tuning for PI (D) controllers“. In: *IFAC-PapersOnLine* 53.2 (2020), S. 4599–4604.
- [38] Heinz Mann u. a. *Einführung in die Regelungstechnik: Analoge und digitale Regelung, Fuzzy-Regler, Regel-Realisierung, Software*. Carl Hanser Verlag GmbH Co KG, 2018.

- [39] Seyedali Mirjalili. „Genetic algorithm“. In: *Evolutionary algorithms and neural networks: theory and applications*. Springer, 2018, S. 43–55.
- [40] DC Meena und Ambrish Devanshu. „Genetic algorithm tuned PID controller for process control“. In: *2017 International Conference on Inventive Systems and Control (ICISC)*. IEEE. 2017, S. 1–6.
- [41] Leonardo Trujillo u. a. „neat genetic programming: Controlling bloat naturally“. In: *Information Sciences* 333 (2016), S. 21–43.
- [42] Rainer Storn und Kenneth Price. „Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces“. In: *Journal of global optimization* 11.4 (1997), S. 341–359.
- [43] Robin Lauckner und Hoshang Kolivand. „Neat algorithm in autonomous vehicles“. In: *Available at SSRN 4644203* ().

## Aufgabenstellung

# Analyse und Test von Methoden zur automatischen Reglersynthese

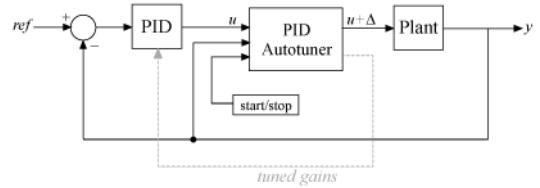
Bachelorarbeit für Alex Krieg

## Beschreibung des Problems und der Aufgaben

### Problemstellung und Ziele

Für die Entwicklung eines Reglers (Reglersynthese) gibt es eine Vielzahl von Methoden. Diese unterscheiden sich zum einen durch ihre mathematischen Herangehensweise und zum anderen durch das benötigte Modellwissen über den Prozess. Ausserdem gibt es Unterschiede wie automatisiert die Reglersynthese abläuft und wie die Methode parametrisiert wird. Der Fokus dieser Arbeit soll auf der Parametrisierung der Methode liegen also darauf wie und in welcher Form der Methode kommuniziert wird welches Verhalten der geschlossene Regelkreis haben soll. Diese Arbeit soll aufzeigen welche Methoden zur automatischen Reglersynthese existieren und für welche Anwendungsfälle diese geeignet sind. Die besten Methoden sollen dann implementiert und auf noch zu definierenden Prozessen erprobt werden. Zuletzt soll eine interaktive GUI erstellt werden mit der die unterschiedlichen Methoden auf einem Prozess angewendet werden können.

Das Ziel der Arbeit ist es eine Aussage zu treffen, welche Methoden, je nach Prozess und gewünschtem Verhalten, zur automatischen Reglersynthese geeignet sind, wie diese funktionieren und welche Vor- und Nachteile diese haben. Zur besseren Vermittlung des gewonnenen Wissens sollen die Methoden mit einer GUI auf einem Beispielprozess implementiert werden.



### Vorgesehene Arbeitsschritte

- Literaturrecherche zu Methoden zur automatischen Reglersynthese
- Clustering der Methoden nach benötigtem Modellwissen, Parametrisierung der Methode, Automatisierungsgrad
- Test der besten Methoden auf Prozessen
- Ausführliche Stellungnahme zu den Vor- und Nachteilen
- Erstellung einer GUI zur Anwendung der Methoden auf einem Prozess zur besseren Visualisierung des gewonnenen Wissens
- Dokumentation der Ergebnisse
- Vorstellung der Ergebnisse

## Angaben zur Durchführung der Arbeit

### Arbeitsplatz / Laborbetrieb:

Raum 2.001.

Für Hardwarebestellungen ist Bruno Vollenweider zuständig. Bestellungen dürfen nicht selbstständig getätigt werden.

### Literatur

Vorlesungsmaterialien, vom Betreuer empfohlene Artikel und Bücher, Recherche in Bibliotheken und Internet.

### Bericht

Die ausgeführten Arbeiten sind in einem schriftlichen Bericht zu dokumentieren. Der Bericht muss die unveränderte Aufgabenstellung, eine Zusammenfassung (1 Seite), einen Zeitplan der Arbeit (Planung), sowie einen abschliessenden Kommentar mit der Unterschrift des Studenten enthalten. Teil des Kommentars muss die Anti-Plagiats-Erklärung sein (siehe letzter Punkt: Weitere Informationen).

Der Bericht muss spätestens am Freitag 19.12.2025 um 23:59 per E-Mail an den Betreuer gesendet werden.

### Gitlab

Software-Code und Daten müssen auf Gitlab abgelegt werden. Eine intensive Nutzung von Gitlab zum Versionsmanagement sowie als Backup wird empfohlen.

### Zeitrahmen

Ausgabe der Arbeit: Montag 01.09.2025

Abgabe des Bericht: Freitag 19.12.2025

### Weekly Meeting

Vor jedem Meeting wird dem Dozenten eine Agenda zugeschickt.

Nach jedem Meeting wird dem Dozenten eine schriftliche Zusammenfassung des Meetings zugeschickt.

### Weitere Informationen

Lesen Sie folgende Dokumente, auf denen sich klärende Informationen zu verschiedenen Detailfragen der Arbeiten in Regelungstechnik finden:

[Arbeiten\\_in\\_Regelungstechnik.htm](#)

[Technische\\_Software.htm](#)

(Beide Dokumente liegen im Ordner \\ost.ch\dfs\bsc.et\work\labors\RT\_Public\Formales\_zu\_Arbeiten).

Rapperswil, 25.07.2025

Prof. Dr. Lukas Ortmann

## A.4 | Zeitplan

Aufgabe	Sept.				Okt.					Nov.			Dez.			
	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
Übersicht verschaffen & Projektplanung																
Literaturrecherche zu Methoden zur automatischen Reglersynthese																
Clustering der Methoden nach benötigtem Modellwissen,																
Parametrisierung der Methode, Automatisierungsgrad																
Test der besten Methoden auf Prozessen																
Erstellung einer GUI zur Anwendung der Methoden auf einem Prozess zur besseren Visualisierung des gewonnenen Wissens																
Zeitre serven																

Abb. 87: Grober Zeitplan für die Bachelorarbeit