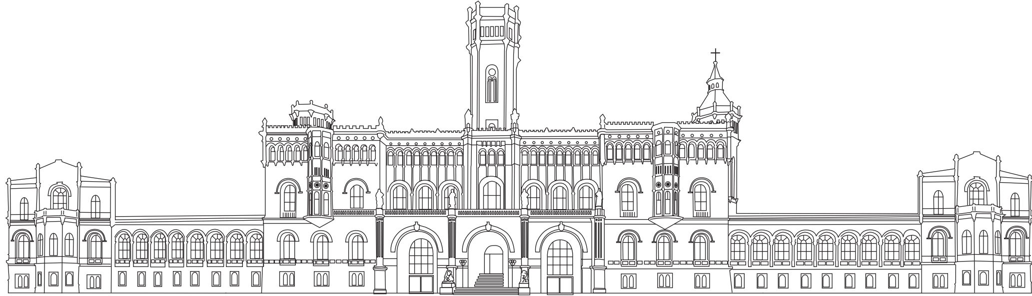


Leibniz Universität Hannover

Institut für Antriebssysteme
und Leistungselektronik

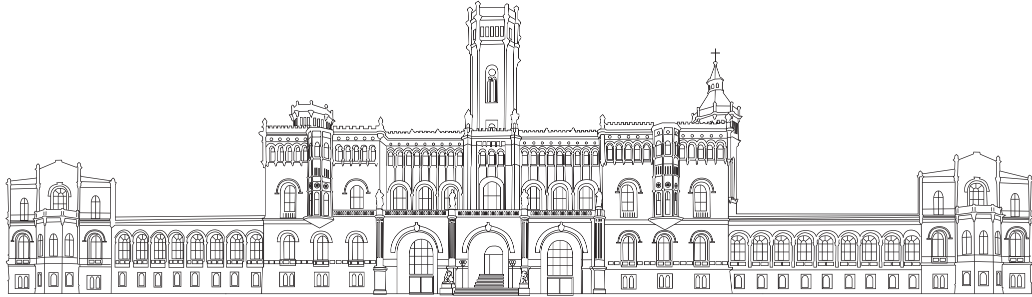
Fachgebiet Leistungselektronik
und Antriebsregelung

Modellierung der Orthogonalen Magnetisierung auf Basis empirischer Materialuntersuchungen

Masterarbeit

Ilka Elisa Marie Schulz
Matrikelnummer: 3132570

Betreuer: M. Sc. Siqi Lin
Erstprüfer: Jun.-Prof. Dr.-Ing. Jens Friebe
Zweitprüfer: Prof. Dr.-Ing. Axel Mertens



Gottfried Wilhelm Leibniz Universität Hannover

Institut für Antriebssysteme und Leistungselektronik

Ilka Elisa Marie Schulz

Ilka Elisa Marie Schulz

Modellierung der Orthogonalen Magnetisierung auf Basis empirischer Materialuntersuchungen

Leibniz Universität Hannover

Erste Auflage 2020
© Ilka Elisa Marie Schulz 2020
Alle Rechte vorbehalten
unveröffentlicht, für den internen Gebrauch bestimmt
Druck: Copyshop Druckservice GmbH, Hannover
Printed in Germany

Inhalt und Recherche
Ilka Elisa Marie Schulz

Gestaltung und Design
Ilka Elisa Marie Schulz

Betreuung
M. Sc. Siqi Lin
Jun.-Prof. Dr.-Ing. Jens Friebe

Besonderer Dank
Dr. Eyup Salih Tez

Für Dr. Leydecker

Aufgabenstellung

Neuartige, schnelle Leistungshalbleiter (Wide-Bandgap-Halbleiter) stellen eine breite Palette an Vorteilen gegenüber herkömmlichen Siliziumhalbleitern bereit, bspw. geringe Schaltverluste, niedrige Durchlasswiderstände, geringe Baugröße u. v. m. Die Nutzung ihres vollen Potentials ist bisher durch verschiedene Einschränkungen jedoch begrenzt, darunter die Unzulänglichkeiten der passiven Komponenten der Leistungselektronik. Verschiedene Verlustmechanismen treten insbesondere in den induktiven Komponenten auf (Skineffekt, Proximityeffekt, Kernresonanz). Diese und weitere Herausforderungen (Masse, Bauvolumen, Preis, EMV, Wellenwiderstand) beim Design induktiver Komponenten erfordern genuin neue Konzepte des Induktivitäts- und Transformatorendesigns, um mit den neuen Entwicklungen auf dem Gebiet der aktiven Leistungselektronik Schritt halten zu können.

Aufgaben

In der Arbeit soll ein bislang technisch ungenutztes Prinzip untersucht werden, bei dem ein ferromagnetisches Material von zwei orthogonal zueinander gerichteten Feldern durchflossen wird.

- Literaturrecherche zum aktuellen Stand der Wissenschaft, Technik und Wirtschaft.
- Entwicklung eines Modells für infinitesimale Gebiete unter idealisierten Bedingungen.
- Erweiterung des Modells auf reale Geometrien und Berücksichtigung realer physikalischer Effekte in typischen und geeigneten Materialien.
- Erweiterung des Modells auf reale Anwendungen und Definition einer Schnittstelle zur verbreiteten Theorie elektrischer Netzwerke.
- Simulative und experimentelle Überprüfung der erstellten Modelle.
- Reflexion der Ergebnisse und Abschätzung des praktischen Potenzials des Konzeptes.

Alle Quelltexte sind angemessen zu kommentieren und dokumentieren. Die Ergebnisse sind schriftlich festzuhalten.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Ort, Datum

Unterschrift

Über die Autorin

Ilka Elisa Marie Schulz studiert Elektrotechnik und Informationstechnik an der Leibniz Universität Hannover und fokussiert sich in ihrem Studium und ihren hilfswissenschaftlichen Tätigkeiten unter der Leitung von Jun.-Prof. Jens Friebe am Institut für Antriebssysteme und Leistungselektronik auf die Grundlagen der orthogonalen Magnetisierung.

In vorangegangenen hilfswissenschaftlichen Tätigkeiten, Abschlussarbeiten, Ehrenämtern und zuvor in internationalen Wettbewerben (insbesondere RoboCup) hat sie Erfahrung im Bereich der Mikrocontrollerprogrammierung, im Umgang mit Linux-Systemen und insbesondere mit der Entwicklung von objektorientierter Software in C++ gesammelt.

Neben ihrem Studium der Elektrotechnik verfolgt sie das Bachelorstudium im Fach Chemie, insbesondere mit dem Ziel, Zusammenhänge auf der Materialebene tiefer zu durchdringen und innovative Materialien und Herstellungsverfahren zu entwickeln.

Außercurricular setzt sie sich bei der Lise-Meitner-Gesellschaft für die Chancengerechtigkeit in akademischen Berufen ein, insbesondere für Frauen.

Vorwort

Da hat man sich beinahe fast gerade erst fürs Elektrotechnikstudium eingeschrieben – und schon sitzt man mit einer Pandemie vor der Haustür an der Masterarbeit.

– frei nach Loriot

Als abstraktes Ziel der vorliegenden Arbeit gilt ein besseres Verständnis eines Phänomens, welches nach Auffassung der Authorin sein technologisches Potential aus vielen Gründen kaum entfalten konnte, auch weil der grundlegende physikalische Mechanismus kaum modelliert ist und bestehende Modelle bislang kaum oder gar nicht praktisch validiert sind. Das Phänomen der orthogonalen Magnetisierung bedarf mikroskopischer und makroskopischer exakter und nähernder Modelle.

Um der Aufgabenstellung gerecht zu werden, sind Messungen und Simulationen benötigt, wobei aufgrund der dünnen Forschungslage noch nicht klar ist, wie diese aussehen sollen. Bspw. herrscht unter den Forschenden Uneinigkeit darüber, wie die Geometrie eines magnetischen Kerns lauten soll, der sich für die orthogonale Magnetisierung eignet. Auch die gewünschten Signalformen von magnetischen Spannungen und Flüssen sind keineswegs definiert. Schließlich ergibt sich das Problem, dass im großen Universum der verfügbaren Software zur Feldsimulation nirgends(!) Rücksicht auf die Effekte der orthogonalen Magnetisierung genommen wird. Sämtliche dieser Werkzeuge (Messaufbau, magnetische Kerne, Simulationssoftware, geeignete Versuche) werden im Rahmen dieser Arbeit entwickelt. Lediglich das Verfahren zur Herstellung gekrümmter Eisenbleche kann in der kurzen Bearbeitungszeit nicht abgeschlossen werden.

Im Laufe der Bearbeitungszeit stellt sich die Frage, ob mit einem einfachen Versuchsaufbau und dafür umso aufwendigerer Hardware (Signalgenerator, Vierkanaloszilloskop, Präzisionsmessgeräte) eine große Menge Messdaten erzeugt und diese mit Näherungsmethoden ausgewertet werden soll oder ob der Fokus nicht besser einem automatisierten Versuchsaufbau mit präziser Auswertungsmethodik gilt, wodurch aber kaum noch Zeit für die eigentlichen Messungen bliebe. Nachdem bereits zuvor die zweite Möglichkeit immer wieder präferiert war, ließen die Infektionsschutzmaßnahmen während der COVID-19-Krise kaum noch ein Umentscheiden zu. Deshalb fokussiert diese Arbeit wesentlich die Herstellung von Versuchsmaterialien, die Entwicklung eines automatisierten Messaufbaus und die Erarbeitung einer mächtigen Simulationssoftware. Die Anwendung dieser Werkzeuge wird nur soweit angerissen, wie es der kleine Umfang der zur Verfügung stehenden Materialien zulässt. Die Aufgabe, weitere magnetische Kerne zu beschaffen und mit den hier entwickelten Werkzeugen zu untersuchen, wird an nachfolgende Arbeiten weitergegeben.

Für den Leitfaden bietet immer wieder eine sehr ausführliche Dissertation aus dem Jahr 1977 [31] Orientierung, bspw. bei der Definition der Aufgabenstellung, der aktuell verfügbaren Modellierung und sogar bei den verwendeten Methoden und Versuchsgeometrien. Besonderer Dank gilt daher dem Author jener Schrift, Dr. Eyup Salih Tez.

Inhaltsübersicht

Einleitung 1
1 Grundlagen 5
2 Messaufbau 51
3 Verfahren zur Herstellung gekrümmter Eisenbleche 111
4 Software zur Netzwerk- und Feldsimulation 119
5 Versuch 139
6 Schlussbetrachtung 165
Literatur 167
A Daten-CD 171
B Programmcode 173
C Handbücher 249

Handbuch zum Softwarepaket „OrthoMeasure“ (englisch)

Handbuch zum Softwarepaket „OrthoSim“ (englisch)

Inhaltsverzeichnis

Einleitung 1

1 Grundlagen 5

1.1 Magnetische Netzwerke . . . 6
1.1.1 Grundlagen . . . 6
1.1.2 Diskretisierung . . . 11
1.1.3 Elemente . . . 14
1.1.4 Transformierte Elemente . . . 19
1.1.5 Topologien . . . 24
1.1.6 Lösung . . . 29
1.2 Magnetische Felder . . . 31
1.2.1 Mathematischer Hintergrund . . . 31
1.2.2 Lösung . . . 33
1.2.3 Einbindung in Netzwerktheorie . . . 33
1.3 Simulation . . . 35
1.3.1 Zeitabhängigkeit . . . 35
1.3.2 Raumabhängigkeit . . . 37
1.3.3 Lineare Gleichungssysteme . . . 39
1.4 Modellierung . . . 42
1.4.1 Keine Interaktion . . . 42
1.4.2 Allgemeine Interaktion . . . 43
1.4.3 Modell von Tez . . . 44
1.4.4 Weitere Modelle . . . 47
1.5 Anwendung . . . 49

2 Messaufbau 51

2.1 Signalgenerator und Messsystem . . . 53
2.1.1 Eigenschaften . . . 53
2.1.2 Einstellungen . . . 53
2.1.3 Programmierung . . . 55

2.2 Schaltungen . . . 57
2.2.1 Tiefpass . . . 62
2.2.2 Kleiner Impedanzwandler . . . 66
2.2.3 Subtrahierer . . . 69
2.2.4 großer Impedanzwandler . . . 73
2.2.5 Messwandler . . . 76
2.2.6 Addierer . . . 83
2.2.7 Über-/Unterspannungsschutz . . . 84
2.3 Programm . . . 87
2.3.1 Hardwarezugriff . . . 87
2.3.2 Werkzeuge . . . 94
2.3.3 Übersetzung . . . 107
2.4 Zusammenfassung . . . 108

3 Verfahren zur Herstellung gekrümmter Eisenbleche 111
3.1 Stand der Technik . . . 113
3.2 Entwicklung eines Verfahrens . . . 114

4 Software zur Netzwerk- und Feldsimulation 119
4.1 Struktur . . . 120
4.1.1 Problemrepräsentation . . . 120
4.1.2 Felder . . . 125
4.1.3 Netzwerke . . . 127
4.2 Modelle . . . 130
4.3 Erweiterbarkeit und Wiederverwendbarkeit . . . 133
4.3.1 Strukturelle Kompatibilität . . . 133
4.3.2 Die Klasse *TMagneticComponent* . . . 135
4.3.3 Werkzeuge . . . 135

5 Versuch 139
5.1 Vorabversuche . . . 140
5.2 Aufbau . . . 141
5.2.1 Magnetischer Rückschluss . . . 141
5.2.2 Geometrien . . . 144
5.2.3 Materialien . . . 147
5.2.4 Simulation . . . 149
5.3 Durchführung . . . 152
5.4 Ergebnisse . . . 154
5.4.1 Zeitveräufe der Netzwerkgrößen . . . 154

5.4.2 Magnetisierungskennlinien 155
5.4.3 Modellierte Magnetisierungskennlinien 157
5.4.4 Stationäre Simulation 159
5.4.5 Zeittransiente Simulation 159
5.5 Auswertung 162

6 Schlussbetrachtung **165**

Literatur **167**

A Daten-CD **171**

B Programmcode **173**

C Handbücher **249**

Abbildungsverzeichnis

1.1 Magnetische Anordnung . . . 9
1.2 Reluktanz . . . 16
1.3 Magnetische Spannungsquelle . . . 16
1.4 Flussquelle . . . 17
1.5 Schaltsymbol einer Transreluktanz . . . 18
1.6 Zerlegung einer realen Reluktanz . . . 20
1.7 Netzwerk mit Verlustreluktanz . . . 21
1.8 Netzwerkgrößen bei Verlustreluktanz . . . 22
1.9 Transformierte Kapazität . . . 24
1.10 Aufbau und Verhalten einer Induktivität . . . 27
1.11 Einfaches elektrisches Netzwerk mit einer Induktivität . . . 27
1.12 Aufbau und Verhalten einer Vormagnetisierten Induktivität . . . 28
1.13 orthogonale Magnetisierung . . . 29
1.14 Orthogonale Magnetisierung und keine Interaktion . . . 43
1.15 Beispiel für eine räumlich zweidimensionale Reluktanz . . . 45
1.16 Graphische Konstruktion der resultierenden Flussdichte bei zwei gegebenen magnetischen Felstärken [31] . . . 46
1.17 Beispiele für magnetische Kerngeometrien mit orthogonaler Magnetisierung [31] . . . 47
1.18 Allgemeines Beispiel für einfache orthogonale Flussinteraktion [31] . . . 48

2.1 Impedanz eines „Jumper-Kabels“ . . . 59
2.2 Fotographie des gesamten Messaufbaus . . . 60
2.3 Schematische Übersicht über den Messaufbau . . . 61
2.4 Welligkeit des Signalausgangs . . . 62
2.5 Schaltplan des Tiefpasses . . . 63
2.6 Fotografie des Tiefpasses . . . 63
2.7 Eingangsimpedanz des Tiefpasses bei offenen Ausgangsklemmen . . . 64
2.8 Transferfunktion des Tiefpasses bei offenen Ausgangsklemmen . . . 65
2.9 Schaltplan des kleinen Impedanzwandlers . . . 66
2.10 Fotografie des kleinen Impedanzwandlers . . . 67
2.11 Transferfunktion des kleinen Impedanzwandlers bei offenen Ausgangsklemmen . . . 68

2.12 Schaltplan des Subtrahierers . . . 70
2.13 Fotografie des Subtrahierers . . . 70
2.14 Eingangsimpedanz des Sutrahierers bei offenen Ausgangsklemmen . . . 72
2.15 Schaltplan des Impedanzwandlers . . . 74
2.16 Fotograpie des großen Impedanzwandlers . . . 74
2.17 Transferfunktion des großen Impedanzwandlers bei einer ohmschen Last von 50 Ω . . . 75
2.18 Schaltplan des Messwandlers . . . 77
2.19 Fotografie des Messwandlers . . . 78
2.20 Impedanz des Shuntwiderstands (1 Ω). . . . 79
2.21 Transferfunktion des Messwandlers bei der Spannungsmessung . . . 80
2.22 Transferfunktion des Messwandlers bei der Strommessung . . . 81
2.23 Transferfunktion des Messwandlers bei der Strommessung . . . 82
2.24 Schaltplan des Über-/Unterspannungsschutzes . . . 84
2.25 Fotografie des Über-/Unterspannungsschutzes . . . 85
2.26 Transferfunktion des Über-/Unterspannungsschutzes . . . 86
2.27 Klassendiagramm von *TInterface* . . . 88
2.28 Klassendiagramm von *TOutput* . . . 90
2.29 Klassendiagramm von *TInput* . . . 91
2.30 Klassendiagramm von *TImpedanceMeasureSetup* . . . 92
2.31 Klassendiagramm von *TSocket* und *TScope* . . . 93
2.32 Sinusschwingung, mit langsamem Ausgang an 47 Ω Last generiert . . . 97
2.33 Sinusschwingung, mit schnellem Ausgang an 47 Ω Last generiert . . . 98
2.34 Zweipolcharakterisierung eines ohmschen Widerstands . . . 101
2.35 Zweipolcharakterisierung eines ohmschen Widerstands . . . 102
2.36 Schematische Übersicht über den improvisierten Messaufbau . . . 108
2.37 Ergebnis einer Vierpolcharakterisierung an einem Transformator . . . 110

3.1 Geometrie aus ebenem und Geometrie aus gekrümmten Blech . . . 111
3.2 3D-gedruckter Halbtorus aus ABS . . . 114
3.3 Graphitisierte Oberfläche . . . 115
3.4 Fotografie vom Galvanisierungsbecken . . . 116
3.5 Galvanisiertes Objekt . . . 117

4.1 Klassendiagramm der Problemrepräsentation . . . 121
4.2 Klassendiagramm von *TElement* . . . 122
4.3 Klassendiagramm von *TSolvable* . . . 124
4.4 Klassendiagramm von *TFdmElement* . . . 125
4.5 Klassendiagramm von *TFdm* . . . 128
4.6 Berechnung eines Netzwerkelementes durch angebundene Feldsimulation . . . 134

4.7 Klassendiagramm von *TMagneticComponent* und *TMagneticComponentOrtho* 136

5.1 Fotografie des Ferrit-U-Kerns 142
5.2 Impedanz der Wicklung auf dem primären Ferrit-U-Kern 143
5.3 Fotografie des „Single-Sheets“ 144
5.4 Schnittskizze eines torusförmigen Kerns 146
5.5 „Single-Cross“ 146
5.6 Fotografie des „Single-Crosses“, an Rückschlüsse angeklemmt 147
5.7 Messung am „Single-Sheet“ aus NO10 in Walzrichtung 155
5.8 Messung am „Single-Cross“ (schmal) aus NO10 mit primärem Fluss orthogonal zur Walzrichtung 156
5.9 Magnetisierungskennlinie des „Single-Sheets“ aus NO10 in Walzrichtung 157
5.10 Magnetisierungskennlinie des „Single-Sheets“ aus NO10 orthogonal zur Walzrichtung 157
5.11 Scheinbare Magnetisierungskennlinie des „Single-Crosses“ (schmal) aus NO10 orthogonal zur Walzrichtung 158
5.12 Simuliertes Klemmenverhalten auf der Primärseite des „Single-Crosses“ (schmal) aus NO10 . . 160
5.13 Simulative Nachbildung der Messung am „Single-Cross“ (schmal) aus NO10 mit primärem Fluss orthogonal zur Walzrichtung 161

Tabellenverzeichnis

2.1 Vergleich zweier kontroverser Philosophien zum Messaufbau 58
2.2 Verdeutlichung des Prinzips bei der Glättung der Quantisierung 106

3.1 Vergleich verschiedener Oberflächenbeschichtungen 116

4.1 Vergleich der Strukturen von Netzwerken und Feldern 120
4.2 Implementierte Modelle 131

5.1 Parameter für die improvisierte Vierpolmessung 152

Abkürzungsverzeichnis

ABS Acrylnitril-Butadien-Styrol-Copolymer

ADC Analog-Digital-Umsetzer, engl. Analog Digital Converter

API Programmierschnittstelle, engl. Application Programming Interface

ASCII Amerikanischer Standard-Code für den Informationsaustausch, engl. American Standard Code for Information Interchange

BMP Windows Bitmap

CPU Prozessor eines Computers, engl. Central Processing Unit

CSV Kommaseparierte Werte, engl. Comma-separated values

DGL Differentialgleichung

EDV Elektronische Datenverarbeitung

ESD Elektrostatische Entladung, engl. Electrostatic Discharge

FDM Finite-Differenzen-Methode

FEM Finite-Elemente-Methode

FPGA im Feld prgrammierbare Logikgatteranordnung, engl. Field Programmable Gate Array

FVM Finite-Volumen-Methode

GCC Compiler-Suite des GNU-Projekts, engl. GNU Compiler Collection

GPIO Allzweckeingabe/-ausgabeport, engl. General Purpose Input/Output

IC Integrierte Schaltung, engl. Integrated Circuit

IDE Integrierte Entwicklungsumgebung, engl. Integrated Development Environment

IP Internetprotokoll, engl. Internet Protocol

KI Künstliche Intelligenz

LAN Lokales Netzwerk, engl. Local Area Network

LGS Lineares Gleichungssystem

LUT Umsetzungstablle, engl. Lookup Table

OPV Operationsverstärker

PC Personal-Computer

PDGL Partielle Differentialgleichung

PM Permanentmagnet

PWM Pulsdauermodulation, engl. Pusle Width Modulation

RAM Hauptspeicher, engl. Random Access Memory

RDP Netzwerkprotokoll für den Zugriff auf entfernte Bildschirminhalte, engl. Remote Desktop Protocol

RTL Laufzeitbibliothek, engl. Runtime Library

SCPI Standardbefehle für Programmierbare Instrumente, engl. Standard Commands for Programmable Instruments

SFTP Dateitransferprotokoll über SSH, engl. SSH File Transfer Protocol

SSH Secure Shell

SST Einzelblechtester, engl. Single-Sheet-Tester

TCP Übertragungssteuerungsprotokoll, engl. Transmission Control Protocol

USB Universelles Serielles Bussystem, engl. Universal Serial Bus

Symbolverzeichnis

Größe	Bezeichnung
a	zeitabhängige Größe
$\overline{a}$	Mittelwert einer zeitabhängigen Größe
$\hat{a}$	Spitzenwert einer zeitabhängigen Größe
A	zeitunabhängige Größe oder Effektivwert
$\mathcal{A}$	laplacetransformierte Größe
$\boldsymbol{a}$	vektorielle Größe
$\boldsymbol{A}$	Matrix
A_0	Offset oder Standardwert
$\boldsymbol{e}_x$	Einheitsvektor in Richtung der x-Achse
$f(t)$	allgemeine zeitabhängige Funktion
$f(...)$	allgemeine von verschiedenen Größen abhängige Funktion
$h(A, B) = 0$	allgemeiner impliziter Zusammenhang zwischen zwei Größen
O	allgemeiner Operator
t	Zeit
ω	Kreisfrequenz einer periodischen Größe
s	Strecke
s_{eff}	effektive Länge unter Berücksichtung realer Effekte
A	Fläche
∂A	Rand einer Fläche
V	Volumen
∂V	Rand eines Volumens
$\boldsymbol{E}$	elektrisches Feld
$\boldsymbol{D}$	elektrische Flussdichte
$\boldsymbol{J}$	elektrisches Strömungsfeld
κ	elektrische Leitfähigkeit
$\boldsymbol{H}$	magnetische Feldstärke
$\boldsymbol{B}$	magnetische Flussdichte

ϕ_{m}	magnetisches Skalarpotential
μ	Permeabilität
μ_0	Vakuumpermeabilität
μ_{r}	relative Permeabilität
μ_{diff}	differentielle Permeabilität

U	elektrische Spannung
U_{q}	Spannung einer Spannungsquelle
I	elektrischer Strom
I_{q}	Strom einer Stromquelle
R	elektrischer Widerstand
R_{V}	verlustverursachender Widerstand
R_{i}	Innenwiderstand
$R_1 \vert\vert R_2$	Gesamtwiderstand der Parallelschaltung zweier Widerstände
$R_1 + R_2$	Gesamtwiderstand der Serienschaltung zweier Widerstände
G	elektrischer Leitwert
L	Induktivität
L_{diff}	differentielle Induktivität
C	Kapazität
N	Windungszahl einer stromführenden Wicklung
V	magnetische Spannung
Φ	magnetischer Fluss (keine Unterscheidung zu verkettetem Fluss)
R_{m}	Reluktanz / magnetischer Widerstand
$R_{\mathrm{m,i}}$	Innenreluktanz
G_{m}	Permeanz / magnetischer Leitwert
$G_{\mathrm{m,i}}$	Innenpermeanz
A_1	Größe auf der Primärseite eines Transformators / Paraformators
A_2	Größe auf der Sekundärseite eines Tansformators / Paraformators
A_{pre}	Größe im vormagnetisierenden magnetischen Netzwerkelement

$\mathcal{A}_{\mathrm{m}}$	A-Charakteristik eines magnetischen Netzwerkelements
$\mathcal{R}_{\mathrm{m}}$	Reluktanz-Charakteristik eines magnetischen Netzwerkelements
$\mathcal{R}_{\mathrm{m,lin}}$	linearer Anteil einer Reluktanz
$\mathcal{R}_{\mathrm{m,nl}}$	nichtlinearer Anteil einer Reluktanz
$\mathcal{R}_{\mathrm{m,V}}$	verlustbehafteter Anteil einer Reluktanz

$G_{\mathrm{m,lin}}$	linearer Anteil einer Permeanz
$G_{\mathrm{m,nl}}$	nichtlinearer Anteil einer Permeanz
$G_{\mathrm{m,V}}$	verlustbehafteter Anteil einer Permeanz

Einleitung

The references is sufficient. However, the quality of references is not very high.

– zweite*r Gutachter*in der Ersteinreichung von [24]

Im Laufe der vergangenen Jahre wurde umfangreicher Aufwand in die Forschung an leistungselektronischen Schaltungen und ihren Elementen investiert, um Leistung, elektrische Spannung und Taktfrequenz zu erhöhen und gleichzeitig das Bauvolumen und den Materialaufwand zu verringern. Es hat sich ein Optimierungswettlauf ergeben, bei dem Leistungshalbleiter die Schaltungen mit zuvor ungeahnten Frequenzen betreiben. Dabei wurde allerdings das Offensichtliche übersehen: leistungselektronische Schaltungen bestehen nicht nur aus Halbleitern, sondern auch aus passiven Komponenten, insbesondere Induktivitäten und Transformatoren.

Die für die Auslegung von moderner Leistungselektronik obligatorische ganzheitliche Betrachtung einer Schaltung stößt derzeit an die Grenzen der physikalischen Beschreibungsmöglichkeiten für magnetische Komponenten, bspw. Induktivitäten. Die Beteiligung an der Forschung ist international extrem dünn und es mangelt an grundlegenden(!) Modellen zur Beschreibung des Zusammenhang zwischen magnetischen Feldstärken und Flussdichten. Dieser Mangel ist nicht überraschend: bei den üblichen Anwendungen von magnetischen Komponenten handelt es sich um Induktivitäten und Transformatoren. Bei beiden lassen sich starke Vereinfachungen treffen, insbesondere ist die räumliche Orientierung des Magnetfeldes zeitlich konstant. Die Bedingung zur Vereinfachung und die Beschränkung auf lediglich diese beiden Arten von Komponenten sind weder physikalisch bedingt, noch sind sie technisch sinnvoll. Es sind zahlreiche weitere Anwendungen mit magnetischen Feldern denkbar, welche in dieser Arbeit nur angerissen werden (siehe Abschnitt 1.5, S. 49). Denn bevor Erfindungen mit magnetischen Aufbauten gemacht werden, bedarf es einer Theorie zur Beschreibung solcher.

Während der Bearbeitungszeit der vorliegenden Masterarbeit haben einige Wissenschaftler vom KDEE-EVS der Universität Kassel, der Prüfer der Arbeit und die Autorin gemeinsam eine Recherche zum Stand der Wissenschaft durchgeführt, wobei zwar einige Ansätze und sogar Umsetzungen von Erfindungen mit magnetischen Aufbauten (außer Induktivitäten und Transformatoren) gefunden sind, aber nahezu keine Theorie. Das Ergebnis ist in [24] festgehalten und sollte dem Prüfer vorliegen. Das einzige Werk, welches sich intensiv mit der Theorie auseinandersetzt ist [31] aus dem Jahr 1977 und beinhaltet lediglich einen sehr einfachen Ansatz mit begrenzter Gültigkeit.

Der ernüchternde bisherige Forschungsstand wird in dieser Arbeit als Chance verstanden, an der Theorie zur Beschreibung beliebiger magnetischer Anordnungen zu forschen:

Zunächst werden die zum weiteren Verständnis notwendigen **Grundlagen** erläutert. Sie umfassen insbesondere die magnetische Netzwerktheorie und Feldtheorie. Dabei werden mächtige Möglichkeiten zur Nutzung dieser Grundlagen gezeigt, wie bspw. die Transformation zwischen elektrischen und magnetischen Netzwerkelementen. Es wird auch auf die bestehenden Modellierung magnetischer Komponenten eingegangen, wie sie in [31] vorgeschlagen sind. Außerdem wird ein Überblick über die mathematischen Methoden gegeben, welche für die Nutzung der Theorie notwendig sind, bspw. numerische Lösungsverfahren für Differentialgleichungen (DGL).

Um eine Theorie aufzustellen und diese anhand empirischer Messungen zu validieren, bedarf es einiger Laborgeräte und Materialien:

Es wird ein **Messaufbau** entwickelt, welcher elektrische Signale generieren und messen kann. Er ist auf die durchzuführenden Versuche optimiert und ermöglicht deshalb ein erhebliches Maß an Automatisierung. Damit werden der Autorin etliche Labortage erspart und es steht ein System zur Verfügung, welches in nachfolgenden Arbeiten wiederverwendet werden kann.

Die Herstellung bestimmter Geometrien aus ferromagnetischen Material bedarf eines **Verfahrens zur Herstellung gekrümmter Eisenbleche**. Weil in der kurzen Bearbeitungszeit das Verfahren nicht abschließend entwickelt werden kann, muss auf bestimmte Geometrien verzichtet werden. Das betrifft insbesondere den in [31] vorgeschlagenen Torus.

Selbst ein fundiertes Materialmodell bietet keinen Nutzen, wenn es keine Anwendung in einer **Software zur Netzwerk- und Feldsimulation** findet. Sie stellt die Schnittstelle zwischen einem Materialmodell, welches die intrinsischen Größen im Raum betrachet, und einem Netzwerkelementmodell, welches das Klemmenverhalten beschreibt, bereit. Zusätzlich kann sie sogar ein Netzwerkelementmodell in ein Netzwerk integrieren und somit eine komplette Anwendung simulieren! Die Software ist so flexibel programmiert, dass sie mit geringem Aufwand in nachfolgenden Arbeiten erweitert werden kann, sogar um ggf. notwendige Abhängigkeiten zu anderen Domänen der Physik, wie der Mechanik.

Nachdem alle Werkzeuge entwickelt sind, kann ein **Versuch** zur Beobachtung magnetischer Felder begonnen werden. Darin werden gezielt Situationen provoziert, welche sich mit den bislang zur Verfügung stehenden Modellen nicht oder nur unzureichend beschreiben lassen. Auf der Basis der Ergebnisse wird das in [31] vorgeschlagene Modell überprüft. Ein eigenes Modell ist im Rahmen dieser Arbeit nicht entwickelt und kann entsprechend nicht experimentell validiert werden.

Schließlich werden die Ergebnisse reflektiert. Weil sich ein großer Teil der Arbeit auf den Rahmen und die Werkzeuge rund um den Versuch beziehen und wenig eigentliche Versuche durchgeführt sind, wird insbesondere auf diese Werkzeuge und das theoretische Rahmenkonstrukt eingegangen.

Um der Erwartung der Lesenden gerecht zu werden, sei bereits vorab erwähnt, woher das Schlagwort „Orthogonale Magnetisierung" rührt, welches im Titel der Arbeit verwendet ist. Im folgenden Kapitel wird detailliert darauf eingegangen, wie magnetische Aufbauten modellierbar sind und es wird eine Definitionslücke gefunden. Diese tritt stets dann auf, wenn das Feld in einem magnetischen Material seine räumliche Orientierung

ändert. Sie tritt auch dann auf, wenn zwei (oder mehr) magnetische Quellen (bspw. Permanentmagneten oder von elektrischen Strömen durchflossene Wicklungen) Felder verursachen, die nicht parallel oder antiparallel zu einander stehen. Eine kurze Übersicht über die Geschichte des Begriffs ist in [31] gegeben.

Für die eiligen Lesenden

Sollte der*die Leser*in lediglich an den Ergebnissen der Arbeit interessiert sein, welche einen wissenschaftlichen Mehrwert im Bereich der orthogonalen Magnetisierug liefern, sei er*sie auf die Seiten 154 bis 165 verwiesen.

An den entwickelten Werkzeugen Interessierte werden beim Messaufbau (siehe Abschnitt 2.3.2, S. 94) und bei der Software (siehe Abschnitt 4.3, S. 133) fündig. Sie seien außerdem auf die abstraktesten Definitionen magnetischer Netzwerkelemente hingewiesen (siehe Abschnitt 1.1.4, S. 19).

1 Grundlagen

Eine gute Theorie ist das Praktischste, was es gibt.

– Gustav Robert Kirchhoff

In den späteren Kapiteln werden einige mathematische Modelle genutzt, um magnetische Experimente aufzubauen und zu modellieren. Dieses Kapitel bietet einen Überblick über die Modelle und ihre Lösungsmethoden und gibt einen Ausblick auf mögliche Anwendungen.

Zunächst werden magnetische Netzwerke, ihre allgemeinen Regeln und ihre Elemente vorgestellt, wobei Wert darauf gelegt wird, ein physikalisch korrektes und gleichzeitig ingenieurstechnisch nützliches Werkzeug der „Netzwerkmodellierung" zu entwickeln.

Anschließend wird an den Grenzen der Möglichkeiten von magnetischen Netzwerken angesetzt und es werden Methoden zur Berechnung von magnetischen Feldern präsentiert, um auch besonders schwierige Probleme zu beschreiben, insbesondere Geometrien mit inhomogenen Feldverteilungen.

Mithilfe der Netzwerk- und der Feldbeschreibung lässt sich ein großes Universum an magnetischen Versuchsaufbauten modellieren. Um die Lösung dieser Modelle zu finden, wird auf einige wenige numerische Methoden zur Lösung von DGL eingegangen, welche für die vorliegende Arbeit von besonderer Relevanz sind.

Nachdem scheinbar sämtliche magnetischen Netzwerke und Felder modellier- und berechenbar sind, wird auf eine Definitionslücke in der Modellierung hingewiesen. Dabei handelt es sich um das Kernthema der Arbeit, die sog. orthogonale Magnetisierung. Die spärliche Sammlung der bisherigen Versuche, dieses Phänomen zu beschreiben, wird kurz beschrieben und es wird Kritik an den Modellen geübt, insbesondere sind dafür zahlreiche, offensichtlich unzulässige Annahmen notwendig.

Schließlich wird dieses Kapitel mit einer kurzen Aussicht abgerundet, welche technischen Anwendungen mithilfe der vorgestellten Methoden beschreibbar sind. Es ist bemerkenswert, dass sich darunter einige finden, welche bislang kaum oder gar nicht technisch genutzt sind.

1.1 Magnetische Netzwerke

Zunächst wird der Aufbau magnetischer Netzwerke beschrieben, wobei untersucht wird, wie sowohl die Netzwerktopologie als auch das Verhalten der Netzwerkelemente das Gleichungssystem determinieren. Nach einem kurzen Überblick über die Besonderheiten magnetischer Materialien werden alle wichtigen Netzwerkelemente vorgestellt und deren Verhalten untersucht. Anschließend werden diese Elemente zu Netzwerken verschaltet und typische Topologien charakterisiert. Schließlich werden Verfahren zur Lösung der Gleichungssysteme erklärt und evaluiert.

1.1.1 Grundlagen

Die Beschreibung magnetischer Phänomene ist mit erheblichem Aufwand verbunden, wenn die Felder berechnet werden müssen. Häufig lassen sich erhebliche Vereinfachungen treffen, indem das betrachtete Volumen in Teilvolumina mit homogenen Feldbildern unterteilt wird. Weil jedes Feld stückweise als homogen angenähert werden kann und in technischen Anwendungen häufig absichtlich oder unabsichtlich große Teilvolumina homogener Felder existieren, bietet sich diese Methode an. Die Darstellung geschieht mithilfe von Netzwerken, sodass die Methoden der Netzwerktheorie zur Beschreibung und Ableitung eines Gleichungssystems genutzt werden können.[1]

Ein magnetisches Netzwerk ist ein zusammenhängender Graph, dessen Knoten mit b gerichteten[2] Kanten verbunden sind. Ein Zug, dessen sämtliche inneren Knoten den Grad zwei aufweisen, heißt Zweig. Ein geschlossener orientierter[3] Zug aus jeweils über einen Knoten mit den im Zug benachbarten Kanten verbundenen Kanten heißt Masche. Jede j-te dieser Kanten, wobei $j \in \{1; ...; b\}$ hat zwei Zustandsgrößen:

- magnetischer Fluss Φ_j, $[\Phi] = \mathrm{Vs}$
- magnetische Spannung V_j, $[V] = \mathrm{A}$

Die Gesamtheit der Zustandsgrößen aller Kanten definiert die Zustandsgrößen des Netzwerkes:

$$\boldsymbol{Z} = \begin{pmatrix} \boldsymbol{V} \\ \boldsymbol{\Phi} \end{pmatrix} \qquad \boldsymbol{V} = \begin{pmatrix} \boldsymbol{V}_1 \\ ... \\ \boldsymbol{V}_b \end{pmatrix} \qquad \boldsymbol{\Phi} = \begin{pmatrix} \boldsymbol{\Phi}_1 \\ ... \\ \boldsymbol{\Phi}_b \end{pmatrix} \tag{1.1}$$

Die Zusammenhänge zwischen den Elementen von $\boldsymbol{Z}$ sind durch den impliziten Zusammenhang $h(\boldsymbol{Z}(\boldsymbol{V}, \boldsymbol{\Phi})) = 0$ definiert. Meist lassen sich Vereinfachungen treffen, sodass dieser vollständige Zusammenhang in mehre voneinander unabhängige oder jeweils nur in eine Richtung abhängige Zusammenhänge umgeformt werden

[1] Fundiertes Wissen über die Grundlagen der Netzwerktheorie und über die Grundlagen der Elektrodynamik wird bei den Lesenden vorausgesetzt. Ansonsten wird die Literatur in [22] empfohlen.

[2] Die Richtung der Kanten ist lediglich für die Festlegung von Vorzeichen bei der Berechnung notwendig. Sie hat keine physikalische Bedeutung.

[3] Die Orientierung des geschlossenen Zugs ist lediglich für die Festlegung vn Vorzeichen bei der Berechnung notwendig.

kann. Jeder dieser einzelnen Zusammenhänge, welcher aus β Kanten besteht und somit 2β Knoten, ggf. aber weniger als 2β verschiedene, verbindet, wird 2β-Pol oder allgemein Netzwerkelement genannt, für den ein expliziter Zusammenhang formuliert werden kann. Im Allgemeinen sind insbesondere Netzwerkeelemente mit $2\beta \in \{2; 4\}$ interessant (siehe Abschnitt 5.2, S. 141). Ein Zweipol mit der rückkopplungslosen Abhängigkeit von den Zustandsgrößen anderer Elemente $f_j(\boldsymbol{Z})$ weist den folgenden expliziten Zusammenhang, das sogenannte Kennlinienfeld auf: [4]

$$\Phi_j = \Phi_j(V_j, f_j(Z), ...) \tag{1.2}$$

Bei einem Vierpol besteht ein ähnlicher Zusammenhang, allerdings mit vier Zustandsgrößen. Eine mögliche Schreibweise für das Kennlinienfeld lautet:

$$\boldsymbol{\Phi}_{j,k} = \begin{pmatrix} \boldsymbol{\Phi}_j \\ \boldsymbol{\Phi}_k \end{pmatrix} = \boldsymbol{\Phi}_{j,k}\left(\begin{pmatrix} \boldsymbol{V}_j \\ \boldsymbol{V}_k \end{pmatrix}, \boldsymbol{f}_{j,k}(\boldsymbol{Z}), ...\right) \tag{1.3}$$

Durch diese Zusammenhänge reduziert sich die Anzahl der Freiheitsgrade des Netzwerks wegen des Satzes der impliziten Funktionen meist um die Dimension der Zusammenhänge. Dabei wird vorausgesetzt, dass alle Kennlinienfelder invertierbar (und entsprechend stetig) sind und dass keine Abhängigkeit von externen Variablen besteht.[5] Sollten externe Abhängigkeiten bestehen, steigt die Anzahl der Freiheitsgrade entsprechend um deren Dimension abzgl. der Dimension der zusätzlich zu erfüllenden Gleichungen. Die Anwendbarkeit des Satzes der impliziten Funktionen soll an dieser Stellte nicht weiter vertieft werden. Typischerweise hat das Kennlinienfeld eines 2β-Pols den Rang β (ebenso sämtliche in dieser Arbeit betrachteten Netzwerkelemente), sodass die Anzahl der Freiheitsgrade halbiert wird.

Weil ein Netzwerk damit noch nicht eindeutig definiert ist, sind weitere Gleichungen notwendig, die sich aus der Topologie des Netzwerks ergeben. Für magnetische genauso wie für elektrische Netzwerke gelten die Kirchhoffschen Sätze. Demnach sind die Summe aller m in einen Knoten fließenden Flüsse und die Summe aller n entlang einer Masche anliegenden magnetischen Spannungen jeweils gleich null: [18]

$$\sum_{k=1}^{m} \boldsymbol{\phi}_k = \mathbf{0} \qquad \sum_{k=1}^{n} \boldsymbol{v}_k = \mathbf{0} \tag{1.4}$$

Für den Beweis der Kirchhoffschen Gesetze müssen zunächst die Netzwerkgrößen aus den Feldgrößen abgeleitet werden. Der Fluss durch eine Fläche ist gleich dem vektoriellen Integral der Flussdichte durch die Fläche und die magnetische Spannung entlang eines Pfades ist gleich dem vektoriellen Integral der magnetischen

[4] Ein solches Kennlinienfeld kann von weiteren Größen abhängen, welche nicht im Zustand des Netzwerkes enthalten sind, bspw. der Zeit oder Größen aus anderen Domänen der Physik.

[5] Eine Abhängigkeit $\boldsymbol{f}(\boldsymbol{Z})$ verstößt nicht gegen den Satz, weil dieser auf die Gesamtheit aller Zusammenhänge der Netzwerkelemente anzuwenden ist.

Feldstärke entlang dieses Pfades:

$$\Phi := \iint_{(\boldsymbol{A})} \boldsymbol{B}\,\mathrm{d}\boldsymbol{A} \qquad V := \int_{(\boldsymbol{l})} \boldsymbol{H}\,\mathrm{d}\boldsymbol{l} \tag{1.5}$$

Der Knotensatz folgt anschließend direkt aus dem Gaußschen Gesetz für Magnetfelder:

$$\mathrm{div}(\boldsymbol{B}) = 0 \quad \Leftrightarrow \quad \oiint_{(\partial \boldsymbol{V})} \boldsymbol{B}\,\mathrm{d}(\partial \boldsymbol{V}) = 0 \quad \Leftrightarrow \quad \sum_{\partial V} \Phi = 0 \tag{1.6}$$

Der Maschensatz folgt aus dem Durchflutungsgesetz:[6]

$$\mathrm{rot}\boldsymbol{H} = \boldsymbol{J} \quad \Leftrightarrow \quad \oint_{(\partial \boldsymbol{A})} \boldsymbol{H}\mathrm{d}(\partial \boldsymbol{A}) = \sum_j N_j I_j \tag{1.7}$$

Während das elektrische Feld wirbelfrei ist,[7] weist die statische magnetische Feldstärke einen Rotor gleich der elektrischen Stromdichte auf (siehe *Gl. 1.7*). Um dennoch den Kirchhoffschen Maschensatz in *Gl. 1.4* herleiten zu können, muss die eingeprägte magnetischen Spannung definiert werden, um die triviale Definition in *Gl. 1.5* zu erweitern:[8]

$$\boldsymbol{V}^e := -N\boldsymbol{I}\boldsymbol{e}_{H,I}\boldsymbol{e}_l \tag{1.8}$$

Dabei zeigt der Einheitsvektor $\boldsymbol{e}_{H,I}$ in die Richtung des H-Feldes, welches durch den Strom I verursacht wird [9] und die physikalisch messbare magnetische Spannung ergibt sich als Differenz der Spannung in *Gl. 1.5* und der eingeprägten Spannung in *Gl. 1.8*. Anschaulich kann die eingeprägte magnetische Spannung als die (negative) Wirkung einer magnetischen Quelle aufgefasst werden. Die Definition ist analog zur eingeprägten elektrischen Spannung, welche in [18] ausführlich behandelt wird. Im Folgenden wird mit dem Formelzeichen V die physikalische magnetische Spannung bezeichnet.

Unter Verwendung dieser Definition ist die integrale Darstellung des Durchflutungsgesetzes in *Gl. 1.7* äquivalent zum Maschensatz in *Gl. 1.4*:

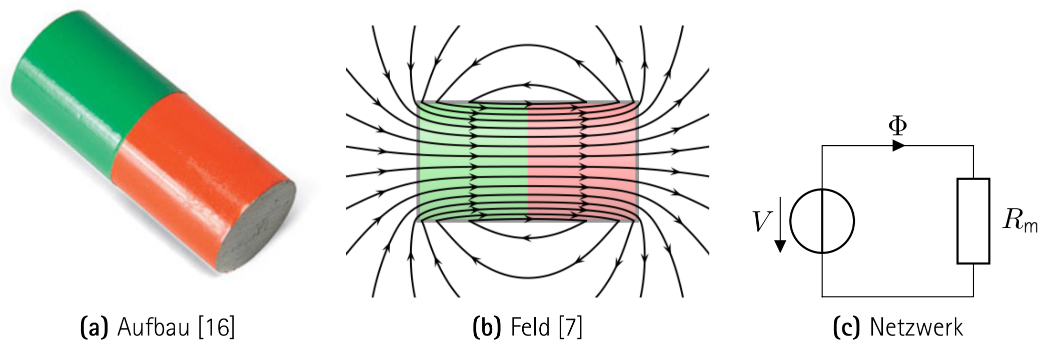
$$\sum_{\partial A} V = 0 \tag{1.9}$$

[6] Das Gesetz ist vereinfacht dargestellt, weil in dieser Arbeit Verschiebungsströme vernachlässigt werden. Diese treten erst bei Frequenzen weit jenseits der hier betrachteten auf. Alternativ kann die Stromdichte als Summe aus Stromdichte durch Ladungstransport und durch die zeitliche Veränderung des D-Feldes aufgefasst werden. Damit ist die Gültigkeit aller folgenden Überlegungen auch für schnelle zeitliche Änderungen bzw. hohe Frequenzen gewährleistet.

[7] Diese Vereinfachung gilt für statische und langsam veränderliche Magnetfelder, wie sie in dieser Arbeit betrachtet werden. Der Rotor des $\boldsymbol{E}$-Feldes $rot(\boldsymbol{E})$ kann ähnlich modelliert werden, wie im Folgenden der Rotor des $\boldsymbol{H}$-Feldes $rot(\boldsymbol{H})$. Dies ist allerdings lediglich für elektrische und nicht für magnetische Netzwerke relevant und wird nicht weiter erläutert.

[8] Die Definition basiert auf der Magnetiserung $\boldsymbol{M}$, auf welche der Übersicht halber nicht weiter eingegangen wird. Sie ist in [22] hergeleitet.

[9] Bei Permanentmagneten (PM) ist der Strom I durch die Summe aller resultierenden atomaren Spins verursacht. [6]



(a) Aufbau [16] **(b)** Feld [7] **(c)** Netzwerk

Abb. 1.1: Magnetische Anordnung. Aufbau bestehend aus Stabmagnet und umgebender Luft (links), graphische Darstellung des B-Feldes mit Flusslinien (mittig) und Modellierung als magnetisches Netzwerk mit Quelle und Reluktanz (rechts).

Den aufmerksamen Lesenden fällt auf, dass die Integrationsvariablen in den integralen Darstellungen der beiden ausgewählten Maxwell-Gleichungen jeweils vektorielle Größen beschreiben, die Netzwerkgrößen zumindest elektrischer Netzwerke hingegen skalar sind. Aufgrund des Superpositionsprinzips in elektrischen Netzwerken lassen sich die Orientierungen der Feldgrößen relativ zu den Integrationsvariablen in elektrischen Netzwerken hinreichend durch die Bezugsrichtungen der Strompfeile und Maschenumlaufpfeile ausdrücken. Die skalaren (auch möglicherweise negativen) elektrischen Größen der elektrischen Spannung und des elektrischen Stroms werden damit lediglich um einen eindimensionalen Einheitsvektor (-1) bzw. (1) erweitert und genügen so der netzwerkischen Vereinfachung der vektoriellen Naturgesetzen. In magnetischen Materialien gilt das Superpositionsprinzip wegen mangelnder Linearität der Materialien nicht (siehe Abschnitt 1.1.2, S. 11) und die Einheitsvektoren für den Bezug der Netzwerkgrößen behalten die gleiche Dimension wie die Integrationsvariablen in den Maxwell-Gleichungen. Es wäre zwar möglich, alle Netzwerkgrößen vektoriell zu notieren und zu berechnen[10], allerdings lässt sich diese Komplexität einfach mithilfe einer weiteren Abstraktionsebene beherrschen.[11] Die Netzwerkgrößen bleiben alle skalar und die Orientierung der Felder wird lediglich in den Netzwerkelementen selbst berücksichtigt. Weil die Eigenschaften der meisten magnetischen Elemente lageunabhängig sind, muss in den Elementen nur die relative Orientierung der einzelnen Knoten zum Körper des Elements beachtet werden.

Schließlich ergibt sich ein Gleichungssystem zur Beschreibung des Netzwerks, welches ggf. über- oder unterbestimmt sein kann. In dieser Arbeit werden allerdings ausschließlich bestimmte Gleichungssysteme betrachtet. Es kann außerdem externe Abhängigkeiten besitzen, bspw. von Größen anderer Domänen der Physik oder einfach von Größen elektrischer Netzwerke. Selbstverständlich kann es mit anderen Gleichungssystemen kombiniert

[10] Dabei würden die Kirchhoffschen Gesetze weiterhin gelten, allerdings wären die Ergebnisse der Summen gleich dem Nullvektor. Auch geometrische Eigenschaften des Netzwerks müssten berücksichtigt werden, bspw. wäre eine Ecke, aus welcher der Fluss in einem anderen Raumwinkel austritt als in sie eintritt durch eine Rotationsmatrix zu beschreiben. Die zu beschreibenden Netzwerke hätten also nicht nur mehr Dimensionen, sondern auch mehr Elemente.

[11] Auch hier gilt der Sinnspruch von David J. Wheeler: „All problems in computer science can be solved by another level of indirection, except for the problem of too many layers of indirection." Die Anzahl der Abstraktionsebenen sollte den Lesenden bis hier hin allerdings zumutbar sein.

werden, um bspw. das Verhalten eines komplexen Systems[12] zu beschreiben. Sowohl die Charakterisierung als auch die Lösung dieser Gleichungssysteme wird erst thematisiert, sobald die Elemente und Topologien hinreichend untersucht sind (siehe Abschnitt 1.1.6, S. 29).

Man beachte, dass bei einem besonders unglücklichen Aufbau des Netzwerks ein widersprüchliches Gleichungssystem entstehen kann. Wenn in einem Zweig bspw. zwei Zweipole mit den Kennlinien $\Phi_1(V_1)$ und $\Phi_2(V_2)$

$$\Phi_1(V_1) = const. \qquad \Phi_2(V_2) = const. \qquad \Phi_1 \neq \Phi_2 \tag{1.10}$$

existieren, widerspricht dies dem Knotensatz und es existiert keine Lösung. Wenn in einer Masche zwei Zweipole mit den Kennlinien $V_1(\Phi_1)$ und $V_2(\Phi_2)$

$$V_1(\Phi_1) = const. \qquad V_2(\Phi_2) = const. \qquad V_1 \neq V_2 \tag{1.11}$$

und keine weiteren Elemente existieren, über welchen eine magnetische Spannung abfällt, steht dies im Widerspruch zum Maschensatz und es existiert ebenfalls keine Lösung. Es existieren zahlreiche weitere Möglichkeiten, Widersprüche im Gleichungssystem zu erzeugen, wie bspw. Kennlinien mit Definitionslücken, Unstetigkeiten oder fehlender Invertierbarkeit. Diese Mängel können alle vermieden werden, wenn alle Kennlinien bijektiv sind. Alternativ können alle Kennlinien im Arbeitspunkt bijektiv und zusätzlich der Anfangszustand des Netzwerkes bekannt sein. Letztere Bedingung wird im Folgenden durch die geeignete Modellierung der Netzwerkelemente (siehe Abschnitt 1.1.3, S. 14) auf Basis geeigneter Materialmodelle (siehe Abschnitt 1.1.2, S. 11) erreicht, sodass in dieser Arbeit keine Netzwerke durch widersrüchliche Gleichungssysteme beschrieben werden.

Zuletzt soll noch auf die Grenzen der Modellierung realer Anordnungen als magnetische Netzwerke eingegangen werden. Es ist bereits erwähnt, dass die Netzwerkelemente geeignete Repräsentationen ihrer physikalischen Vorbilder darstellen müssen, um mathematische Widersprüche zu vermeiden. Gleichzeitig stellen die Netzwerkelemente als diskrete Elemente eine Zusammenfassung eines ganzen physikalischen Volumens dar und können somit unmöglich alle Teilvolumina vollständig abbilden. Diese Diskretisierung ist zum Zwecke der Abstraktion gewünscht, sodass stets eine Abwägung notwendig ist, zwischen einer detailreichen und einer abstrakten Modellierung. Obwohl diese beiden Konzepte sich nicht grundsätzlich ausschließen, bietet die magnetische[13] Netzwerktheorie keine Werkzeuge zu deren Vereinigung.

Mögliche Lösungsansätze zur Behebung dieses Konflikts beinhalten die Anpassung der Modelle, auf welche hier nicht weiter eingegangen werden soll. Alternativ kann der reale Aufbau verändert werden. Dabei ist stets eine Homogenisierung der Felder und eine geringe Streuung anzustreben, indem geeignete Geometrien sowie Materialien mit einer hohen Permeabilität angewandt werden.

[12] Ein komplexes System folgt lokal verschiedenen Gesetzen. Bspw. kann ein System aus einem magnetischen und einem elektrischen Netzwerk bestehen, die miteinander interagieren, aber jeweils eigenen Gesetzen folgen. Solche Schnittstellen werden noch mehrfach thematisiert (siehe Abschnitt 1.1.2, S. 11; siehe Abschnitt 1.1.3, S. 14; siehe Abschnitt 1.1.5, S. 24).

[13] ebenso die elektrische, thermische, mechanische, etc.

1.1.2 Diskretisierung

Der Zusammenhang $h(\mathbf{\Phi}, \boldsymbol{V}) = 0$ eines 2β-Pols wird durch die extrinsischen (d. h. Geometrie und Verknüpfung mit netzwerkexternen und netzwerkelementexternen Größen) und intrinsischen Eigenschaften (d. h. Material) bestimmt.

Geometrie

Innerhalb des Volumens, welches als ein Netzwerkelement modelliert wird, müssen im Allgemeinen alle Felder homogen sein, sowohl die magnetische Feldstärke $\boldsymbol{H}$ als auch die magnetische Flussdichte $\boldsymbol{B}$, Materialparameterfelder wie die Permeabilität $\boldsymbol{\mu}$ und die elektrische Leitfähigkeit $\boldsymbol{\kappa}$ und sämtliche weiteren Größen.[14] Eigtl. muss lediglich der Zusammenhang $h(\boldsymbol{B}, \boldsymbol{H}) = 0$ konstant sein, durch die Nichtlinearität (s. u.) lässt sich dies meist nur durch Erfüllung der hinreichenden Bedingung erreichen, dass sämtliche Felder homogen sind. Im Sinne einer ingenieurmäßigen Näherung werden allerdings kleine Abweichungen ignoriert.

Externe Größen

Weil es nicht immer einfach ist, solche Gebiete homogener Felder zu finden, können ggf. nicht trivial geformte Gebiete mithilfe von Computersimulationen identifiziert werden. Diese erfordern meist einen initial hohen Rechenaufwand, die Modellierung des realen Aufbaus mithilfe eines Netzwerks ermöglicht anschließend aber einfachere weitergehende Berechnungen. Dabei werden typischerweise eine detaillierte Berechnung des Feldbildes mit der Finite-Elemente-Methode (FEM) angestellt und an den Feldlinien die einzelnen Teilvolumina definiert. Die Länge, Breite und Tiefe der einzelnen Volumina ist davon abhängig, in welchem Maße die Felder in der Nähe als homogen angenähert werden können. Bei dieser Methode ist allerdings zu berücksichtigen, dass das Feldbild wegen ggf. vorhandener Nichtlinearität (s. u.) von der Feldstärke und ggf. auch von externen Größen abhängt. Die Geometrie der Teilgebiete hängt ebenfalls davon ab. Auf die genaue Funktionsweise und die Anwendung dieser Methode wird an dieser Stelle nicht weiter eingegangen, allerdings wird die Methode in vereinfachter Form später angewandt (siehe Kapitel 6, S. 139).

Sollte eine Abhängigkeit von einem anderen Netzwerkelement bestehen, kann ein neues Element definiert werden, welches das Verhalten beider einzelnen Elemente sowie deren Interaktion beschreibt. Die Anzahl der Pole sowie die Anzahl der Gleichungen addieren sich entsprechend. Interaktionen mit den Zustandsgrößen anderer Netzwerkelemente werden in diesem Abschnitt deshalb ignoriert, ohne eine Definitionslücke entstehen zu lassen. Wenn ein Netzwerkelement von netzwerkexternen Größen abhängt, bspw. der Temperatur, muss das Gleichungssystem des Netzwerks um entsprechend viele Gleichungen erweitert werden, um eine eindeutige Lösung zu erhalten, bspw. um das Stromwärmegesetz.

[14] Für die vereinfachte Betrachtung in dieser Arbeit müssen lediglich $\boldsymbol{H}$, $\boldsymbol{B}$ und $\boldsymbol{\mu}$ homogen sein.

Material

Magnetische Materialen selbst haben ausschließlich intrinsische Eigenschaften, sodass nur der Zusammenhang $h(\boldsymbol{B}, \boldsymbol{H}) = 0$ untersucht werden muss.[15] Sie weisen gleich mehrere Besonderheiten auf, welche die Beschreibung magnetischer Netzwerkelemente gegenüber elektrischen verkompliziert, weshalb der Einfluss des Materials auf die Beschreibung des Netzwerkelements besonderer Aufmerksamkeit bedarf.

Für eine einfache Beschreibung des Zusammenhangs $h(\boldsymbol{B}, \boldsymbol{H}, ...) = 0$ wird der Begriff der Permeabilität $\boldsymbol{\mu}$ verwendet.[16] Sie ist das Produkt aus der Naturkonstante der Vakuumpermeabilität μ_0 und der materialabhängigen relativen Permeabilität $\boldsymbol{\mu}_{\mathrm{r}}$.

$$\boldsymbol{B} = \boldsymbol{\mu} \begin{pmatrix} \boldsymbol{H} \\ \dots \end{pmatrix} = \mu_0 \boldsymbol{\mu}_{\mathrm{r}} \begin{pmatrix} \boldsymbol{H} \\ \dots \end{pmatrix} \tag{1.12}$$

Wenn $\boldsymbol{B}$ lediglich von $\boldsymbol{H}$ abhängt, ist die Matrix $\boldsymbol{\mu}_{\mathrm{r}}$ quadratisch und das Material heißt **unabhängig** von externen Einflüssen. Dies ist selten der Fall, weil meist die Temperatur und das zeitliche Differential des H-Feldes die Permeabilität beeinflussen. Dennoch wird häufig die Näherung getroffen, dass die Permeabilität unabhängig sei und auch in dieser Arbeit werden alle Materialien ohne Rücksicht auf ihre Unabhängigkeit so dargestellt. In dem Fall können außerdem lediglich die Diagonalelemente der Matrix besetzt sein. Dabei wird die quadratische Matrix einfach als Funktion der Größen definiert, welche das Material beeinflussen:

$$\boldsymbol{\mu}_{\mathrm{r}} = \boldsymbol{\mu}_{\mathrm{r}}(...) \tag{1.13}$$

Diese Definition ist äquivalent zu der oben genannten. Im Falle einer gegenseitigen Abhängigkeit muss für die Lösung des sich letztendlich ergebenden Gleichungssystems nun allerdings ein iteratives Verfahren angewandt werden (sofern es nicht wieder in die ursprüngliche Form umgestellt wird.)

Anstatt eine evtl. bestehende Abhängigkeit von einer Größe $a(t)$ direkt zu formulieren, wird diese als Abhängigkeit von der Zeit definiert:[17]

$$\boldsymbol{\mu}_{\mathrm{r}} = \boldsymbol{\mu}_{\mathrm{r}}(a(t)) = \boldsymbol{\mu}_{\mathrm{r}}(t) \tag{1.14}$$

Obwohl dies das Gleichungssystem nicht beeinflusst, wird die Darstellung einiger Zusammenhänge in den kommenden Abschnitten dadurch verständlicher.

Sofern alle Matrixelemente entweder nur statische oder gar keine Operatoren beinhalten, heißt das Material **verlustfrei**. Dies heißt insbesondere, dass bei einer Änderung einer einzelnen Zustandsgröße und einer anschließenden Zurückänderung der exakt gleiche Zustand erreicht wird. Der Begriff „Verlust" bezieht sich in diesem

[15] $\boldsymbol{H}$ ist in der Gleichung das H-Feld und h ist eine beliebige Funktion.
[16] Es wird davon ausgegangen, dass der implizite Zusammenhang ein explizites Äquivalent besitzt. Bei verlustbehafteten Materialien kann die Umformung eine Herausforderung darstellen, die im Rahmen der vorliegenden Arbeit nicht weiter thematisiert wird.
[17] Da alle Felder innerhalb eines Netzwerkelements homogen sind, muss keine Ortsabhängigkeit beachtet werden.

Zusammenhang auf die im magnetischen Feld gespeicherte Energie[18] oder auf Energie, welche mit dem Magnetfeld gekoppelt ist, bspw. die Energie, welche in einem elektrischen Netzwerk transportiert wird, welches auf geeignete Weise mit der betrachteten magnetischen Anordnung gekoppelt ist.[19] In dieser Arbeit sollen die physikalischen Effekte, welche zum Verlust von magnetischer Energie führen, nicht untersucht werden. Der Stromwärmeeffekt sollte zudem bekannt sein.

Weil in magnetischen Feldern ein energetischer Zustand gespeichert wird, anders als in elektrischen Strömungsfeldern, die einen Leistungszustand darstellen, und weil Energie eine Größe ist, die dem Netzwerk zugeführt oder entnommen werden kann, sind die Kennlinien der magnetischen Widerstände nicht zwingend zeitlich konstant, sondern von der gespeicherten Energie abhängig. Wenn Energieänderungen bei sich periodisch ändernden Feldern mit der gleichen Periodizität auftreten wie die Feldgrößen, können sie durch eine nicht bijektive Kennlinie modelliert werden. Dies soll zunächst am Beispiel der Kernverluste[20] erläutert werden: Beim Wechsel des Vorzeichens der magnetischen Flussdichte durch eine Reluktanz entstehen energetische Verluste. Dadurch, dass die im Material gespeicherte Energiedichte durch den Zustand von Feldstärke und Flussdichte $\frac{\partial E(t)}{\partial V} = \int H(B(t), t)\mathrm{d}B(t)$ [21] definiert ist, erreicht die Reluktanz bei dem erneuten Vorzeichenwechsel des Flusses nicht mehr den gleichen Zustand. Die Kennlinie weist deshalb eine Hysterese auf, deren Flächeninhalt dem Energieverlust bei einmaligem Durchlaufen der gesamten Hystereseschleife gleich ist.

Die Modellierung solcher Verluste kann auf beliebige Weise geschehen, bspw. mithilfe von Verlustreluktanzen, die mithilfe einer Transformation als elektrische Widerstände dargestellt werden (siehe Abschnitt 1.1.4, S. 19). Diese Darstellungen sind typischerweise nichtlinear.[22] Eine einfach Modellierung ist außerdem mit dem Jiles-Atherton-Modell möglich, welches erstmals in [19] vorgestellt ist. Im Rahmen der vorliegenden Arbeit werden nur verlustfreie Materialien verwendet (siehe Abschnitt 5.2.3, S. 147) und modelliert (siehe Abschnitt 4.2, S. 130).

Wenn in anderen Domänen der Physik Verluste auftreten, können diese als Zweipol mit entsprechender Hysterese modelliert werden (siehe Abschnitt 1.1.4, S. 19).

Wenn keines der Matrixelemente ein Operator ist, heißt das Material **linear**. Typischerweise sind dann alle Einträge konstant und die Permeabilität ist unabhängig von der Feldstärke. Während paramagnetische Materialien, wie Luft, eine konstante Permeabilität in Abhängigkeit der Felstärke vorweisen können, sind ferromagnetische Materialien (bspw. Eisenlegierungen, keramische Ferrite) vom Phänomen der Sättigung betroffen. Dabei sinkt die Permeabilität mit steigender magnetischer Feldstärke, sodass die Permeabilität für hohe Feldstärkebeträge

[18] Selbstverständlich handelt es sich beim „Verlust" um eine Wandlung der Energie in Wärme und nicht um eine Vernichtung der Energie.

[19] Eine solche Kopplung wird später definiert (siehe Abschnitt 1.1.3, S. 14).

[20] Oberbegriff für die Umwandlung magnetischer Energie in Wärme im Volumen eines magnetischen Materials

[21] Hier ist die Kennlinie $H(B,t) := |\boldsymbol{H}|(|\boldsymbol{B}|,t) = \frac{1}{\mu_0}\left|\boldsymbol{\mu}_r^{-1}(...(t))\boldsymbol{B}\right|$ als expliziter Zusammenhang beschrieben, welcher äquivalent zur Beschreibung mit der Permeabilität, aber übersichtlicher auszuschreiben ist. Man beachte, dass alle externen Abhängigkeiten auf die die Zeitabhängigkeit reduziert sind, wie bereits zuvor beschrieben.

[22] Der Autorin ist bekannt, das Prof. Dr.-Ing. P. Zacharias am KDEE-EVS bereits Erfolge bei der Modellierung von verlustbehafteten magnetischen Materialien als nichtlineare elektrische Widerstände hatte. Ihr sind allerdings bislang keine Veröffentlichungen zu dem Thema bekannt.

$|\boldsymbol{H}|$ abflacht. Ein Beispiel ist das Material „NO10“, welches im Versuch verwendet wird (siehe Abschnitt 5.4, S. 154). Damit wird die Definition der differentiellen Permeabilität nützlich:

$$\mu_{\mathsf{diff}}(H) := \frac{\partial B(H)}{\partial H} \tag{1.15}$$

Meist wird der Sättingungseffekt mithilfe der Kennlinie $B(H)$ ausgedrückt, wie in *Abb. 5.9*. Für die analytische oder numerische Modellierung der Kennlinie bestehen zahlreiche Möglichkeiten, [31] auf welche aufgrund ihres Umfangs an dieser Stelle nicht eingegangen werden soll. Ein konkretes Beispiel ist später vorgestellt (siehe Abschnitt 4.2, S. 130).

Für den Fall, dass die Matrix quadratisch und alle Einträge identisch sind[23], heißt das Material **isotrop**. In dem Fall ist der Zusammenhang $B(H)$ unabhängig von der Richtung der Felder.

Es sei angemerkt, dass alle von externen Einflüssen unabhängigen, verlustfreien Materialien nach dem Verständnis der Autorin eine **positiv** definite Permeabilitätsmatrix aufweisen müssen. Andernfalls wäre es möglich, in einem magnetischen Material eine „negative“ Energie zu speichern, was für eine ingenieursmäßige Betrachtung unsinnig wäre.

1.1.3 Elemente

Für die Modellierung der Netzwerkelemente müssen die zuvor genannten Eigenschaften magnetischer Aufbauten miteinander verknüpft werden (siehe Abschnitt 1.1.2, S. 11). Zunächst werden alle Zweipole untersucht (Reluktanzen, magnetische Spannungsquellen und Flussquellen), womit alle zuvor genannten Eigenschaften magnetischer Anordnungen beschrieben sind. Dabei wird zwischen aktiven (solche, die eine Energiequelle enthalten) und passiven Elementen unterschieden. Für die anschauliche Beschreibung mehrerer voneinander abhängiger Zweipole werden anschließend Multipole (Transreluktanzen) definiert. Abhängigkeiten von externen physikalischen Größen werden im hierauf folgenden Abschnitt (siehe Abschnitt 1.1.4, S. 19) untersucht.

Reluktanz

Ein Netzwerkelement, welches typischerweise ein physikalisches Raumgebiet der Fläche A und der Länge l beschreibt, in welchem sich ein Material mit der invertierbaren Permeabilität $\boldsymbol{\mu}_{\mathsf{r}}$ befindet, wird durch den Zusammenhang $h(\Phi, V)$ in der expliziten Form

$$V(\Phi) = \boldsymbol{R}_{\mathsf{m}}\Phi \qquad \boldsymbol{R}_{\mathsf{m}}^{-1} = \mu_0 \boldsymbol{\mu}_{\mathsf{r}} \frac{A}{l} \tag{1.16}$$

[23] Anmerkung zur Veröffentlichung: In der Abgabeversion war statt der Bedingung identischer Einträge die Bedingung einer symmetrischen Matrix genannt. Diese ist unzureichend.

beschrieben und heißt Reluktanz und ist passiv.[24] Der Zusammenhang wird somit für jede Reluktanz vollständig durch dessen gleichnamige Kenngröße $\boldsymbol{R}_\text{m}$ charakterisiert. Die Gegenfunktion heißt Permeanz $\boldsymbol{G}_\text{m}$:

$$\boldsymbol{G}_\text{m} = \boldsymbol{R}_\text{m}^{-1} \tag{1.17}$$

Sollte das Raumgebiet nicht den Ansprüchen zur Modellierung als einzelne Reluktanz genügen (siehe Abschnitt 1.1.2, S. 11), kann es als ein Netzwerk aus Reluktanzen betrachtet werden. Diese können ggf. zu einer sog. verteilten Reluktanz zusammengefasst werden. Im Folgenden wird, soweit möglich, die Existenz einer positiven, skalaren Reluktanz angenommen:[25]

$$\forall \boldsymbol{R}_\text{m} \in \mu_0 \cdot \text{diag}\left(O_1, O_2, ..., O_{\text{dim}(V)}\right) : R_\text{m} := \boldsymbol{e}_H^\text{T} \boldsymbol{R}_\text{m} \boldsymbol{e}_H \tag{1.18}$$

Für deren Berechnung ist kein Wissen über die tatsächliche Richtung des $\boldsymbol{H}$-Feldes notwendig, sondern lediglich über die Orientierung des Bezugsvektors für die magnetische Spannung V siehe Abschnitt 1.1.1, S. 6). Eine reale Reluktanz zeichnet sich außerdem dadurch aus, dass sie nicht nur skalar, sondern auch positiv und ihre Permeabilität auch invertierbar und in ferromagnetischen Materialien verlustbehaftet und nichtlinear ist. Eine ideale Reluktanz ist im Gegensatz zur realen linear und verlustfrei. Die beiden Schaltsymbole sowie die Kennlinie einer realen Reluktanz sind in *Abb. 1.2* abgebildet. Der Übersicht halber werden im Weiteren Abhängigkeiten der Reluktanzen von anderen Größen (insbesondere zeitabhängigen Feldern) nicht explizit ausgeschrieben.

Wenn bei der Berechnung der Lösung eines Netzwerkes eine kleine Änderung in der Nähe eines Arbeitspunkts betrachtet wird (siehe Abschnitt 1.1.6, S. 29), ist die Definition der differentiellen Permeabilität $R_\text{m,diff}$ nützlich:

$$R_\text{m}^{-1} = \frac{\Phi}{V} \qquad R_\text{m,diff}^{-1}(V) = \frac{\partial \Phi(V)}{\partial V} \tag{1.19}$$

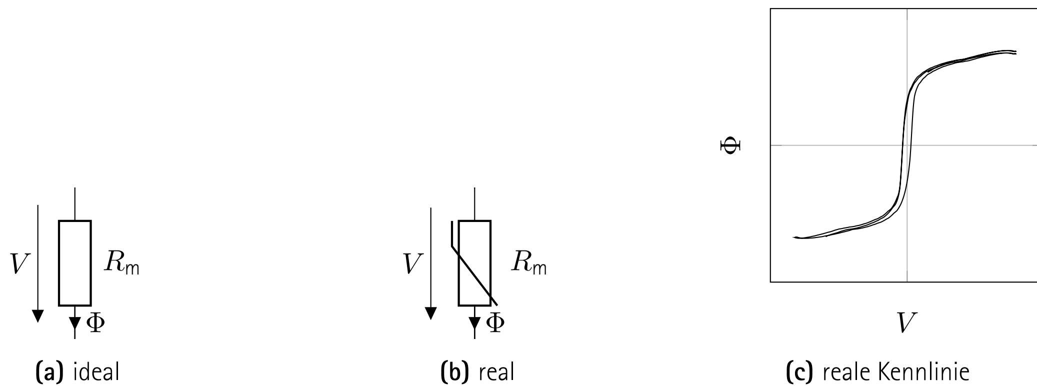
Magnetische Spannungsquelle

Eine ideale magnetische Spannungsquelle[26] ist ein aktiver Zweipol in magnetischen Netzwerken, welches sich durch eine konstante Spannung über dem Fluss auszeichnet. In der Realität wird sie mithilfe einer stromdurchflossenen Spulenwicklung dargestellt. Durch die Nichtlinearität des umwickelten Kernmaterials und die darin entstehenden Verluste sind reale magnetische Spannungsquellen stets nichtlinear und verlustbehaftet. Ein Modell, welches diese Effekte berücksichtigt, heißt reale magnetische Spannungsquelle. Analytisch ergibt sich dessen Kennlinie durch Addition einer idealen Spannungsquelle mit der einer realen Reluktanz, der soge-

[24] Die Reluktanz verhält sich ähnlich zum elektrischen Widerstand in elektrischen Netzwerken und wird deshalb auch magnetischer Widerstand genannt.

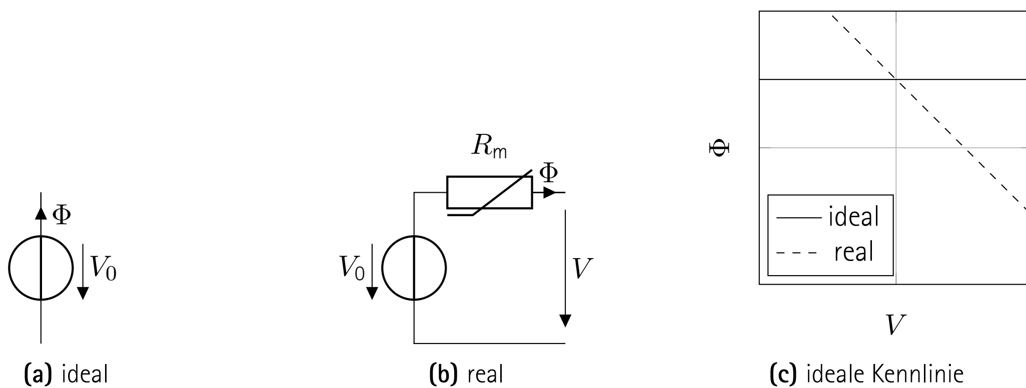
[25] Hier ist O ein beliebiger Operator.

[26] Eine magnetische Spannungsquelle verhält sich ähnlich zur elektrischen Spannungsquelle, daher rührt die namentliche Ähnlichkeit.



(a) ideal **(b)** real **(c)** reale Kennlinie

Abb. 1.2: Reluktanz. Schaltsymbol für ideale Reluktanz (links) und reale Reluktanz (mittig). Charakteristische Funktion am Beispiel einer Reluktanz mit sehr homogenem Feld aus dem Material „NO10" (rechts).



(a) ideal **(b)** real **(c)** ideale Kennlinie

Abb. 1.3: Magnetische Spannungsquelle. Schaltsymbol für ideale Quelle (links) und reale Quelle (mittig). Charakteristische Funktion ohne und mit idealer Innenreluktanz (rechts).

nannten Innenreluktanz $R_{m,i}$.

$$V(\Phi) = V_0 - V_R(\Phi) = V_0 - R_{m,i}\Phi \tag{1.20}$$

Die Schaltbilder und die reale Kennlinie sind in *Abb. 1.3* dargestellt. Auf die Zusammenhänge zwischen dem elektrischen Netzwerk, in welches die Wicklung eingebunden wird, und dem magnetischen Netzwerk, Teil dessen die magnetische Spannungsquelle ist, wird in später genauer eingegangen (siehe Abschnitt 1.1.5, S. 24).

Flussquelle

Eine Flussquelle[27] ist ein aktiver Zweipol, dessen Kennlinie einen konstanten Fluss über der magnetischen Spannung aufweist. In der Realität wird sie mithilfe eines Permanentmagneten oder einer spannungsintegralgesteu-

[27] Eine Flussquelle verhält sich ähnlich zur elektrischen Stromquelle und wird deshalb auch magnetische Stromquelle genannt.

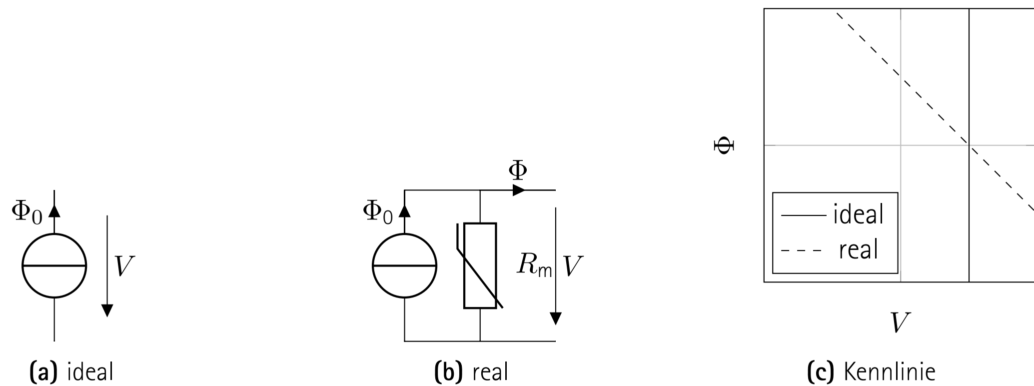


Abb. 1.4: Flussquelle. Schaltsymbol für ideale Quelle (links) und reale Quelle (mittig). Charakteristische Funktion ohne und mit idealer Innenreluktanz (rechts).

erten Spule[28] implementiert. Weil eine Flussquelle stets an ein physikalisches Volumen eines magnetischen Materials gekoppelt ist, stellt dessen Reluktanz die Innenreluktanz dar. Die Kennlinie ergibt sich zu:

$$\Phi(V) = \Phi_0 - \Phi_R = \Phi_0 - R_{\mathrm{m}}^{-1} V \tag{1.21}$$

Die Schaltbilder und die reale Kennlinie sind in *Abb. 1.4* illustriert. Die Ursache des Magnetfeldes in PM ist bereits untersucht (siehe Abschnitt 1.1.1, S. 6). Der Zusammenhang zwischen dem elektrischen und dem magnetischen Netzwerk, in welches eine entkoppelte Spule eingebunden ist, wird später betrachtet (siehe Abschnitt 1.1.5, S. 24).

Transreluktanz

Aufgrund der Nichtlinearität realer Reluktanzen ist bei der Einprägung mehrerer Felder in das selbe Raumgebiet nicht durch das Superpositionsprinzip entkoppelt. Entsprechend interagieren mehrere magnetische Spannungen an einer bzw. mehrere Flüsse durch eine Reluktanz miteinander. Wenn die verschiedenen Felder parallel oder antiparallel orientiert sind, können die sie verursachenden Netzwerkelemente einfach entsprechend angeordnet [29] und in das Netzwerk integriert werden. Sollten die verursachten Felder aber in einem anderen Raumwinkel zueinander orientiert sein, ist die Definition eines 2β-Pols notwendig, wobei β gleich der Anzahl der unabhängigen Felder ist. Man beachte, dass parallele Feldkomponenten bereits im Netzwerk beschrieben werden können, sodass β sich auf die Anzahl orthogonal zueinander stehender Feldkomponenten reduziert und somit nicht größer als drei werden kann.

Für einen solchen 2β-Pol hat sich in der Literatur noch keine eindeutige Bezeichnung durchgesetzt, Vorschläge beinhalten Paraformer (oder Paraformator), Parametrisches Gerät, Transpermeanz, Transreluktanz und Trans-

[28] Anders als bei der magnetischen Spannungsquelle, deren Spule von einem bestimmten Strom durchflossen wird, ist bei der magnetischen Flussquelle das zeitliche Integral der elektrischen Spannung bestimmt.

[29] Magnetische Spannungsquellen werden entsprechend seriell oder antiseriell und Flussquellen parallel oder antiparallel angeordnet.

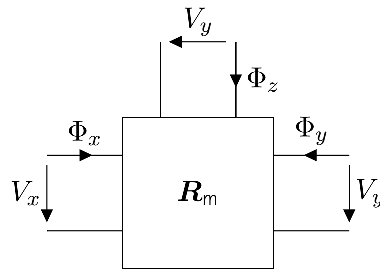


Abb. 1.5: Schaltsymbol einer Transreluktanz. Drei Polpaare für je eine magnetische Spannung und je einen Fluss in je einer Raumdimension. Polpaar kann ignoriert werden, wenn magnetische Spannung zwischen beiden Polen und Fluss durch beide Pole zu null werden. Darstellung mit Reluktanzmatrix $\boldsymbol{R}_\mathrm{m}$. Gesamtes Element erfüllt den Knotensatz.

luktanz.[30] Der ersten beiden Begriffe stammen aus der Betrachtung des elektrischen Klemmenverhaltens zweier als magnetische Flussquellen dienenden Wicklungen, die in ein magnetisches Netzwerk mit einem solchen Element integriert sind (siehe Abschnitt 1.1.5, S. 24), weshalb sich die Bezeichnung korrekterweise auf das gesamte Netzwerk und nicht auf das darin verwendete zentrale Element beziehen sollte, wie in [2] geschehen. Weil in dieser Arbeit das Verhalten des Multipols in der Reluktanzschreibweise (s. u.) bevorzugt wird, bietet sich der Begriff der Transreluktanz an.

Allgemein definiert eine Transreluktanz den Zusammenhang $h(V_x, \Phi_x, V_y, \Phi_y, V_z, \Phi_z) = 0$. Das Netzwerkelement ist in *Abb. 1.5* abgedruckt. Der implizite Zusammenhang lässt sich explizit formulieren (siehe Abschnitt 1.1.2, S. 11). Wenn die Interaktion lediglich zweier orthogonaler Komponenten betrachtet wird $V_z = 0, \Phi_z = 0$, sind verschiedene Schreibweisen möglich. Dazu gehört die Notation als Kettenmatrix, auch A-Charakteristik genannt:

$$\begin{pmatrix} V_x \\ \Phi_x \end{pmatrix} = \boldsymbol{A}_\mathrm{m} \begin{pmatrix} V_y \\ \Phi_y \end{pmatrix} \tag{1.22}$$

Auf diese Weise wird die Berechnung mehrerer in Kette geschalteter Elemente einfacher, weil sich die Kettenmatrix der gesamten Schaltung als das Produkt aller einzelner Elemente ergibt. Diese und weitere Schreibweisen mit ähnlichen Vorteilen sind zur Beschreibung von Transreluktanzen nur bedingt geeignet, weil sie eine Anzahl orthogonale Feldkomponenten von $\beta = 2$ voraussetzt. In dieser Arbeit wird ausschließlich die Schreibweise als Reluktanzmatrix $\boldsymbol{R}_\mathrm{m}$ verwendet, auch Z-Charakteristik genannt:

$$\begin{pmatrix} V_x \\ V_y \end{pmatrix} = \boldsymbol{R}_\mathrm{m} \begin{pmatrix} \Phi_x \\ \Phi_y \end{pmatrix} \tag{1.23}$$

Diese zeichnet sich zum einen durch den Vorteil aus, dass sie unabhängig von der Dimensionalität des zu

[30] Diese Vorschläge sind insbesondere [31] und dem mündlichen Wort von Prof. Dr.-Ing. P. Zacharias vom KDEE-EVS entnommen.

beschreibenden Raumgebiets definierbar ist:

$$\begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} = \boldsymbol{R}_{\mathrm{m}} \begin{pmatrix} \Phi_x \\ \Phi_y \\ \Phi_z \end{pmatrix} \tag{1.24}$$

Zum anderen legt sie die Betrachtung des physikalischen Objektes als Reluktanz nahe, welche ein intuitives Verständnis der in späteren Kapiteln identifizierten Zusammenhänge erleichtern soll.

Die Herleitung der Reluktanzmatrix aus den physikalischen Größen (siehe Abschnitt 1.1.3, S. 14) ist bislang selbst für einfache praktische Anwendungen unzureichend. Die bereits geleisteten Modellierungsarbeiten werden später vorgestellt (siehe Abschnitt 1.4, S. 42).

1.1.4 Transformierte Elemente

Zuvor sind die eigentlichen magnetischen Netzwerkelemente untersucht (siehe Abschnitt 1.1.3, S. 14), welche ausschließlich magnetische Phänomene beschreiben. Uneigentliche magnetische Netzwerkelemente modellieren die Phänomene aus anderen Domänen der Physik[31] oder stellen magnetische Elemente auf eine andere Weise dar, als zuvor definiert (siehe Abschnitt 1.1.3, S. 14). Das Ziel ist die Beschreibung von Nichtlinearität, Verlustbehaftung und Kopplung und verteilter Größen.

Zerlegte Reluktanzen

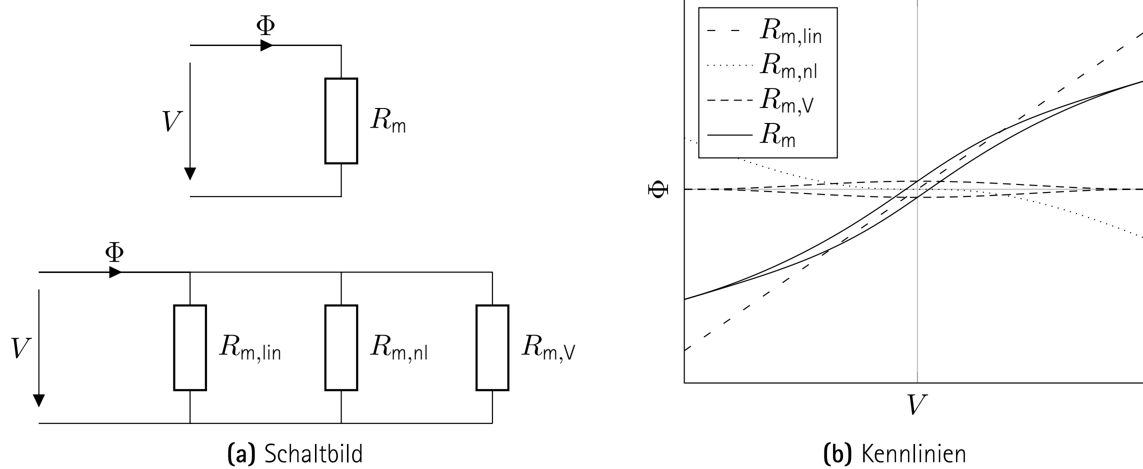
Eine nichtlineare, verlustbehaftete Reluktanz kann aufgeteilt werden in einen linearen, einen nichtlinearen und einen verlustbehafteten Anteil, die wahlweise in Reihe oder parallel geschaltet werden. Bei einer Reihenschaltung ergibt sich die gesamte Reluktanz als Summe ihrer Anteile:

$$R_{\mathrm{m}} = R_{\mathrm{m,lin}} + R_{\mathrm{m,nl}} + R_{\mathrm{m,V}} \tag{1.25}$$

Dabei wird die Hystereseschleife in einen oberen Teil $R_{\mathrm{m},+}$ und einen unteren Teil $R_{\mathrm{m},-}$ aufgeteilt. Der Mittelwert der Hystereseschleife wird als verlustfreier Teil betrachtet, die Differenz ist entsprechend der verlustbehaftete Teil:

$$R_{\mathrm{m}} - R_{\mathrm{m,V}} = R_{\mathrm{m,lin}} + R_{\mathrm{m,nl}} = \frac{R_{\mathrm{m},+} + R_{\mathrm{m},-}}{2} \tag{1.26}$$

[31] Die Modellierung von nichtmagnetischen Effekten als besondere magnetische Netzwerkelemente ist zwar allgemein immer möglich, allerdings nur sinnvoll, wenn diese mit den Größen des Netzwerkes gekoppelt sind und die Beschreibung durch die Kopplung eines eigentlichen magnetischen Netzwerkelementes mit einem netzwerkexternen Größe nicht gewünscht ist.



(a) Schaltbild

(b) Kennlinien

Abb. 1.6: Zerlegung einer realen Reluktanz in linearen, nichtlinearen und verlustbehafteten Teil mit gleicher Kennlinie. Addition der Kennlinien $\Phi(V)$ entsprechend Parallelschaltung (siehe Code, S. 179).

Die Anfangsreluktanz des verlustfreien Anteils wird als linearer Teil verwendet:

$$R_{\mathrm{m,lin}} = R_{\mathrm{m,0}} = R_{\mathrm{m,diff}}|_{V=0,\Phi=0} \tag{1.27}$$

Bei einer Parallelschaltung addieren sich die Permeanzen,

$$G_{\mathrm{m}} = G_{\mathrm{m,lin}} + G_{\mathrm{m,nl}} + G_{\mathrm{m,V}} \tag{1.28}$$

die Berechnung der Anteile geschieht entsprechend. Die Zerlegung ist in *Abb. 1.6* graphisch präsentiert. Diese Methode erleichtert die manuelle Netzwerkanalyse: Weil Reluktanzen mit gleicher Kurvenform einfach zusammengefasst werden können, lassen sich so mindestens alle linearen Reluktanzen eines reinen Reluktanznetzwerks zusammenfassen, bspw. Luftspalt-, ungesättigte und lineare Anteile anderer Reluktanzen. Außerdem können spezifischere Lösungsverfahren genutzt werden (siehe Abschnitt 1.1.6, S. 29).

Verlustreluktanzen

Wenn in anderen Domänen der Physik Verluste auftreten, können diese als Zweipol mit entsprechender Hysterese modelliert werden. Der Zusammenhang zwischen Materialhysterese und Energiedichte ist bereits zuvor beschrieben (siehe Abschnitt 1.1.2, S. 11). Die Verlustleistung ist das Volumenintegral ihrer (volumetrischen) Dichte, welche wiederum die zeitliche Ableitung der Energiedichte ist.

Beispielhaft sollen hier die Gleichstromverluste einer Wicklung betrachtet werden. Zunächst werden das elektrische und magnetische Netzwerk mit allen Elementen gebildet, welche die physikalischen Objekte repräsen-

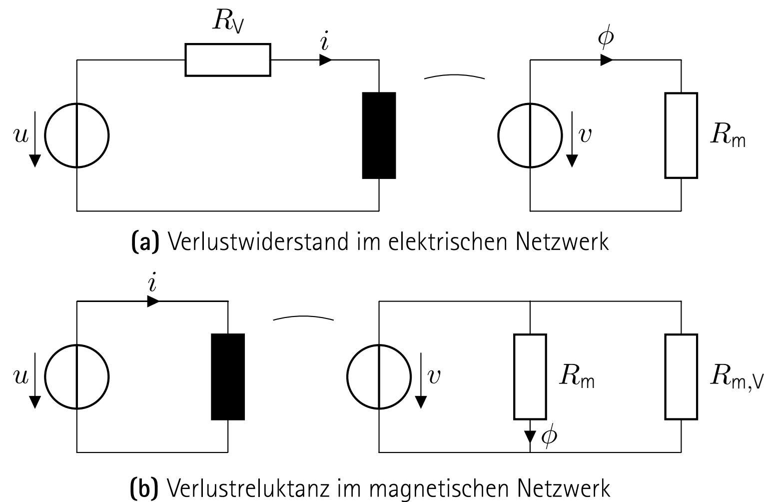


Abb. 1.7: Netzwerk mit Verlustreluktanz. Elektrischer Widerstand aus (oben) transformiert in Verlustreluktanz (unten). Magnetischer Fluss ϕ bleibt definiert als derjenige durch die Reluktanz R_m. Eingezeichnete Induktivität ergibt sich aus gesamter Reluktanz an den Klemmen der magnetischen Spannungsquelle.

tieren, wie in *Abb. 1.7a* gezeigt. Es ergibt sich ein unterbestimmtes Gleichungssystem, aus dem sich der Zusammenhang zwischen der magnetischen und der elektrischen Spannungsquelle herleiten lässt.

$$\left| \begin{array}{rcl} \phi & = & \frac{1}{N}\int(u - R_\mathrm{V} i)\mathrm{d}t \\ v & = & R_\mathrm{m}\phi \\ v & = & Ni \end{array} \right| \Rightarrow \phi = \frac{\frac{N^2 \frac{\mathrm{d}}{\mathrm{d}t}}{R_\mathrm{V}}}{R_\mathrm{m} + \frac{N^2 \frac{\mathrm{d}}{\mathrm{d}t}}{R_\mathrm{V}}} \frac{1}{N} \int u \mathrm{d}t \tag{1.29}$$

Der elektrische Widerstand R_V kann durch einen Kurzschluss ersetzt werden, wenn das magnetische Netzwerk erweitert wird. Ein Koeffizientenvergleich mit der Flussteilerformel

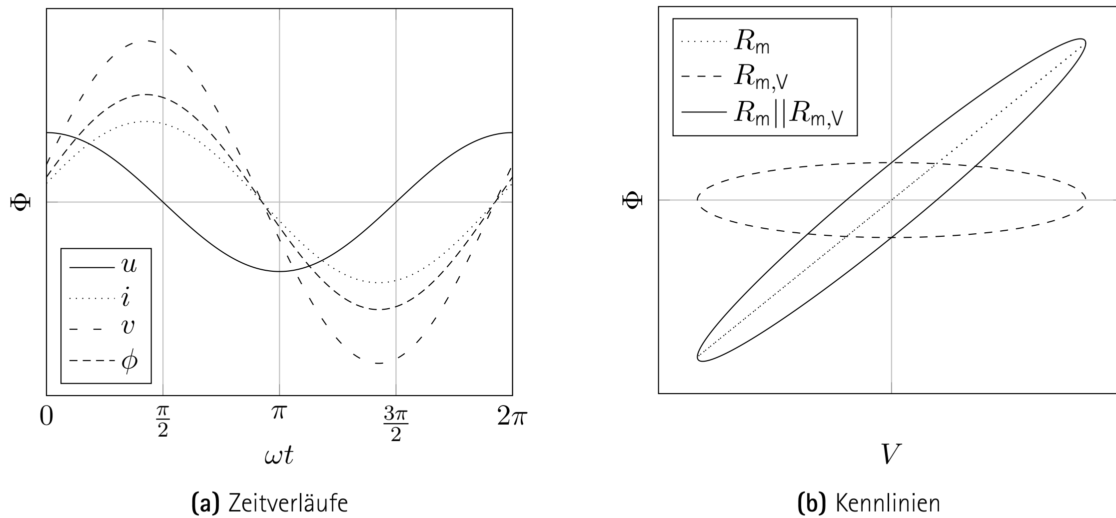
$$\phi_1 = \frac{R_{\mathrm{m},2}}{R_{\mathrm{m},1} + R_{\mathrm{m},2}}(\phi_1 + \phi_2) \qquad \phi_2 = \frac{R_{\mathrm{m},1}}{R_{\mathrm{m},1} + R_{\mathrm{m},2}}(\phi_1 + \phi_2) \tag{1.30}$$

ergibt, dass eine Reluktanz

$$R_\mathrm{m,V} = \frac{N^2 \frac{\mathrm{d}}{\mathrm{d}t}}{R_\mathrm{V}} \tag{1.31}$$

parallel zur physikalischen Reluktanz R_m zu einem Netzwerk führt, wie in *Abb. 1.7b* festgehalten, und dass beide Netzwerke durch den gleichen Zusammenhang $h(u, i, v, \phi) = 0$ in Abhängigkeit der Parameter R, R_m und N beschrieben werden.

Mithilfe der komplexen Netzwerkrechnung lässt sich der Differentialoperator in *Gl. 1.31* bei einer Kreisfrequenz



(a) Zeitverläufe

(b) Kennlinien

Abb. 1.8: Netzwerkgrößen bei Verlustreluktanz. Harmonische Anregung durch Spannungsquelle. Lineare Reluktanz und linearer elektrischer Widerstand beispielhaft mit $\frac{\omega L}{R} \approx 4{,}2; L = \frac{N^2}{R_\mathrm{m}}$ (siehe Code, S. 175).

ω als

$$\mathcal{L}\left\{\frac{\mathrm{d}}{\mathrm{d}t}f(t)\right\} = j\omega\mathcal{F}(j\omega) \tag{1.32}$$

ausdrücken. Die komplexe Verlustreluktanz ergibt sich zu:

$$R_\mathrm{m,V} = \frac{N^2 j\omega}{R_\mathrm{V}} \tag{1.33}$$

Sollte die Frequenzabhängigkeit der Verluste nicht proportional sondern ganzrational vom Grad k sein, ist die Modellierung einer frequenzabhängigen Hysterese notwendig. Diese enthält ein Integral der Ordnung $k-1$. Im Beispiel des transformierten elektrischen Widerstands ist die Verlustleistung unabhängig von der Frequenz ($k=0$), was durch den Differentialoperator erster Ordnung ausgedrückt wird.

Man beachte, dass für die Gültigkeit dieser Rechenmethode alle davon betroffenen Netzwerkelemente linear und zeitinvariant sein und alle Zustandsgrößen harmonische zeitliche Schwingungen beschreiben müssen. Bei der oben gemachten Vereinfachung $s = j\omega$ wird außerdem ein stationärer Zustand angenommen.[32] Auch die Flussteilerformel in *Gl. 1.30* setzt lokal lineare, zeitinvariante Elemente voraus. Wie bereits zuvor beschrieben, weisen physikalische Reluktanzen im Bereich niedriger Feldstärken und kleiner Netzwerkgrößenbeträge dieses Verhalten annähernd auf (siehe Abschnitt 1.1.2, S. 11). Auf Nichtlinearitäten soll in dieser Arbeit lediglich im Zusammenhang mit Transreluktanzen eingegangen werden (siehe Abschnitt 1.4, S. 42).

Selbstverständlich ist auch die Transformation eines elektrischen Widerstands möglich, welcher nicht in Reihe,

[32] Auf die vielen Annahmen der komplexen Wechselstromrechnung und ihre Herleitung soll hier nicht weiter eingegangen werden, sondern es sei lediglich auf [22] verwiesen.

sondern parallel zum induktiven Element geschaltet ist, welches die Kopplung zwischen elektrischem und magnetischem Netzwerk auf der elektrischen Seite symbolisiert. In dem Fall ist die transformierte Verlustreluktanz in Reihe zur physikalischen Reluktanz geschaltet. Die Berechnung des beschreibenden Operators ist identisch und geschieht wie in *Gl. 1.31* definiert.

Transformierte Induktivität

Ebenso wie ein elektrischer Widerstand kann auch eine elektrische Induktivität in ein magnetisches Netzwerk transformiert werden. Eine Induktivität L_1 setze sich aus einer Reluktanz $R_{\mathrm{m},1}$ und einer Wicklung mit der Windungszahl N_1, die in einem elektrischen Netzwerk verwendet sei, zusammen. Sie kann entfernt werden, wenn einer anderen Induktivität im selben elektrischen Netzwerk mit der Windungszahl N_2 die Reluktanz

$$R_{\mathrm{m},2} = \frac{N_2}{N_1} R_{\mathrm{m},1} \tag{1.34}$$

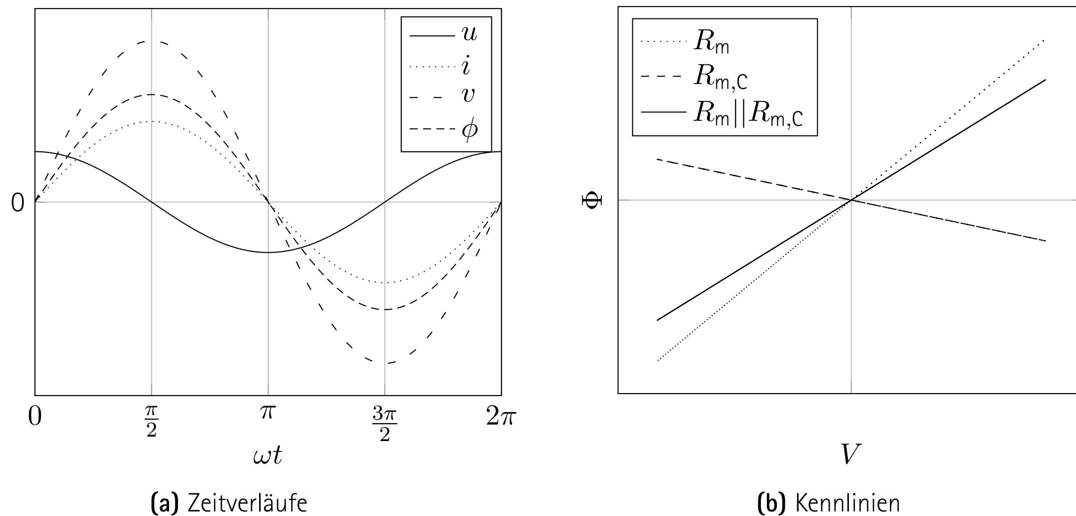
hinzugefügt wird. Bei einer Reihenschaltung von Induktivitäten müssen die Reluktanzen parallel geschaltet sein und umgekehrt. Das Verfahren ist selbstverständlich umkehrbar, sodass auch Reluktanzen entfernt und als weitere Induktivität im elektrischen Netzwerk aufgeführt sein können. Ein häufig genutztes Beispiel ist die Modellierung der Streureluktanz einer Wicklung als eine weitere Induktivität im elektrischen Netzwerk.

Transformierte Kapazität

Elektrische Kapazitäten lassen sich ebenfalls zu Reluktanzen transformieren, die in elektrischen Netzwerken verwendet werden:

$$R_{\mathrm{m,C}} = N^2 (j\omega)^2 C = -N^2 \omega^2 C \tag{1.35}$$

Weil physikalische Kapazitäten genauso wie physikalische Induktivitäten stets positiv sind, ergibt sich bei der Transformation einer elektrische Kapazität in ein magnetisches Netzwerk eine negative Reluktanz. Auch hier gilt, dass Reihenschaltungen im elektrischen Netzwerk mit Parallelschaltungen im magnetischen korrespondieren und umgekehrt. Ein Beispiel ist in *Abb. 1.9* abgebildet, worin deutlich wird, dass die effektive Reluktanz an den Klemmen einer magnetischen Quelle durch die transformierte elektrische Kapazität sinkt. Später sind im Messaufbau immer wieder parasitäre Kapazitäten beobachtbar (siehe Abschnitt 2.2, S. 57), allerdings sind diese so klein, dass sie erst oberhalb des genutzten Frequenzbereichs (siehe Abschnitt 5.3, S. 152) relevant werden. Auf die Theorie transformierter Kapazitäten wird deshalb an dieser Stelle nicht weiter eingegangen.



(a) Zeitverläufe

(b) Kennlinien

Abb. 1.9: Transformierte Kapazität. Elektrische Kapazität in Reihe zu induktivem Bauteil. Deshalb zur Reluktanz transformierte Kapazität parallel zur physikalischen Reluktanz. Beispielhaft $\omega L \cdot \omega C \approx 4{,}0$; $L = \frac{N^2}{R_\text{m}}$ (siehe Code, S. 178).

Weitere Transformationen

Physikalische Größen, die mit einem magnetischen Netzwerk gekoppelt sind, können im Allgemeinen in dieses „hineintransformiert" werden. Von besonderem technischen und kommerziellen Interesse ist dabei die Transformation mechanischer Größen, weil die in mechanischen Aufbauten vorkommenden Energiedichten erheblich höher sind als die in magnetischen. Leider bietet die kurze Bearbeitungszeit für die vorliegende Arbeit keine Möglichkeit, das Phänomen zu untersuchen.

Kernresonanz

Bei hohen Frequenzen ist das magnetische Feld innerhalb einer Reluktanz ggf. nicht mehr homogen verteilt. Die Modellierung als ein einziges Netzwerkelement ist damit prinzipiell nicht mehr möglich (siehe Abschnitt 1.1.2, S. 11). Zudem treten elektrische (Verschiebungs-)Ströme im magnetischen Material auf, sodass die getrennte Modellierung von magnetischem und elektrischem Netzwerk nicht mehr möglich ist. Weil der Effekt der Kernresonanz erst bei weitaus höheren Frequenzen auftritt als bei den in dieser Arbeit betrachteten, wird kein Versuch unternommen, das Phänomen korrekt zu modellieren und in ein magnetisches Netzwerk zu transformieren.

1.1.5 Topologien

Nachdem die Elemente magnetischer Netzwerke vorgestellt sind, sollen in diesem Abschnitt die üblichsten Netzwerktopologien untersucht werden. Wegen der in der Praxis geringen Vielfalt magnetischer Netzwerke ist die folgende Liste fast erschöpfend, wie sie Induktivitäten, vormagnetisierte Induktivitäten, Transformatoren,

Transduktivitäten und Paraformatoren umfasst. Zu jeder dieser Netzwerktopologien werden im Folgenden der Aufbau, Eigenschaften und Besonderheiten genannt.

Induktivität

Das einfachste magnetische Netzwerk beinhaltet lediglich eine Quelle und eine Reluktanz. Wenn es sich bei der Quelle um eine magnetische Spannungsquelle in Form einer Wicklung handelt, wird diese Anordnung Induktivität genannt. Der magnetische Widerstand setzt sich aus mehreren in Reihe geschalteten Reluktanzen zusammen, der parasitären Reluktanz eines Kreises aus ferromagnetischem Material ("Kern") und hauptsächlich der eines kleinen Luftspalts in diesem (s. *Abb. 1.10a*).

Die wichtigste Eigenschaft des Bauteils Induktivität ist die gleichnamige elektrodynamische Netzwerkgröße mit dem Formelzeichen L. Sie wird stets in Abhängigkeit des elektrischen Stroms I angegeben, welcher die magnetische Spannung V der Quelle bestimmt. Man beachte, dass die feldstärkeabhängige effektive Länge des magnetischen Pfades l_{eff} für die Berechnung irrelevant ist, denn:

$$H(I) = \frac{1}{l_{\text{eff}}(H(I))} NI \qquad \Rightarrow^{(1.19)} \qquad V(H(I)) = V(I) = NI \tag{1.36}$$

Die Definiton der Induktivität folgt zu:

$$L(I) := \frac{|\Phi(V(I))|}{|I|} \tag{1.37}$$

Der Zusammenhang $\Phi(V)$ ist bereits bei der Definition der Reluktanz ausgeführt(siehe Abschnitt 1.1.3, S. 14). Der Zusammenhang $V(I)$ ist stets eine bijektive Funktion.[33] Die in *Gl. 1.37* definierte Induktivität soll präzise als absolute Induktivität L beeichnet werdern, um sie von der differentiellen Induktivität L_{diff} abzugrenzen:

$$L_{\text{diff}}(I) := \frac{\partial \Phi(V(I))}{\partial I} \tag{1.38}$$

Diese Definition ermöglicht eine intuitive Schnittstelle zwischen dem magnetischen Netzwerk, welches die Induktivität beschreibt und dem elektrischen Netzwerk, in welches sie so eingebunden ist, dass über den Klemmen der Wicklung die Spannung u abfällt und durch die Wicklung der Strom i fließt, in Form einer gewöhnlichen eindimensionalen DGL (sogenannte „Spulengleichung“):

$$u(i) = \frac{\partial}{\partial t}\left(l_{\text{diff}}(i) \cdot i\right) \tag{1.39}$$

Diese Schnittstelle lässt sich einfach aus dem Induktionsgesetz herleiten:

$$u = \oint_{(\partial \boldsymbol{A})} \boldsymbol{E} \mathrm{d}\boldsymbol{s} = -\oiint_{(\boldsymbol{A})} \frac{\partial \boldsymbol{B}}{\partial t} \mathrm{d}\boldsymbol{A} = -N\frac{\mathrm{d}\phi}{\mathrm{d}t} = -\frac{\mathrm{d}\phi}{\mathrm{d}t} = -\frac{\mathrm{d}}{\mathrm{d}t}\left(l_{\text{diff}} i\right) = l_{\text{diff}}\frac{\partial i}{\partial t} + i\frac{\partial l}{\partial t} \tag{1.40}$$

[33] Der Proportionalitätsfaktor N darf als konstant angenommen werden.

Man beachtete, dass ein sehr einfaches elektrisches Netzwerk aus lediglich dem Zweipol, welcher durch die Klemmen der Wicklung gebildet wird, und einem weiteren Zweipol bestehen kann, die miteinander verbunden sind, siehe *Abb. 1.11a*. Selbst wenn der weitere Zweipol eine statische Kennlinie aufweist, wie beispielsweise eine reale elektrische Spannungsquelle $u_\mathrm{q} = u_0 - R_\mathrm{i} i$, wird das Netzwerk mit der Maschengleichung

$$\begin{aligned} u_\mathrm{q} &= u \\ u_0 - R_\mathrm{i} i &= \frac{\partial}{\partial t}(l(i)i) \\ &= l(i)\frac{\partial i}{\partial t} + \frac{\partial l(i)}{\partial t} i \end{aligned} \tag{1.41}$$

bereits durch eine DGL erster Ordnung beschrieben:

$$l(i)\frac{\partial i}{\partial t} + \left(\frac{\partial l(i)}{\partial t} + R_\mathrm{i}\right) i = u_0 \tag{1.42}$$

Im folgenden wird ein Netzwerk mit einer idealen elektrischen Spannungsquelle und einer realen Induktivität betrachtet. Das verwendete magnetische Netzwerk ist in *Abb. 1.10* zusammen mit seiner Kennlinie und das gesamte Netzwerk in *Abb. 1.11* zusammen mit dem elektrischen Netzwerk und der LI-Kennlinie dargestellt. Die Ordnung des gesamten Systems wird durch die Ordnung der DGL der Spulengleichung in *Gl. 1.39* bestimmt.[34]

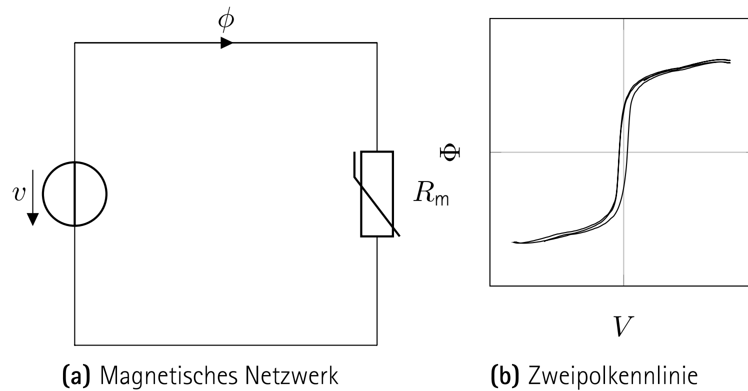
Wegen der Nichtlinearität der Funktion $\Phi(V(I))$ bei Reluktanzen aus typischen Kernmaterialien (siehe Abschnitt 1.1.2, S. 11) sinkt die Induktivität bei betragsmäßig hohen elektrischen Strömen, wie in *Abb. 1.11c* illustriert. Während in praktischen Anwendungen die Ströme meist innerhalb des Bereichs der konstanten Induktivität bleiben sollen, wird im Rahmen dieser Arbeit stets ein größerer Bereich der Kennlinie untersucht, weil das Prinzip der orthogonalen Magnetisierung insbesondere in den Betriebsbereichen der sinkenden Induktivität genutzt wird (siehe Abschnitt 1.4.3, S. 44).

Auf magnetische Netzwerke, die keine elektrisch gespeiste Quelle, sondern bspw. nur Permanentmagneten als Quellen enthalten, lässt sich die Definition der Induktivität nicht anwenden. Sie spielen in der Elektrotechnik kaum eine Rolle und werden deshalb hier nicht behandelt.

Vormagnetisierte Induktivität

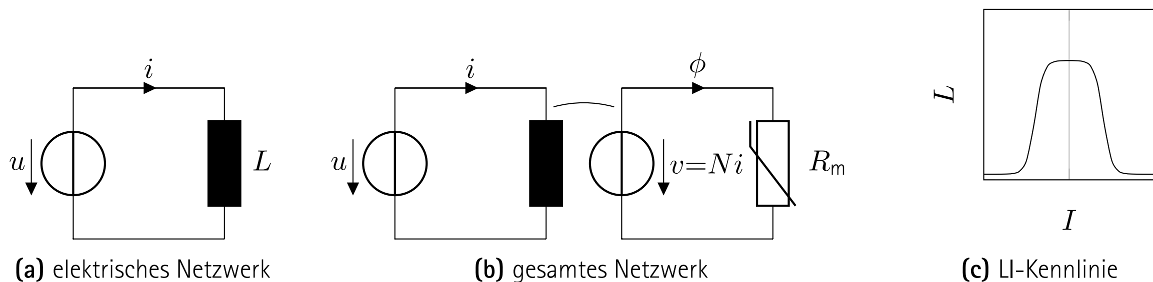
Bei einer Induktivität können beliebig viele nichtlineare Reluktanzen stets zu einer Gesamtreluktanz zusammengefasst werden. Bei einer Erhöhnung der Anzahl an Quellen ist eine Zusammenfassung jedoch nicht trivial, weil diese Quellen unterschiedliche Abhängigkeiten von Größen außerhalb des magnetischen Netzwerkes haben können. Bspw. können zwei Wicklungen unabhängig voneinander angesteuert werden. Dieses Prinzip wird bei sog. Vormagnetisierten Induktivitäten genutzt, wo typischerweise eine Flussquelle in Form eines Per-

[34] Für den stark vereinfachten Fall, dass der ohmsche Widerstand sehr klein wird $R \to 0\,\Omega$, handelt es sich um eine unechte DGL, die durch einfache Integration gelöst werden kann. Diese Vereinfachung wird in der Leistungselektronik manchmal angenommen.



(a) Magnetisches Netzwerk **(b)** Zweipolkennlinie

Abb. 1.10: Aufbau und Verhalten einer Induktivität. Magnetisches Netzwerk (links) und beispielhafte Magnetisierungskennlinie mit Material „NO10" (rechts).



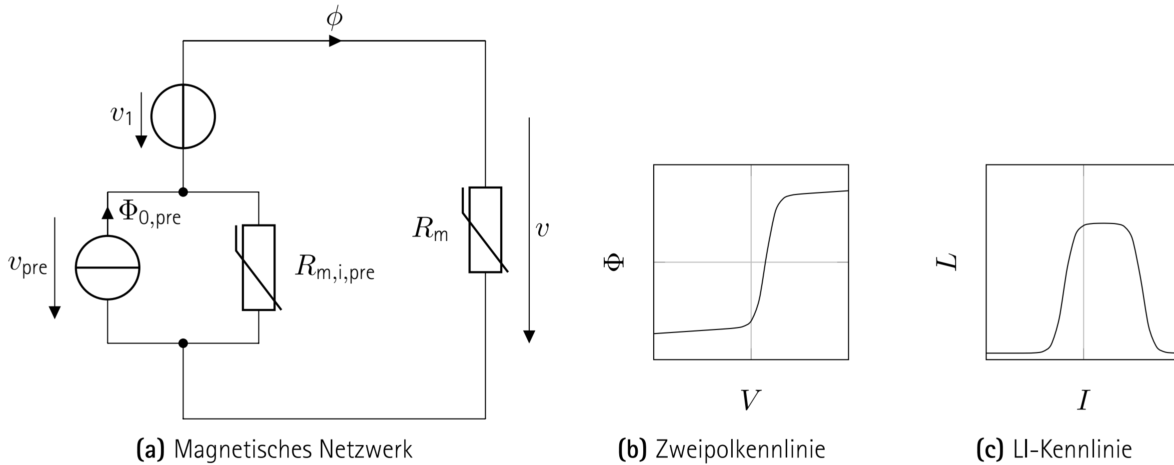
(a) elektrisches Netzwerk **(b)** gesamtes Netzwerk **(c)** LI-Kennlinie

Abb. 1.11: Einfaches elektrisches Netzwerk mit einer Induktivität. Typische Darstellung mit verstecktem magnetischem Netzwerk (links), vollständige Darstellung mit Kopplung (mittig) und typische Charakterisierung mit beispielhafter Kurve (rechts).

manentmagneten in Reihe mit einer Spannungsquelle in Form einer Wicklung geschaltet ist:

$$L_{\mathrm{diff}}(I) := \frac{\mathrm{d}}{\mathrm{d}I}\Phi(V_1(I) + V_{\mathrm{pre}}(\Phi)) \tag{1.43}$$

Die Innenreluktanz der Flussquelle kann meist vernachlässigt werden, sodass die Berechnung der „Vorspannung" V_{pre} über dem PM einfach ist: Es ergibt sich eine von der Reluktanz R_{m} abhängige, aber von der magnetischen Spannung v_1 und damit auch vom elektrischen Strom unabhängige Vorspannung. Die Unabhängigkeit gilt natürlich nur solange, wie der Überlagerungssatz gilt, d. h., die Reluktanz muss im linearen Bereich betrieben werden, was in der Leistungselektronik ohnehin fast immer angestrebt wird. Durch das Hinzufügen der Flussquelle wird effektiv die Kennlinie $L(I)$ an der I-Achse verschoben, wie in *Abb. 1.12c* dargestellt. Auf die Herleitung des Einflusses der Innenreluktanz $R_{\mathrm{m,i,pre}}$ der vormagnetisierenden Quelle soll der Übersicht halber nicht eingegangen werden. Ggf. bringt der Permanentmagnet noch eine weitere Nichtlinearität für $\Phi(V)$ und damit auch für $L(I)$ sowie eine Frequenz-, Temperatur- oder sonstige Abhängigkeit in das System ein, welche hier nicht weiter betrachtet werden. Die veränderte Netzwerktopologie und die sich ergebende Zweipolkennlinie sind in *Abb. 1.12* abgedruckt.



(a) Magnetisches Netzwerk **(b)** Zweipolkennlinie **(c)** LI-Kennlinie

Abb. 1.12: Aufbau und Verhalten einer Vormagnetisierten Induktivität. Magnetisches Netzwerk mit magnetischer Quelle aus Wicklung v_1, Permanentmagnet $\Phi = \Phi_{0,\text{pre}} - \frac{V_{\text{pre}}}{R_{\text{m,i,pre}}}$ und Reluktanz R_{m} (links). Durch PM verschobene Zweipolkennlinie (mittig) und verschobene LI-Kennlinie (rechts).

Transformator

Die zuvor betrachteten magnetischen Netzwerkelemente weisen alle eine statische Kennlinie auf, was ihre Beschreibung einfach gestaltet. Transformatoren sind nahezu identisch wie Vormagnetisierte Induktivitäten aufgebaut, verwenden allerdings mindestens zwei Quellen in Form von Wicklungen.[35] Deshalb können die magnetische Spannung $v(h(i(t)))$, die Flussänderung $\frac{\partial\phi(t)}{\partial t} = \frac{1}{n}u(t)$ und die Kennlinie $\Phi(V)$ nicht mehr als unabhängige Variablen angenommen werden, sondern sie werden durch das elektrische Netzwerk bestimmt, in welches die Wicklung eingebunden ist. Durch Energiespeicher in den elektrischen Netzwerken (Induktivitäten und Kapazitäten) sowie durch aktive Bauelemente (Schalter), kann die DGL quasi beliebig kompliziert werden. Das betrifft Nichtlinearität, Nichtdefinitheit, Störgrößen und Ordnung der DGL. Es gilt aber vereinfachend:

$$\frac{\partial\phi(t)}{\partial t} = \sum_{j=1}^{n} \frac{1}{n_j} u_j(t) \qquad v(t) = \sum_{j=1}^{n} n_j i_j(t) \tag{1.44}$$

$$\frac{\partial\phi(t)}{\partial t} = f\left(\frac{1}{N_1}u_1(t) + \frac{1}{N_2}u_2, t, ...\right) \tag{1.45}$$

Weil die Charakterisierung von Transformatoren den erwarteten Umfang dieses Abschnitts weit übersteigen würde, sei hier lediglich auf [18, S. 227 ff.] verwiesen.

[35] Im einfachsten Fall ist dabei der PM in *Abb. 1.12a* durch eine Wicklung ersetzt.

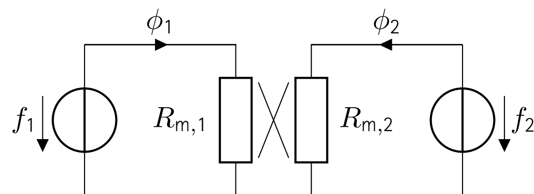


Abb. 1.13: orthogonale Magnetisierung

Transduktivität und Paraformator

Beim Transformator (s. o.) erzeugen zwei von elektrischen Strömen durchflossene Wicklungen zwei parallel zueinander stehende magnetische Felder. Wenn diese Felder stattdessen orthogonal zueinander stehen, heißt die Anordnung entweder Transduktivität oder Paraformator, abhängig davon, wie sie verwendet wird. In beiden Fällen wird der Mechanismus der orthogonalen Magnetisierung genutzt, der erst später vorgestellt wird (siehe Abschnitt 1.4, S. 42).

Bei einer Transduktivität ist die primäre Wicklung in ein elektrisches Netzwerk eingebunden, wo sie als Induktivität (s. o.) dient. Ein Strom durch die sekundäre Wicklung kann die primäre Induktivität beeinflussen, indem das magnetische Material in die Sättigung getrieben wird (siehe Abschnitt 1.4, S. 42). Eine auf diese Weise beeinflussbare Induktivität bietet vielfältige technische Anwendungsmöglichkeiten. [25] [17]

Beim Paraformator wird die sekundäre Wicklung nicht nur verwendet, um die primäre zu beeinflussen, sondern es steht eine Energieübertragung zwischen beiden Wicklungen im Vordergrund, genau wie beim Transformator. Der Paraformator ist ausführlich in [31] untersucht.

1.1.6 Lösung

Wie bereits zuvor beschrieben (siehe Abschnitt 1.1.1, S. 6), wird ein magnetisches Netzwerk durch die Maschen- und Knotengleichungen seiner Topologie und durch die charakteristischen Kennlinien der Netzwerkelemente beschrieben. Falls Abhängigkeiten zu anderen Systeme bestehen, sind deren Zusammenhänge ebenfalls zu beachten. Sofern das Netzwerk eine reale physikalische Anwendung gerecht darstellt, weist es eine Lösung auf. Diese ist im Allgemeinen nur dann eindeutig, wenn das Gleichungssystem zur Beschreibung keine DGL enthält, was allerdings nahezu nie der Fall ist. Eindeutigkeit kann auch erreicht werden, indem für sämtliche DGL in der Zeitdomäne Anfangswerte und für sämtliche DGL in der Raumdomäne umfassende Randbedingungen gelten. [4]

Einen Einstieg in das Lösen von nichtlinearen Netzwerken gibt [18], wobei nicht auf verlustbehaftete und anisotrope Materialien eingegangen wird. Einen Überblick über magnetische Netzwerke mit verlustbehafteten Reluktanzen bietet [21]. Darin sind insbesondere die Eindeutigkeit, Stabilität, Verfahren zur Annäherung, die Input-Output-Charakteristik und die Berechnung letzterer thematisiert. Allgemeine Verfahren zur Charakterisierung und Lösung von nichtlinearen DGL können [23] entnommen werden. In [24] sind Quellen für die Mo-

dellierung von magnetischen Netzwerken mit orthogonaler Magnetisierung zusammengetragen, insbesondere [31].

Ein konkretes Verfahren zur Lösung von Netzwerkproblemen ist später implementiert (siehe Kapitel 5, S. 119). Für nachfolgende Arbeiten sei darauf hingewiesen, dass die „Kompetenzgruppe Magnetische Netzwerke" am betreuenden Institut zum Ergebnis kommt, dass nichtlineare elektrische Widerstände, welche nichtlineare Reluktanzen repräsentieren sollen, bei der Simulation mit der Software „PLECS" einen geradezu absurd hohen Rechenaufwand verursachen. Die Verwendung einer Umsetzungstablle, engl. Lookup Table (LUT) scheint effizienter zu sein.

1.2 Magnetische Felder

Im vorangegangenen Abschnitt sind magnetische Netzwerke untersucht, bei welchen es sich stets um eine Abstraktion magnetischer Felder handelt. Wenn die für diese Abstraktion notwendigen Voraussetzungen (siehe Abschnitt 1.1.2, S. 11) nicht erfüllt sind, oder wenn ein geeignetes Netzwerkmodell eine unzweckmäßig hohe Komplexität aufweisen würde, muss häufig auf Feldsimulationen zurückgegriffen werden. Weil das Thema bereits eigenständig mehrere Bücher füllen kann, wird in diesem Abschnitt nur ein kurzer Blick auf den theoretischen Hintergrund mitsamt den hier getroffenen Vereinfachungen gegeben. Anschließend geschieht ein Ausblick auf die Lösungsmethoden und die Feldtheorie wird mit der magnetischen Netzwerktheorie verknüpft. Die in diesem Abschnitt vorgestellten Erkenntnisse werden später verwendet, um darauf aufbauend eine Software zu entwickeln (siehe Kapitel 5, S. 119).

1.2.1 Mathematischer Hintergrund

Bei den untersuchten magnetischen Feldern handelt es sich um höchstens langsam veränderliche Größen und damit um quasistationäre Felder. Deren Ursachen liegen außerhalb des zu untersuchenden Raumes, d. h. die elektrischen Wicklungen oder Permanentmagneten werden nicht mitbeobachtet. Stattdessen werden ausschließlich die reluktant wirkenden Raumgebiete betrachtet, also der magnetische Kern und ggf. Luftspalte. Unter diesen Annahmen vereinfacht sich das Durchflutungsgesetz:

$$\begin{aligned} rot(\boldsymbol{H}) \quad &= \boldsymbol{J} + \frac{\partial}{\partial t}\boldsymbol{D} \\ &= \boldsymbol{0} \end{aligned} \tag{1.46}$$

Das wirbelfreie $\boldsymbol{H}$-Feld kann deshalb als Gradient einer Skalarfunktion beschrieben werden. Definitionsgemäß sei ϕ_m das magnetische Skalarpotential: [22]

$$\boldsymbol{H} =: -\,\mathrm{grad}(\phi_\mathrm{m}) \tag{1.47}$$

Im Folgenden wird davon ausgegangen, dass der Zusammenhang $\boldsymbol{B}(\boldsymbol{H})$ bekannt ist. Außerdem soll er zu jedem Zeitpunkt stückweise linear sein (siehe Abschnitt 1.1.2, S. 11).

Verlustfreie, isotrope Permeabilität

Zunächst wird der Fall einer verlustfreien, isotropen Permeabilität betrachtet: (siehe Abschnitt 1.1.2, S. 11)

$$\boldsymbol{B} := \mu\boldsymbol{H}; \qquad \mu \in \mathbb{R}\cdot 1\frac{\mathrm{Vs}}{\mathrm{Am}} \tag{1.48}$$

Durch Einsetzen in die Definition der Quellenfreiheit der magnetischen Flussdichte ergibt sich die zugrundeliegende Differentialgleichung der Magnetostatik:

$$\begin{aligned} 0 &= \mathrm{div}(\boldsymbol{B}) && |\boldsymbol{B} = \mu\boldsymbol{H} \\ &= \mathrm{div}(\mu\boldsymbol{H}) && |\mu \in \mathbb{R}\cdot 1\frac{\mathrm{Vs}}{\mathrm{Am}} \\ &= \mu\,\mathrm{div}(\boldsymbol{H}) + \boldsymbol{H}\,\mathrm{grad}(\mu) && |\boldsymbol{H} := -\,\mathrm{grad}(\phi_\mathrm{m}) \\ &= -\,\mu\,\mathrm{div}(\mathrm{grad}(\phi_\mathrm{m})) - \mathrm{grad}(\phi_\mathrm{m})\,\mathrm{grad}(\mu) \\ &= \Delta\phi_\mathrm{m} + \frac{1}{\mu}\cdot\mathrm{grad}(\mu)\cdot\mathrm{grad}(\phi_\mathrm{m}) \end{aligned} \tag{1.49}$$

Das Ergebnis ist eine elliptische DGL zweiter Ordnung, welche den energetisch geringsten Zustand beschreibt. [22]

Verlustfreie, anisotrope Permeabilität

Obwohl die oben genannte DGL häufig als der Magnetostatik zugrundeliegend bezeichnet wird, [22] vernachlässigt sie anisotrope Materialien:

$$\boldsymbol{B} := \mu\boldsymbol{H} \qquad \mu = \mathrm{diag}(\mu_x, \mu_y, \mu_z) \qquad \mu_x, \mu_y, \mu_z \in \mathbb{R}\cdot 1\frac{\mathrm{Vs}}{\mathrm{Am}} \tag{1.50}$$

In dem Fall kann *Gl. 1.49* nicht in der vereinfachten Form angewandt werden. Weil die Permeabilität μ_x, μ_y, μ_z nicht nur stückweise konstant, sondern auch eine gerade Funktion ist, gilt weiterhin die Quellenfreiheit der magnetischen Feldstärke. Die zugrundeliegende DGL lautet entsprechend:

$$0 = \mathrm{div}(\mu\,\mathrm{grad}(\phi_\mathrm{m})) \tag{1.51}$$

Verlustbehaftete, anisotrope Permeabilität

Die zuvor getroffenen Einschränkungen für die Gültigkeit der DGL lassen sich aufheben, indem ein verlustbehaftetes, anisotropes Material betrachtet wird. Nahezu sämtliche reale magnetischen Materialien lassen sich damit modellieren.

Der Ansatz für ist denkbar einfach: zu dem durch den Gradienten des Skalarfeldes $\mu\,\mathrm{grad}(\phi_\mathrm{m})$ erzeugten B-Feld wird ein weiteres Feld B_0 addiert, welches eine Raum- und eine Zeitabhängigkeit aufweisen kann. Es stellt für ein infinitesimales Raumgebiet zu einem bestimmten Zeitpunkt den $\boldsymbol{B}$-Achsenabschnitt der Magnetisierungskennlinie $\boldsymbol{B}(\boldsymbol{H})$ dar. Die gültige DGL lautet deshalb:

$$0 = \mathrm{div}(\boldsymbol{B}_0 + \mu\,\mathrm{grad}(\phi_\mathrm{m})) \tag{1.52}$$

1.2.2 Lösung

Nachdem die zugrundeliegende DGL eines Problems identifiziert ist, kann eine Lösung in Abhängigkeit der Randbedingungen bestimmt werden. Bei einfachen Problemen ist eine analytische Lösung oder immerhin eine analytische Näherung möglich. Meist werden aber die FEM oder Finite-Differenzen-Methode (FDM) angewandt. Wenn das Problem zeitabhängig ist, weist dessen Lösung sicherlich ebenfalls eine Zeitabhängigkeit auf, die bspw. mit finiten Differenzen, der komplexen Wechselgrößenrechnung oder mit der Regelungstechnik entlehnten Näherungsbetrachtungen bearbeitet werden kann. In einem folgenden Abschnitt wird genauer auf die in dieser Arbeit verwendeten Methoden eingegangen (siehe Abschnitt 1.3, S. 35). Außerdem werden sie in einer Software implementiert (siehe Kapitel 5, S. 119).

Für alle betrachteten Fälle ergibt sich eine homogene DGL. Entsprechend ergibt sich immer die triviale Lösung. Für die physikalische und eindeutige Lösung werden Randbedingungen benötigt, die sich typischerweise aus dem magnetischen Netzwerk ergeben, in welche das untersuchte Raumgebiet eingebunden ist (siehe Abschnitt 1.2.3, S. 33). Möglich sind Dirichlet-Randbedingungen, bei denen das magnetische Skalarpotential auf dem Rand festgelegt ist und Neumann-Randbedingungen, welche die Normal- oder Tangentialkomponente des H-Feldes bestimmen.

1.2.3 Einbindung in Netzwerktheorie

Die Definitionen der Netzwerkgrößen V und Phi aus den Feldgrößen $\boldsymbol{H}$ und $\boldsymbol{B}$ sind bereits zuvor behandelt (siehe Abschnitt 1.1.1, S. 6). Die umgekehrte Berechnung wird allerdings ebenfalls benötigt, insbesondere zur Festlegung von Randbedingungen. Die Vorgabe einer Netzwerkgröße führt im Allgemeinen zu nicht eindeutigen Randbedingungen.

Das magnetische Skalarpotential und die magnetische Spannung weisen die Ähnlichkeit auf, dass sie als das Wegintegral des H-Feldes bestimmt werden können. Aufgrund dieser Analogie können sie bei magnetostatischen Problemen äquivalent verwendet werden:

$$\int_a^b \boldsymbol{H} \,\mathrm{d}\, s = \phi_{\mathrm{m}}(b) - \phi_{\mathrm{m}}(a) \tag{1.53}$$

Bei der Berechnung des magnetisches Skalarpotentialfeldes ergeben sich damit Dirichlet-Randbedingungen, deren Niveau frei gewählt werden kann.

Die magnetische Flussdichte ergibt sich einfach aus dem Quotienten von Fluss und Fläche, wobei sie normal zur Fläche fließt:

$$\boldsymbol{B} = \frac{\Phi}{A} \cdot \frac{\boldsymbol{A}}{A} \tag{1.54}$$

Für die Berechnung des Skalarpotentials ergeben sich Neumann-Randbedingungen:

$$-\mu \,\mathsf{grad}(\phi_\mathsf{m}) = \frac{\Phi}{A} \cdot \frac{\boldsymbol{A}}{A} \tag{1.55}$$

Dabei wird eine homogen über den Randflächenbereich verteilte Flussdichte angenommen. Sollte diese Annahme unzulässig sein, bspw. weil das Material Inhomogenitäten aufweist, muss der betrachtete Raumbereich vergrößert werden. In dieser Arbeit sollen außerhalb der Bereiche von Feldbetrachtungen allerdings magnetische Rückschlüsse verwendet werden, die eine geeignete Feldform herbeiführen.

1.3 Simulation

In den vorangegangenen Abschnitten sind diverse DGL aufgestellt, die noch unvollständige Lösungsverfahren aufweisen. Bspw. ergibt sich bei der Netzwerkberechnung eine Gleichungssystem in Abhängigkeit der charakteristischen Funktionen der Netzwerkelemente. Insbesondere Zeit- und Raumabhängigkeiten des Gleichungssystems sind bislang nicht betrachtet. Die Lösung eines solchen, im Allgemeinen nichtlinearen Systems von DGL wird im Folgenden Abschnitt thematisiert.

Zunächst wird die Berechnung von zeitabhängigen Größen untersucht, wobei das Euler-Polygonzug- und das Heun-Verfahren vorgestellt werden. Anschließend wird der Ansatz der Differenzenquotienten auf Raumabhängigkeiten erweitert und das mathematische Fundament der FDM betrachtet. Die sich dabei ergebenden Lineare Gleichungssysteme (LGS) erhalten schließlich geeignete Lösungsverfahren. Alle Überlegungen gelten sowohl für lineare als auch für nichtlineare Partielle Differentialgleichungen (PDGL) beliebig hoher Ordnung.

1.3.1 Zeitabhängigkeit

Sowohl bei der Berechnung von Netzwerken als auch von Feldern werden häufig zeitabhängige Vorgänge untersucht. Sofern keine einfache Darstellung dieser Abhängigkeit, bspw. mit der komplexen Wechselgrößenrechnung, möglich ist, wird eine zeittransiente Betrachtung notwendig. Folgend wird die Problemkategorie analysiert und ein geeignetes Verfahren zur Lösung vorgestellt.

Bei der Simulation von zeitabhängigen Systemen lassen sich die Eigenheiten dieser physikalischen Dimension zunutze machen:

- Zeit wird bei transienten Vorgängen als eindimensional betrachtet, bei stationären Zuständen sogar nur als nulldimensional.
- Der Zustand eines physikalischen Systems kann nur von seinem in der Vergangenheit unmittelbar zuvor liegenden Zustand abhängen, d. h. Informationen können sich nur unidirektional ausbreiten und beeinflussen nur jene im unmittelbar folgenden Zeitpunkt. Entsprechend ist es bei einer Diskretisierung des betrachteten Zeitraums nicht nötig, ein Gleichungssystem aufzustellen, sondern der Zustand zu einem Zeitpunkt kann aus dem vorangegangenen Zustand vollständig berechnet werden.[36]

Betrachtet wird ein System, welches durch die allgemeine DGL beschrieben wird:

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{y}(t) = \boldsymbol{f}(\boldsymbol{y}(t), t) \tag{1.56}$$

[36] Für die Berechnung mancher Zustandsgrößen, bspw. zweiter oder höherer Ableitungen nach der Zeit, werden in der praktischen Umsetzung letztlich doch weitere Zeitpunkte für die Berechnung herangezogen, auch in dieser Arbeit (siehe Kapitel 5, S. 119). Für die Lösung von DGL n-ter Ordnung werden bei üblichen Lösungsalgorithmen n vorangegangene Zeitpunkte zur Berechnung des aktuellen betrachteten benötigt. Generell lassen sich allerdings alle DGL n-ter Ordnung auf DGL erster Ordnung transformieren. [4]

Außerdem sei ein Anfangswert $\boldsymbol{y}(t_0)$ gegeben. Diese Betrachtungsweise eignet sich auf für Systeme höherer Ordnung. Diese sollen in mehrdimensionale Probleme erster Ordnung entsprechend der Transformationsvorschrift in [4] umgewandelt werden. Das Zeitgebiet wird in n aneinander angrenzende Teilgebiete diskretisiert, sodass nur noch die Zeitpunkte $t_0, ..., t_n$ betrachtet werden. Im einfachsten Fall wird das Eulersche Polygonzug-Verfahren angewandt, welches auf Basis des Differenzenquotienten

$$\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{y}(t_k) = \lim_{t_{k+1}-t_k \to 0} \left(\frac{\boldsymbol{y}(t_{k+1}) - \boldsymbol{y}(t_k)}{t_{k+1} - t_k} \right) \tag{1.57}$$

die folgende Iterationsvorschrift beinhaltet: [4]

$$\boldsymbol{y}(t_{k+1}) = \boldsymbol{y}(t_k) + (t_{k+1} - t_k)\boldsymbol{f}(\boldsymbol{y}(t_k), t_k) \tag{1.58}$$

Das Euer-Polygonzugverfahren konvergiert linear, d. h. der Approximationsfehler ist antiproportional zur Schrittweite $t_{k+1} - t_k$. Es sind weitere Verfahren mit höheren Konvergenzordnungen bekannt, bspw. das Heun- oder das Euler-Runge-Verfahren, die beide quadratisch konvergieren. Dabei wird für die Berechnung des Zustandes $y(t_{k+1})$ jeweils der Zustand $y(t_{k+2})$ prädiziert. Das klassische Runge-Kutta-Verfahren konvergiert sogar quartisch, wofür es drei Prädiktionen pro Iterationsschritt durchführt. Im Folgenden wird das Heun-Verfahren mit Euler-Prädiktor vorgestellt.

Das Heun-Verfahren approximiert das Inkrement[37] besser als das Eulerverfahren, indem es die Inkrementfunktion $\boldsymbol{f}$ an der Stelle t_k und t_{k+1} auswertet und den Mittelwert bildet: [4]

$$\boldsymbol{y}(t_{k+1}) = \boldsymbol{y}(t_k) + (t_{k+1} - t_k)\frac{\boldsymbol{f}(\boldsymbol{y}(t_k), t_k) + \boldsymbol{f}(\boldsymbol{y}_{\text{est}}(t_{k+1}), t_{k+1})}{2} \tag{1.59}$$

Man beachte, dass die Inkrementfunktion nicht mit dem tatsächlichen Wert von $\boldsymbol{y}(t_{k+1})$ ausgewertet werden kann, weil dieser erst in jedem Iterationsschritt berechnet wird. Er wird deshalb zunächst mit dem Euler-Prädiktor geschätzt:[38]

$$\boldsymbol{y}(t_{k+1}) \approx \boldsymbol{y}_{\text{est}}(t_{k+1}) = \boldsymbol{y}(t_k) + (t_{k+1} - t_k)\boldsymbol{f}(\boldsymbol{y}(t_k), t_k) \tag{1.60}$$

Dabei sollte das Ergenis von $\boldsymbol{f}(\boldsymbol{y}(t_k), t_k)$ noch gespeichert sein, sodass der Prädiktor sich nicht wesentlich auf den Bedarf an Rechenleistung des gesamten Verfahrens auswirkt. Der etwas mehr als doppelt so hohe Rechenaufwand für einen Iterationsschritt beim Heun-Verfahren als beim Eulerschen Polygonzug-Verfahren verschwindet bei der Anforderung von besonders geringen Fehlern in der Lösung wegen der dafür weniger fein benötigten Diskretisierung des Zeitgebiets.

[37] Das Inkrement ist anschaulich die Steigung der Funktion $\boldsymbol{y}$ und für eine unendlich feine Diskretisierung gleich dem Funktionswert von $\boldsymbol{f}$.

[38] Es ist prinzipiell möglich, das Ergebnis des Iterationsschritts zu verwenden, um damit iterativ bessere Ergenisse für den prädizierten zu erhalten und somit den Mittelwert der Inkrementfunktionen zu verbessern, worauf hier nicht weiter eingegangen werden soll.

1.3.2 Raumabhängigkeit

Bei der Berechnung von Feldern liegt, anders als bei einem einzelnen Netzwerkelement, eine Raumabhängigkeit der Lösung vor. Für einfache Geometrien und Materialparameter lassen sich diese noch analytisch bestimmen. Häufig muss allerdings auf numerische Methoden, insbesondere die FEM und die FDM, zurückgegriffen werden oder diese Methoden bieten einen geringeren Arbeitsaufwand oder eine geringere Fehleranfälligkeit. Im Folgenden werden die FEM und FDM begründet und ihre mathematische Grundlage angerissen. Die Anwendung geschieht in einer eigenen Software (siehe Kapitel 5, S. 119).

Es sei D ein beliebiger Differentialoperator von höchstens zweiter Ordnung. Außerdem sei ϕ eine beliebige skalare (Potential-)Funktion, μ eine beliebige Funktion und f eine beliebige skalare (Stör-)Funktion. In dieser Arbeit werden ausschließlich DGL betrachtet, welche in der Form beschrieben werden:

$$D(\phi, \mu) = f \tag{1.61}$$

Man beachtet, dass μ eine beliebige Funktion und nicht zwingend die Permeabilität eines magnetischen Materials sein muss (aber kann). In den bereits zuvor gegebenen Beispielen für solche DGL kann μ auch ein Operator sein (siehe Abschnitt 1.2.1, S. 31). Wie bereits beschrieben, wird dieser dann durch eine variable Funktion angenähert (siehe Abschnitt 1.1.2, S. 11).

Ähnlich zur im vorangegangenen Abschnitt diskutierten Diskretisierung der Zeit muss auch der Raum diskretisiert werden. Wenn eine kartesische Einteilung mit in jeder Dimension je konstanten Schrittweiten gewählt wird, heißt die darauf aufbauende Methode FDM bzw. für den Sonderfall von drei Raumdimensionen Finite-Volumen-Methode (FVM). Wegen ihrer Ähnlichkeit werden im Folgenden beide als FDM bezeichnet. Wenn die Raumdiskretisierung variabel ist, heißt die Methode FEM. Die Vor- und Nachteile dieser Methoden sind:

- Der wichtigste Vorteil der FEM ist die Möglichkeit, komplizierte Geometrien zu bewältigen, während die FDM dafür eine deutlich feinere Diskretisierung im gesamten Raum benötigt. [10]
- Die FEM ist mathematisch „sauberer" und führt häufig zu richtigeren Ergebnissen, zumal die Approximation zwischen zwei Stützstellen der Diskretisierung bei der FDM ungenau ist.
- Bei rechteckigen Geometrien mit ohnehin gleichmäßiger Vernetzung ist die FDM etwas schneller und kann unter Umständen auch deutlich genauer sein.
- Die FDM ist einfacher zu implementieren.

Im Folgenden wird von einer konstanten Schrittweite in allen Dimensionen innerhalb eines dreidimensionalen Volumens ausgegangen, was der FDM entspricht. Auf die Gründe für die Entscheidung zur Methode wird bei der Implementierung eingegangen (siehe Kapitel 5, S. 119). Das Prinzip ist allerdings auf eine variable Konkretisierung erweiterbar.

Zur Darstellung des Differentialoperators D genügt eine Verknüpfung des linearen partiellen Differentialope-

rators erster Ordnung

$$\frac{\partial}{\partial x} : y \mapsto \frac{\partial y}{\partial x} \tag{1.62}$$

mit sich selbst und ggf. anderen Operatoren, die keine Differentialoperatoren sind. Dementsprechend kann D mithilfe der Differenzenquotienten auf dem diskretisierten Raum dargestellt werden: [4]

vorderer Differenzenquotient der ersten Ableitung:

$$\frac{\partial}{\partial x} y(x) \approx \frac{y(x+h) - y(x)}{h} \tag{1.63}$$

hinterer Differenzenquotient der ersten Ableitung:

$$\frac{\partial}{\partial x} y(x) \approx \frac{y(x) - y(x-h)}{h} \tag{1.64}$$

zentraler Differenzenquotient der ersten Ableitung:

$$\frac{\partial}{\partial x} y(x) \approx \frac{y(x+h) - y(x-h)}{2h} \tag{1.65}$$

Mithilfe des zentralen Differenzenquotienten der ersten Ableitung lässt sich der zentrale Differenzenquotient der zweiten Ableitung bestimmen:
zentraler Differenzenquotient der zweiten Ableitung:

$$\frac{\partial^2}{\partial x^2} y(x) \approx \frac{y(x+h) - 2y(x) + y(x-h)}{h^2} \tag{1.66}$$

Auf diese Weise kann jeder beliebige Differentialoperator D hergeleitet werden:

$$\text{grad}\, f(x,y,z) \approx \frac{1}{2h} \begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{1.67}$$

$$\begin{aligned} \text{div}\, f(x,y,z) \approx \frac{1}{2h} \Big(& f(x+h,y,z) - f(x-h,y,z) + \\ & f(x,y+h,z) - f(x,y-h,z) + \\ & f(x,y,z+h) - f(x,y,z-h) \Big) \end{aligned} \tag{1.68}$$

$$\begin{aligned} \Delta f(x,y,z) \approx \frac{1}{h^2} \Big(& f(x+h,y,z) + f(x-h,y,z) + \\ & f(x,y+h,z) + f(x,y-h,z) + \\ & f(x,y,z+h) + f(x,y,z-h) - 6f(x,y,z) \Big) \end{aligned} \tag{1.69}$$

$$\ldots$$

Funktionswerte zwischen den Stützstellen des diskretisierten Raumes werden mithilfe linearer Splines angenähert.

Auf diese Weise ergibt sich ein LGS, dessen Koeffizientenmatrix die Werte der linken Seite der DGL in *Gl. 1.61* an den Stützstellen repräsentiert und dessen rechte Seite die Werte der rechten Seite der DGL an den selben Stützstellen annimmt. Der Lösungsvektor beinhaltet die Funktionswerte von ϕ an den Stützstellen. Die Koeffizientenmatrix ist im Allgemeinen dünn besetzt, weil in jeder Gleichung nur die Stützstellen in der Nachbarschaft zueinander betrachtet werden. Bei meist einigen tausend Stellen enthält die Koeffizientenmatrix entsprechend fast nur Nullen. Die Lösung des LGS wird im folgenden Abschnitt thematisiert (siehe Abschnitt 1.3.3, S. 39).

Es fällt auf, dass das LGS nur in Abhängigkeit von μ beschrieben werden kann. Wie bereits zuvor erwähnt, soll hier μ die Permeabilität eines Materials sein. Es ergibt sich deshalb die Notwendigkeit, dass μ als stückweise konstant betrachtet werden kann. Wenn μ von ϕ abhängt, kann der Banach'sche Fixpunktsatz angewandt werden. Dabei wird das Problem für eine geratene Funktion für μ gelöst. Aus dem Ergebnis wird die neue Funktion für μ bestimmt. Diese Iterationsvorschrift wird bis zur hinreichend guten Konvergenz wiederholt. Auf das Konvergenzkriterium des Banach'schen Fixpunktsatzes soll an dieser Stelle nicht weiter eingegangen werden.[39]

Prinzipiell können Symmetrien bei den Raumgeometrien, innerhalb deren eine Feldberechnung durchgeführt werden sollen, genutzt werden, um den Rechenaufwand zu reduzieren. Dabei wird nur ein Teil der Geometrie simuliert und mithilfe geeigneter Randbedingungen wird sichergestellt, dass die Feldverteilung in den anderen Gebieten durch einfaches Spiegeln an den Symmetrieachsen bestimmt werden kann. Weil dieses Hilfsmittel im Folgenden nicht verwendet wird (siehe Kapitel 5, S. 119), werden seine Grundlagen hier nicht erläutert.

1.3.3 Lineare Gleichungssysteme

Vorangegangen sind Verfahren erläutert, mit welchen sich zeit- und raumabhängige Differentialprobleme lösen lassen und die „lediglich" noch die Lösung von LGS erfordern. Deren Lösung und insbesondere deren rechenökonomische Lösung werden folgend erarbeitet.

Ein LGS kann in der Matrixschreibweise mit der Koeffizientenmatix A, dem Lösungsvektor $\boldsymbol{x}$ und der rechten Seite $\boldsymbol{b}$ dargestellt werden:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{1.70}$$

Mit einer regulären Koeffizientenmatrix[40] lautet die allgemeine Lösung:

$$\boldsymbol{x} = A^{-1}\boldsymbol{b} \tag{1.71}$$

[39] Es ist in [4] hergeleitet und charakterisiert.

[40] Eine reguläre Matrix ist quadratisch und hat vollen Rang. Das bedeutet, dass ein zu der Koeffizientenmatrix gehöriges LGS mit n Variablen durch n linear unabhängige Gleichungen beschrieben wird und dementsprechend genau eine Lösung aufweist.

Weil die Berechnung der inversen Matrix, bspw. mithilfe des Gauß-Algorithmus, bereits bei mittelgroßen LGS immens rechenaufwendig ist und sich keineswegs für die Lösung von großen LGS eignet,[3] werden für die Berechnung häufig lineare Iterationsverfahren verwendet. Bei den sogenannten Zerlege-Methoden wird die Koeffizientenmatrix zerlegt: [3]

$$A = N^{-1} + (A - N^{-1}) \tag{1.72}$$

Mit einem geschätzten Startvektor $\boldsymbol{x}^{(0)}$ lautet die Iterationsvorschrift entsprechend: [3]

$$\boldsymbol{x}^{(k+1)} = N\left(\boldsymbol{b} - (A - N^{-1})\boldsymbol{x}^{(k)}\right) \tag{1.73}$$

Zerlegemethoden bieten gegenüber direkten Verfahren zur Lösung von LGS einige Vor- und Nachteile:

- Die häufig dünn besetzte Koeffizientenmatrix muss nicht explizit gespeichert und verarbeitet werden, sondern lediglich ihre von Null verschiedenen Einträge und deren Positionen. Der Speicher- und Rechenaufwand steigt somit im besten Fall nur linear[41] und nicht quadratisch mit der Anzahl der Variablen.
- Iterationsverfahren können vorkonditioniert werden, indem der Startvektor auf Basis von physikalischen Überlegungen gut geschätzt wird.
- Es ist keine exakte Berechnung der Lösung möglich. Allerdings kann dies in einem erheblich verringerten Rechenaufwand resultieren.
- Für geänderte rechte Seiten muss das Verfahren von vorn und mit dem vollen Rechenaufwand ausgeführt werden.

In dieser Arbeit wird insbesondere das Jacobi-Verfahren angewandt, für welches gilt: [3]

$$N^{-1} = D_A \qquad\qquad A - N^{-1} = L_A + R_A \tag{1.74}$$

Dabei ist D_A eine Matrix, welche lediglich die Diagonalelemente der Koeffizientenmatrix und anderstellig nur Nullen enthält. R_A und L_A enthalten entsprechend nur die linken unteren bzw. die rechten oberen Elemente. Dadurch, dass N^{-1} hier eine Diagonalmatrix ist, wird die Invertierung trivial. Auf andere Verfahren mit komplizierteren N^{-1} soll an dieser Stelle nicht eingegangen werden.

Sofern die Koeffizientenmatrix strikt zeilen- oder strikt spaltendiagonaldominant ist, dient dies als hinreichende Bedingung für die Konvergenz des Jacobi-Verfahrens. [3] Sollte das nicht der Fall sein, kann ein Dämpfungsfaktor k im Bereich $0 \leq k < 1$ verwendet werden. Die Iterationsvorschrift in *Gl. 1.73* wird dafür geändert:

$$\boldsymbol{x}^{(k+1)} = (1-k)N\left(\boldsymbol{b} - (A - N^{-1})\boldsymbol{x}^{(k)}\right) + k\boldsymbol{x}^{(k)} \tag{1.75}$$

[41] Besonders große LGS mit zigtausenden oder mehr Variablen ergeben sich bei Feldberechnungen. Die dabei entstehenden Koeffizientenmatrizen sind sehr dünn besetzt (siehe Abschnitt 1.3.2, S. 37).

1.4 Modellierung

Ein umfangreicher Überblick über die bisherige Forschung am Prinzip der orthogonalen Magnetisierung ist in [24] gegeben. Es ist bemerkenswert, wie kurz und insbesondere mit wie wenigen Quellen belegt der darin enthaltene theoretische Teil ist. In diesem Abschnitt werden die bislang verfügbaren Theorien vorgestellt, wie die orthogonale Flussinteraktion modellierbar sein kann. Zunächst wird das Modell einer nicht vorhandenen Interaktion und seine Annahmen gezeigt. Weil im Rahmen dieser Arbeit aber explizit sämtliche Annahmen gebrochen werden sollen, um eine Interaktion zwischen orthogonalen Flüssen zu provozieren (siehe Kapitel 6, S. 139), werden anschließend ein allgemeines Modell und ein spezifisches Modell vorgestellt, welche von Dr. Eyup Salih Tez entwickelt und in [31] festgehalten sind. Schließlich wird darauf eingegangen, inwiefern ein Bedarf an weiteren Modellen besteht.

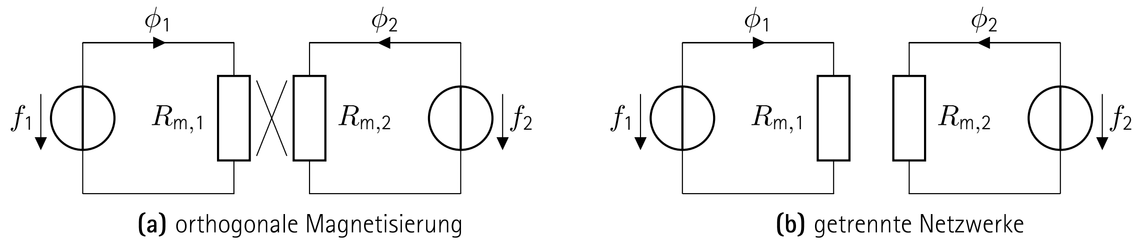
1.4.1 Keine Interaktion

Im einfachsten Fall besteht keine Interaktion zwischen zwei orthogonalen magnetischen Flüssen. Die dafür notwendigen Bedingungen sind identisch mit denen des der Modellierung zugrundeliegenden Superpositionsprinzips:

1. Die zur Anwendung kommenden physikalischen Gesetze sind lineare (Differential-)Gleichungen.

Bezogen auf reale magnetische Anwendungen lässt sich diese Voraussetzung erfüllen, wenn eine der folgenden, hinreichenden Bedingungen gegeben ist:

1. Das magnetische Material im Raumgebiet der Interaktion hat einen linearen Bereich und die maximale resultierende Feldstärke liegt darin $\hat{H} = \sqrt{\hat{H}_1^{\,2} + \hat{H}_2^{\,2}} < H_{\text{sat}}$.
2. Das magnetische Material im Raumgebiet der Interaktion hat einen linearen Bereich und durch passende Zeitabhängigkeiten der beiden Feldstärken bleibt die maximale resultierende Feldstärke zu jedem einzelnen Zeitpunkt darin $\hat{h}(t) = \sqrt{\hat{h}_1^{\,2}(t) + \hat{h}_2^{\,2}(t)} < H_{\text{sat}}$.
3. Das magnetische Material im Raumgebiet der Interaktion hat einen linearen Bereich und durch passende räumliche Inhomogenitäten der beiden Feldstärken bleibt die maximale resultierende Felstärke darin $H(\boldsymbol{x}) = \sqrt{H_1^2(\boldsymbol{x}) + H_2^2(\boldsymbol{x})} < H_{\text{sat}}$.
4. Eine Kombination der beiden vorangegangenen Bedingungen ist stets möglich: $h(\boldsymbol{x}, t) = \sqrt{h_1^2(\boldsymbol{x}, t) + h_2^2(\boldsymbol{x}, t)} < H_{\text{sat}}$
5. Eine der beiden Feldstärken ist genau null $H_1 = 0 \vee H_2 = 0$. An diesem Betriebspunkt lässt sich auch ohne genaue Kenntnisse über die Interaktionsmechanismen feststellen, dass die Mechanismen eine Nullstelle aufweisen und damit äquivalent zu keiner Interaktion seind.
6. Es tritt keine orthogonale Magnetisierung auf.



(a) orthogonale Magnetisierung **(b)** getrennte Netzwerke

Abb. 1.14: Orthogonale Magnetisierung und keine Interaktion. Äquivalente Modellierung als zwei getrennte Netzwerke.

Die mathematische Beschreibung der Zusammenhänge zwischen den beiden Flüssen und den beiden magnetischen Spannungen zeichnet sich dadurch aus, dass in der Reluktanzmatrix lediglich die Diagonalelemente besetzt sind:

$$\begin{pmatrix} V_1 \\ V_2 \\ V_3 \end{pmatrix} = \begin{pmatrix} R_{\mathrm{m},1} & 0 & 0 \\ 0 & R_{\mathrm{m},2} & 0 \\ 0 & 0 & R_{\mathrm{m},3} \end{pmatrix} \begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \Phi_3 \end{pmatrix} \tag{1.76}$$

Damit ist die Beschreibung identisch zu getrennten magnetischen Netzwerken, wie in *Abb. 1.14* beispielhaft mit zwei orthogonalen Feldern präsentiert. In [11, S. 41] wird diese Theorie erwähnt, allerdings wird nicht weiter darauf eingegangen.

1.4.2 Allgemeine Interaktion

In dieser Arbeit soll explizit die Interaktion orthogonaler magnetischer Felder untersucht werden, wenn keine der o. g. Vereinfachungen gelten (siehe Abschnitt 1.4.1, S. 42). Das im folgenden vorgestellte Modell ist [31] entnommen, insbesondere den Kapiteln 2 und 3. Die Interaktion lässt sich demnach allgemein in einer charakteristischen Funktion ausdrücken, die nicht nur eine skalare Felstärke mit einer skalaren Flussdichte verknüpft, sondern mehrere orthogonale Komponenten enthält:

$$\begin{pmatrix} \Phi_1 \\ \Phi_2 \\ \Phi_3 \end{pmatrix} = \Phi \begin{pmatrix} V_1 \\ V_2 \\ V_3 \end{pmatrix} \tag{1.77}$$

Der Einfachheit halber wird im Folgenden exemplarisch nur eine der drei Funktionen betrachtet und lediglich die Abhängigkeit von zwei orthogonalen Feldern:

$$\Phi_1 = \Phi_1(V_1, V_2) \qquad \boldsymbol{\Phi} = \begin{pmatrix} \Phi_1 \\ \Phi_2 \end{pmatrix} \qquad \Phi = \sqrt{\Phi_1^2 + \Phi_2^2} \tag{1.78}$$

Dies erleichtert den Aufbau von Versuchen und verbessert die Übersichtlichkeit erheblich. Die Ergebnisse sollten übertragbar sein. Zur Herleitung einer charakteristischen Funktion aus den Materialeigenschaften kann folgendes Vorgehen genutzt werden:

1. Identifikation des betragsmäßigen Zusammenhangs $\Phi(V_1, V_2)$
2. Bestimmung der Orientierung $\boldsymbol{e}_\Phi = \frac{\boldsymbol{\Phi}}{\Phi}$
3. Zerlegung in orthogonale Komponenten und Darstellung als Netzwerkentitäten

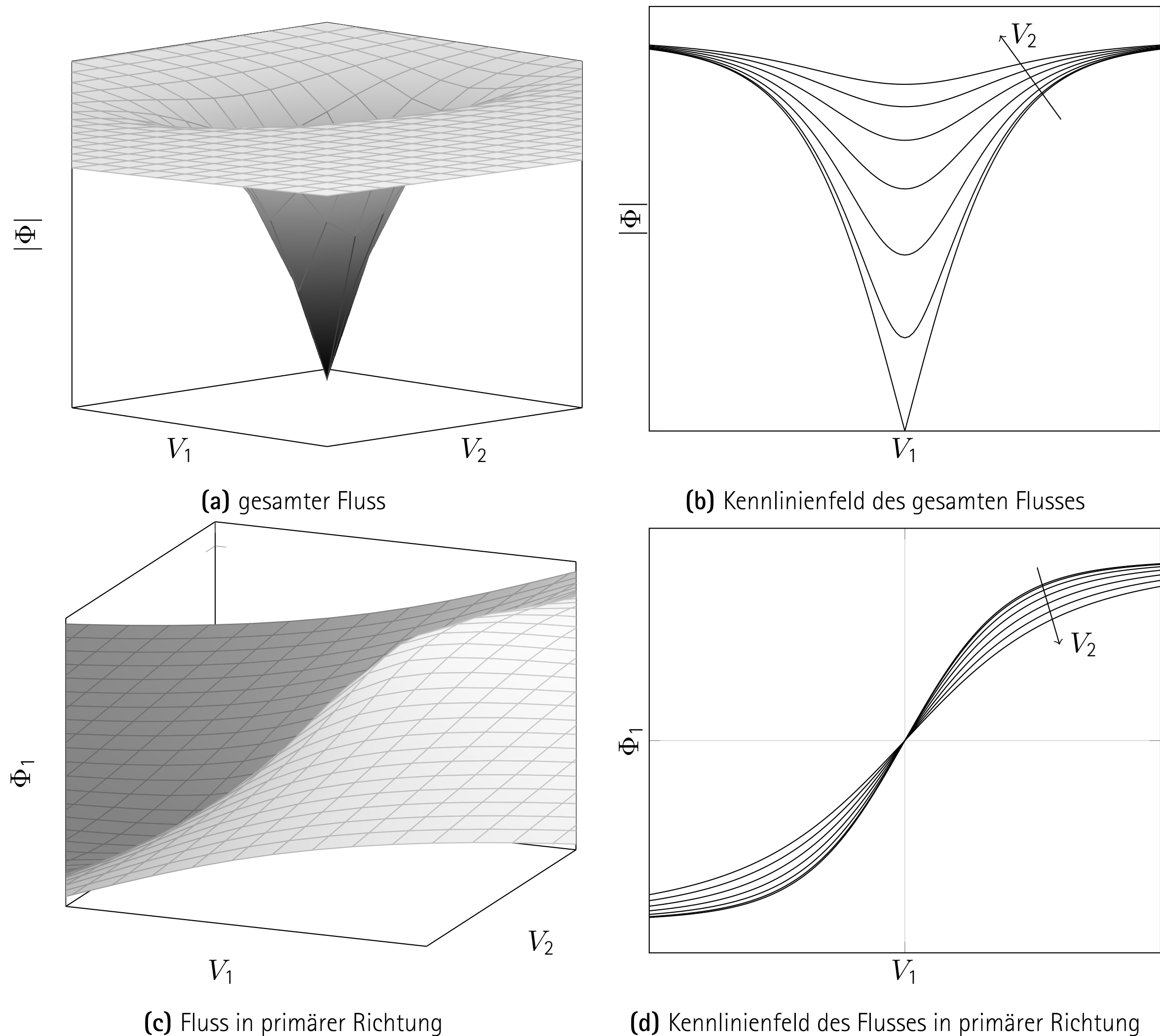
Ein qualitatives Beispiel für den ersten Schritt ist in *Abb. 1.15* (oben) gezeigt, eines für den zweiten und dritten Schritt in *Abb. 1.15* (unten). Obwohl für die Berechnung des Beispiels eine konkrete Interaktion angenommen ist, dient es nur der qualitativen Veranschaulichung! Das Finden einer geeigneten quantitativen Beschreibung der Interaktion ist Kernbestandteil der Aufgabenstellung dieser Arbeit.

1.4.3 Modell von Tez

Für den Fall, dass einige Vereinfachungen getroffen werden können, wird in [31] ein Modell auf Basis einfacher Vektoranalysis hergeleitet. Das im folgenden vorgestellte Modell ist der selben Quelle entnommen, insbesondere den Kapiteln 2 und 3. Dabei wird die grundlegende Idee verfolgt, dass zwei gegebene Feldstärken sich vektoriell addieren lassen. Der resultierende Betrag und die resultierende Orientierung dienen dann der Berechnung des Betrags und der Orientierung der sich ergebenden Flussdichte. Dabei ist der Betrag irrelevant für die Berechnung der Orientierung und die Orientierung irrelevant für die Berechnung des Betrags.

Obwohl in der Quelle nur wenige der notwendigen Bedingungen für die Gültigkeit des Modells genannt werden, lassen sich folgende identifizieren:

1. Das magnetische Material ist isotrop, sodass eine skalare Magnetisierungskurve $B(H)$ existiert.
2. Die Magnetisierungskurve ist nichtlinear.
3. Die Magnetisierungskurve steigt streng monoton, sodass die Lösung des Netzwerkes eindeutig ist und die Inverse der Magnetisierungskurve $H(B)$ eine Funktion ist.
4. Die Magnetisierungskurve geht durch den Koordinatenursprung.
5. Die Magnetisierungskurve ändert sich nicht unter Einflüssen, wie bspw. der Frequenz der Feldgrößen.
6. Das Material ist verlustfrei.
7. Das magnetische Material ist homogen.
8. Das Raumgebiet der Interaktion ist ordentlich definiert.
9. Innerhalb des Raumgebiets der Interaktion sind die Flüsse homogen.
10. Das Raumgebiet der Interaktion ändert sich nicht unter Einflüssen, wie bspw. der Frequenz der Stärke der Feldgrößen.



(a) gesamter Fluss

(b) Kennlinienfeld des gesamten Flusses

(c) Fluss in primärer Richtung

(d) Kennlinienfeld des Flusses in primärer Richtung

Abb. 1.15: Beispiel für eine räumlich zweidimensionale Reluktanz. Beispielhaft charakteristische Funktion $\Phi(V_1, V_2) = k \cdot V \cdot \left(1 + 100 \tanh\left(\frac{V}{[V]}\right)\right); V := \sqrt{V_1^2 + V_2^2}$ für isotropes, verlustfreies Material mit gleicher Geometrie in primärer und sekundärer Richtung. Deutlich erkennbare Zunahme des gesamten Flusses mit beiden magnetischen Spannungen V_1 und V_2. Zunahme des primären Flusses mit primärer Feldstärker, aber Abnahme mit sekundärer Feldstärke.

11. Die Flüsse stehen exakt orthogonal innerhalb des Raumgebiets der Interaktion.
12. Die resultierende Feldform ist unabhängig von der Amplitude.
13. Die Flüsse interagieren nicht anderswo oder diese Interaktion ist separat und kompatibel modelliert.
14. Es gibt kein Streufeld.

Sollten alle diese notwendigen Bedingungen erfüllt sein, folgt daraus quasi direkt der vektorielle Zusammenhang zwischen H- und B-Feld. Dabei bestimmt sich der Betrag des B-Feldes durch den Betrag des H-Feldes

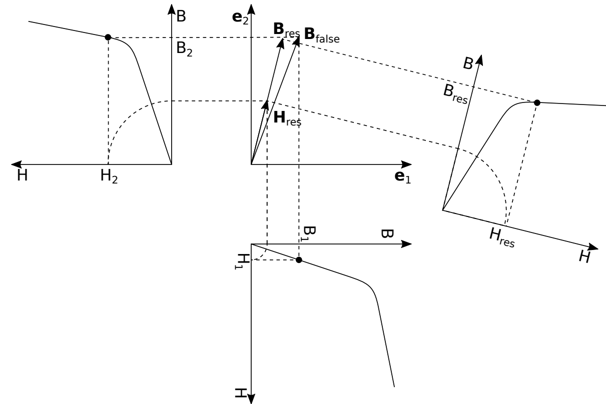


Abb. 1.16: Graphische Konstruktion der resultierenden Flussdichte bei zwei gegebenen magnetischen Felstärken. Zwei gegebene orthogonale magnetische Feldstärken H_1 und H_2 mit größerer und falsch orientierter resultierender Flussdichte unter Annahme von keiner Flussinteraktion $\boldsymbol{B}_{\text{false}} = B(H_1)\boldsymbol{e}_1 + B(H_2)\boldsymbol{e}_2$ als die stattdessen resultierende Flussdichte $\boldsymbol{B}_{\text{res}}$ gemäß *Gl. 1.79*. [31]

und die Magnetisierungskurve $B(H)$; die Richtung beider Felder ist identisch:

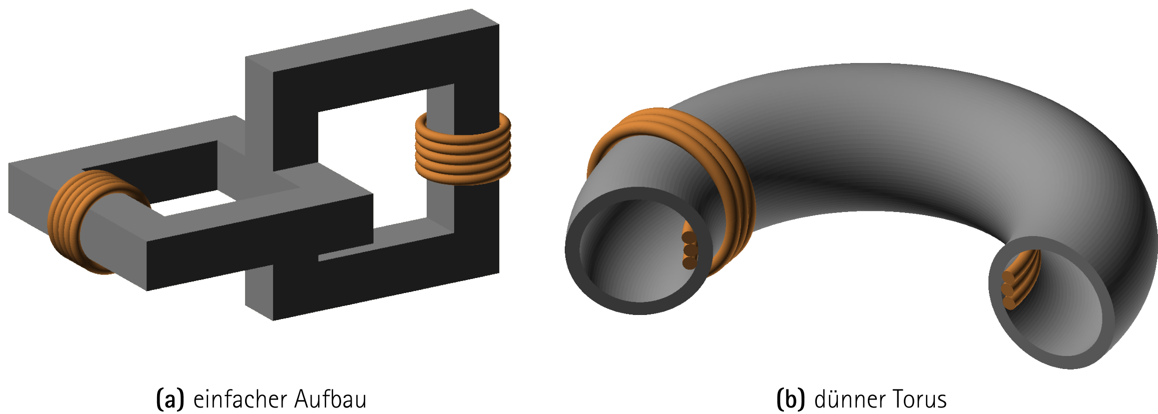
$$\boldsymbol{B}(H_1, H_2) = \boldsymbol{B}(H_1\boldsymbol{e}_1 + H_2\boldsymbol{e}_2) = B(\sqrt{H_1^2 + H_2^2}) \cdot \frac{\begin{pmatrix} H_1 \\ H_2 \end{pmatrix}}{\sqrt{H_1^2 + H_2^2}} \tag{1.79}$$

Der Zusammenhang gilt für beliebig viel-dimensionale Räume entsprechend,

$$\boldsymbol{B}(\boldsymbol{H}) = B(H) \cdot \frac{\boldsymbol{H}}{H} \qquad \boldsymbol{H} = \sum_j H_{\text{j}}\boldsymbol{e}_{\text{j}} \tag{1.80}$$

wobei die graphische Konstruktion für zwei Dimension erheblich einfacher ist. Die umgekehrte Berechnung des H-Feldvektors aus zwei gegebenen B-Feldern funktioniert entsprechend, wobei die Umkehrfunktion der Magnetisierungskurve benötigt wird. In *Abb. 1.16* ist diese Konstruktion des B-Feldvektors aus zwei gegebenen H-Feldvektoren festgehalten. Man bemerke, dass der Zusammenhang keine Interaktion zwischen den Feldern aufweist, wenn alle bis auf eine Feldstärke gleich null sind. Damit unterscheidet sich die orthogonale Interaktion erheblich von der parallelen und Anwendungen mit orthogonaler Magnetisierung erfordern ggf. eine „Startaktion" (siehe Abschnitt 1.5, S. 49).

Die Bedingungen 1 – 7 müssen von dem Material des magnetischen Kerns erfüllt werden. Die weiteren Bedingungen werden hauptsächlich durch eine geeignete Wahl der Kerngeometrie erfüllt, welche auch zur Herleitung der Netzwerkgrößen aus den Felgrößen dient (siehe Abschnitt 1.1, S. 6). Im Folgenden werden lediglich Versuche unternommen, die Voraussetzungen für genau zwei orthogonale Felder zu erfüllen und die Netzwerkgrößen dafür zu beschreiben. Die Bedingungen 10 – 13 können ausschließlich mit einem torusförmigen magnetischen Kern erfüllt werden (siehe *Abb. 1.17b*), wobei eine analytische Lösung laut [31] hier bereits unzweckmäßig kompliziert würde. Die Voraussetzungen 8 – 9 können einfach mit einem sehr dünnwandigen Torus erreicht werden,



(a) einfacher Aufbau

(b) dünner Torus

Abb. 1.17: Beispiele für magnetische Kerngeometrien mit orthogonaler Magnetisierung [31]

dessen Feldbilder sich einfacher analytisch beschreiben lassen und dessen effektive Flächen und effektiven Längen als A_1, l_1, A_2 und l_2 definiert seien. In dem Fall folgt aus dem Zusammenhang zwischen den Feldern in *Gl. 1.79* die charakteristische Funktion: [31]

$$v_{\mathrm{k}}(\phi_1, \phi_2) = \frac{l_{\mathrm{k}}}{A_{\mathrm{k}}} \phi_{\mathrm{k}} \cdot \frac{h\left(\sqrt{\frac{\phi_1^2}{A_1^2} + \frac{\phi_2^2}{A_2^2}}\right)}{\sqrt{\frac{\phi_1^2}{A_1^2} + \frac{\phi_2^2}{A_2^2}}} \qquad \forall k \in \{1; 2\} \tag{1.81}$$

Schließlich ist Annahme 14 notwendig, um sicherzustellen, dass die Flüsse auch tatsächlich durch das Raumgebiet der Interaktion fließen, was bei einem ordentlichen Kerndesign der Fall sein sollte. Bei Arbeitspunkten weit im Gebiet der Sättigung, welche für das Prinzip der orthogonalen Flussinteraktion notwendig sind (siehe Abschnitt 1.4.2, S. 43), kann sich dieser Verhalt allerdings ändern.

1.4.4 Weitere Modelle

Wegen der zahlreichen Annahmen, welche für die Gültigkeit des Modells von Tez notwendig sind (s. o.), sollten weitere Modellierungen entwickelt werden. Wegen der guten Ergebnisse mit dem Modell von Tez und weil für die Entwicklung weiterer Modelle Kernmaterialien benötigt würden, welche im Rahmen der Arbeit nicht zur Verfügung stehen (siehe Kapitel 6, S. 139), wird die Aufgabe an nachfolgende Arbeiten weitergegeben. Sie lässt sich in zwei Teile analysieren:

1. Mikroskopisches Modell, welches auch anisotrope und verlustbehaftete Materialien berücksichtigt.
2. Makroskopisches Modell, welches der Einbindung des mikroskopischen Modells in ein magnetisches Netzwerk dient.

Letztere Aufgabe wird später insofern gelöst, dass ein Werkzeug zur Feldsimulation entwickelt wird, welches

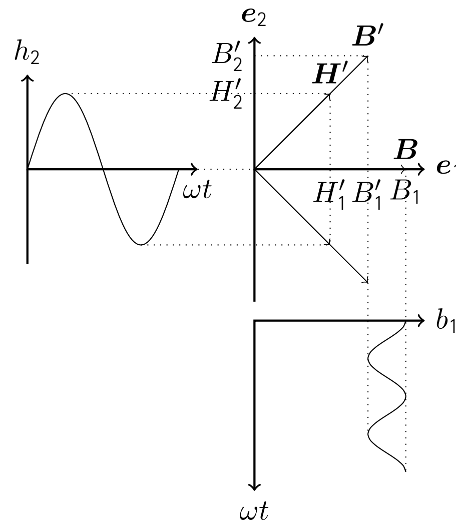


Abb. 1.18: Allgemeines Beispiel für einfache orthogonale Flussinteraktion. Eine zeitvariante sekundäre Feldstärke h_2 verursacht einen Zeitverlauf in der primären Flussdichte b_1, obwohl die primäre Feldstärke h_1 konstant ist. $\boldsymbol{B}$ an den Stellen mit $h_2 = 0$ und $\boldsymbol{B}'$ an der Stelle des maximalen h_2. Die Kurvenform der resultierenden primären Flussdichte $b_1(\omega t)$ weist gerade harmonische Oberschwingungen auf und hängt von Kurvenform und Amplitude von der sekundären Felstärke $h_2(\omega t)$ und der Magnetisierungskurve $b(h)$ ab. [31]

die Integration des mikroskopischen Modells in ein Netzwerk berechnet (siehe Kapitel 5, S. 119).

1.5 Anwendung

Nachdem die Eigenschaften von magnetischen Feldern und Netzwerken erörtert sind, stellt sich im Ingenieursbereich die Frage nach einer sinnvollen Anwendung. Auf konventionelle Induktivitäten und Transformatoren wird an dieser Stelle nicht eingegangen; ihre Funktionsweise sollte bekannt sein. Stattdessen werden technische Anwendungen vorgestellt, welche sich mit den zuvor erläuterten Beschreibungen und Methoden charakterisieren lassen. Weil dieser Abschnitt einen Ausblick liefert und nicht zum Erreichen der Aufgabenstellung nützt, werden die Anwendungen nur kurz angerissen und sofern möglich wird weiterführende Literatur genannt.

Vormagnetisierte Speicherinduktivität

Die bereits diskutierten vormagnetisierten Induktivitäten (siehe Abschnitt 1.1.5, S. 24) lassen sich in leistungselektronischen Anwendungen nutzen, um mit einer bestimmten Menge an magnetischem Kernmaterial und Wickeldraht einen größeren um den Mittelwert des elektrischen Stroms symmetrischen Bereich konstanter Induktivität zu erreichen. Das Prinzip bietet natürlich nur einen Vorteil, wenn der Mittelwert des Stroms ungleich null ist, wie bspw. bei Gleichrichtern oder Stromzwischenkreisen. Das Prinzip ist ausführlich in [14] untersucht.

Übersättigte Induktivität

Eine interessante Form der Amplitudenfilterung lässt sich mit einer sog. übersättigten Induktivität erreichen. Es handelt sich um eine vormagnetisierte Induktivität (siehe Abschnitt 1.1.5, S. 24), bei welcher die LI-Kennlinie so weit verschoben ist, dass der gesamte Bereich hoher Induktivität bei Strömen ungleich null auftritt. Für kleine Stromamplituden wirkt das Bauteil damit wie eine geringe Induktivität, für große Amplituden stellt es (bei nur einem Vorzeichen) eine große Induktivität dar. Das Prinzip ist in [5] beschrieben.

Transduktivität

Bei der allgemeinen Modellierung der orthogonalen Magnetisierung (siehe Abschnitt 1.4.2, S. 43) stellt sich heraus, dass eine Reluktanz vergrößert werden kann, indem ein zum primären Feld orthogonal stehendes sekundäres Feld das magnetische Material in die Sättigung treibt (siehe Abb. 1.15). Dieses Prinzip kann genutzt werden, um die primäre Reluktanz und damit eine Induktivität gezielt zu beeinflussen. Ein solches Bauteil wird bspw. als stellbare Induktivität oder Transduktivität bezeichnet. Eine breite Übersicht über den Forschungsstand zu diesem Bauteil und seiner Anwendung ist in [24] gegeben.

Paraformator

Wenn bei einer Transduktivität (s. o.) sowohl das primäre Feld als auch das dazu orthogonal stehende sekundäre Feld durch mit elektrischen Strömen durchflossenen Wicklungen verursacht sind, lässt sich die Anordnung als Paraformator nutzen. Sofern beide Ströme ungleich null sind, führt die Änderung des Stroms durch eine Wicklung zur Induktion einer Spannung in der anderen Wicklung. Das Prinzip weist einige Vorteile gegenüber Transformatoren auf, bspw. eine gute Filterung im Amplitudenbereich. Der Aufbau ist sehr detailliert in [31] untersucht.

Transformator mit zusätzlicher orthogonaler Magnetisierung

In einem gewöhnlichen Transformator stehen die beiden magnetischen Flüsse parallel zueinander. Wenn eine weitere Wicklung aufgebracht wird, deren Feld orthogonal zu den beiden Feldern der Primär- und der Sekundärwicklung steht und die in Reihe mit der Primärwicklung geschaltet ist, lassen sich einige Vorteile des Paraformators nutzen. Bspw. wird der magnetische Kern bei hohen Amplituden auf der Primärseite gesättigt und die magnetische Kopplung nimmt ab, weil nun ein höherer Anteil des primären magnetischen Flusses durch die Streureluktanz fließt, sodass die Sekundärseite teilweise gegen Signalspitzen geschützt ist.

Neue Anwendungen

Das Ziel der vorliegenden Arbeit besteht darin, ein Modell zu entwickeln und zu validieren, welches die orthogonale Magnetisierung beschreibt. Es kann davon ausgegangen werden, dass sobald dieses Ziel erreicht ist, neue Anwendungsmöglichkeiten ersichtlich werden.

2 Messaufbau

Es geht doch nichts über einen guten Workaround.

– Tobias Riedel

Für die später beschriebenen Versuche (siehe Kapitel 6, S. 139) werden nicht nur Prüfkörper benötigt, sondern auch ein Aufbau, welcher magnetische Signale generieren und messen kann. Wegen der hohen Diversität der verschiedenen Prüfline wird die Wandlung zwischen elektrischen und magnetischen Signalen mithilfe von Drahtwicklungen dem jeweiligen Spezimen überlassen. Daraus folgt, dass der in dieser Arbeit entwickelte Messaufbau folgenden Funktionsumfang bereitstellen muss:

1. Funktionsgenerator
   - mindestens zwei unabhängige Kanäle
   - frei programmierbare Signalformen
   - geringe Ausgangsimpedanz, um auch Lasten im unteren Ω-Bereich zu treiben
   - keine Dämpfung bis ca. 1 kHz
   - gute Linearität bis ca. 1 A
   - einfache Anbindung an Elektronische Datenverarbeitung (EDV), bspw. Personal-Computer (PC)
2. Sensoren
   - mindestens vier unabhängige Kanäle, davon zwei zur Messung von Spannungen und zwei zur Messung von Strömen
   - Schutz vor Bedienungsfehlern
   - hohe Eingangsimpedanz, um die Messungen nicht zu verfälschen
   - Abtastrate von mindestens 10 kS/s
   - einstellbarer Messbereich für verschiedene Lasten
   - gute Linearität über den gesamten Messbereich
   - einfache Anbindung an EDV, bspw. PC
3. Software

- lauffähig auf „STEMlab 125-10 RedPitaya“[1], vorgeschrieben vom betreuenden Institut zu Evaluationszwecken
- einfach verständlicher und einfach erweiterbarer Programmcode für nachfolgende Arbeiten durch andere Mitarbeitende
- einfache Anbindung an weiterverarbeitende EDV, bspw. Auswertungssoftware (siehe Kapitel 5, S. 119).

Im Folgenden werden zunächst das Linux-basierte Entwicklungsboard „RedPitaya“ und daran vorgenommene Anpassungen vorgestellt, sodass das Herzstück des Messaufbaus betriebsbereit ist. Anschließend werden einige Schaltungen entwickelt, um einen Funktionsgenerator und Sensoren zu erhalten, die den Ansprüchen genügen. Schließlich ist eine Programmierung auf dem „RedPitaya“ erforderlich, um die Schaltungen korrekt anzusprechen und die Daten weiterzuverarbeiten.

Mit der voranschreitenden Programmierung stellt sich heraus, dass die vom Hersteller gemachten Spezifikationen des „RedPitayas“ nicht stimmen. Der Messaufbau wird deshalb stark verändert (siehe Abschnitt 2.3.2, S. 94) und das Ziel eines automatischen, intelligenten Messaufbaus wird damit verfehlt. Allerdings kann so überhaupt ein Versuch stattfinden (siehe Kapitel 6, S. 139). Der finale Aufbau wird am Ende des Kapitels vorgestellt.

[1] Das Herstellerunternehmen verwendet verschiedene Namen für das Gerät und verschiedene Schreibweisen für den Unternehmensnahmen und den Gerätenamen. Der Einfachheit halber wird im Folgenden die Bezeichnung „RedPitaya“ verwendet.

2.1 Signalgenerator und Messsystem

Mit dem „RedPitaya“ stellt das betreuende Institut einen Einplatinencomputer zur Verfügung, der hochfrequente Signale auf mehreren Kanälen generieren und messen und der während der Coronapandemie auch im Home Lab verwendet werden kann. Er bildet das Herzstück des Messaufbaus und schafft eine Verbindung zwischen Hardware und Software. Im Folgenden werden seine Eigenschaften zusammengefasst, die getroffenen Einstellungen dokumentiert und die Werkzeuge zur Programmierung definiert.

2.1.1 Eigenschaften

Die **Hardware** wird über einen 5 V-Universelles Serielles Bussystem, engl. Universal Serial Bus (USB)-Anschluss mit Spannung versorgt, verfügt über einen „ARM Cortex A9 Dual Core“-Prozessor, 256 MB Hauptspeicher, engl. Random Access Memory (RAM), einen 1 Gbit-Ethernetanschluss, einen USB-2.0-Port und eine im Feld prgrammierbare Logikgatteranordnung, engl. Field Programmable Gate Array (FPGA) „Xilinx Zync 7010“. Letztere bietet zwei analge Eingänge und zwei analoge Ausgänge mit einer Auflösung von je 10 bit und einer Tastrate von je 125 MS/s und außerdem vier analoge Eingänge und vier analoge Ausgänge mit einer Auflösung von je 12 bit und einer Tastrate von je 100 kS/s. Die vier langsamen analogen Ausgänge umfassen den Spannungsbereich 0 V...1,8 V, die zwei schnellen −1 V...1 V. Die vier langsamen analogen Eingänge messen Spannungen im Bereich 0 V...3,5 V, die zwei schnellen −20 V...20 V. Es sind 16 digitale Allzweckeingabe/-ausgabeports, engl. General Purpose Inputs/Outputs (GPIO) und verschiedene digitale Bussysteme verfügbar. [9]

Die **Software** ist ein stark angepasstes und beschnittenes Linux-System (Ubuntu 16.04). Es bietet eine Vielfalt von Kommunikationsschnittstellen zu einem Computer, bspw. über Web, Secure Shell (SSH), Dateitransferprotokoll über SSH, engl. SSH File Transfer Protocol (SFTP) oder Standardbefehle für Programmierbare Instrumente, engl. Standard Commands for Programmable Instruments (SCPI) und ist durch zusätzliche Software erweiterbar, bspw. um das Netzwerkprotokoll für den Zugriff auf entfernte Bildschirminhalte, engl. Remote Desktop Protocol (RDP). Auf dem FPGA läuft ein vorkompiliertes Image zur Bereitstellung einer Programmierschnittstelle, engl. Application Programming Interface (API). Das Laden eigener Images ist möglich.

2.1.2 Einstellungen

Bevor das Gerät verwendet werden kann, müssen einige Einstellungen vorgenommen, Bugs behoben und Home-Verzeichnisse geschützt werden. Dazu dient die Kommandozeile über SSH über Lokales Netzwerk, engl. Local Area Network (LAN). Die werkseitigen Zugangsdaten lauten:

```
user: root
pass: root
```

Das Passwort zur **Authentifizierung** des Benutzers „root“ wird von der Voreinstellung „root“ zu „vfquazhGNz2QxqBzKoItpXt4DThPK“ geändert. Die Authentifizierung mit einem Passwort anstatt eines Public Keys über

SSH wird nicht deaktiviert, um nachfolgenden Nutzenden des Systems einen einfachen Zugang zu ermöglichen. Die Zugangsdaten sind trotz des Sicherheitsrisikos auf dem Gehäuse vermerkt. Obwohl dies anscheinend nicht vom Hersteller vorgesehen ist, wird gemäß guter Praxis ein Account „ilka" hinzugefügt, um eine Abgrenzung verschiedener User zu erzielen. Die Authentifizierung ist mit einem Passwort oder einem Public Key möglich. Letztere bietet folgende Vorteile:

- bessere Sicherheit, weil kein (geheimes) Passwort über das Netzwerk übertragen werden muss, sondern nur der öffentliche Schlüssel.
- einfachere Nutzbarkeit, weil SSH-Clients nun keine Authentifizierung mehr bei jedem einzelnen Login fordern; insbesondere bei Automatisierung nützlich

Außerdem wird die Nachfrage nach dem Passwort durch *sudo* für den Account „ilka" entfernt. Zu der Konfigurationsdatei, die mit dem Befehl

```
sudo visudo
```

bearbeitet werden kann, wird dazu am Ende die Zeile hinzugefügt:

```
ilka ALL=(ALL) NOPASSWD: ALL
```

Bevor das Gerät programmiert werden kann, müssen einige **Bugs** behoben werden:

- Der Datei */etc/environment* wird die Zeile hinzugefügt:

  ```
  LD_LIBRARY_PATH=/opt/redpitaya/lib
  ```

  Ohne diese finden Programme, welche auf die API des „RedPitayas" zugreifen, nicht die nötigen Laufzeitbibliotheken, engl. Runtime Libraries (RTL).

- Die Datei */etc/hosts* wird erstellt und erhält die beiden Zeilen:

  ```
  127.0.0.1       localhost
  127.0.0.1       rp-f046ab
  ```

  Sonst besteht keine Hostauflösung für den in der Datei */etc/hostname* festgelegten Hostnamen „rp-f046ab", was bei diversen Programmen in Fehlermeldungen resultiert.

- Der Datei */etc/rc.local* wird die Zeile hinzugefügt:

  ```
  sudo cat /opt/redpitaya/fpga/fpga_0.94.bit > /dev/xdevcfg
  ```

  Andernfalls ist nicht das richtige FPGA-Image geladen und Programme, die auf die API des „RedPitayas" zugreifen, werfen den Fehler „Bus Error".

- Updates müssen installiert werden. Die Update-Funktion des Webinterfaces versagt bei dieser Aufgabe auch dann, wenn sie Erfolg mitteilt. Stattdessen gelingt die Installation von Updates mit den beiden Befehlen:

  ```
  sudo apt update
  sudo apt upgrade
  ```

- Das Paket *bash-completion* muss mit dem Befehl

  ```
  sudo apt install bash—completion
  ```

  nachgeladen werden, um mit der Kommandozeile effektiv arbeiten zu können.
- Kernelmodule können nicht dynamisch geladen werden. Die Behebung dieses Bugs ist zu umfangreich für den Rahmen der vorliegenden Arbeit.
- Für eine bequeme Arbeit mit der Kommandozeile werden außerdem die Pakete *time* und *tree* installiert.

Die „Home"-Verzeichnisse der Accounts werden nicht **verschlüsselt**. Obwohl sich die Pakete *ecryptfs-utils* und *cryptsetup* installieren lassen, hat der Hersteller einige, teils sehr invasive Eingriffe in das funktionierende Betriebssystem vorgenommen. Deshalb können keine Kernel-Module nachgeladen werden. Bei der Weitergabe des Geräts ist deshalb darauf zu achten, alle erstellten Accounts wieder zu entfernen. Die auf das Gehäuse aufgeklebten Zugangsdaten für den Account „root" sind somit ebenfalls ein noch größeres Sicherheitsrisiko.

2.1.3 Programmierung

Die Programmierung wird in der Programmiersprache C umgesetzt. Die Kompilierung und die Ausführung der Programme geschieht auf dem Gerät selbst. Codedateien werden entweder über das SFTP übertragen oder direkt mit einem Kommandozeileneditor über SSH bearbeitet. Für die Kompilierung wird der Befehl

```
make
```

mit dem Makefile des Herstellers genutzt: [12]

```
CFLAGS  = —g —std=gnu99 —Wall —Werror
CFLAGS += —I/opt/redpitaya/include
LDFLAGS = —L/opt/redpitaya/lib
LDLIBS = —lm —lpthread —lrp

SRCS=$(wildcard *.c)
OBJS=$(SRCS:.c=)

all: $(OBJS)

%.o: %.c
$(CC) —c $(CFLAGS) $< —o $@

clean:
$(RM) *.o
$(RM) $(OBJS)
```

Es definiert die Übersetzung jeder Datei mit C-Code in ein ausführbares Programm. Dazu teilt es dem Compiler und dem Linker die nötigen Bibliotheken mit. Das Makefile kann angepasst werden, um spezifische Abhängigkeiten oder Übersetzungseinheiten ohne Einstiegspunkt zu implementieren. Das vollständig erweiterte Makefile wird später vorgestellt (siehe Abschnitt 2.3.3, S. 107).

Für den Zugriff auf die signalgenerierende und messende Hardware des Geräts wird die Bibliothek des Herstellers in den C-Code eingebunden:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "redpitaya/rp.h"  // includes the Redpitaya API function declarations

int main (int argc, char **argv) {
   // Initialization of API
   if (rp_Init() != RP_OK) {
      fprintf(stderr, "Red Pitaya API init failed!\n");
      return EXIT_FAILURE;
   }

   // —— user code here ——

   // Releasing resources
   rp_Release();

   return EXIT_SUCCESS;
}
```

Die im Rahmen dieser Arbeit entwickelte Software wird in einem eigenen Abschnitt behandelt (siehe Abschnitt 2.3, S. 87).

Backups werden automatisiert über das SFTP angelegt (siehe Code, S. 180)

2.2 Schaltungen

Der Messaufbau umfasst einige elektronische Schaltungen, die für die Erzeugung und die Messung von Leistungssignalen notwendig sind. Für den Entwicklungsansatz sind zwei verschiedene Philosophien möglich, die in *Tab. 2.1* gegenübergestellt sind. In einer vorangegangenen hilfswissenschaftlichen Tätigkeit sind schnell gute Ergebnisse mit der Lösung aus Standardkomponenten erzielt. [27] Die folgenden Gründe sind ausschlaggebend für die Wahl der spezifischen Lösung:

1. Der Messaufbau soll auch nach Abschluss der Arbeit von weiteren Mitarbeitenden wiederverwendet werden.
2. Das „Redpitaya" ist vom betreuenden Institut neu angeschafft und soll bzgl. seiner Tauglichkeit für genau solche Aufgaben bewertet werden.
3. In Zukunft soll eine Vielzahl von Messungen durchgeführt werden, weshalb ein hoher Grad an Automatisierung angestrebt wird. Die nahtlose Anbindung an eine Software zur Auswertung ist allgemein anzustreben.
4. Wegen der Pandemie während der Bearbeitungszeit sind die Aufgaben in der Privatwohnung der Autorin durchzuführen und der Zugang zum Labor ist streng untersagt. Das Messsystem muss entsprechend aufgebaut sein, um autark zu funktionieren.

Wie aus *Tab. 2.1* ersichtlich wird, umfasst die gewählte, spezifische Lösung einige selbst anzufertigende elektronische Schaltungen. Obwohl diese zum größten Teil sehr einfach aufgebaut sind und mitunter lediglich Standardschaltungen beinhalten, muss deren Implementierung charakterisiert werden. Dabei werden parasitäre Größen und unerwünschte Effekte identifiziert und teils werden die Schaltungen daraufhin überarbeitet.

Dieser Abschnitt ist in die Vorstellung der verschiedenen Schaltungstypen gegliedert, wobei unterschiedliche Implementierungen einzeln aufgelistet sind, insbesondere die Impedanzwandler mit unterschiedlichen Betriebsbereichen. Obwohl Notwendigkeit, Aufbau, Funktion, Charakterisierung und Zusammenspiel mit verbundenen Komponenten jeder einzelnen gelöteten Platine erst zusammen mit der jeweiligen Vorstellung abgedruckt ist, soll ein Überblick über den gesamten Aufbau dennoch im Vorhinein gegeben werden: Der gesamte Aufbau ist in *Abb. 2.2* fotographiert und schematisch in *Abb. 2.3* abgebildet. Die Schaltungen werden mithilfe von „Jumper-Kabeln" verbunden, welche im betrachteten Frequenzbereich nur geringe parasitäre Effekte aufweisen. Der Frequenzgang einer Verbindung mit einem „Jumper-Kabel" und dessen Unbedenklichkeit sind in *Abb. 2.1* dargestellt.

Es sei erwähnt, dass in den nachfolgenden Schaltskizzen die Anschlüsse für die Spannungsversorgung der Operationsverstärker (OPV) nicht eingezeichnet sind. Für die Messungen mit dem Gerät „Bode 100" von „Omciron Lab"[2] wird eine Versorgungsspannung von $\pm 5\,\mathrm{V}$ verwendet, um dem empfindlichen Messgerät keine versehentlichen Schäden beizufügen. In sämtlichen anderen Anwendungen wird eine Versorgungsspannung von

[2] https://www.omicron-lab.com/products/vector-network-analysis/bode-100/

Tab. 2.1: Vergleich zweier kontroverser Philosophien zum Messaufbau. Geringerer zeitlicher und technischer Aufwand für die Standardlösung, besserer Wiederverwendbarkeit der spezifischen Lösung.

	Standardlösung	spezifische Lösung
Geräte	Funktionsgenerator, Oszilloskop, Strommesszangen, Spannungsquellen	„Redpitaya", Spannungsquellen
Schaltungen	Impedanzwandler	Tiefpass, kleiner Impedanzwandler, Subtrahierer, Impedanzwandler, Messwandler, Addierer, Spannungsschutz
analoge Signale	hochauflösend	ausreichend auflösend
Kalibrierung	nachträglich bei der Auswertung	auf der Hardware (Potentiometer, Redpitaya)
Automatisierung und Anbindung	vollständig manuell, hoher Nachbearbeitungsaufwand, kaum portierbarer Aufbau	vollständig automatisierbar, direkte Anbindung an Software zur Auswertung, portierbarer Aufbau
(Fehler-)Charakterisierung	geringe Fehler, einfach charakterisierbar	fehleranfällig, einige Bodediagramme notwendig
Programmierung	keine	einige triviale Algorithmen
Entwicklungsaufwand	ein Tag	ein Monat

$\pm 14\,\mathrm{V}$ gewählt, sofern nichts anderes explizit beschrieben ist.

Alle impedanzwandelnden Schaltungen mit OPV zeigen im Bode-Diagramm eine statische Verstärkung von 2, obwohl mit Oszilloskopmessungen gezeigt werden kann, dass die statische Verstärkung zumindest für niederfrequente Signale etwa bei 1 liegt. Der Phänomen ist einfach zu erklären: Das Geräte „Bode100" verwendet einen Widerstand von $50\,\Omega$ in Serie zu seinem Ausgang und einen gleichen parallel zu seinem Eingang. Bei der Kalibrierung sind die Ausgangsklemmen niederimpedant an die Eingangsklemmen angeschlossen. Die Widerstände bilden einen Spannungsteiler und der das Gerät wird auf diesen kalibriert. Bei der Messung wird zwischen den Ausgang und den Eingang statt der niederimpedanten Verbindung ein Impedanzwandler angeschlossen. Ein idealer Impedanzwandler hat einen unendlich hohen Eingangswiderstand und einen Ausgangswiderstand von $0\,\Omega$. Es liegt kein Spannungsteiler vor, sondern der Sensor im Eingang misst die gleiche Spannung wie sie von der Quelle im Ausgang ausgegeben wird. Im Folgenden werden die originalen Bode-Diagramme abgedruckt und lediglich ein Hinweis auf diesen Absatz gegeben.

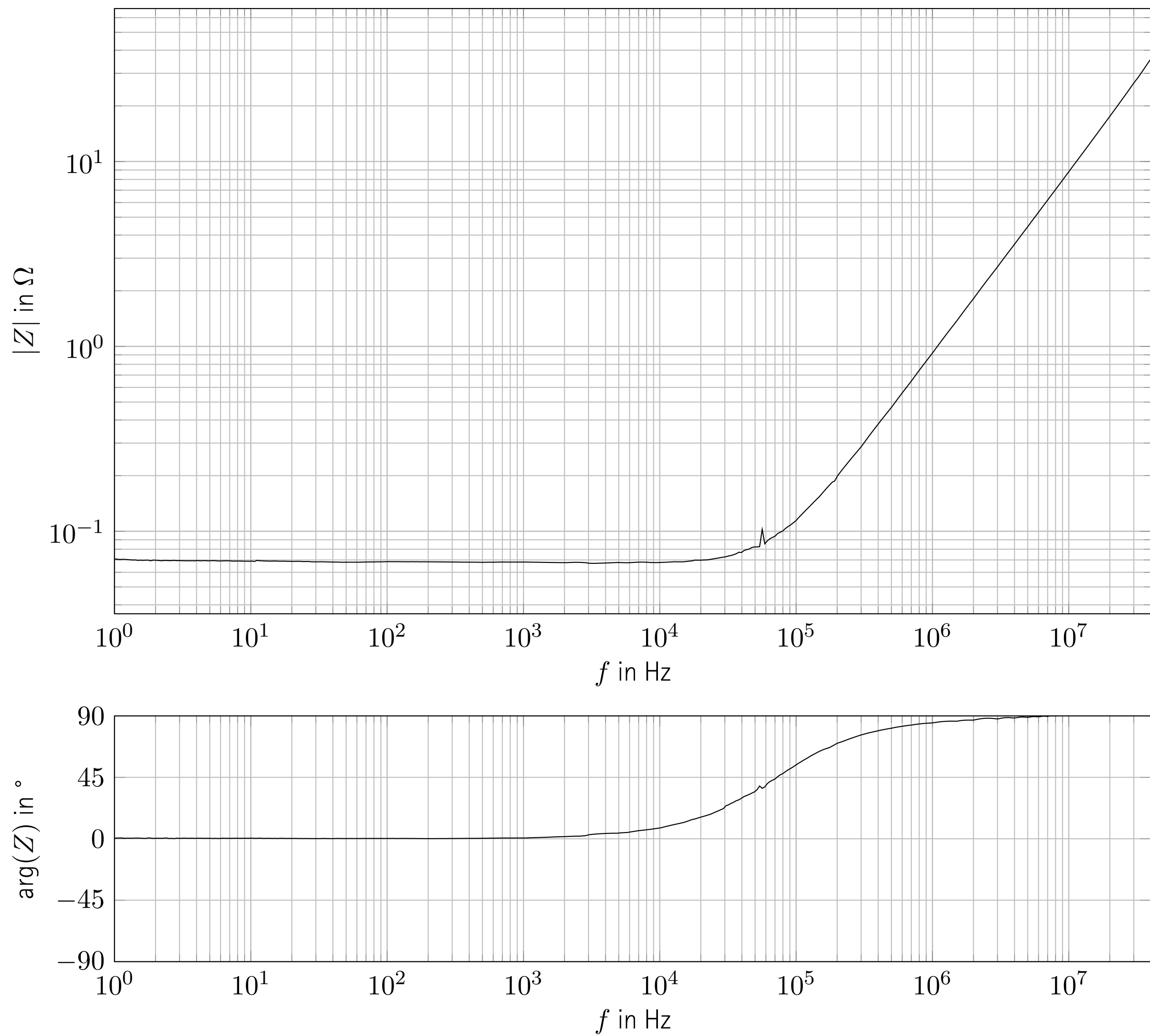


Abb. 2.1: Impedanz eines „Jumper-Kabels". 401 Messpunkte, aufgenommen mit Bode100 (Ein-Port-Messung, Empfängerbandbreite: 1 Hz). Geringe ohmsche Impedanz bis ca. 100 kHz, ausgeprägte parasitäre Induktivität bei höheren Frequenzen.

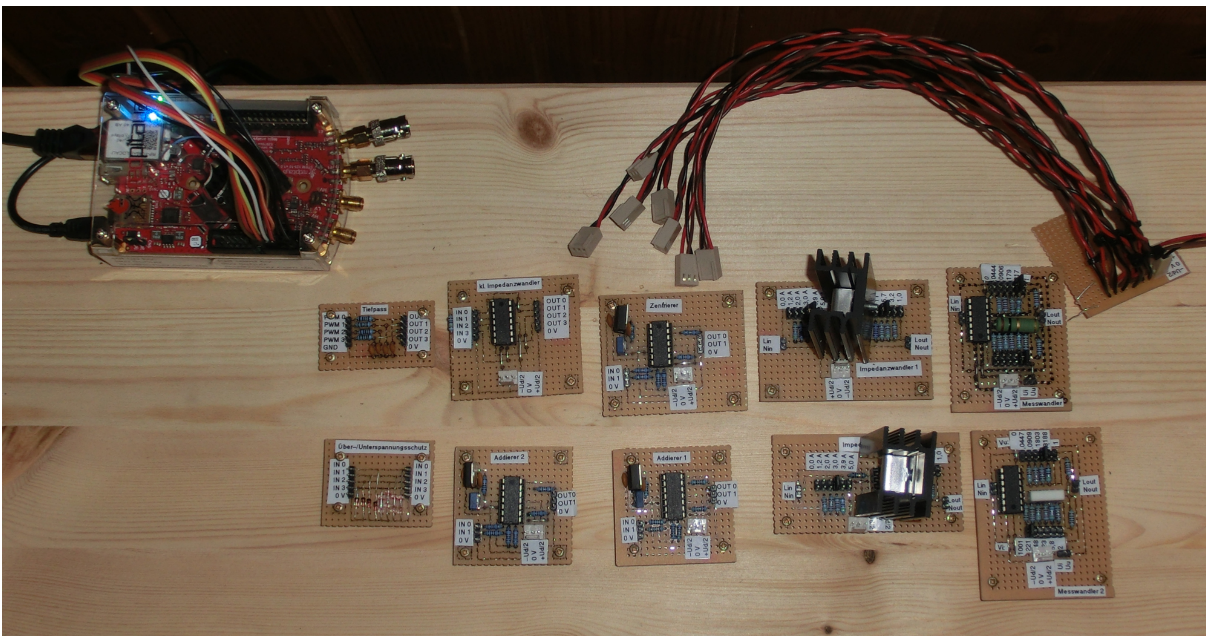


Abb. 2.2: Fotographie des gesamten Messaufbaus. „Jumperkabel" entfernt für bessere Übersicht. Alle selbstgelöteten Platinen sind festgeschraubt. „RedPitaya" (oben links), Spannungsversorgung (oben rechts). Platinen von oben links im Uhrzeigersinn: Tiefpass, kleiner Impedanzwandler, Subtrahierer, großer Impedanzwandler (Kanal 1), Messystem (Kanal 1), Messsystem (Kanal 2), großer Impedanzwandler (Kanal 2), Addierer (Spannungen), Addierer (Ströme), Über-/Unterspanungsschutz.

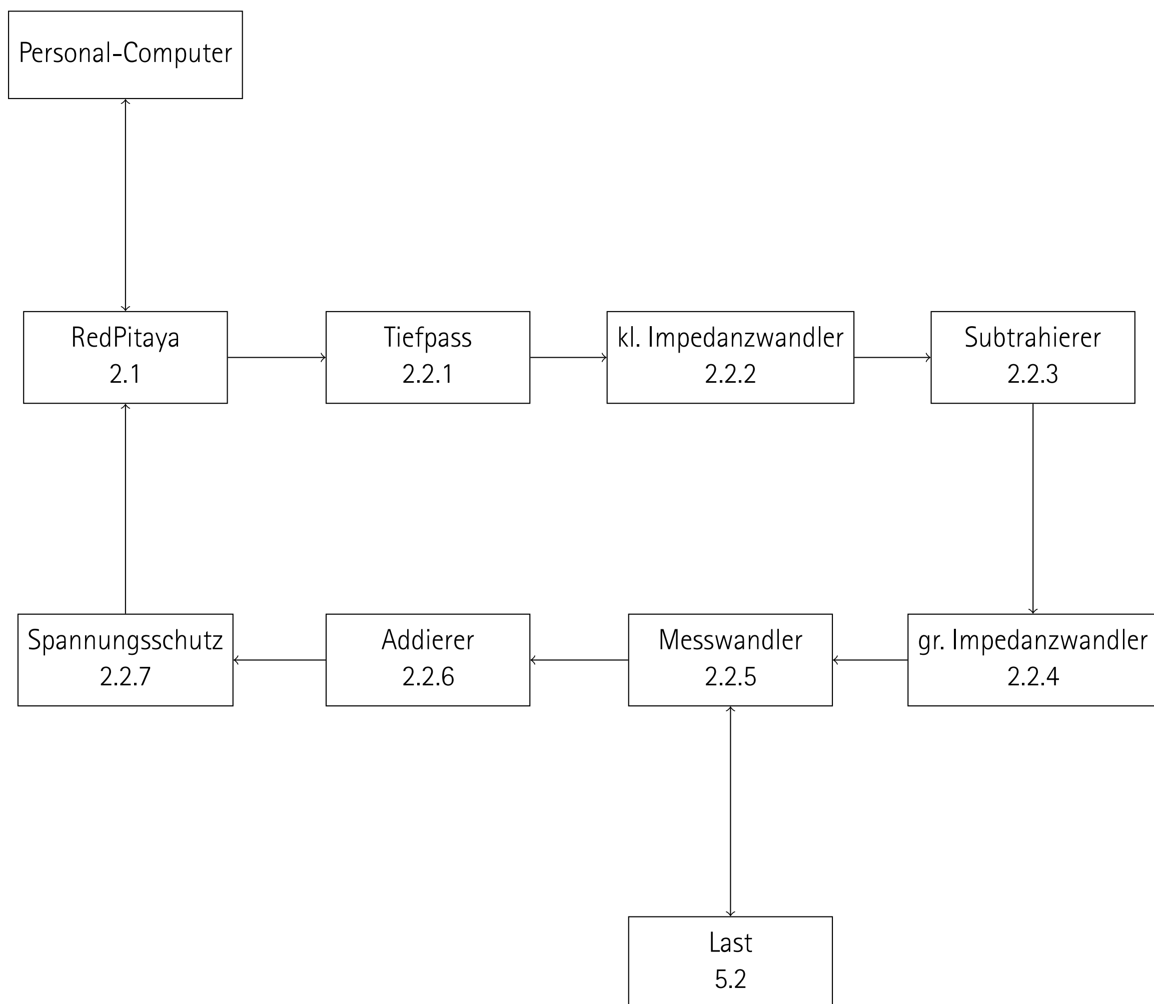


Abb. 2.3: Schematische Übersicht über den Messaufbau. Netzwerkverbindung zwischen Personal Computer und RedPitaya zur Programmierung und Datenübertragung (oben links). Ausgangssignale von RedPitaya gefiltert, impedanzgewandelt und an die Last angelegt (von RedPitaya aus im Uhrzeigersinn). Messgrößen vom Messwandler erfasst, dann bearbeitet und sicher an die Eingänge des RedPitaya angelegt (von Messwandler aus im Uhrzeigersinn). Nummer des jeweils beschreibenden Abschnitts unter dem Schriftzug jedes Elements.

2.2.1 Tiefpass

Der Einplatinencomputer verfügt über vier analoge Ausgänge, die jeweils ein tiefpassgefiltertes Signal einer Pulsdauermodulation, engl. Pusle Width Modulation (PWM) bereitstellen.[8] Das ausgegebene analage Signal weist eine sehr große Restwelligkeit auf, wie in *Abb. 2.4* illustriert. Die Welligkeit lässt sich nahezu vollständig eliminieren, indem ein geeignetes Filter verbaut wird. Im Folgenden wird der Schaltplan eines Tiefpasses gezeigt, anschließend die Implementierung und deren Charakterisierung.

Schaltung

Der Schaltplan ist in *Abb. 2.5* abgedruckt. Der RC-Tiefpass erster Ordnung hat eine Grenzfrequenz von:

$$f_0 = \frac{1}{RC} \approx 21\,\text{kHz} \tag{2.1}$$

Der Eingang ist hochohmig genug, um ihn direkt mit dem Signal des Einplatinencomputers zu speisen. Der Ausgang ist zu hochohmig, um das Signal direkt weiterzuverarbeiten. Ein geeigneter Impedanzwandler wird

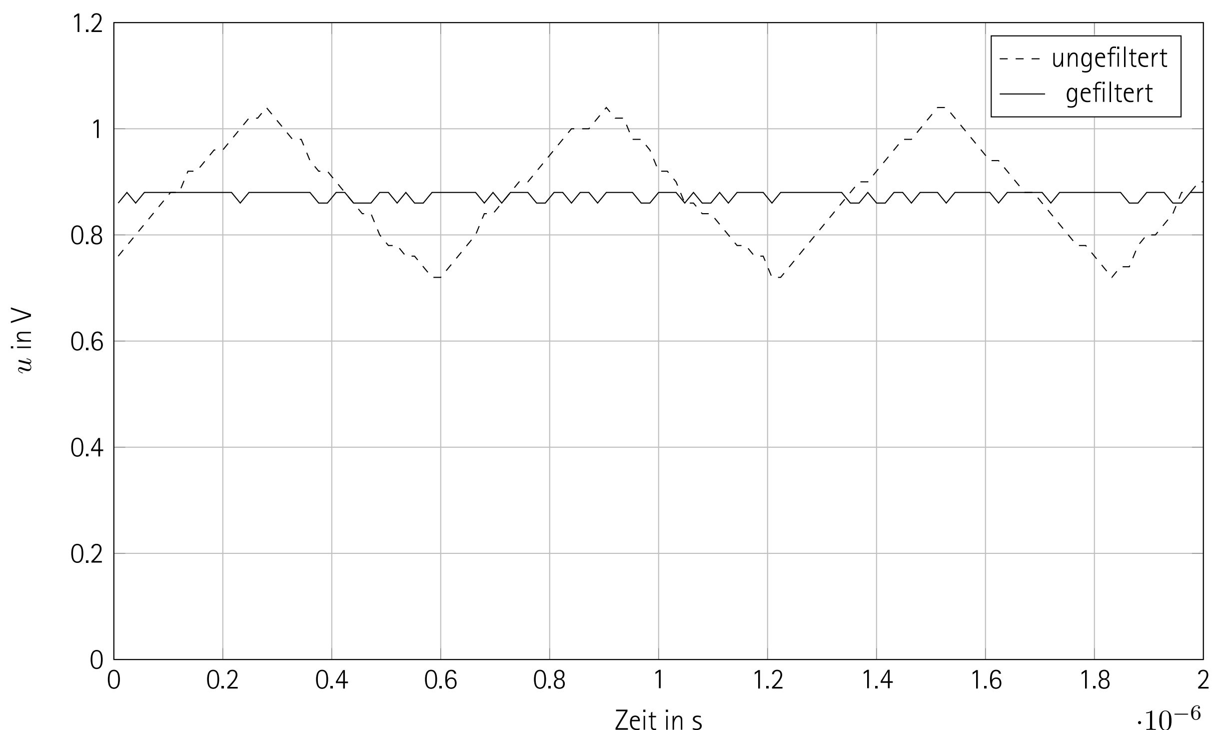


Abb. 2.4: Welligkeit des Signalausgangs. Ungefiltertes Signal etwas durch „Jumper-Kabel" gedämpft. Gefiltertes Signal mit Reihenschaltung aus Tiefpass und kleinem Impedanzwandler. Aufgenommen mit Oszilloskop „Tektronix DPO 2012B". Geringe Signalauflösung. Vollständige Unterdrückung des „Ripples" durch das Filter. Verbleibendes Ripple ist artefaktisch überzeichnet.

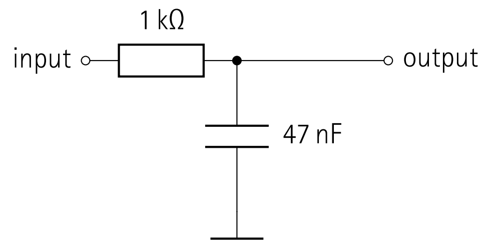


Abb. 2.5: Schaltplan des Tiefpasses. Einfacher RC-Tiefpass mit einer Grenzfrequenz von 21 kHz. Kurzschlussfester Ausgang, mindestens mittelhochimpedanter Eingang und Notwendigkeit zur Impedanzwandlung des Ausgangssignals durch mittelhohen ohmschen Widerstand.

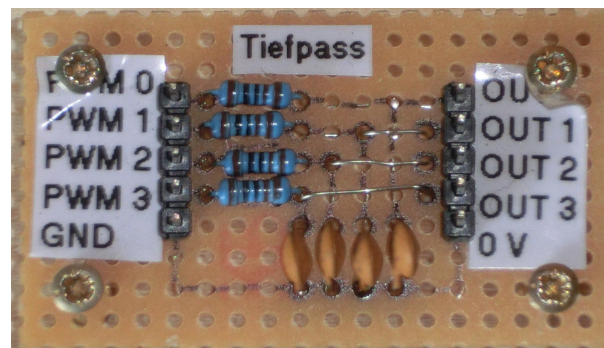


Abb. 2.6: Fotografie des Tiefpasses. Eingang für das verrauschte Signal (links), Metallschichtwiderstände und Keramikkondensatoren für vier parallele RC-Tiefpässe. Ausgang für gefiltertes Signal (rechts).

im folgenden Abschnitt vorgestellt (siehe Abschnitt 2.2.2, S. 66).

Implementierung

Die fertiggestellte Schaltung ist in *Abb. 2.6* präsentiert. Die Schaltung beinhaltet vier Kanäle und deckt somit alle „langsamen Ausgänge“ des signalgenerierenden Computers ab.[3] Die Anschlüsse sind mit einer Nummerierung beschriftet.

Der maximale Strom wird durch die Belastbarkeit des Widerstandes bestimmt:

$$I_{\mathrm{max}} = \sqrt{\frac{0{,}25\,\mathrm{W}}{1\,\mathrm{k\Omega}}} \approx 16\,\mathrm{mA} \tag{2.2}$$

Die maximale Spannung gegenüber dem Bezugspotential ist von dem Keramikkondensator auszuhalten und beträgt somit 50 V.

[3] Bei den später durchgeführten Versuchen werden nur zwei Kanäle benötigt (siehe Abschnitt 2.3.2, S. 94). Die Schaltung soll aber „zukunftssicher“ sein, falls mehr Kaäle benötigt werden.

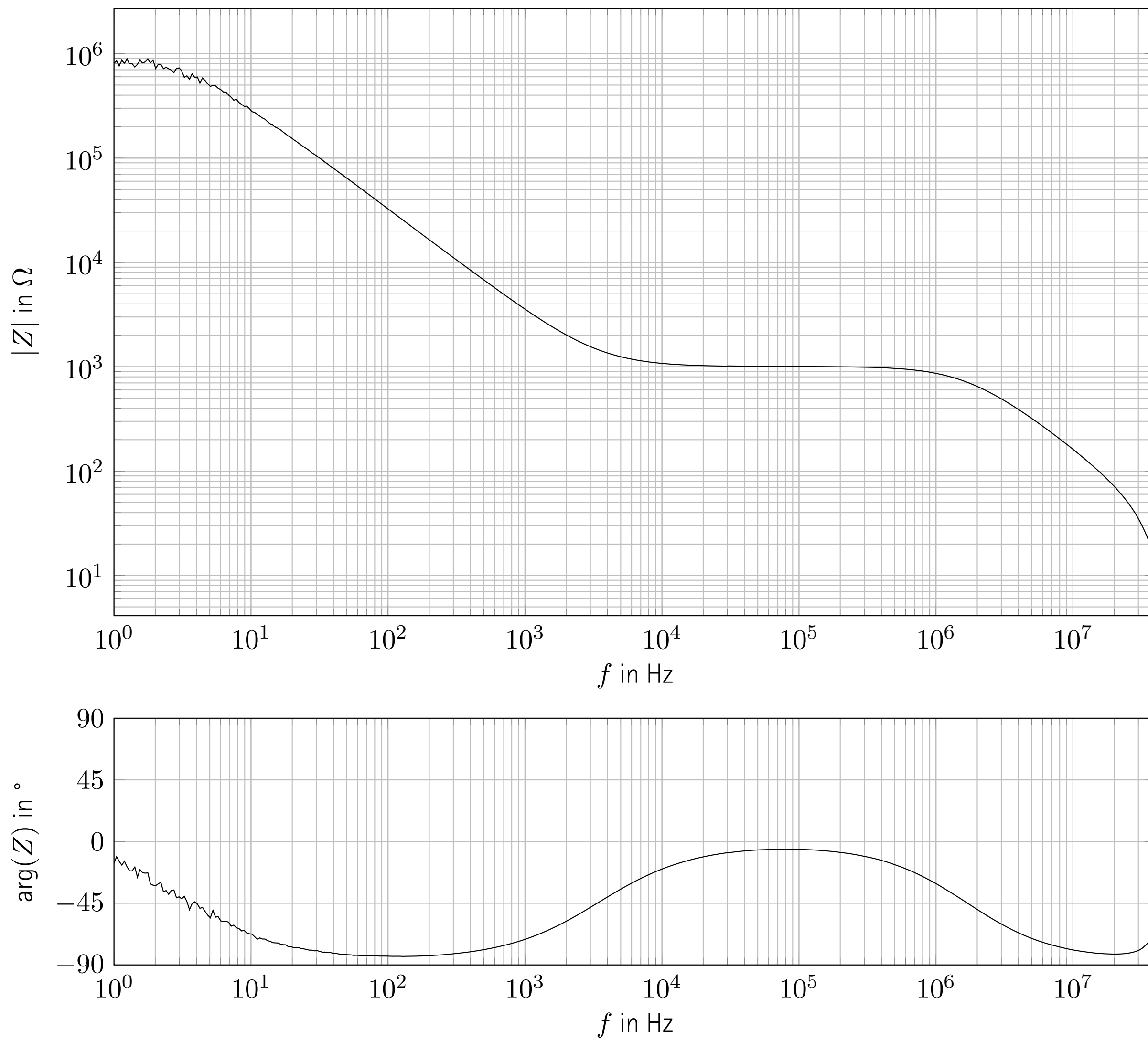


Abb. 2.7: Eingangsimpedanz des Tiefpasses bei offenen Ausgangsklemmen. 401 Messpunkte, aufgenommen mit Bode100 (Ein-Port-Messung, Empfängerbandbreite: 1 Hz). Hoch- bis mittelimpedanter Eingang, große parasitäre Kapazität im MHz-Bereich.

Charakterisierung

Die Eingangsimpedanz ist in *Abb. 2.7* und die Transferfunktion in *Abb. 2.8* gezeigt. Daraus wird ersichtlich, dass die Quelle nur hochohmig belastet, das Nutzsignal transferiert und die Störung herausgefiltert wird. Die Schaltung ist nur eingeschränkt für harmonische Anteile oberhalb von 10 MHz geeignet.

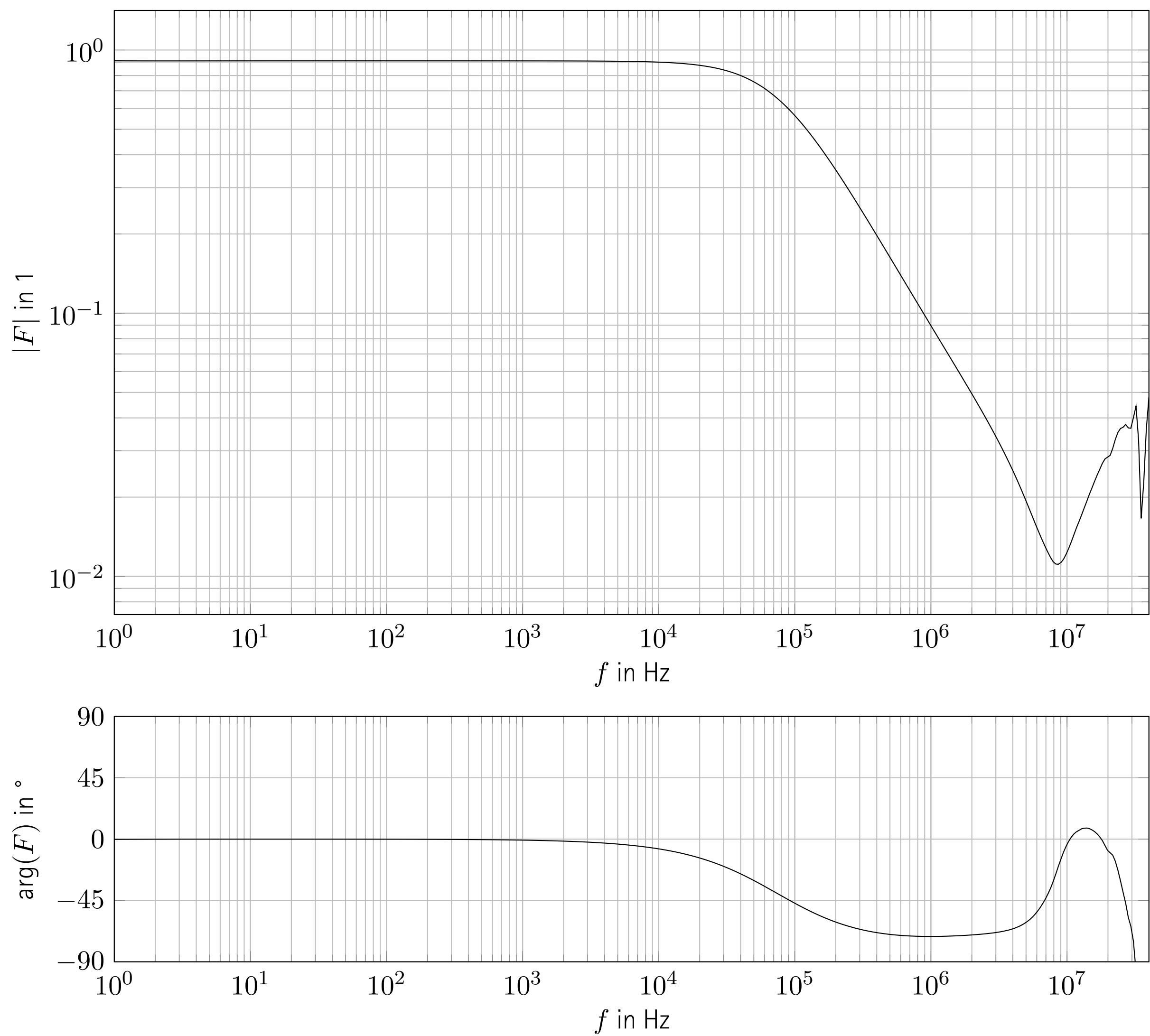


Abb. 2.8: Transferfunktion des Tiefpasses bei offenen Ausgangsklemmen. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Nahezu ideales Verhalten bis in den unteren MHz-Bereich.

2.2.2 Kleiner Impedanzwandler

Der Ausgang des Tiefpasses (siehe Abschnitt 2.2.1, S. 62) ist zu hochohmig, um dessen Signal direkt weiterzuverarbeiten, weshalb ein Impedanzwandler angeschlossen wird. Dessen Ströme sind um etwa zwei Größenordnungen stärker begrenzt als die eines weiteren, später vorgestellten Impedanzwandlers (siehe Abschnitt 2.2.4, S. 73), weshalb das Ausgangssignal des „kleinen" Impedanzwandlers lediglich für die Signalverarbeitung geeignet ist. Die Schaltung und ihre Implementierung werde folgend vorgestellt und charakterisiert.

Schaltung

Der Schaltplan ist in *Abb. 2.9* festgehalten. Es handelt sich um eine nichtinvertierende OPV-Schaltung mit einem Verstärkungsfaktor von 1. Das verwendete Integrierte Schaltung, engl. Integrated Circuit (IC) ist kostengünstig und weit verbreitet. Es beinhaltet vier OPV in einem Gehäuse, sodass alle vier Ausgangssignale des Tiefpasses auf einer einzigen Platine Impedanzgewandelt werden können. Der große Widerstand dient dem Schutz des Eingangs gegenüber einkoppelnden hohen Spannungen bei einem „schwebenden Potential".

Implementierung

Die Platine ist in *Abb. 2.10* abgebildet und beinhaltet vier Instanzen der Schaltung, um einen kleinen Impedanzwandler für jeden Tiefpass bereitzustellen. Die Anschlüsse sind auf die selbe Weise nummeriert und beschriftet wie die Platinen mit den Tiefpässen, um Fehler beim Aufbau zu vermeiden.

Charakterisierung

Der Ausgang des OPV kann einen Strom zwischen 10 mA und 40 mA treiben, worauf die nachfolgenden Schaltungen auszulegen sind. Der Ausgang ist kurzschlussfest. Die Eingangsströme liegen im nA-Bereich, sodass der Eingang den angeschlossenen Ausgang des Tiefpasses kaum belastet wird. [30] Die Übertragungsfunktion

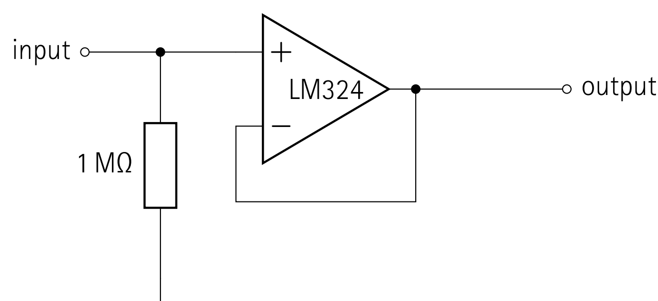


Abb. 2.9: Schaltplan des kleinen Impedanzwandlers. Nichtinvertierende Verstärkerschaltung mit dem Verstärkungsfaktor 1. Standardbaustein „LM324" zur einfachen Implementierung von vier Instanzen auf einer Platine. Schutzwiderstand gegen „schwebendes Potential"

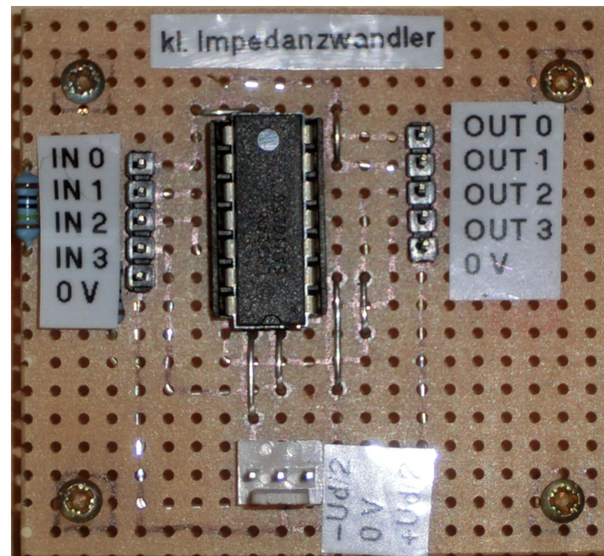


Abb. 2.10: Fotografie des kleinen Impedanzwandlers. Vier Eingänge mit Schutzwiderständen (links), IC *LM324* (mittig), vier Ausgänge (rechts) und Stecker für die Spannungsversorgung (unten).

ist in *Abb. 2.11* dargestellt. Daraus wird deutlich, dass die Schaltung ein nahezu ideales Verhalten bis zu einer Frequenz von ca. 100 kHz und anschließend eine Dämpfung aufweist.

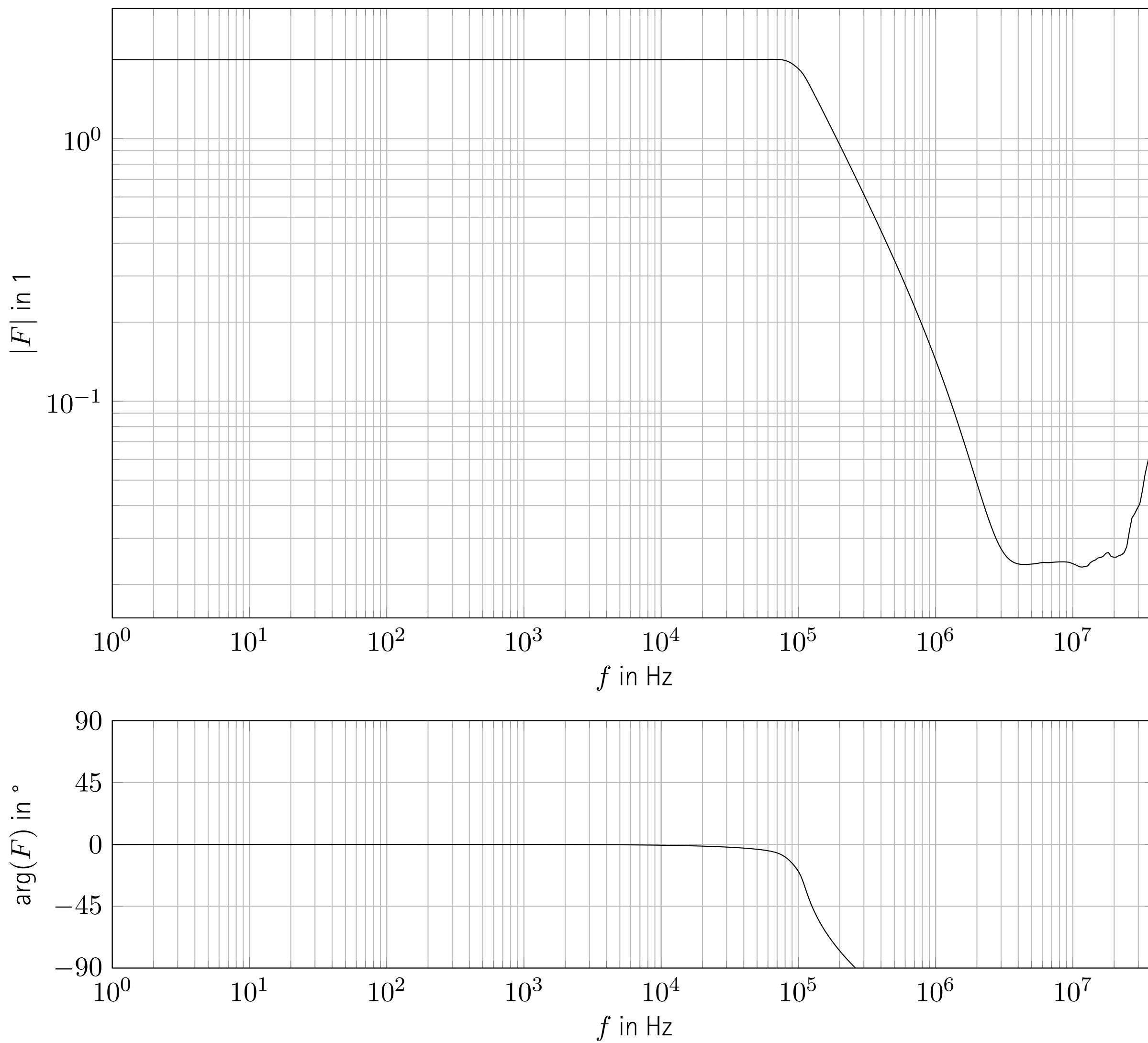


Abb. 2.11: Transferfunktion des kleinen Impedanzwandlers bei offenen Ausgangsklemmen. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Nahezu ideales Verhalen bis ca. 100 kHz, anschließend Dämfung (in etwa Tiefpass erster Ordnung). Siehe Einleitung wegen statischem Faktor 2 (siehe Abschnitt 2.2, S. 57).

2.2.3 Subtrahierer

Weil einige Ausgänge des Signalgenerator einen Spannungsbereich aufweisen, der nicht symmetrisch um 0 V liegt, muss das generierte Signal ggf. noch verschoben werden. Dazu dient ein einfacher Subtrahierer auf Basis einer Operationsverstärkerschaltung. Zunächst wird die Schaltung vorgestellt, anschließend ihre Implementierung. Schließlich wird letztere charakterisiert.

Schaltung

Der Schaltplan ist in *Abb. 2.12* illustriert. Die spannungsregelnde IC „7905" erzeugt aus der negativen Versorgungsspannung $-\frac{U_\mathrm{d}}{2}$ die konstante negative Spannung von $-5\,\mathrm{V}$. Die beiden Keramikkondensatoren dienen der Stabilisierung des IC. Weitere, bspw. größere Elektrolyt-, Kondensatoren sind nicht erforderlich, weil die Last konstant und gering ist. Der Spannungsregler wird mit einem Spannungsteiler belastet, dessen oberer[4] Widerstand einstellbar ist. Die Mittelspannung des Teilers wird von einem OPV mit dem Faktor 1 nichtinvertierend verstärkt, um den Teiler nicht zu belasten und das Ergebnis somit nicht zu verfälschen.

Die negative Spannung wird mithilfe einer Additionsschaltung, bestehend aus einem weiteren OPV und drei Präzisionswiderständen[5] zum eingehenden Signal addiert. Das Ergebnis steht als Spannung gegenüber dem Massepotential am Ausgang der Schaltung zur Verfügung. Die addierende Schaltung wirkt invertierend, was bei der softwareseitigen Kalibrierung später beachtet werden muss.

Implementierung

Die gelötete Platine ist in *Abb. 2.13* abgedruckt. Die Schaltung implementiert zwei Subtrahierer, die sich eine gemeinsame negative Spannung teilen. Auf diese Weise werden nicht nur Materialien und Aufwand gespart, sondern es wird sichergestellt, dass zwei Signale nahezu identisch verschoben werden. Die Anschlüsse sind mithilfe eines Etikettendruckgeräts beschriftet.

Durch die Wahl eines Trimmpotentiometers mit einem Nennwiderstand von 10 kΩ lässt sich das eingehende Signal um 0,22 V bis 5 V verschieben. Der OPV „LM324" begrenzt die Zwischenkreisspannung auf den Bereich $U_\mathrm{d} \in [3\,\mathrm{V}; 32\,\mathrm{V}]$ und die maximale Strombelastung des Ausgangs auf 10 mA. [30]

[4] Durch die negative Versorgungsspannung für den Teiler ist der obere Widerstand hier derjenige, der mit dem niedrigen (negativeren) Potential verbunden ist.

[5] Bei der Herstellung der Platine sind keine Präzisionwiderstände verfügbar, weshalb Metallschichtwiderstände mit einer Toleranz von 1 % genutzt werden. Kleine resultierende Fehler können weitestgehend durch eine geeignete softwareseitige Kalibrierung kompensiert werden.

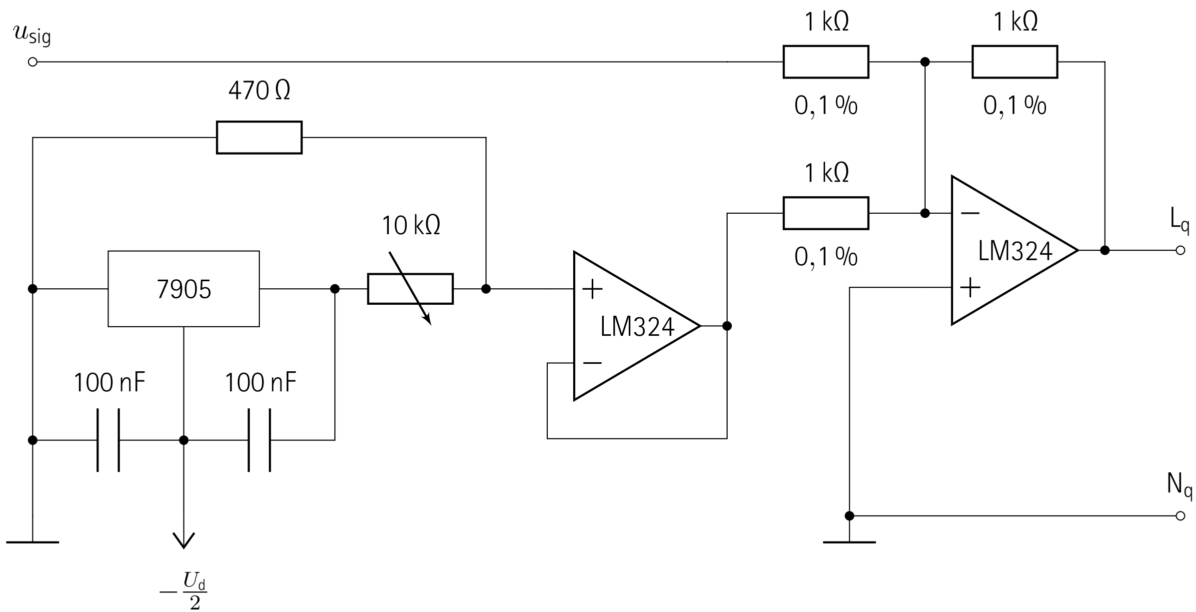


Abb. 2.12: Schaltplan des Subtrahierers. Signaleingang (links), Phase und Neutralleiter (rechts). Bereitstellung einer negativen Spannung von −5 V durch IC „7905" (links). Stufenlos einstellbare negative Spannung durch Spannungsteiler (links) und Impedanzwandler (mittig). Addition der negativen Spannung mit Signaleingang durch OPV und Präzisionswiderstände (rechts). Versetzt ein eingehendes Signal gegenüber Masse um eine manuell stufenlos einstellbare negative Spannung gegenüber Masse. Hochohmiger Ausgang.

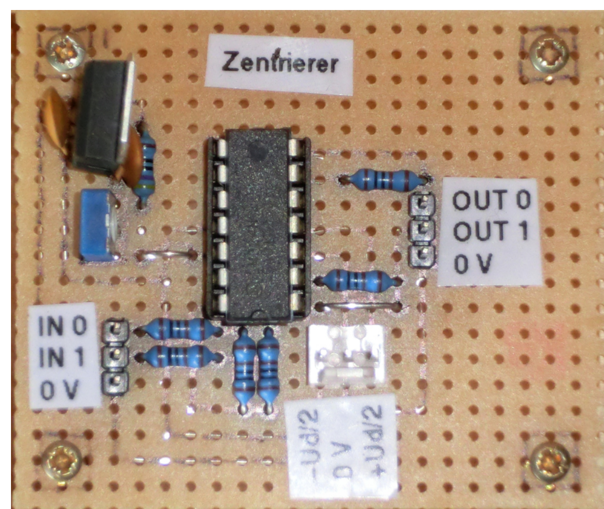


Abb. 2.13: Fotografie des Subtrahierers. Zwei Eingänge (links unten), Spannungsregler und Spannungsteiler (links oben), OPV in invertierender Addiererschaltung (mittig), zwei Ausgänge (rechts) und Stecker für die Spannungsversorgung (unten). Synonyme Beschriftung „Zentrierer".

Charakterisierung

Auf die Messung der Übertragungsfunktion wird bei dieser Schaltung verzichtet. Weil der Aufbau ähnlich zur vorangegangenen ist (siehe Abschnitt 2.2.2, S. 66), kann auch eine ähnliche Frequenzabhängigkeit der Übertragungsfunktion erwartet werden. Die Verschiebung des Ausgangssignals gegenüber dem Potential 0 V verhindert eine brauchbare Messung mithilfe des „Bode 100".

Die Eingangsimpedanz ist in *Abb. 2.14* präsentiert. Durch den Widerstand von 1 kΩ zwischen dem Signaleingang und dem auf Masse liegenden invertierenden Eingang des OPV ergibt sich eine ohmsche Last von 1 kΩ bis zu einer Frequenz von ca. 100 kHz. Daran schließst sich eine kapazitive Charakteristik an.

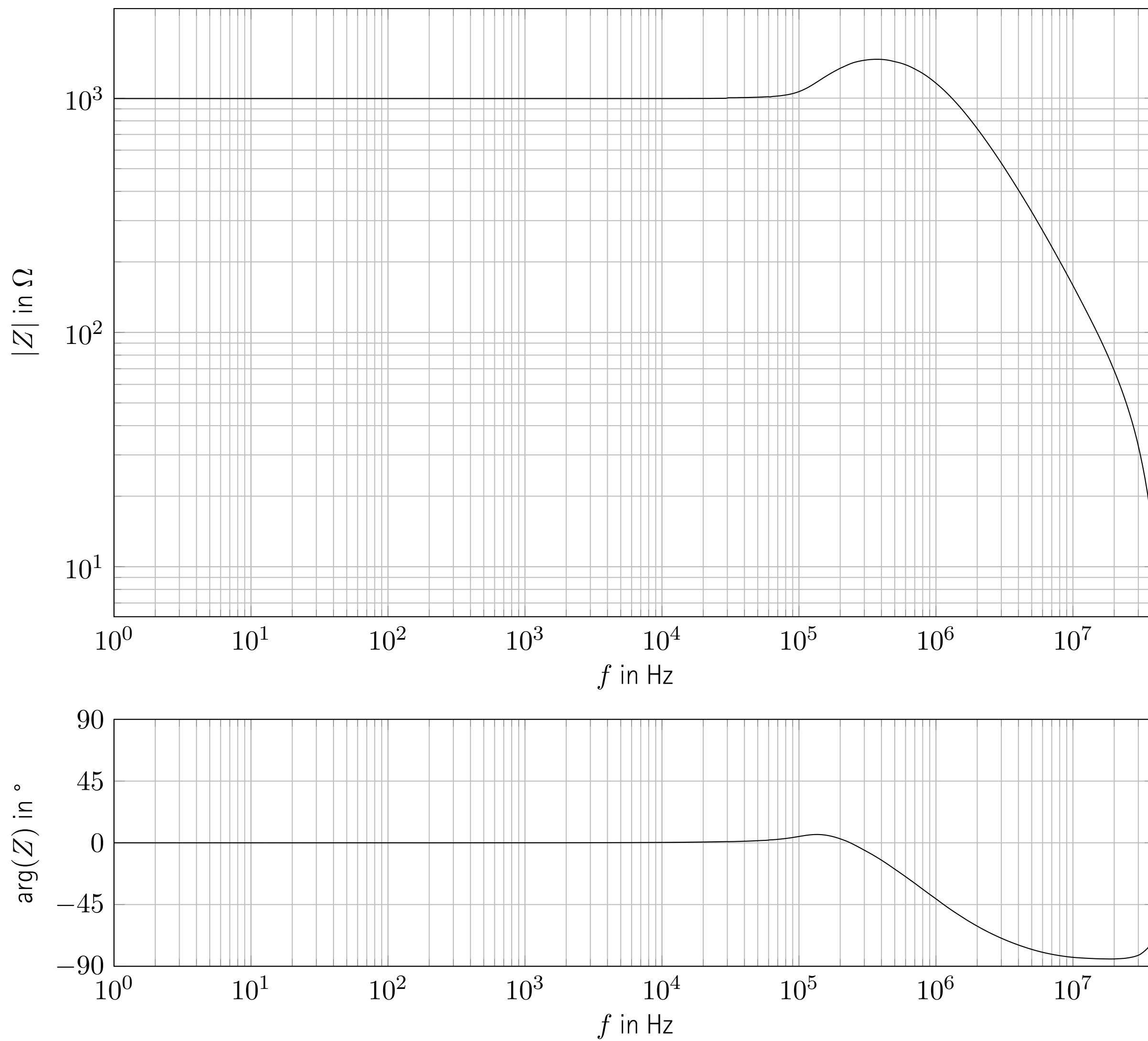


Abb. 2.14: Eingangsimpedanz des Subtrahierers bei offenen Ausgangsklemmen. 401 Messpunkte, aufgenommen mit Bode100 (Ein-Port-Messung, Empfängerbandbreite: 1 Hz). Nahezu ideale ohmsche Last von 1 kHz bis ca. 100 kHz, anschließend leichte Resonanz und Dämpfung.

2.2.4 großer Impedanzwandler

Die zu vermessenden Netzwerkelemente weisen unterschiedliche Impedanzen auf, teils auch sehr niedrige. Weil der Signalgenerator und andere signalverarbeitende Schaltungen, wie der Subtrahierer (siehe Abschnitt 2.2.3, S. 69) hochohmige Ausgänge aufweisen, wird ein Impedanzwandler benötigt. Er dient hier als leistungselektronische Treiberstufe. Zunächst wird die Schaltung vorgestellt, anschließend ihre Implementierung. Schließlich wird letztere charakterisiert.

Schaltung

Der Schaltplan ist in *Abb. 2.15* gezeigt. Die Stützkondensatoren sind der Übersicht halber nicht eingetragen.[6] Er beinhaltet lediglich einen OPV des Typs „OPA548" und dessen notwendige Beschaltung. Der OPV wird in der nicht-invertierenden Verstärkerschaltung betrieben, deren Verstärkungsfaktor sich durch einen einstellbaren Widerstand einfach ändern lässt. Weil das verwendete IC sehr hohe Ströme treiben kann, stellt es bereits einen funktionsfähigen Impedanzwandler dar.

Die Phase wird quellenseitig durch einen geringen Widerstand gegen versehentliche Kurzschlüsse geschützt. Außerdem werden die Eingänge des OPV über einen hohen Widerstand verbunden, um im Fall einer getrennten Quelle eine undefinierte Potentialdifferenz zu vermeiden, welche sonst zur Zerstörung des IC führen kann.

Das IC beinhaltet neben dem OPV weitere Komponenten, bspw. eine Strombegrenzung. Für deren Einstellung wird der entsprechende Pol über einen einstellbaren Widerstand mit der negativen Versorgungsspannung verbunden.

Implementierung

Die gelötete Platine ist in *Abb. 2.16* festgehalten. Das IC ist mit einem Kühlkörper ausgestattet. Die beiden einstellbaren Widerstände sind als Reihe ausgeführt, deren Elemente mit jeweils einer Steckbrücke ausgewählt werden können. Auf diese Weise können alle Widerstandselemente einmal gemessen und deren Wirkung genau notiert werden. Die Berechnung der Strombegrenzung geschieht mithilfe der Formel: [29, S. 11]

$$R_{\mathrm{CL}} = \frac{15\,000 \cdot 4{,}75\,\mathrm{V}}{I_{\mathrm{lim}}} - 13\,750\,\Omega \tag{2.3}$$

Die berechneten Strombegrenzungen und Verstärkungsfaktoren sind mit einem Etikettiergerät auf der Platine vermerkt.

[6] Für die IC empfiehlt das Datenblatt einen Kondensator von 100 µF und einen von 100 nF je für die positive und für die negative Vesorgungsspannung dicht an der IC.

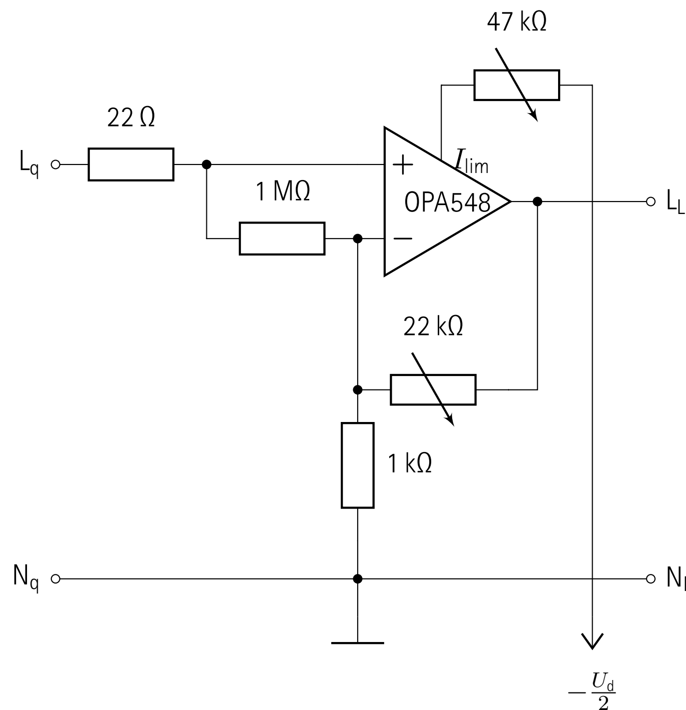


Abb. 2.15: Schaltplan des Impedanzwandlers. Anschlüsse für Signalquelle (links) und Last (rechts). Schutzwiderstand für Quelle im Fall eines ungewollten Kurzschlusses, Sicherheitswiderstand zwischen Eingängen des OPV zum Schutz vor nicht angeschlossenem Eingang und Elektrostatische Entladung, engl. Electrostatic Discharge (ESD). OPV in einstellbarer, nicht-invertierender Verstärkerschaltung. Einstellbarer Widerstand zur internen Strombegrenzung des IC. Neutralleiter und Masse kurzgeschlossen. Verstärkung des eingehenden Signals und Verringerung der Impedanz bei gleichzeitiger Strombegrenzung. Vier Stützkondensator der Übersicht halber nicht eingetragen.

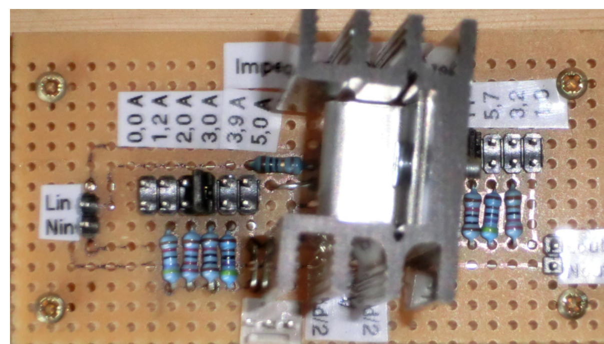


Abb. 2.16: Fotograpie des großen Impedanzwandlers. Signaleingang, einstellbare Strombegrenzung, IC „OPA548" mit Kühlkörper, einstellbaree Verstärkung, Ausgang (von links nach rechts). Stecker für die Spannungsversorgung (unten).

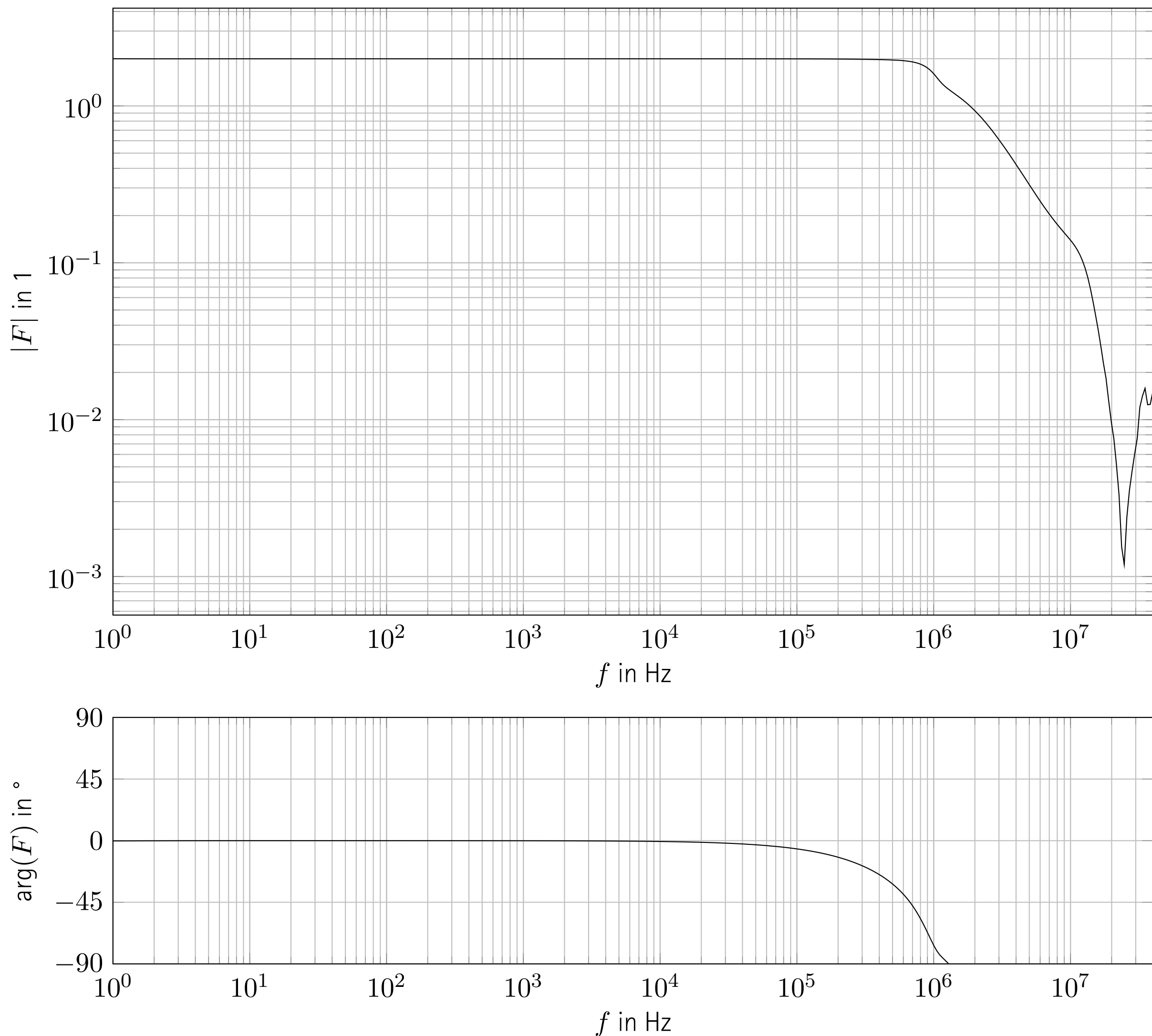


Abb. 2.17: Transferfunktion des großen Impedanzwandler bei einer ohmschen Last von 50 Ω. Verstärkungsfaktor $V = 1$. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Nahezu ideale Charakteristik bis ca. 40 kHz, zunehmende Phasenverschiebung bis ca. 600 kHz, anschließend Dämpfung. Siehe Einleitung wegen statischem Faktor 2 (siehe Abschnitt 2.2, S. 57).

Charakterisierung

Der Ausgang ist auf die mit einer Kontaktbrücke ausgewählte Stromstärke begrenzt, höchstens jedoch 5 A. Der Ausgang ist indirekt kurzschlussfest, weil das IC bei zu hohen Temperaturen von selbst abschaltet. Das Spannungssignal kann mit einer Kontaktbrücke um den Faktor 1 bis 23 verstärkt werden. Sofern keine weiteren Angaben gemacht sind, wird im Rahmen dieser Arbeit immer der Faktor 1 ausgewählt. Die Übertragungsfunktion ist in *Abb. 2.17* abgebildet und zeigt das nahezu ideale Verhalten bis zu einer Frequenz von ca. 50 kHz und die anschließend einsetzende Dämpfung.

2.2.5 Messwandler

Die zu messenden Ströme und Spannungen benötigen jeweils eines Messwandlers, um ein (Spannungs-)Signal in einem geeigneten Bereich zu generieren, welches vom Messsystem digitalisiert werden kann. Zunächst wird die Schaltung vorgestellt, anschließend ihre Implementierung. Schließlich wird letztere charakterisiert.

Schaltung

Der Schaltplan ist in *Abb. 2.15* dargestellt. Die Phasenspannung wird mithilfe eines einstellbaren Spannungsteilers geteilt und das Ergebnis mit einem OPV in nicht-invertierender Verstärkerschaltung mit dem Verstärkungsfaktor 1 an den Messausgang angelegt. Der Neutralleiterstrom wird mit einem unpräzise gefertigten, aber festen und niederohmigen Shuntwiderstand gemessen, indem die darüber abfallende Spannung mit einem weiteren OPV in nicht-invertierender Verstärkerschaltung mit einstellbarem Verstärkungsfaktor an den Messausgang angelegt wird.

Der Neutralleiter ist quellenseitig mit dem Massepotential verbunden. Deshalb ist eine Vertauschung der Quellen- und der Lastseite nicht möglich.

Implementierung

Die gelötete Platine ist in *Abb. 2.16* illustriert. Sowohl die Phase als auch der Neutralleiter sind niederohmig durchgeführt, um eine Verfälschung der Messergebnisse zu minimieren. Beide einstellbaren Widerstände sind als Reihe ausgeführt, deren Elemente mit jeweils einer Steckbrücke ausgewählt werden können. Auf diese Weise können alle Widerstandselemente einmal gemessen und deren Wirkung genau notiert werden. Die berechneten Faktoren sind mit auf Etiketten auf der Platine festgehalten.

Charakterisierung

Das eingespeiste Signal wird nahezu unverändert an die Last durchgegeben. Wegen der extrem geringen Innenimpedanz des vorgeschalteten großen Impedanzwandlers stellt die niedrige Querimpedanz von 2,2 kΩ bis 59,2 kΩ eine vernachlässigbar kleine Last dar (siehe Abschnitt 2.2.4, S. 73). Die nur aus dem Shuntwiderstand und parasitären Größen gebildete Längsimpedanz muss zu der angeschlossenen Last passen und darf entsprechend nicht zu groß sein. Ein zu kleiner Shuntwiderstand bedarf allerdings einer unpraktischen Verstärkung, die zu erheblichem Signalrauschen führen kann. Für die in dieser Arbeit verwendeten Lasten (siehe Abschnitt 5.2, S. 141) wird eine Shuntwiderstand mit einem Nennwiderstand von 1 Ω verwendet. Die zwei Shuntwiderstände für die zwei Instanzen der implementierten Schaltungen werden mit der Ein-Port-Impedanzmessung des Bode100 bei einer Frequenz von 100 Hz bestimmt: $R_{\text{shunt},1} = 1{,}131\,\Omega$; $R_{\text{shunt},2} = 1{,}133\,\Omega$ [7]

[7] Die Empfängerbandbreite beträgt 1 Hz. Die Phasen sind mit $\phi_{\text{shunt},1} = 0{,}048°$ und $\phi_{\text{shunt},2} = 0{,}047°$ zu vernachlässigen.

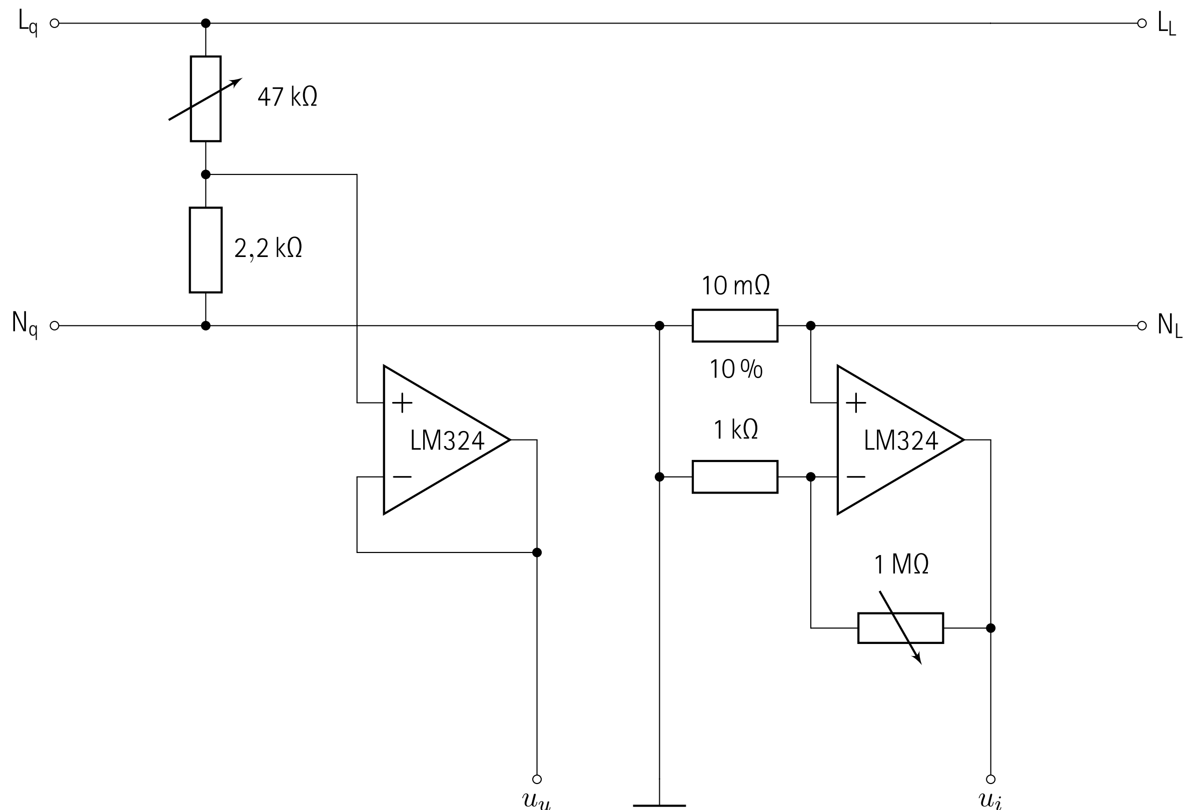


Abb. 2.18: Schaltplan des Messwandlers. Anschlüsse für Quelle (links), Last (rechts) und Messgeräte (unten). Messung der Leiterspannung mit einstellbarem Spannungsteiler (links) und Operationsverstärker. Messung des Neutralleiterstroms mit Shuntwiderstand und einstellbarer Messverstärkerschaltung (rechts). Phase und Neutralleiter niederohmig durchgeführt, Neutralleiter quellenseitig auf Masse gelegt, analoge, symmetrische Messsignale gegen Masse.

Die Messung mithilfe des Messwandlers geschieht stromrichtig, allerdings kann die korrekte Spannung einfach berechnet werden, weil der verwendete Shuntwiderstand als ideal angenommen werden kann. Er wird durch den Pulsbetrieb (siehe Abschnitt 2.3.2, S. 94) typischerweise nur geringfügig erwärmt und nähert ausreichend gut das ideale Frequenzverhalten eines ohmschen Widerstands an, wie in *Abb. 2.20* abgedruckt.

Die gemessene Spannung kann durch eine Kontaktbrücke mit einem Faktor zwischen $\frac{1}{21.4}$ und 1 verstärkt werden oder auch auf 0 V gesetzt werden, um den Ausgang zu deaktivieren. Das erhaltene Signal gilt gegenüber dem Massepotential und wird an den Ausgang zur Spannungsmessung angelegt. Der Strom wird als Spannung über dem Shunt-Widerstand gemessen, wodurch sich der o.g. einheitenbehaftete Faktor ergibt. Er kann zusätzlich durch eine Kontaktbrücke auswählbar mit einem Faktor zwischen 1 und 1000 verstärkert werden, um kleine Signale einfacher zu messen. Das erhaltene Signal gilt gegenüber dem Massepotential und wird an den Ausgang zur Strommessung angelegt. Beide Signale bedürfen wegen Fertigungstoleranzen einer linearen Kalibrierung.

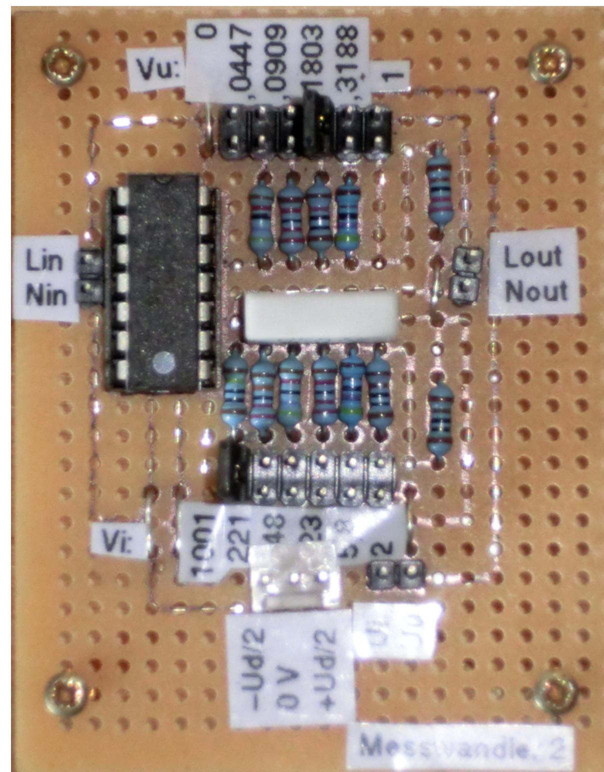


Abb. 2.19: Fotografie des Messwandlers. Eingang und IC mit vier OPV (links), einstellbarer Spannungsteiler (oben), einstellbarer Strommesswandler (unten), Shunt-Widerstand (mittig), Leistungsausgang (rechts), Signalausgang und Stecker für Spannungsversorgung (unten).

Die Übertragungsfunktion für die Spannungsmessung mit einem Verstärkungsfaktor von 1 ist in *Abb. 2.21* präsentiert und zeigt ein nahezu ideales Verhalten bis zu einer Frequenz von 10 kHz sowie eine starke Dämpfung im Bereich zwischen 80 kHz und 8 MHz.

Bei einer zuvor durchgeführten Messung mit einem Verstärkungsfaktor von $1000 \cdot 10\,\mathrm{m\Omega}$ besteht das nahezu ideale Verhalten bis lediglich ca. 100 Hz, wie in *Abb. 2.23* gezeigt. Danach tritt eine unerwünschte Dämpfung auf. Um die starke Frequenzabhängigkeit zu reduzieren, wird der Shunt-Widerstand von 10 mΩ gegen einen anderen (1 Ω) ausgetauscht. Die Impedanz des neuen Widerstands ist in *Abb. 2.20* und die Übertragungsfunktion der Schaltung in *Abb. 2.22* festgehalten. Es ist deutlich zu erkennen, wie der Bereich des nahezu idealen Verhaltens durch den größeren Shunt-Widerstand bis zu einer Frequenz von 10 kHz erweitert ist.

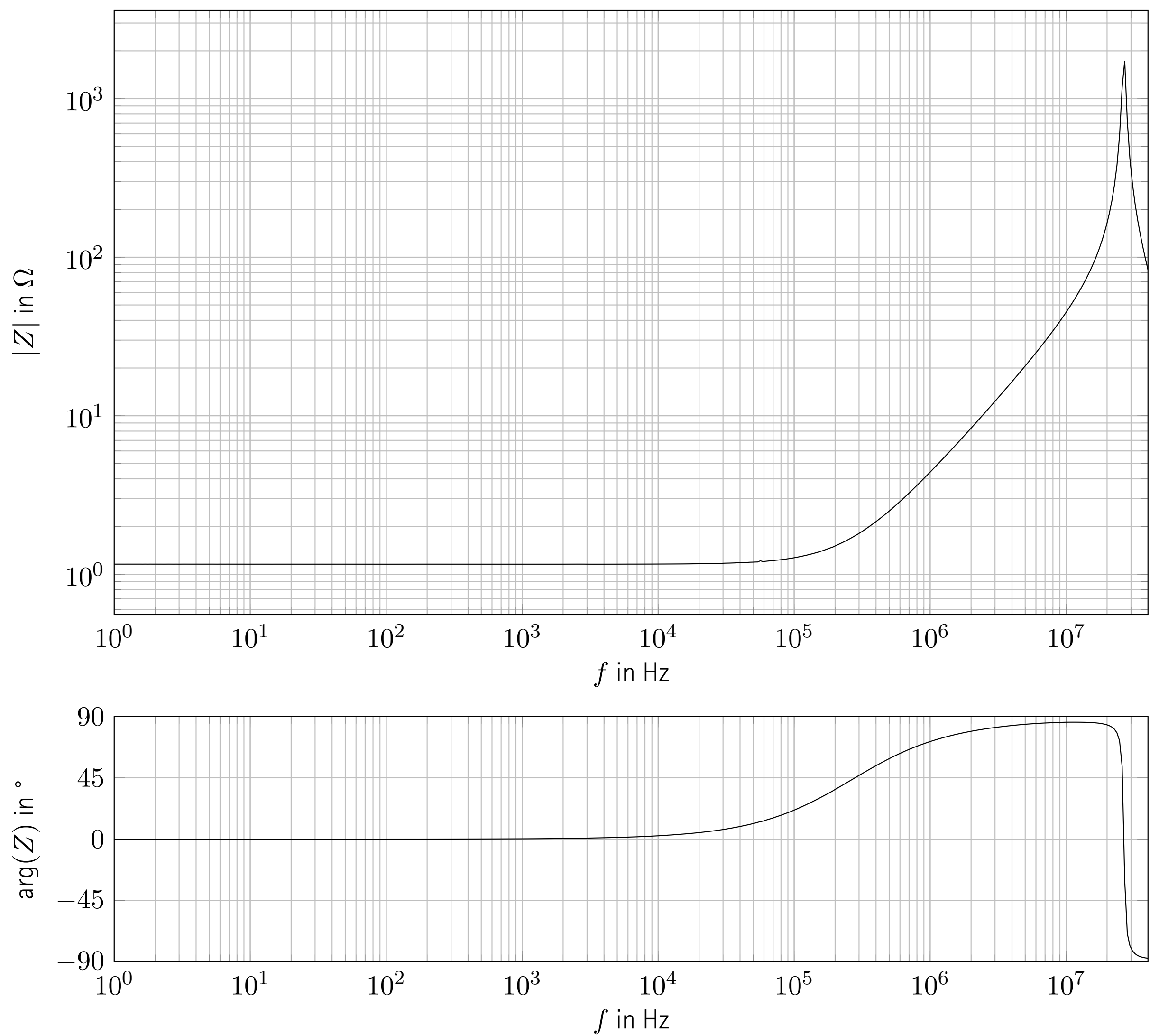


Abb. 2.20: Impedanz des Shuntwiderstands (1 Ω). 401 Messpunkte, aufgenommen mit Bode100 (Ein-Port-Messung, Empfängerbandbreite: 1 Hz). Nahezu ideale Charakteristik bis ca. 10 kHz, anschließend dominierende parasitäre Induktivität, Resonanz bei 27 MHz

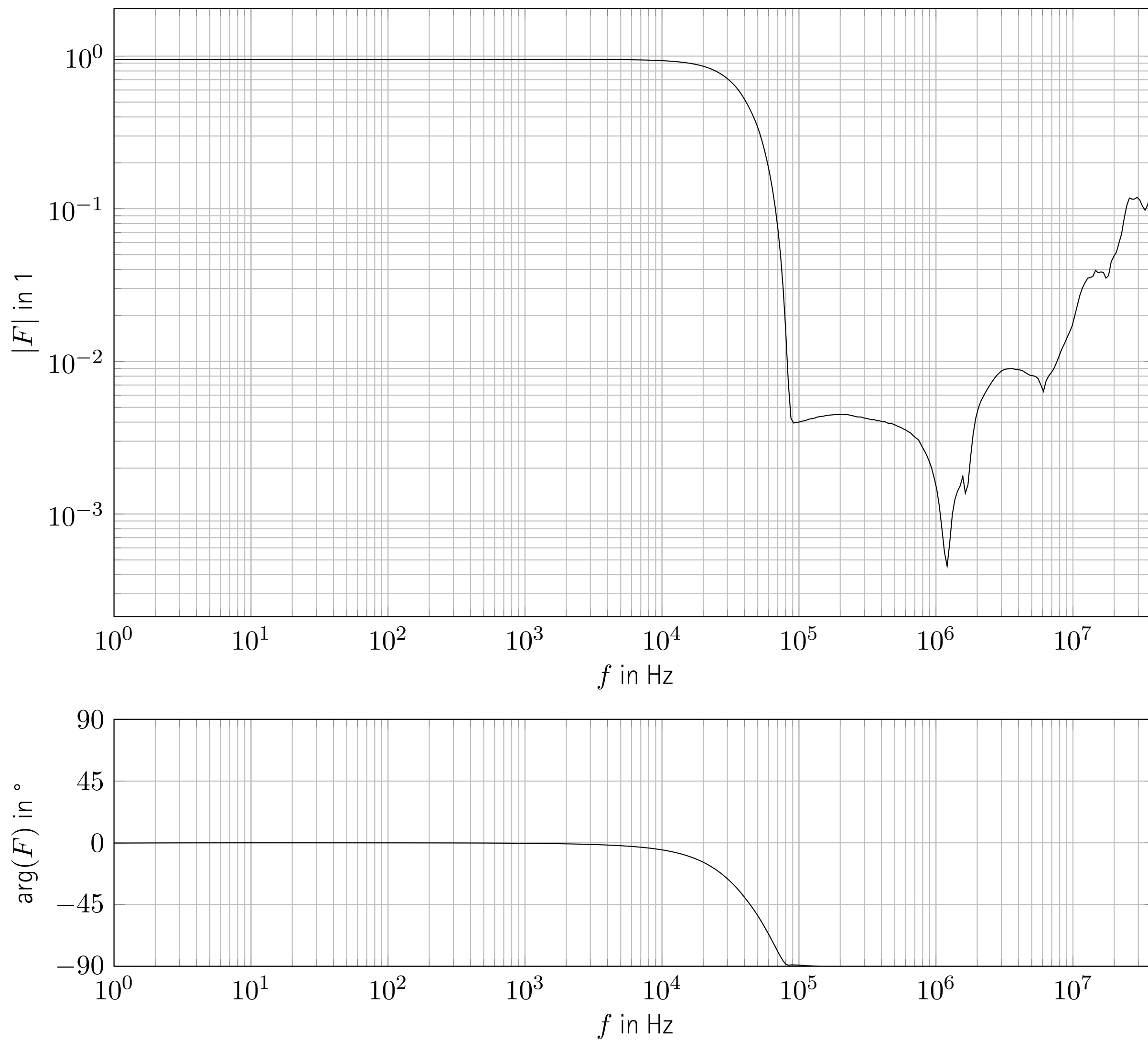


Abb. 2.21: Transferfunktion des Messwandlers bei der Spannungsmessung. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Verstärkungsfaktor auf 1 eingestellt. Nahezu ideales Verhalten bis ca. 10 kHz, anschließende Dämpfung.

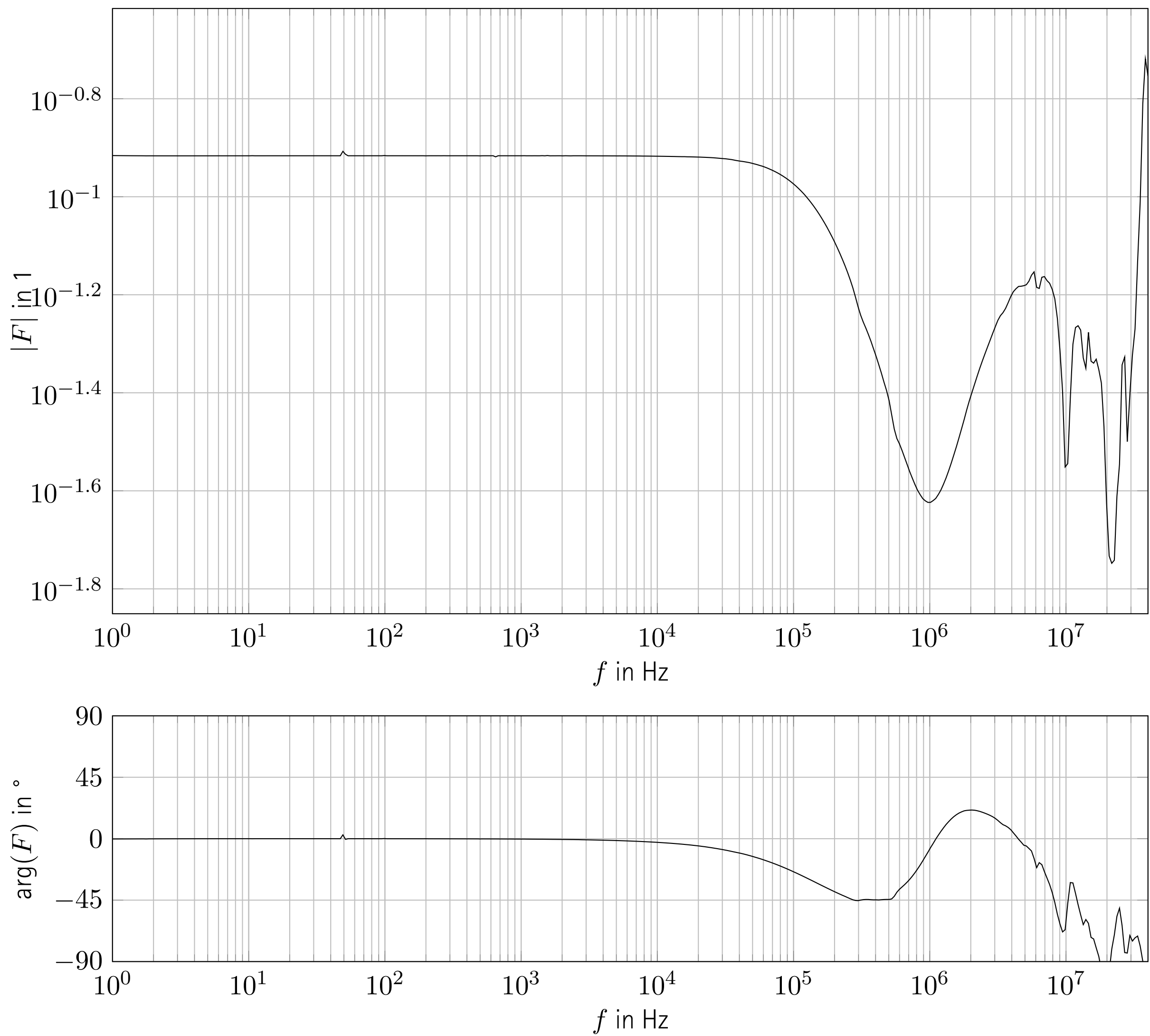


Abb. 2.22: Transferfunktion des Messwandlers bei der Strommessung. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Verstärkungsfaktor auf 5,8 · 1 Ω eingestellt, ohmsche Last von 47 Ω. Nahezu ideales Verhalten bis ca. 10 kHz, anschließend unerwünschte Dämpfung. Siehe Einleitung wegen statischem Faktor 2 (siehe Abschnitt 2.2, S. 57).

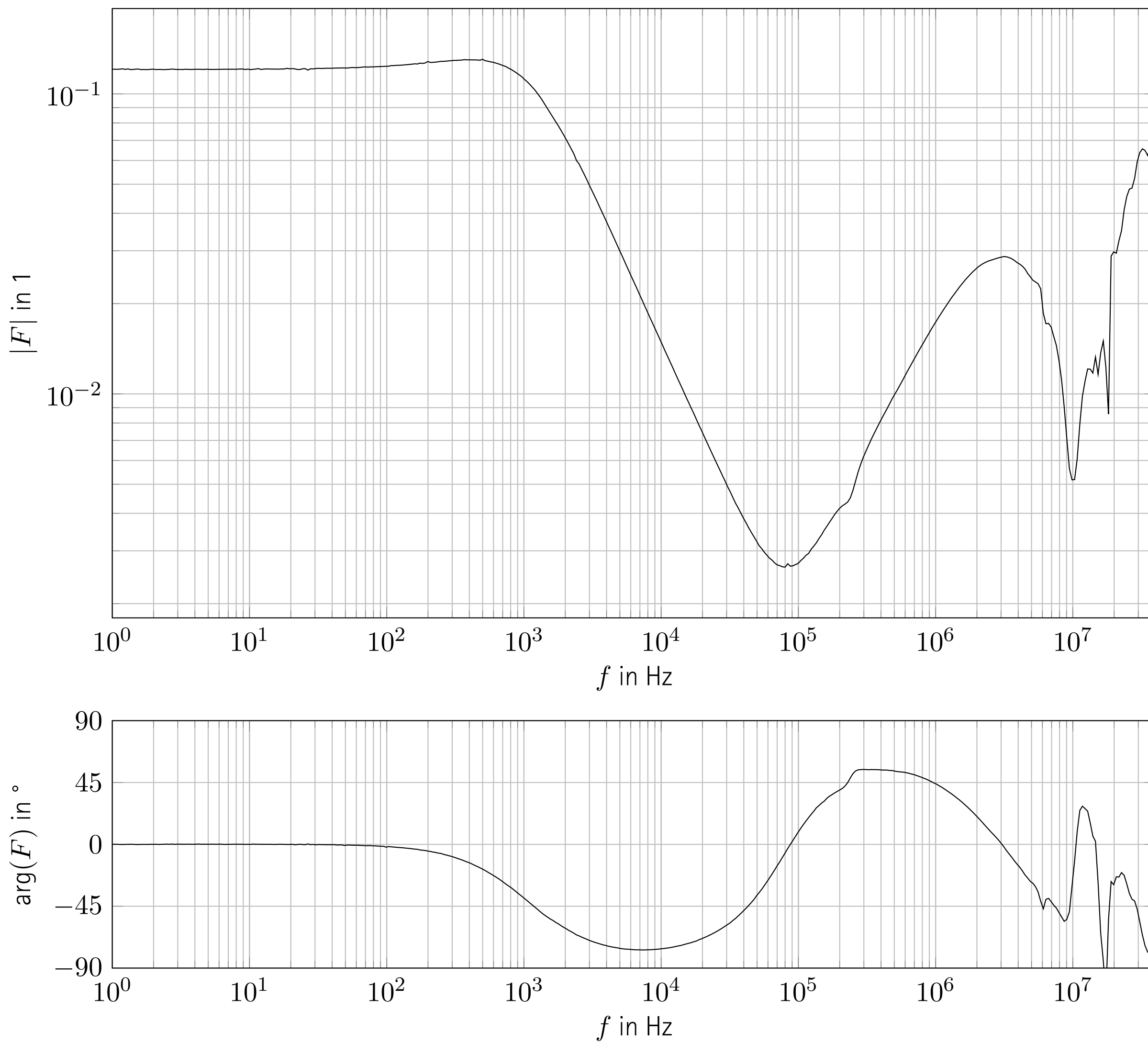


Abb. 2.23: Transferfunktion des Messwandlers bei der Strommessung. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Verstärkungsfaktor auf 1000 · 10 mΩ eingestellt, ohmsche Last von 47 Ω. Nahezu ideales Verhalten bis ca. 100 Hz, anschließend unerwünschte Dämpfung. Siehe Einleitung wegen statischem Faktor 2 (siehe Abschnitt 2.2, S. 57).

2.2.6 Addierer

Der Messwandler (siehe Abschnitt 2.2.5, S. 76) gibt zwei (Spannungs-)Signale aus, welche proportional zu der gemessenen Spannung und dem gemessenen Strom verlaufen. Dementsprechend können die Signale auch negative Werte annehmen, aber die Eingänge des „RedPitayas" umfassen lediglich einen positiven Bereich. Zu den Signalen muss entsprechend eine konstante Spannung von 1,9 V addiert werden, welche den Mittelpunkt des Eingangsbereichs des „RedPitayas" markiert.

Weil die bereits zuvor als Subtrahierer verwendete Schaltung invertierend wirkt (siehe Abschnitt 2.2.3, S. 69), muss lediglich ein geeignetes konstantes Spannungssignal addiert werden und sie kann als Addierer wiederverwendet werden. Der Spannungsteiler besteht hier aus einem festen Widerstand von 1 kΩ und einem einstellbaren Widerstand bis 5 kΩ, sodass das Signal um 0,83 V bis 5 V verschoben werden kann.

Wegen der starken Ähnlichkeit zu der Subtrahiererschaltung wird auf eine Charakterisierung verzichtet.

2.2.7 Über-/Unterspannungsschutz

Die langsamen analogen Eingänge des „RedPitaya" sind wahrscheinlich nicht über den gesamten Bereich spannungsfest, der von den vorgestellten Schaltungen verwendet wird, weshalb ein Über-/Unterspannungsschutz zwischen die Eingänge und die entwickelten Schaltungen (insbesondere die zwei Addierer, siehe Abschnitt 2.2.6, S. 83) gesetzt wird.

Schaltung

Der Hersteller des „RedPitayas" machte keine Angaben über die Spannungsfestigkeit der langsamen analogen Eingänge, weshalb ein Über-/Unterspannungsschutz nach „gesundem Ingenieurinnenverstand" entwickelt wird. Der Schaltplan ist in *Abb. 2.24* abgebildet. Auf vier Kanälen werden Überspannungen gegenüber dem Massepotential mit Zehnerdioden abgeleitet, Unterspannungen werden mit Germaniumdioden kurzgeschlossen.

Implementierung

Die fertige Umsetzung ist in *Abb. 2.25* dargestellt. Die Elemente sind auf geringe Ströme mit hohen Spitzen ausgelegt. Durch den Verzicht auf Filter im Frequenzbereich wird die Gefahr von Resonanzen minimiert.

Charakterisierung

Die Übertragungsfunktion ist in *Abb. 2.26* illustriert und zeigt eine statische Verstärkung von fast 1 bis zu einer Frequenz von ca. 1 MHz. Anschließend tritt einige Dämpfung auf, insbesondere ab ca. 20 MHz. Weil der geringe

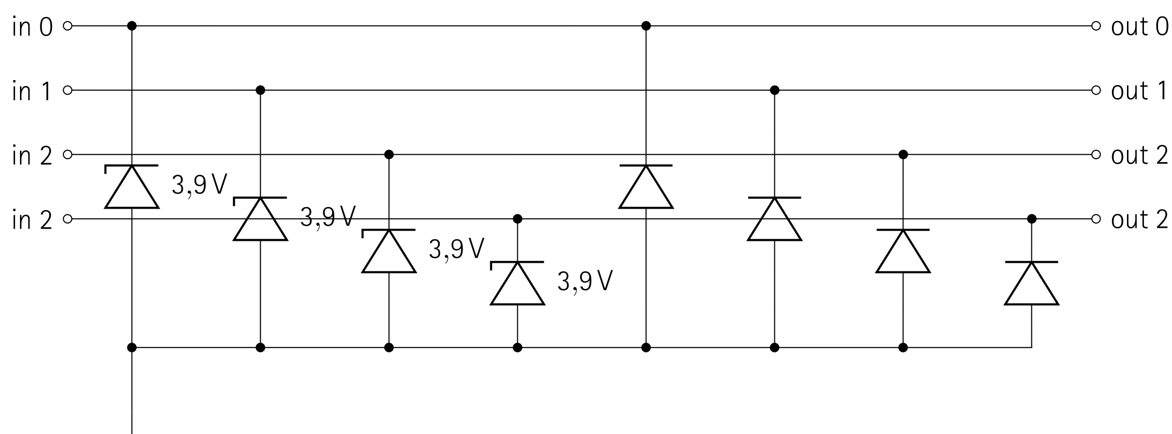


Abb. 2.24: Schaltplan des Über-/Unterspannungsschutzes. Vier Signalleitungen (oben) durch Zehnerdioden (links) gegen Überspannung und durch Germaniumdioden (rechts) gegen Unterspannung, jeweils gegenüber dem Massepotential, geschützt.

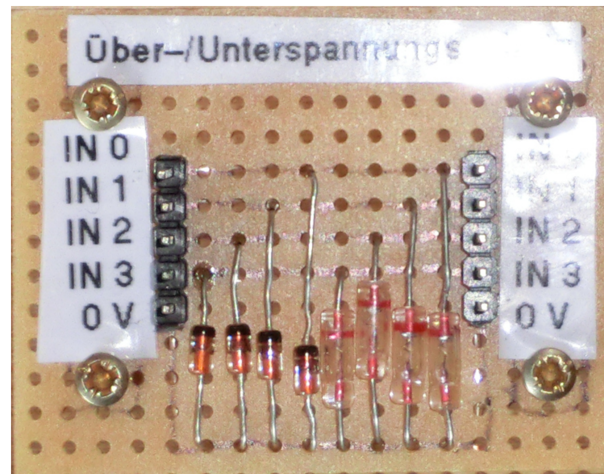


Abb. 2.25: Fotografie des Über-/Unterspannungsschutzes. Kontakte (links und rechts), Zehnerdioden (links) und Germaniumdioden (rechts).

Fehler bei der statischen Verstärkung bis zu sehr hohen Frequenzen konstant ist, kann er durch eine lineare Kalibrierung kompensiert und das Frequenzverhalten für das Nutzsignal als ideal angenommen werden.

Es sind keine geeigneten Messgeräte verfügbar, um die Kapazität der Anordnung zu messen. Eine Impedanzmessung mit dem Bode100 zeigt bei einer konstanten Leistung von 1 mW und offenen Klemmen eine geringe statische Impedanz von 1 kHz und ein kapazitives Verhalten ab ca. 1 kHz. Dabei handelt es sich vermutlich um ein Messartefakt, welches durch den Betrieb mit einer konstanten Leistung zu erklären ist, wodurch die Dioden im nichtlinearen Bereich angesteuert werden. Vor dem Hintergrund der sehr guten Übertragungsfunktion soll auf dieses Phänomen nicht weiter eingegangen werden.

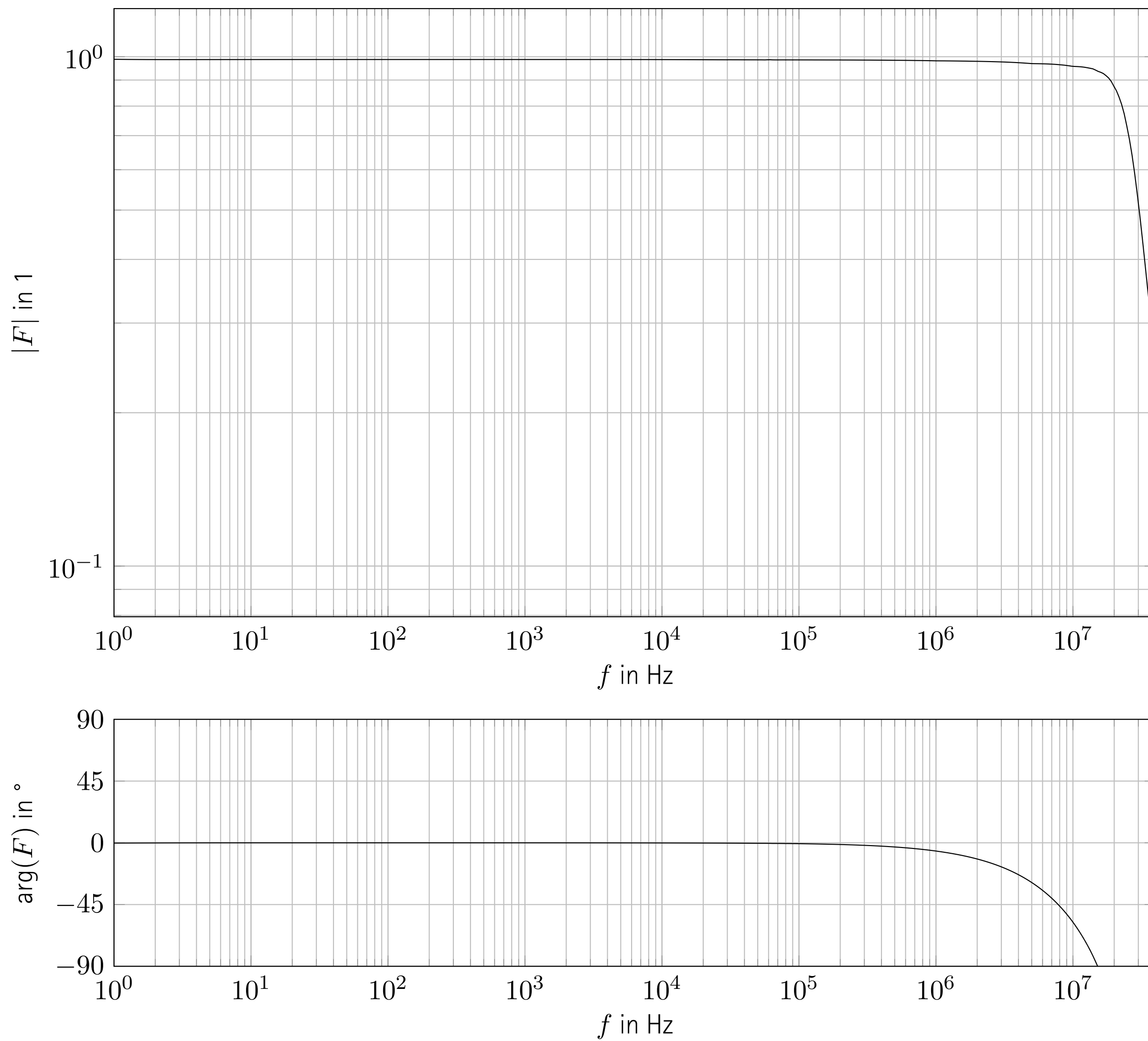


Abb. 2.26: Transferfunktion des Über-/Unterspannungsschutzes. 401 Messpunkte, aufgenommen mit Bode100 (Transferfunktionsmessung, Empfängerbandbreite: 1 Hz). Belastung mit 50 Ω, Beaufschlagung mit Kleinsignal (Leistung von ca. 1 mW). Nahezu ideales Verhalten bis ca. 1 MHz, anschließende Dämpfung.

2.3 Programm

Der Vorteil des gewählten Aufbaus gegenüber der zuvor präsentierten Alternative (siehe Abschnitt 2.2, S. 57) wird wesentlich mithilfe der Programmierbarkeit des „RedPitayas“ umgesetzt. Dieser Abschnitt behandelt die gesamte Programmierung, wobei auf Übersichtlichkeit geachtet ist. Eine detaillierte, mit Doxygen erstellte Dokumentation ist der Arbeit angehängt (siehe Kapitel 11, S. 249).

Zunächst wird eine Softwarebibliothek entwickelt, mithilfe derer der Hardwarezugriff einfach und einheitlich gestaltet ist. Sie selbst und auch die damit erstellten Konfigurationen[8] sollen eine hohe Wiederverwendbarkeit aufweisen.

Anschließend wird die Bibliothek genutzt, um damit Kommandozeilenwerkzeuge zu entwickeln. Diese bieten den Anwendenden Zugang zu der Funktionalität der Bibliothek und implementieren eigene Anwendungsfälle.

Schließlich wird darauf eingegangen, wie der geschriebene Programmcode in ausführbare Dateien übersetzt wird. Das ist insbesondere deshalb wichtig, weil für die Entwicklung keine Integrierte Entwicklungsumgebung, engl. Integrated Development Environment (IDE) zur Verfügung steht, welche diese Aufgabe normalerweise übernehmen würde.

2.3.1 Hardwarezugriff

Obwohl die API des Herstellers bereits Funktionen zum Zugriff auf die analogen Ein- und Ausgänge bereitstellt, reichen diese nicht aus, um eine produktive Schnittstelle für den Hardwarezugriff bereitzustellen, weil sie lediglich die Spannung am Port auslesen bzw. festlegen und das noch nicht einmal korrekt.[9] In diesem Abschnitt werden Klassen vorgestellt, die den Zugriff auf die o. g. Ports ermöglichen und außerdem eine Kalibrierung anwenden und Einstellungen dauerhaft im Dateisystem des „RedPitaya“ speichern können. Die bereitgestellten Schnittstellen sind auf Wiederverwendbarkeit optimiert.

Zusätzlich sind zwei Klassen für den Zugriff auf ein über das LAN angeschlossenes Oszilloskop entwickelt. Die Nutzung dieser Klassen wird bei der Vorstellung der „improvisierten Vierpolcharakterisierung“ erläutert (siehe Abschnitt 2.3.2, S. 94).

Die Klasse *TInterface*

Zunächst wird eine Klasse benötigt, welche die allgemeinen Funktionen eines Interfaces implementiert, bspw. die eines analogen Eingangs oder einen Ausgangs, prinzipiell aber auch die von abstrakteren oder komplizierteren Interfaces. Die Klasse *TInterface* beinhaltet lediglich einen Namen für jedes Objekt und gibt die Aufgabe der Definition weiterer Eigenschaften an spezifischere, abgeleitete Klassen weiter, wie in *Abb. 2.27* abgedruckt.

[8] Jedes Interface kann konfiguriert und gespeichert werden, wie im entsprechenden Abschnitt vertieft.

[9] Die Werte der von der API bereitgestellten Funktionen weisen Fehler von mehreren Prozent auf. Aufgrund der Reproduzierbarkeit der nicht statistisch verteilten Fehler ist eine Kalibrierung möglich.

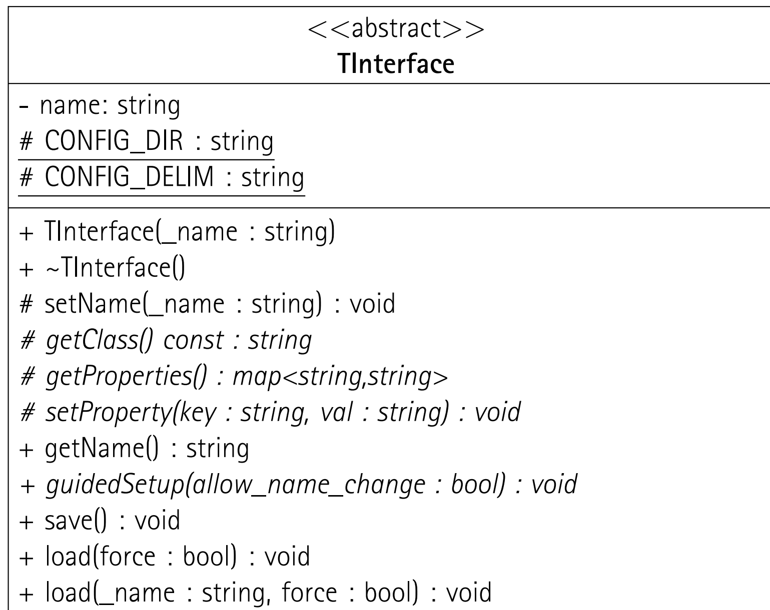


Abb. 2.27: Klassendiagramm von *TInterface*. Name des Objekts als einzige Eigenschaft, weitere werden von abgeleiteten Klassen definiert. Statische Konstanten , Methoden und abstrakte Funktionen zur Implementierung von Konfigurationsdateien. Virtueller Destruktor für polymorpthe Klassen. Rein virtuelle Methode *guidedSetup* zur einfachen Nutzerinteraktion.

Sie implementiert insbesondere die Möglichkeit, die Eigenschaften einer Klasse, wie bspw. Hardwareport oder Kalibrierung, permanent im Dateisystem zu speichern. Für jedes Objekt ist eine eigene Konfigurationsdatei vorgesehen, deren Pfad sich wie folgt zusammensetzt:

```
TInterface::CONFIG_DIR+"s"+"/"+TInterface()::getClass()+"/"+TInterface::getName()+".conf"
```

Daraus resultiert bspw. Der Pfad:

```
/home/ilka/OrthoMeasure/conf/inputs/u1-nogain-calibrated.conf
```

Für die einwandfreie Funktion der Konfigurationsdateien müssen die rein virtuellen Funktionen *getClass*, *getProperties* und *setProperty* von in den abgeleiteten Klassen implementiert sein. Außerdem müssen die Konfigurationsdateien schreibar sein und dürfen nicht extern mit ungültigen Inhalten gefüllt werden.

Die Klasse *TOutput*

Der Zugriff auf die langsamen und die schnellen analogen Ausgänge des „RedPitayas" wird durch die Klasse *TOutput* definiert, welche von *TInterface* abgeleitet ist und entsprechend alle rein virtuellen Methoden implementiert. Sie legt in ihren Eigenschaften den Port fest und speichert eine Kalibrierung. Der direkte Hardwarezugriff ist mit den privaten Methoden *write* und *getCurrentPwm* möglich. Der kalibrierte Zugriff auf den analogen Ausgang der Hardware geschieht durch die öffentliche Funktion *putValue*. Gelegentlich kann ein direkter Hardwarezugriff benötigt werden, sodass die öffentliche Methode *putRaw* insbesondere für Debugging-Zwecke

behalten wird. Sie bietet außerdem die Möglichkeit, ein Signal am Ausgang zu erzeugen, indem *putSignal* aufgerufen wird. Für die langsamen Ausgänge gibt die Methode *isSelfRefreshing* dann den Wert *false* zurück und die Methode *refreshSignal* muss andauernd aufgerufen werden. Die schnellen Ausgänge sind hardwareseitig programmierbar und geben das Signal selbstständig aus.

Der Konstruktor versucht, die Einstellungen des Objektes aus dem Dateisystem zu laden. Deshalb muss der Name bereits dem Konstruktor bekannt sein und kann nicht später gesetzt werden. Weil dieses Verhalten nicht deaktivierbar sein soll, muss der Konstruktor außerdem erfahren, ob bei einem Fehlschlag beim Laden der Konfiguration ein Fehler geworfen oder lediglich eine Warnung ausgegeben werden soll.

Später entwickelte Programme, welche auf diese Klasse zugreifen, werden häufig zeitliche Schwingungen mit bestimmten Formen am Ausgang erzeugen, indem sie schnell aufeinanderfolgend die Methode *putValue* aufrufen. Weil einige Funktionsformen üblich sind und auch eine übliche textuale Repräsentation aufweisen, werden unter der Klasse *TOutput* außerdem eine Aufzählungsdatentyp *TWaveform*[10] und die statischen Methoden *waveformToString* und *stringToWaveform* deklariert. Die Funktionen sind in der selben Übersetzungseinheit definiert.

Die Klasse *TInput*

Der Zugriff auf die langsamen analogen Eingänge des „RedPitayas" wird durch die Klasse *TInput* definiert, welche von *TInterface* abgeleitet ist und entsprechend alle rein virtuellen Methoden implementiert. Sie legt in ihren Eigenschaften den Port fest und speichert eine Kalibrierung. Der direkte Hardwarezugriff ist mit der privaten Methode *read* möglich, der kalibrierte Zugriff auf den analogen Eingang der Hardware geschieht durch die öffentliche Funktion *measureValue*. Der Zugriff auf den „rohen" Messwert ist für externe Funktionen nicht möglich.

Der Konstruktor ist identisch zu dem der Klasse *TOutput*.

Die Klasse *TImpedanceMeasureSetup*

Die Messungen, die später durchgeführt werden sollen (siehe Abschnitt 2.3.2, S. 94, siehe Abschnitt 5.3, S. 152), beinhalten typischerweise die Messung von (mehreren) Zweipolen. Dafür wird ein Interface benötigt, das von der Klasse *TImpedanceMeasureSetup* bereitgestellt wird:

1. Eingang zur Messung der am Zweipol anliegenden Spannung
2. Eingang zur Messung des durch den Zweipol fließenden Stroms
3. Ausgang zur Manipulation des elektrischen Zustandes, bspw. mithilfe einer steuerbaren Spannungsquelle

[10] Obwohl es sich um Signalformen und damit um die Formen von Funktionen handelt, ist die Bezeichnung „waveform" ist gewählt, weil diese im Englischen üblich ist.

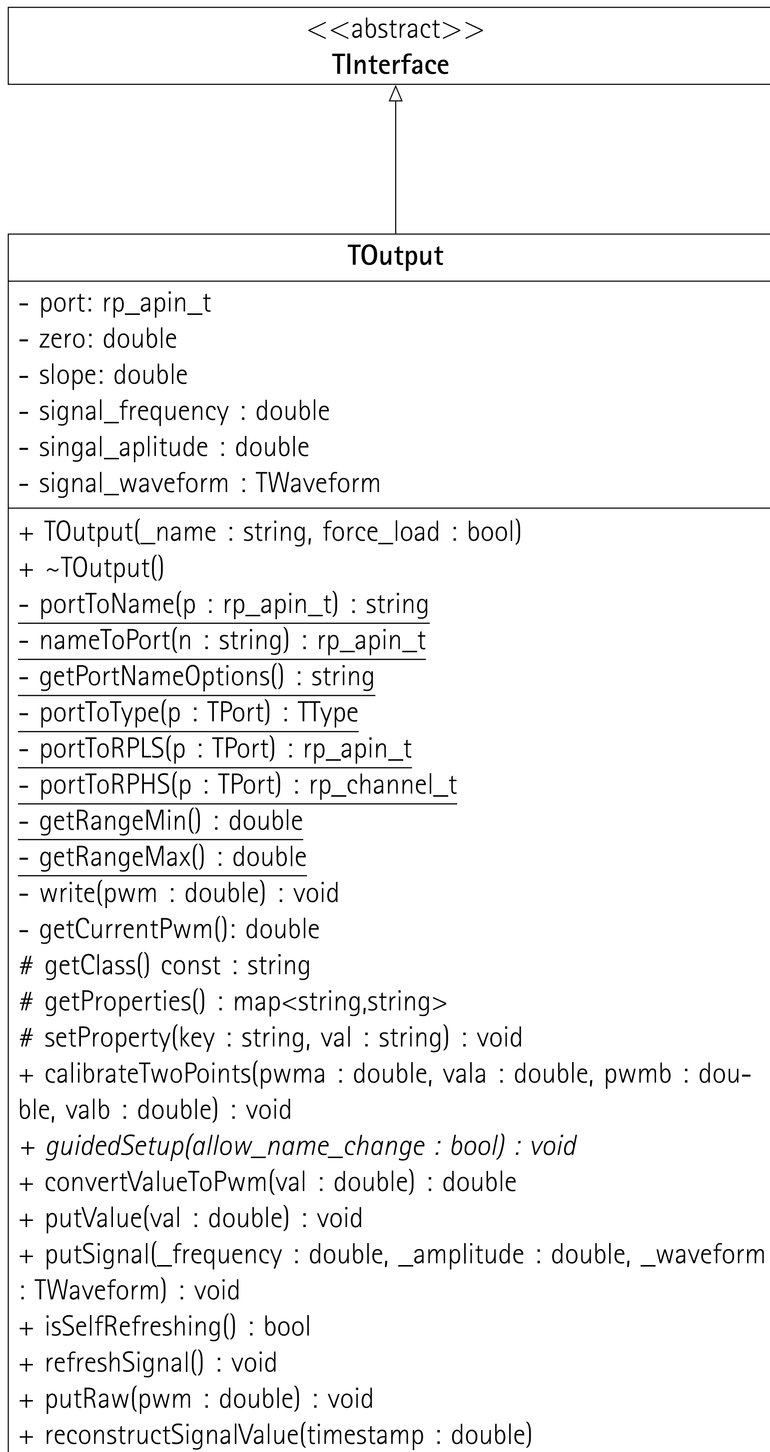


Abb. 2.28: Klassendiagramm von *TOutput*. Eigenschaften legen Hardwareport , Kalibrierung und aktuelles Signal fest. Konstruktor definiert Namen und lädt Konfiguration, virtueller Destruktor für polymorphe Klasse. Statische Funktionen zur menschenlesbaren Darstellung des Ports und interne Konvertierungen, *write*, *putSignal*, etc. für Hardwarezugriff, Implementierung der virtuellen Funktionen von *TInterface*, Methoden zur Kalibrierung und Einstellung. Zugriffsmethode für Hardwareausgangsport.

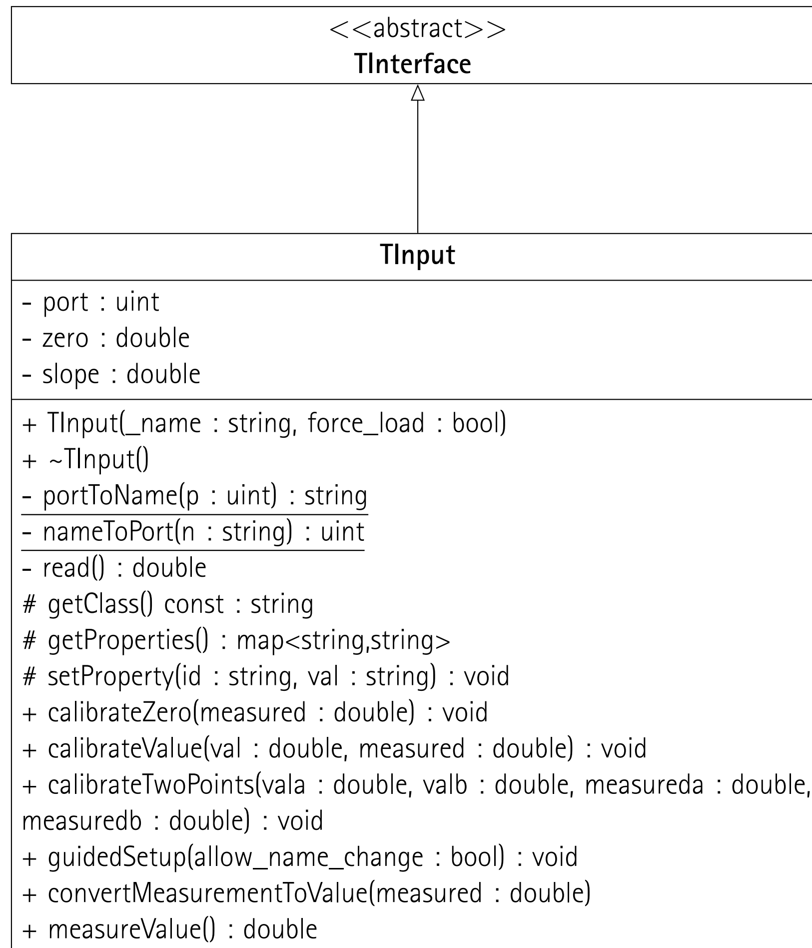


Abb. 2.29: Klassendiagramm von *TInput*. Eigenschaften zur Speicherung des Ports und der Kalibrierung. Konstruktor definiert Namen und lädt Konfiguration, virtueller Destruktor für polymorphe Klasse. Statische Funktionen zur menschenlesbaren Darstellung des Ports, *read* für Hardwarezugriff, Implementierung der virtuellen Funktionen von *TInterface*, Methoden zur Kalibrierung und Einstellung. Zugrifssmethode für Hardwareeingangsport.

Die Klasse beinhaltet dafür Zeiger auf die Klassen *TInput* und *TOutput*. Sie ist von *TInterface* abgeleitet und implementiert entsprechend alle rein virtuellen Methoden. Sie enthält zusätzlich lediglich Zugriffsmethoden und dient entsprechend nur der internen Struktur der Interfaces, ohne selbst Funktionalität hinzuzufügen. Für weiterführende Arbeiten ist es denkbar, Algorithmen bereitzustellen, wie bspw. eine Impedanzspektrometrie.

Die Klassen *TSocket* und *TScope*

Später wird ein Werkzeug zur „improvisierten Vierpolcharakterisierung" implementiert, welches über das LAN auf ein Oszilloskop zugreift. Die Klasse *TSocket* stellt eine objektorientierte Schnittstelle für den Zugriff auf eine Socket über das Übertragungssteuerungsprotokoll, engl. Transmission Control Protocol (TCP) bereit. Die

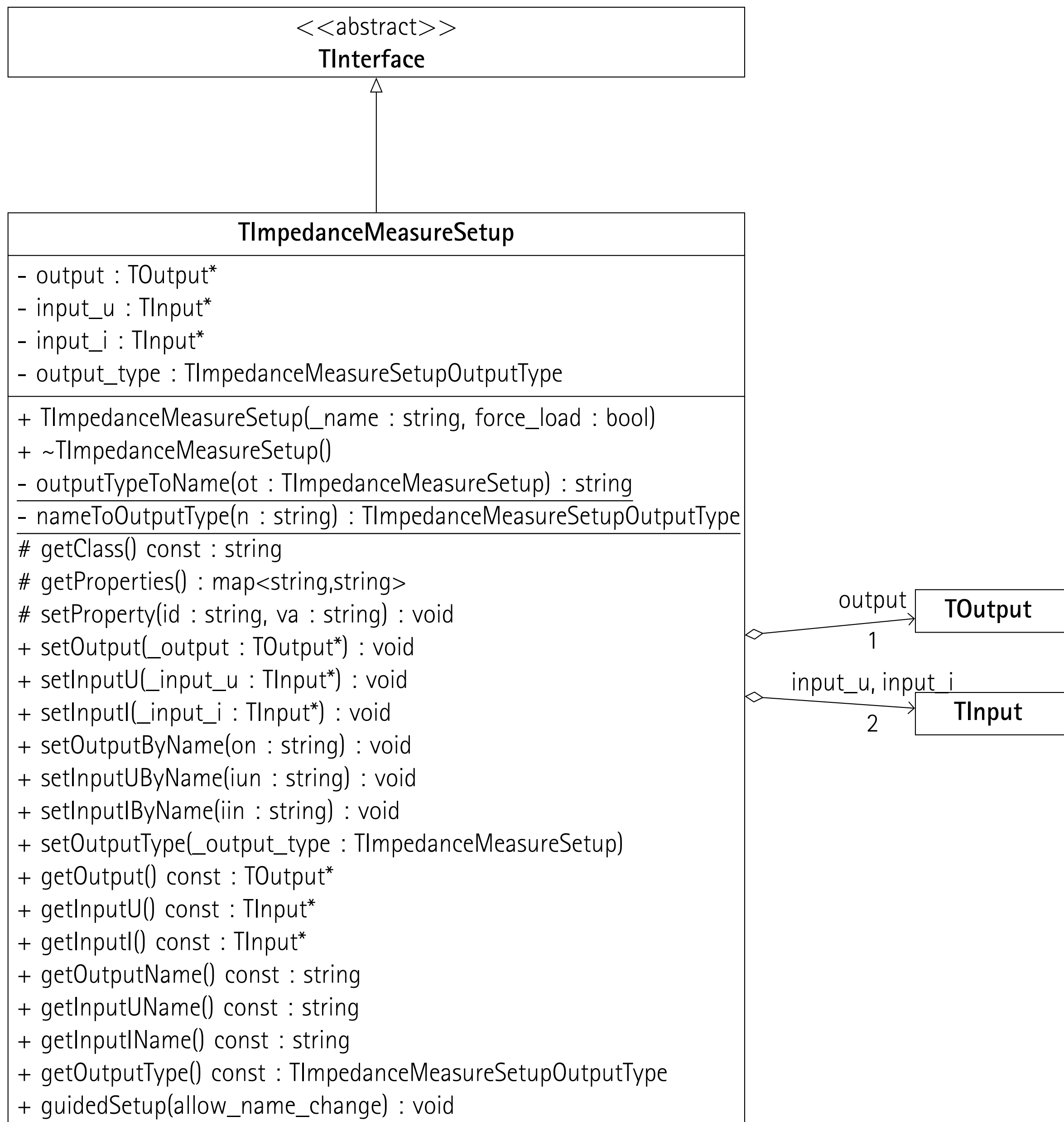


Abb. 2.30: Klassendiagramm von *TImpedanceMeasureSetup*. Zusammenfassung eines Ausgangs und zweier Eingänge als Eigenschaften, Ausgangstyp. Konstruktor definiert Namen und lädt Konfiguration, virtueller Destruktor zerstört Interfaces. Statische Funktionen zur menschenlesbaren Darstellung des Ausgangstyps. Implementierung der rein virtuellen Methoden von *TInterface*, get- und set-Methoden.

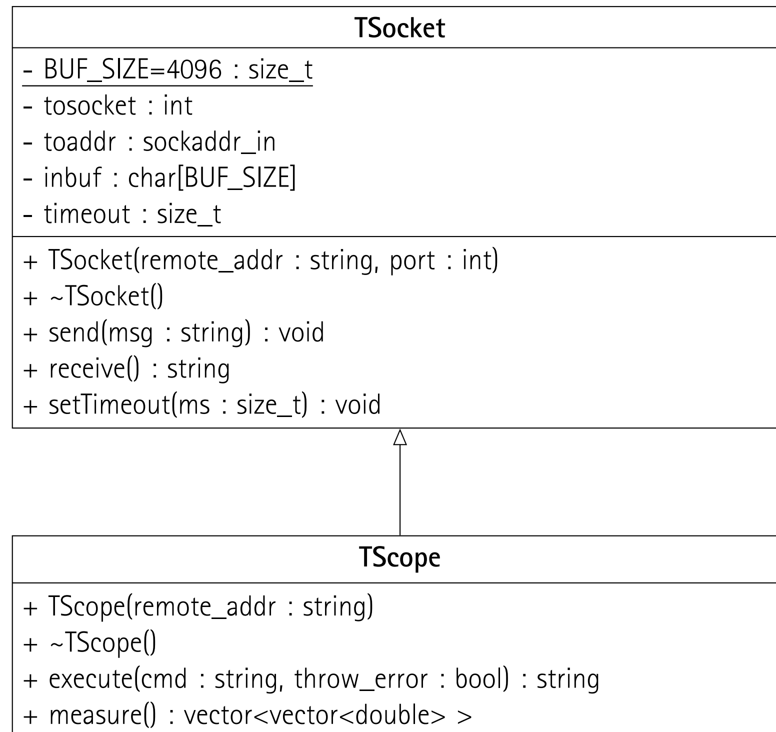


Abb. 2.31: Klassendiagramm von *TSocket* und *TScope*. Erstere greift auf Linux-eigene Bibliothek zu und stellt minimale Funktionalität bereit. Letztere beherrscht herstellereigenes Protokoll und führt Befehle aus. Komplexe Funktion *TScope::measure* wartet auf Auslösung der Messung und gibt Zeiten und Messwerte zurück.

Klasse *TScope* erbt von ihr und ermöglicht die Aufnahme einer getriggerten Messung von einem Oszilloskop aus der Modellserie „TBS 2000“ des Herstellers „Tektronix“ über das LAN. Das Klassendiagramm ist in *Abb. 2.31* präsentiert.

Die Klasse *TSocket* greift intern auf die Linux-eigene Bibliothek für socketbasierte Netzwerkverbindungen zu. Der Code ist zu einem großen Teil dem Beispiel in [32] entnommen. Eine Verbindung wird sofort beim Aufruf des Konstruktors etabliert und erst im Destruktor getrennt. Entsprechend muss der aufrufende Code die Lebensdauer des Objektes sorgfältig festlegen. Die Methoden *send*, *receive* und *setTimeout* sind selbsterklärend. Es sei angemerkt, dass die Funktion „receive“ im Falle eines leeren Eingangspuffers nicht auf eingehende Daten wartet, sondern eine leere Zeichenkette zurückgibt. Die Klasse ist nur mit Zeichen des Amerikanischen Standard-Codes für den Informationsaustausch, engl. American Standard Code for Information Interchange (ASCII) kompatibel und weist andernfalls ein undefiniertes Verhalten auf.

Die Klasse *TScope* erbt direkt und geschützt von *TSocket* und trifft bereits im Konstruktor einige Einstellungen. Der Konstruktor legt die Timeoutdauer für die Netzwerkverbindung fest und teilt die selbe Timeoutdauer dem Oszilloskop mit. Er erwartet außerdem die Eröffnungsnachricht des Oszilloskops und wirf einen Fehler, falls diese nicht ankommt oder nicht den erwarteten Inhalt aufweist. Die Methode *execute* schickt einen Befehl an das

Oszilloskop und wartet auf eine Antwort. Die Antwort wird bereinigt[11] und zurückgegeben. Falls die bereinigte Antwort mindstens ein Zeichen beinhaltet und der Parameter *throw_error* auf *wahr* gesetzt ist, handelt es sich wahrscheinlich um eine Fehlermeldung vom Oszilloskop, die als Ausnahme geworfen wird. Die Methode *measure* stellt den Messbetrieb des Oszilloskops auf „Single“, sodass nach einer Auslösung nur ein Puffer voll Daten messen wird. Sobald die Messung „getriggert“ ist, werden die Zeitpunkte und Messdaten ausgelesen und gespeichert. Es kommt häufig vor, dass die Messbereiche des Oszilloskopkanäle nicht für die Amplituden der zu messenden Signale geeignet sind. Falls ein Messbereich zu groß oder zu klein eingestellt ist, ruft die Methode *measure* automatisch die Methode *execute* mit geeigneten Parametern auf, um den Messbereich des betroffenen Kanals zu ändern. Die Messung wird anschließend wiederholt. Der Rückgabewert der Methode besteht aus fünf Listen gleicher Länge, welche eben diese Daten enthalten.

2.3.2 Werkzeuge

Sowohl für die praktische Nutzung als auch zum Debugging werden alleinstehende Programme benötigt, welche auf die Funktionalität der Softwarebibliotheken zugreifen. Im Rahmen dieser Arbeit werden ein Kalibrierungswerkzeug, ein Funktionsgenerator, ein Messgerät, ein Zweipolcharakterisierer und ein intelligenter Vierpolcharakterisierer entwickelt. Letzterer wird später für die eigentlichen Messungen bei der Beobachtung von orthogonaler Magnetisierung verwendet, die anderen Programme sind als Hilfswerkzeuge nützlich.

Aus Gründen der Übersichtlichkeit wird darauf verzichtet, den Quellcode der Software vorzustellen. Es handelt sich im wesentlich einfachen, prozeduralen Code, welcher in den jeweiligen Quelldateien hinreichend kommentiert ist (siehe Kapitel 10, S. 173).

Kalibrierung

Für die Nutzung der zuvor entwickelten Softwarebibliothek ist das Vorhandensein von geeigneten Konfigurationen im Dateisystem notwendig. Weil jede Schnittstellenklasse (*TOutput*, *TInput* und *TImpedanceMeasureSetup*) bereits die Methode *guidedSetup* implementieren muss und damit Konfigurationen erstellt werden können (siehe Abschnitt 2.3.1, S. 87), soll ein einfaches Kommandozeilenprogramm lediglich diese aufrufen. Das Kalibrierungswerkzeug trägt den Namen *calib* und der dazugehörige Code ist in der Datei *calib.cpp* abgelegt. Wenn die ausführbare Datei ohne Parameter aufgerufen wird, gibt sie eine selbst erklärende Hilfe aus:

```
usage: calib --help
usage: calib [OPTIONS] CLASS1 NAME1 [CLASS2 NAME2 [...] ]

options:
 -h --help                  prints this message
 -v --verbose               verbose output

classes:
```

[11] Bpsw. endet jede Antwort mit einem Zeilenumbruch, gefolgt von einem Leerzeichen und einer schließenden Spitzen Klammer.

```
  -i --input                    analog input pin
10  -o --output                 analog output pin
  -z --impedancemeasuresetup    impedance measure setup

names:
 * run ./lscalib to list available names sorted by class
15 * OR specify an existing name to edit that interface
 * OR specify an non-existent name to create that interface

For full documentation, see master thesis by Ilka Schulz or contact:
Leibniz Universität Hannover
20 Institut für Antriebssystem und Leistungselektronik
z. Hd. Ilka Schulz
Welfengarten 1
30167 Hannover
Germany
```

Der kurze Code ist der Arbeit angehängt (siehe Code, S. 184).

Zusätzlich wird ein Bash-Skript mit dem Dateinamen *lscalib* geschrieben, welches einen Überblick über die aktuell installierten Konfigurationen gibt:

```
#!/bin/bash

echo "available configurations: "
cd conf && tree --noreport *
```

Die Ausgabe enthält UTF-8-kodierte Zeichen, sodass an dieser Stelle aus Kompatibilitätsgründen auf ein veranschaulichendes Beispiel verzichtet werden muss.

Weil sämtliche Programme mit root-Rechten ausgeführt werden müssen, um Zugriff auf die API des „RedPitayas“ zu erhalten, gehören die ggf. erzeugten Konfigurationsdateien ebenfalls dem User „root“. Insbesondere zu Debugging-Zwecken ist es praktisch, die Konfigurationsdateien dennoch ohne erhöhte Rechte bearbeiten zu können. Ein Bash-Skript mit dem Dateinamen *ownconf* kann von dem Account aus aufgerufen werden, welchem wieder Zugriff auf die Dateien gegeben werden soll, um die Eigentümerschaft an diesen zurückzugeben:

```
#!/bin/bash

sudo chown "$UID:$UID" conf -R
```

Funktionsgenerator

Der Funktionsgenerator ist ein einfaches Werkzeug, welches insbesondere bei der Fehlersuche an den gelöteten Platinen hilft (siehe Abschnitt 2.2, S. 57). Die ausführbare Datei *fungen* benötigt folgende Kommandozeilenparameter, um ein zeitliches Signal an einen der Ausgänge des „RedPitaya“ zu generieren:

- Frequenz des Signals, bspw. 100 Hz
- Amplitude des Signals, bspw. 1 V

- Form des Signals, bspw. sinus-förmig
- Ausgangsport, entweder *TOutput* oder *TImpedanceMeasureSetup*
- Sicherheitswartezeit, bevor das Signal generiert wird, bspw. 5 s
- Betriebsdauer, bspw. 10 s
- Standardwert, der ausgegeben wird, bevor und nachdem das Signal generiert wird

Wenn das Programm ohne Parameter ausgeführt wird, druckt es eine selbst erklärende Hilfe in die Konsole:

```
usage: fungen --help
usage: fungen OPTIONS

options:
 -h --help                    prints this message
 -v --verbose                 verbose output
 -f --frequency  FREQ         set frequency to FREQ hertz. 0 or 'DC' for DC
 -a --amplitude  AMP          set amplitude to AMP volts or amps
 -w --waveform   WAVEFORM     will create the waveform WAVEFORM (see below for valid values,
     default:sin)
 -o --output     OUTNAME      use output with name OUTNAME
 -z              IMPNAME      use output specified in impedance measure setup with name IMPNAME
 -s --safety     TIME         wait for TIME seconds before enabling output (default: 5)
 -T --duration   TIME         generate output for TIME seconds (default: 1)
 -d --default    VALUE        set output to constant (calibrated physical) VALUE when ideling,
     e.g. after ending (default: 0)
 -da --default-analog VALUE   set output to constant (PWM) VALUE when ideling, e.g. after
     ending (default: using --default)

waveforms (case-insensitive):
 dc                           direct current
 sin                          sine wave
 cos                          cosine wave
 square                       square

For full documentation, see master thesis by Ilka Schulz or contact:
Leibniz Universität Hannover
Institut für Antriebssystem und Leistungselektronik
z. Hd. Ilka Schulz
Welfengarten 1
30167 Hannover
Germany
```

Selbstverständlich greift das Programm auf die Kalibrierung des Ausgangs zurück. Lediglich falls die Option *--default-analog* gewählt ist, wird für den Standardwert ein „roher" Wert akzeptiert und die Methode *TOutput::putRaw* aufgerufen.

Zur Charakterisierung des Programms wird beispielhaft eine Sinusschwingung mit einer Frequenz von 100 Hz und einer Amplitude von 1 V am zuvor erwähnten Ausgang *fullrange0* generiert. Der zugehörige Befehl lautet:

```
make fungen && ./fungen -f 100 -a 1 -w sin -o fullrange0 -d 0 && ./ownconf
```

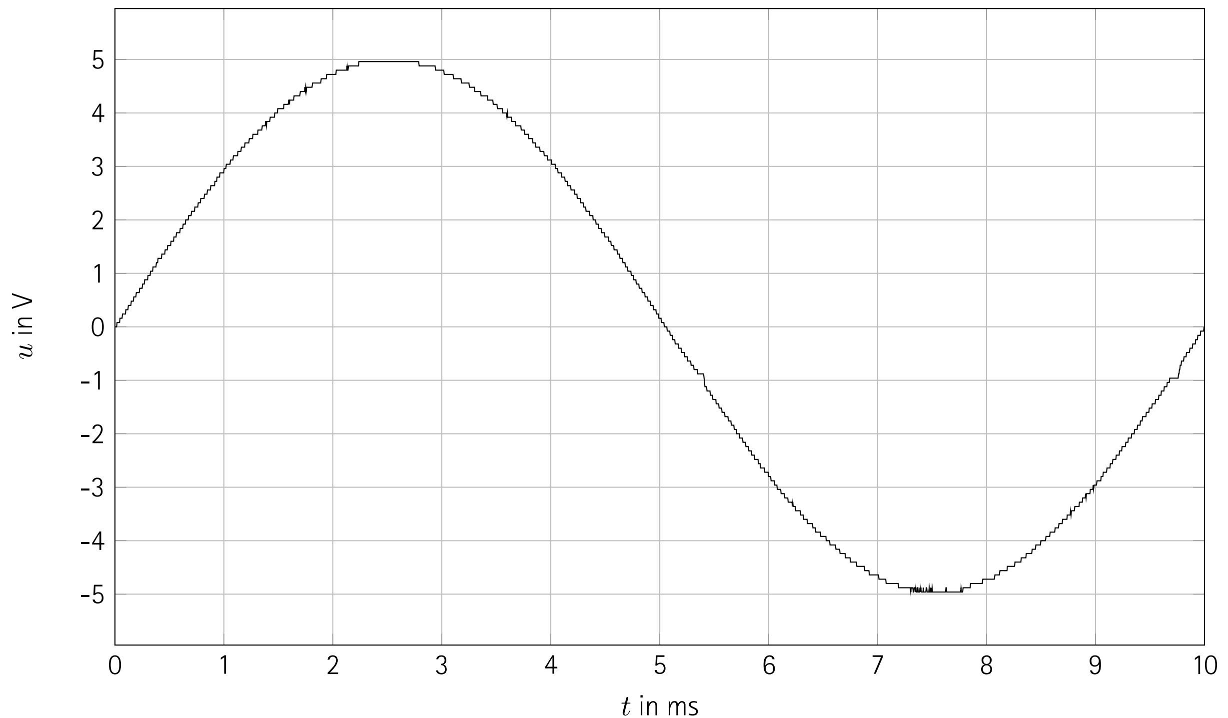
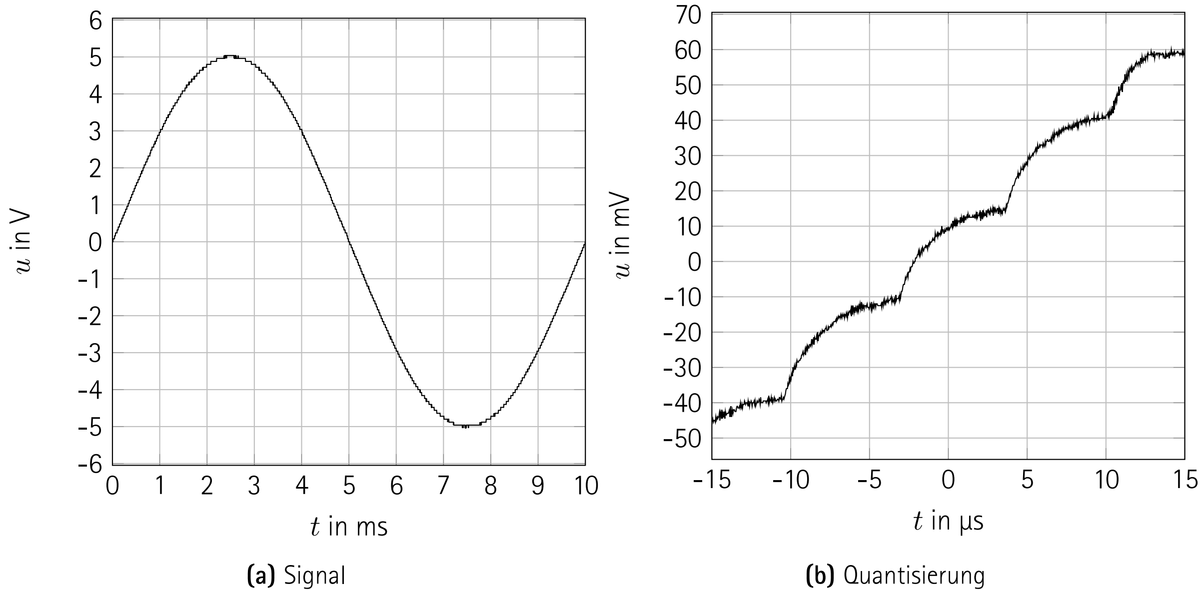


Abb. 2.32: Sinusschwingung, mit langsamem Ausgang an 47 Ω Last generiert. Frequenz $f = 100$ Hz, Amplitude $\hat{u} = 5$ V, sinusförmiges Signal. Kalibrierter Ausgang auf Kanal *RP_AOUT1* und angeschlossenem Tiefpass, Subtrahierer und sekundärem großem Impedanzwandler mit Verstärkungsfaktor 11. Gut passende Kalibrierung, zeitlich verzögerter Nulldurchgang des Signals bei der fallenden Flanke. Geringe Signalauflösung des Oszilloskops deutlich sichtbar. Signalrauschen bei negativer Spitze. Signalfehler bei -1 V.

Die mit einem Oszilloskop gemessene Schwingung ist in *Abb. 2.32* gezeigt. Dafür wird der zweite langsame analoge Ausgang des „RedPitayas", *RP_AOUT1*, genutzt. Obwohl der Funktionsgenerator die angestrebte Sinusfunktion erzeugt und die Amplitude und Frequenz einhält, weist diese einige Mängel auf:

- Die Signalauflösung des Oszilloskops ist so gering, dass viele Stufen in der Funktion sichtbar sind. Weitere Messungen ergeben, dass der Signalausgang des „RedPitayas" eine höhere Signalauflösung erreicht.
- Das Signal weist ein geringes Rauschen auf.
- Der Nulldurchgang der fallenden Flanke ist zeitlich verzögert.
- Bei ca. -1 V ist eine „Ecke" in der Signalform beobachtbar, die typisch für die verbauten OPV, das Modell LM324, ist.

Die Fehler scheinen zu einem Teil durch den „langsamen" Ausgang des „RedPitayas" verursacht zu sein, welches im Rahmen dieser Arbeit wegen der Vorgabe durch das betreuende Institut nicht ausgetauscht werden darf. Um die Mängel zu beheben, wird einer der „schnellen" analogen Ausgänge verwendet und direkt an den großen Impedanzwandler angeschlossen. Der Tiefpass und der kleine Impedanzwandler werden für diesen Aufbau nicht weiter benötigt. Das Messergebnis ist in *Abb. 2.33* festgehalten. Es ist deutlich erkennbar, dass die Signalform



(a) Signal **(b)** Quantisierung

Abb. 2.33: Sinusschwingung, mit schnellem Ausgang an 47 Ω Last generiert. Frequenz $f = 100$ Hz, Amplitude $\hat{u} = 5$ V, sinusförmiges Signal. Kalibrierter Ausgang auf Kanal *RP_CH_1* und angeschlossenem großem Impedanzwandler mit Verstärkungsfaktor 11. Gut passende Kalibrierung und Signalform. Quantisierungdes Ausgangs nicht ganz gleichmäßig auf Schritte von 21 mV begrenzt. Geringes Signalrauschen, keine unerwünschten Schwingungen.

stimmt und die Amplitude richtig kalibriert ist. Die Quantisierung durch den Ausgang ist befriedigend.

Messgerät

Um die Funktionlität der Eingänge zu überprüfen, wird ein Software benötigt, welche Sensordaten einliest und in der Konsole ausgibt. Die ausführbare Datei *meter* kann beliebig viele Eingänge (*TInput*) und Impedanzmessaufbauten (*TImpedanceMeasureSetup*) als Parameter übergeben bekommen und diese mit einer ebenfalls einstellbaren Frequenz auslesen, wie der Hilfehinweis des Programms erläutert:

```
usage: meter --help
usage: meter OPTIONS

options:
 -h --help                      prints this message
 -v --verbose                   verbose output
 -i --input      INNAME         use input with name INNAME
 -z --setup      IMPNAME        use inputs specified in impedance measure setup with name IMPNAME
 -d --delay      TIME           delay for TIME seconds between two measurements (default: 0.5
    seconds)
 -c --delimiter  STRING         use STRING as table column delimiter

For full documentation, see master thesis by Ilka Schulz or contact:
Leibniz Universität Hannover
```

```
   Institut für Antriebssystem und Leistungselektronik
15 z. Hd. Ilka Schulz
   Welfengarten 1
   30167 Hannover
   Germany
```

Im Betrieb gibt das Programm sämtliche Hinweise über den Fehlerkanal (Kanal 2 unter GNU/Linux, *cerr* in C++) und die Messdaten über den Standardausgabekanal (Kanal 1 unter GNU/Linux, *cout* in C++) aus. Die Messdaten werden als Kommaseparierte Werte, engl. Comma-separated values (CSV) formatiert, sodass die Standardausgabe ohne weitere Bearbeitung von der Kommandozeile aus in eine maschinenlesbare Datei umgeleitet werden kann.

Die Verzögerung zwischen zwei Messungen ist im Rahmen der technischen Möglichkeiten exakt und wird nicht durch die Dauer der Messdatenaufnahme oder der Konsolenausgabe beeinflusst. Der Zeitstempel „driftet“ entsprechend nicht mit der Dauer der Aufnahme.

Zweipolcharakterisierung

Bevor der intelligente Vierpolcharakterisierer entwickelt wird, sollen zunächst Erfahrungen mit der Implementierung eines Zweipolcharakterisierers gesammelt werden. Das Programm ist außerdem nützlich, um die Funktionsweise der analogen Aus- und Eingänge zu beobachten. Dazu generiert es eine Funktion an einem Ausgang, ähnlich wie der Funktionsgenerator. Nach einer bestimmten Anzahl an Perioden, welche als Einschwingdauer angenommen wird, beginnt die Aufnahme von Messpunkten, typischerweise über eine Periode. Die Messdaten werden in eine Datei geschrieben. Die Hilfemeldung der Software lautet:

```
   usage: impmsr --help
   usage: impmsr OPTIONS

   options:
5   -h --help                        prints this message
    -v --verbose                     verbose output
    -f --frequency  FREQ             set frequency to FREQ hertz. 0 or 'DC' for DC
    -a --amplitude  AMP              set amplitude to AMP volts or amps
    -w --waveform   WAVEFORM         will create the waveform WAVEFORM (see below for valid values)
10  -z --setup      IMPNAME          use output specified in impedance measure setup with name IMPNAME
    -s --safety     TIME             wait for TIME seconds before enabling output (default: 5)
    -Np             CYCLES           prime output for CYCLES cycles (default: 20)
    -Nm             CYLES            measure from inputs for CYCLES cycles after priming outputs
         (default: 1)
    -d --default    VALUE            set output to constant (calibrated physical) VALUE when ideling,
        e.g. after ending (default: 0)
15  -da --default-analog VALUE       set output to constant (PWM) VALUE when ideling, e.g. after
        ending (default: using --default)

   data output settings:
    -c --delimiter  STRING           use STRING as table column delimiter
    -pt --pretty-timestamp           let timestamp start with zero, regardless of priming time (see
        option -Np)
```

```
20   -tf --time-factor FACTOR     multiply time with FACTOR
     -if --current-factor FACTOR  multiply current with FACTOR
     -udtf --udt-factor FACTOR    multiply voltage integral with FACTOR
     -or --output-raw FILE        save raw measurements to csv file FILE
     -oc --output-characteristic  FILE  save evaluated data to csv file FILE
25  waveforms (case-insensitive):
     dc                           direct current
     sin                          sine wave
     cos                          cosine wave
     square                       square
30
    For full documentation, see master thesis by Ilka Schulz or contact:
    Leibniz Universität Hannover
    Institut für Antriebssystem und Leistungselektronik
    z. Hd. Ilka Schulz
35  Welfengarten 1
    30167 Hannover
    Germany
```

In *Abb. 2.34* ist eine aufgenommene Messkurve abgebildet, bei der ein angeschlossener Metallschichtwiderstand von 47 Ω mit einem „langsamen" analogen Ausgang und einer Stromwandlung mit dem Faktor $1000 \cdot 10\,\mathrm{m\Omega}$ vermessen ist. Man sieht die bereits zuvor diskutierte schlechte Qualität des ausgegebenen Spannungssignals. Die aufgenommenen Messpunkte weisen die für den *LM324* üblichen Ecken auf, sowie Rauschen, eine schlechte Signalauflösung und eine schlechte zeitliche Auflösung.[12]

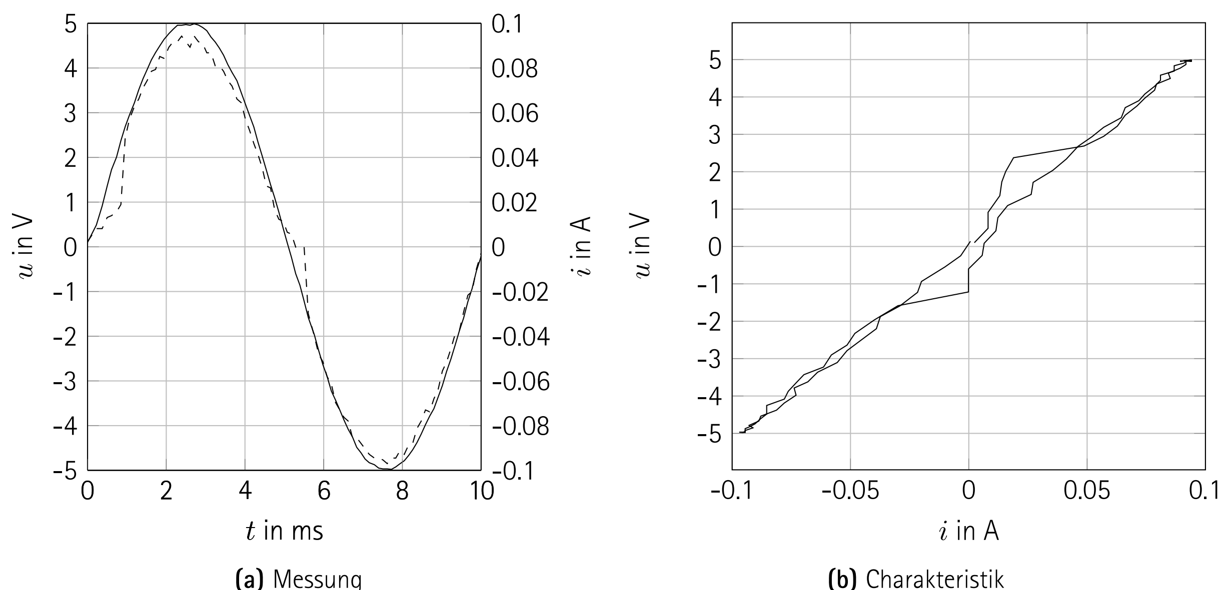
Um den Problemen zu begegnen, wird einer der „schnellen" analogen Ausgänge verwendet (s. o., Funktionsgenerator) und für die Stromwandlung wird der Faktor $5.7 \cdot 1\,\Omega$ gewählt. In *Abb. 2.35* ist die entsprechend besser aufgelöste Messung dargestellt. Die Signalauflösung ist befriedigend, aber die Signale scheinen verrauscht zu sein. Die zeitliche Auflösung ist mit 9 kS/s kaum noch ausreichend und erweckt sogar den Anschein einer Phasenverschiebung.

Intelligente Vierpolcharakterisierung

Um einen Vierpol zu charakterisieren, wird ein einfacher Vorgang gewählt: Der Vierpol wird in zwei Zweipole aufgeteilt und an einem wird eine Wechselspannung und an dem anderen eine Gleichspannung angelegt. Die Spannungen und Ströme werden aufgezeichnet und können ausgewertet werden (siehe Kapitel 5, S. 119). Die Gleichspannung wird variiert, um einen größeren Überblich über das nichtlineare Verhalten des Vierpols zu bekommen. Dieser Algorithmus ist einfach zu implementieren und kann noch weiter automatisiert werden, indem die konkreten Werte für beide Spannungen durch eine Künstliche Intelligenz (KI) gewählt werden.

Die zeitliche Auflösung der Messsignale bei der Vierpolcharakterisierung ist gegenüber der Zweipolcharakterisierung noch einmal halbiert, weil nun doppelt so viele Signale aufgenommen werden. Die Ergebnisse sind mangelhaft. Es kommt deshalb zu keiner Implementierung des Algorithmus, sondern es wird ein externes Os-

[12] Die Messfrequenz von 9 kS/s kann nicht mithilfe eines intelligenteren Speichermanagements erhöht werden, sondern ist wesentlich durch die lange Ausführungsdauer der Funktion *rp_ApinGetValue* der API bedingt.



(a) Messung

(b) Charakteristik

Abb. 2.34: Zweipolcharakterisierung eines ohmschen Widerstands. Langsamer Signalausgang, Strommessung mit Faktor $1000 \cdot 10\,\text{m}\Omega$. „Ecken" nach Nulldurchgängen, Signalrauschen, geringe Signalauflösung, geringe zeitliche Auflösung.

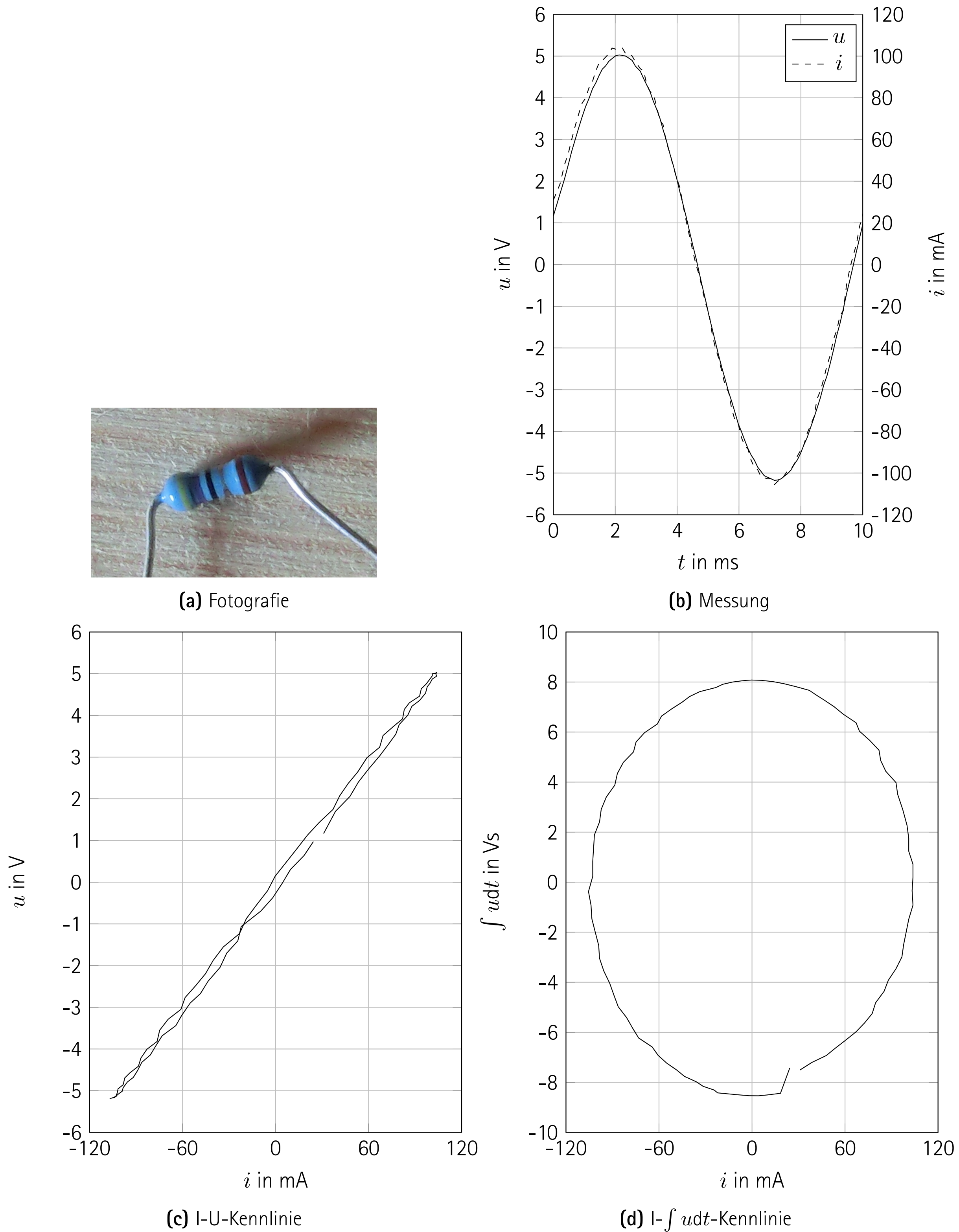
zilloskop an die Messwandler angeschlossen. Weil die signalgenerierende Hardware („RedPitaya") und die signalmessende Hardware (Oszilloskop) nun von einander getrennt sind, ist eine Automatisierung nur möglich, wenn eine geeignete Kommunikation zwischen den beiden Systemen etabliert werden kann.

Für die Automatisierung der Vierpolcharakterisierung kann das Oszilloskop über LAN vom „RedPitaya" angesprochen werden. Leider ist das dafür benötigte Passwort nicht bekannt. Weil außerdem die Programmierung der KI einigen Aufwand bedeutet, wird vor dem Hintergrund der kurzen Bearbeitungszeit auf die Implementierung dieses Werkzeugs verzichtet.

Primitve Vierpolcharakterisierung

Wenn kein Kommunikationskanal zwischen dem Oszilloskop und dem „RedPitaya" etabliert werden kann und das Oszilloskop nicht programmierbar ist, muss die Kalibrierung nachträglich angewandt werden (s. u., Nachträgliche Kalibrierung), was den manuellen Aufwand deutlich erhöht. Außerdem muss jeder Wert für die Gleichspannung einzeln gewählt und dazu eine Messung am Oszilloskop manuell ausgelöst werden. Diese beiden Nachteile eines externen Oszilloskops sollten ursprünglich mit dem „RedPitaya" eliminiert werden. Nun ist der zeitliche Aufwand für die manuell durchzuführenden Messungen gegenüber einer automatischen Messung um mehrere Größenordnungen erhöht.[13] Durch diesen Mehraufwand ist das ursprünglich angestrebte Ziel, einen vollautomatischen Messaufbau zu erhalten, nicht erreicht.

[13] Eine einzelne Messung würde automatisiert weniger als eine Sekunde dauern, wie bereits bei der Zweipolcharakterisierung gezeigt. Die manuelle Aufnahme der Daten, sowie die nachträgliche Kalibrierung dauern mehrere Minuten pro eingestellter Gleichspannung.



(a) Fotografie

(b) Messung

(c) I-U-Kennlinie

(d) I-$\int udt$-Kennlinie

Abb. 2.35: Zweipolcharakterisierung eines ohmschen Widerstands. Befriedigende Signalauflösung, kaum ausreichende zeitliche Auflösung von 9 kS/s und sogar leichte artefaktische Phasenverschiebung. Schlechte Synchronisation von Signal und Trigger

Improvisierte Vierpolcharakterisierung

Um dennoch einen verwendbaren Messaufbau zu erhalten, wird ein „Workaround" angestrebt. Das Oszilloskop kann nach jedem Einschalten so konfiguriert werden, dass es eine Verbindung über das Internetprotokoll, engl. Internet Protocol (IP) anbietet, für die kein Passwort benötigt wird.[14] Weil dieser „Workaround" erst kurz vor dem Ende der Bearbeitungszeit für die vorliegende Arbeit angefangen ist, bittet die Autorin um Verständnis für die improvisierte Struktur der Software. Die Funktionalität dieser Software ist trotzdem gegeben.

Die Spannungen werden direkt an den Klemmen der Last gemessen. Die Ströme werden mithilfe von zwei Shuntwiderständen aufgenommen (siehe Abschnitt 2.2.5, S. 76). Die Messbereiche und die Auslöseeinstellungen müssen manuell am Oszillskop vorgenommen werden.

Das Programm bietet einige Optionen zur Einstellung. Die Werte Kalibrierung der Shuntwiderstände, wird der Einfachheit halber ebenfalls als Kommanozeilenparameter übergeben, die Position jener ist fest programmiert und kann nicht geändert werden.

```
usage: orthomsr-scope --help
usage: orthomsr-scope OPTIONS

options:
 -h --help                  prints this message
 -v --verbose               verbose output
 -o1           ONAME        use high speed output ONAME for primary signal
 -f1           FREQ         set frequency of primary signal to FREQ hertz. 0 or 'DC' for DC
    (default: 100)
 -a1           AMP          set maximum amplitude of primary signal to AMP volts or amps
 -w1           WAVEFORM     will create the waveform WAVEFORM on primary output (see below
     for valid values, default: sin)
 -d1           VALUE        set primary output to constant (calibrated physical) VALUE when
     ideling, e.g. after ending
 -o2           ONAME        use high speed output ONAME for secondary signal
 -f2           FREQ         set frequency of secondary signal to FREQ hertz. 0 or 'DC' for
     DC (default: 0)
 -a2           AMP          set maximum amplitude of secondary signal to AMP volts or amps
 -w2           WAVEFORM     will create the waveform WAVEFORM on secondary output (see below
     for valid values, default: dc)
 -d2           VALUE        set secondary output to constant (calibrated physical) VALUE
     when ideling, e.g. after ending
 -o            FILE         print results to FILE (see below for variables; specify this
     option multiple times to save the same data to different files)
 -s --safety   TIME         wait for TIME seconds before enabling outputs (default: 5
     seconds)
 -Tp           TIME         prime output for TIME seconds (default: 0.1 seconds)
 -N1           NUMBER       measure NUMBER different values for primary output (default: 1
     ohms)
 -N2           NUMBER       measure NUMBER different values for secondary output (default: 1
     ohms)
```

[14]Nach jedem Einschalten muss am Oszilloskop dafür vor Ort die Einstellung „Protocol" im Menü „Utility->Socket Server" von „None" zu „Terminal" geändert werden. Die über das Webinterface ansprechbare Schnittstelle ist weiterhin nicht verfügbar, weil das Passwort unbekannt ist.

```
   --scope-addr    NETADDR     scope must be reachable under IPv4 address NETADDR
      (default192.168.1.103)
   -r1             RES         assume RES ohms for primary shunt (default: 1.131
   -r2             RES         assume RES ohms for primary shunt (default: 1.133
25 -tf             FACTOR      multiply time values with FACTOR when saving to file (default: 1)

   waveforms (case-insensitive):
    dc                         direct current
    sin                        sine wave
30  cos                        cosine wave
    square                     square

   filename variables:
    %u1                        primary voltage
35  %i1                        primary current
    %u2                        secondary voltage
    %i2                        secondary current
    %n                         measurement counting index
    %N                         measurement counting index plus one
40  %n1                        primary measurement counting index
    %n2                        secondary measurement counting index
    %N1                        primary measurement counting index plus one
    %N2                        secondary measurement counting index plus one

45 For full documentation, see master thesis by Ilka Schulz or contact:
   Leibniz Universität Hannover
   Institut für Antriebssystem und Leistungselektronik
   z. Hd. Ilka Schulz
   Welfengarten 1
50 30167 Hannover
   Germany
```

Die Messdaten müssen anschließend um die Spannungsintegrale erweitert werden (s. u).

Eine durchgeführte Vierpolcharakterisierung wird zusammen mit den notwendigen Bearbeitungen, wie der Berechnung der Spannungsintegrale, am Ende des Kapitels vorgestellt (siehe Abschnitt 2.4, S. 108).

Prinzipiell eignet sich dieses Programm auch für die Zweipolcharakterisierung, wenn der sekundäre Ausgang bspw. auf 0 V festgelegt wird.

Berechnung von $\int u dt$

Um das zeitliche Integral der Spannung zu berechnen, wird ein kurzes Programm geschrieben, das bereits aufgezeichnete Messdaten für die Berechnung einliest. Die Daten müssen als Datei mit CSV vorliegen, sodass die Datei manipuliert werden kann. Dieses „offline" laufende Programm bietet den Vorteil, dass bereits aufgezeichnete Daten nachträglich bearbeitet werden können und die Messung von der Aufarbeitung der Daten zeitlich und algorithmisch getrennt ist. Diese Trennung wird insbesondere benötigt, wenn Daten mit einem nicht programmierbaren Messgerät aufgenommen werden, bspw. mit einem Oszilloskop statt des „RedPitayas". Für die

Berechnung wird die Simpson-Regel verwendet. Nach Abschluss der Berechnung wird das Spannungsintegral um seinen zeitlichen Mittelwert reduziert, es wird „zentriert". Diese Maßnahme wird getroffen, weil die Berechnung wegen der Integrationskonstante nicht eindeutig ist. Um die Messergebnisse untereinander vergleichbar zu gestalten wird deshalb der Mittelwert des Spannungsintegrals zu null definiert.

Die programmeigene Hilfe erläutert die Funktionen:

```
usage: udt --help
usage: udt OPTIONS FILE1 [FILE2 [FILE3 [...]]]

options:
 -h --help                 prints this message
 -v --verbose              verbose output
 -t            NAME        column name containing times (default: t)
 -u            NAME        column name containing voltages (default: u)
 -udt          NAME        column name to contain voltage integrals (default: udt)
 -f            FREQ        frequency of signal for voltage correction, use NaN for none
     (default: nan)
 -c            STRING      use STRING as delimiter (default: ;)

For full documentation, see master thesis by Ilka Schulz or contact:
Leibniz Universität Hannover
Institut für Antriebssystem und Leistungselektronik
z. Hd. Ilka Schulz
Welfengarten 1
30167 Hannover
Germany
```

Nachträgliche Kalibrierung

Wie bereits zuvor beschrieben (siehe primitive Vierpolcharakterisierung), werden Messungen teils mithilfe eines nicht programmierbaren Oszilloskops durchgeführt. In diesen Fällen muss eine Kalibrierung nachträglich angewandt werden. Dazu werden Kalibrierungsdateien manuell angelegt, welche mit keinem Hardwareport des „RedPitayas" assoziiert werden.[15] Die Daten werden wieder als CSV in einer Datei gespeichert, sodass die Kalibrierung nachträglich angewandt werden können. Das Werkzeug bietet selbst die Hilfe:

```
usage: converter --help
usage: converter OPTIONS FILE

options:
 -h --help                 prints this message
 -v --verbose              verbose output
 -i --input    INAME       name of the input containing calibration
 -zu           SETUP       setup containing the voltage input containing calibration
 -zi           SETUP       setup containing the current input containing calibration
 -r --raw-col  NAME        column name containing raw measurements (default: )
 -v --val-col  NAME        column name to contain converted values (default: )
```

[15] Der Einfachheit halber werden die Klasse *TInput* und das Kalibrierungswerkzeug nicht angepasst, sondern es wird ein beliebiger Hardwareport eingetragen, der niemals ausgelesen wird.

Tab. 2.2: Verdeutlichung des Prinzips bei der Glättung der Quantisierung.

Zeitpunkt	Messung	Glättung
0	0	0.75
1	0	0
2	0	1.25
3	0	1.5
4	1	1.75
5	1	1
6	1	1.25
7	1	1.5
8	2	1.75
9	2	2
10	2	2.25
11	2	2.5
...	...	...

```
 -c              STRING       use STRING as delimiter (default: ;)

For full documentation, see master thesis by Ilka Schulz or contact:
Leibniz Universität Hannover
Institut für Antriebssystem und Leistungselektronik
z. Hd. Ilka Schulz
Welfengarten 1
30167 Hannover
Germany
```

Es sei angemerkt, dass für die „improvisierte Vierpolcharakterisierung" keine nachträgliche Kalibrierung benötigt ist, weil die Werte der Shuntwiderstände in ihrer improvisierten Struktur enthalten sind. Dieser „Workaround" soll in nachfolgenden Arbeiten besser strukturiert werden.

Glättung der Quantisierung

Ein Analog-Digital-Umsetzer, engl. Analog Digital Converter (ADC) weist stets Quantisierungschritte größer als null auf. Es kommt deshalb vor, dass die Messdaten Stufen aufweisen. Um diese Unregelmäßigkeit zu beheben, wird ein Programm entwickelt, welches die Quantisierungsschritte erkennt und eine lineare Näherung durchführt. Es sucht nach „Plateaus" in den Messwerten, findet deren Mittelstellen und verwendet diese als Stützstellen für stückweise lineare Splines. Das Prinzip ist in *Tab. 2.2* verdeutlicht. Das Programm ist wie folgt zu verwenden:

```
usage: smth --help
usage: smth OPTIONS FILE1 [FILE2 [FILE3 [...]]]

options:
 -h --help                  prints this message
 -v --verbose               verbose output
```

```
   −t                NAME        column name containing times (default: t)
   −c                STRING      use STRING as delimiter (default: ;)

10 For full documentation, see master thesis by Ilka Schulz or contact:
   Leibniz Universität Hannover
   Institut für Antriebssystem und Leistungselektronik
   z. Hd. Ilka Schulz
   Welfengarten 1
15 30167 Hannover
   Germany
```

2.3.3 Übersetzung

Die Übersetzung des Quellcodes geschieht mit der Compiler-Suite des GNU-Projekts, engl. GNU Compiler Collection (GCC), welche auf mit dem Betriebssystem des „RedPitayas" mitgeliefert wird. Außerdem wird das Werkzeug *make* genutzt, das Makefile ist stark an das des Herstellers angelehnt und wegen seiner Länge der Arbeit angehängt (siehe Code, S. 182). Allerdings sind die ausführbaren Dateien im Ziel „all" explizit aufgeführt und ihre Abhängigkeiten sind ebenfalls explizit genannt. Auf diese Weise kann bei einer späteren Erweiterung der Software erheblich Rechenleistung gespart werden, weil nicht sämtliche Übersetzungseinheiten für jedes Ziel neu übersetzt werden müssen.

Quellcodedateien werden, anders als durch das Makefile des Herrstellers, nicht automatisch als alleinstehendes Programm interpretiert, sondern können nun auch Code ohne Einstiegspunkt enthalten. Es ist unbekannt, warum der Hersteller in seinem Makefile diese unübliche Wahl getroffen hat, die hiermit korrigiert ist.

Die Zeitersparnis durch die Nutzung eines Makefiles ist erheblich. Die Übersetzung sämtlicher Einheiten und Linkung zu den ausführbaren Dateien dauert ca. 115 s. Die Übersetzung des Kalibrierungswerkzeuges *calib* und dessen Linkung benötigen hingegen lediglich 4,5 s, wenn zuvor alle anderen Übersetzungseinheiten übersetzt sind. Weil nach der Entwicklung der Bibliotheken meist nur Änderungen am Code des Hauptprogramms vorgenommen werden, bevor die GCC erneut aufgerufen wird, beeinflusst das Makefile den Arbeitsfluss bei der Entwicklung maßgeblich positiv.[16]

[16] Der extrem hohe Zeitaufwand für die Übersetzung trotz des geringen Codeumfangs ist auf die sehr geringe Rechenleistung des Systems im Vergleich zu üblichen PCs zurückzuführen.

2.4 Zusammenfassung

Wegen der großen Änderungen am Messaufbau, die während der Entwicklung der Software und insbesondere der improvisierten Vierpolcharakterisierung gemacht sind, soll abschließend ein Überblick über den finalen Messaufbau gegeben werden, wie er später verwendet wird (siehe Kapitel 6, S. 139).

Der Aufbau ist schematisch in *Abb. 2.36* illustriert. Darin werden die Signale vom „RedPitaya" erzeugt und von den zwei großen Impedanzwandlern gewandelt und verstärkt. Die Messung der Spannungen geschieht direkt an der Last durch ein Oszilloskop. Die Ströme werden vom selben Oszilloskop als Spannungen über jeweils einem Shuntwiderstand gemessen. Das „RedPitaya" kommuniziert über eine primitive Netzwerkverbindung mit dem Oszilloskop und erhält so die Messdaten. Der Messvorgang ist dann abgeschlossen und kann mit anderen Ausgangssignalen wiederholt werden.

Nach der Messung werden die Daten von er Software auf dem „RedPitaya" noch bearbeitet. Es wird eine Kalibrierung angewandt und die zeitlichen Spannungsintegrale werden berechnet. Die dafür notwendigen Befehle lassen sich in einem Bash-Skript zusammenfassen:

```
#!/bin/bash
```

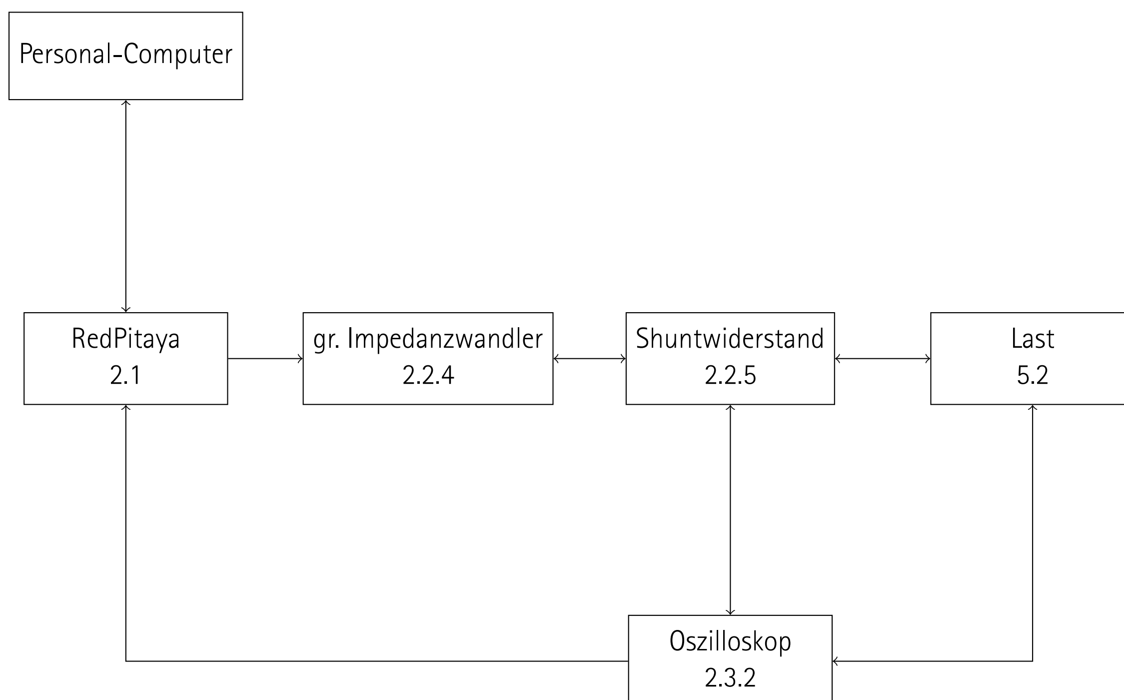


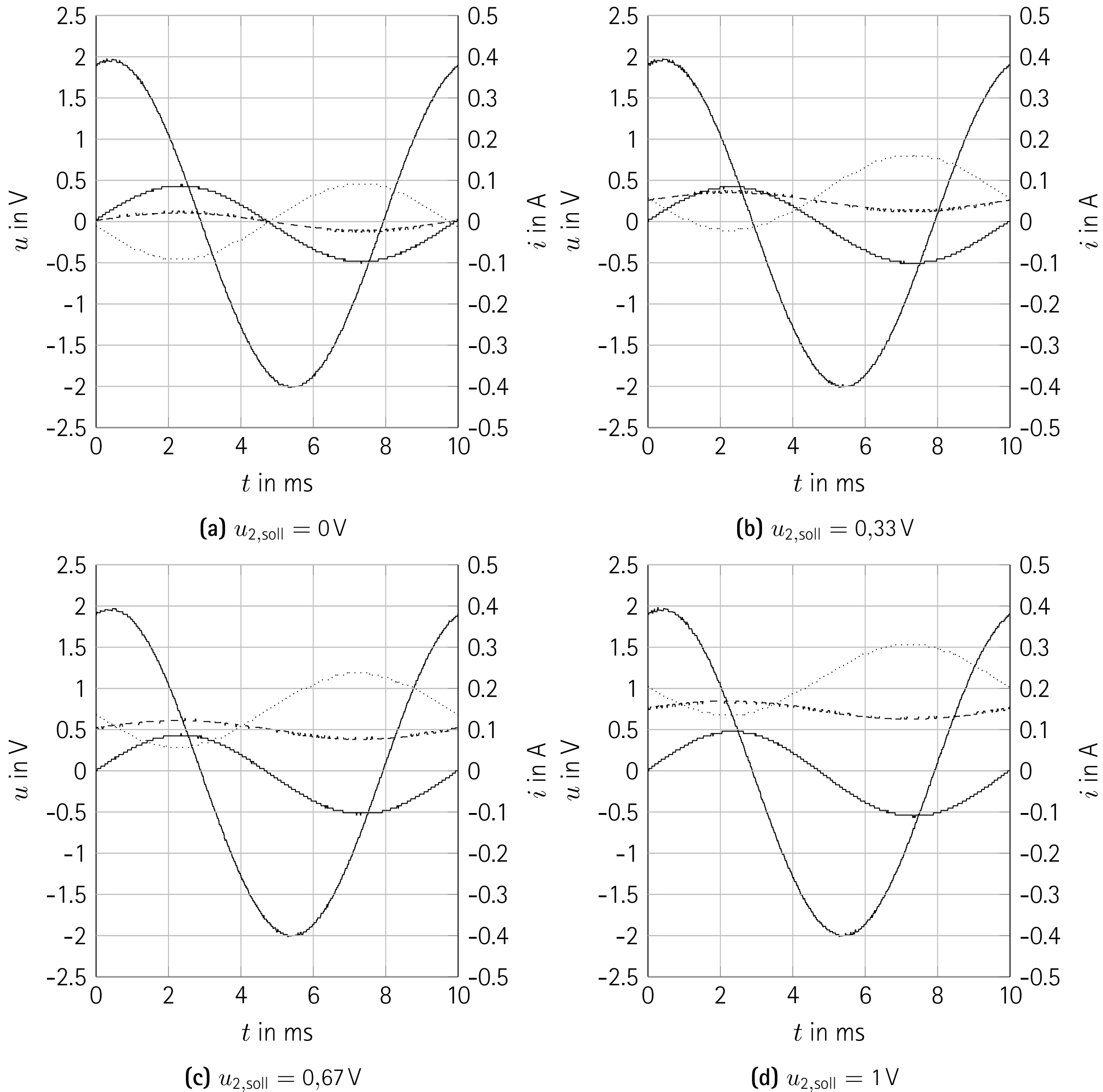
Abb. 2.36: Schematische Übersicht über den improvisierten Messaufbau. Netzwerkverbindung zwischen Personal Computer und RedPitaya zur Programmierung und Datenübertragung (oben links). Ausgangssignale von RedPitaya impedanzgewandelt an die Last angelegt (mittig). Messgrößen vom Oszilloskop (unten) direkt und über einem Shuntwiderstand (einer je Kanal) erfasst (rechts) und über Netzwerk an „RedPitaya" übertragen (unten). Nummer des jeweils beschreibenden Abschnitts unter dem Schriftzug jedes Elements.

```bash
set -e

echo "------------ cleaning directory ------------"
TARGETDIR="measurements"
mkdir -p "$TARGETDIR"
rm "$TARGETDIR"/* -r -f
echo "done"

echo "------------ compiling ------------"
make orthomsr-scope
make udt
make smth

echo "------------ taking measurements ------------"
sudo ./orthomsr-scope -f1 100 -a1 10 -o1 u1hs -d1 0 -f2 0 -a2 5 -o2 u2hs -d2 0 -s 0 -Tp 1 -o
    "$TARGETDIR/test-10-%n2.csv" -N1 1 -N2 6 -tf 1000

echo "------------ processing data ------------"
echo "please be patient, this can take several minutes..."
sudo chown ilka:ilka  "$TARGETDIR"/test-*.csv
chmod ug+rw "$TARGETDIR"/test-*.csv
BACKUPDIR=$(mktemp -d)
echo "saving backup to $BACKUPDIR"
cp "$TARGETDIR" "$BACKUPDIR" -R
echo "smoothing..."
./smth "$TARGETDIR"/test-*.csv
echo "adding u1dt"
./udt -u u1 -udt u1dt -f 0.1 "$TARGETDIR"/test-*.csv
echo "adding u2dt"
./udt -u u2 -udt u2dt -f 0   "$TARGETDIR"/test-*.csv
echo "done"

echo "------------ results ------------"
tree "$TARGETDIR"
```

Die Ergebnisse werden als CSV in einer Datei auf dem „RedPitaya" gespeichert und können anschließend ausgewertet werden (siehe Kapitel 5, S. 119). Beispielhafte Ergebnisse einer solchen Messungen sind in *Abb. 2.37* abgedruckt. Ein kompletter Durchlauf des Bash-Skripts dauert bei diesem Beispiel 13,5 s.

Für nachfolgende Arbeiten sollte das Ziel angestrebt werden, alle Ausgänge und Eingänge in einer einfachen Hardware, wie dem „RedPitaya", zu integrieren. Für einen ggf. notwendigen Wechsel auf andere Plattformen ist die Software vorbereitet, sodass nur wenige Zeilen Code geändert werden müssen. Außerdem soll eine intelligente Vierpolcharakterisierung implementiert werden, welche die Ausgangssignale mithilfe einer KI bestimmt. Langfristig wird eine Möglichkeit zur Strommessung benötigt, welche eine kleinere Rückwirkung auf die Spannung hat, wie aus *Abb. 2.37* hervorgeht; für die hohen Impedanzen in den durchzuführenden Versuchen (siehe Kapitel 6, S. 139) sind die Shuntwiderstände allerdings ausreichend.

Aus Gründen der Zeit und dem Mangel an einer alternativen Hardware wird im Versuch dieser Arbeit (siehe Abschnitt 5.2, S. 141) nur die „improvisierte Vierpolcharakterisierung" verwendet.



(a) $u_{2,\text{soll}} = 0\,\text{V}$ **(b)** $u_{2,\text{soll}} = 0{,}33\,\text{V}$

(c) $u_{2,\text{soll}} = 0{,}67\,\text{V}$ **(d)** $u_{2,\text{soll}} = 1\,\text{V}$

Abb. 2.37: Ergebnis einer Vierpolcharakterisierung an einem Transformator. Primärspannung (durchgezogen, Amplitude von 2 V), Primärstrom (durchgezogen, Amplitude von ca. 0,1 A), Sekundärspannung (gestrichelt), Sekundärstrom (gepunktet). Beide Wicklungen mit gleicher Windungszahl $N_1 = N_2 = 300$. Transformator primärseitig an ein Wechselspannung ($\hat{u}_1 = 2\,\text{V}$; $f_1 = 100\,\text{Hz}$) und sekundärseitig an eine Gleichspannung ($u_{2,\text{soll}} = 0\,\text{V}$ (oben links) bis $u_{2,\text{soll}} = 1\,\text{V}$ (unten rechts)) angeschlossen. Über Shunt-Widerstand abfallender Anteil der Sekundärspannung deutlich zu sehen, bei Primärspannung nur schwierig erkennbar. Durch Primärstrom verursachter Wechselstrom durch die Sekundärwicklung deutlich zu sehen. Keine Nichtlinearität beobachtbar.

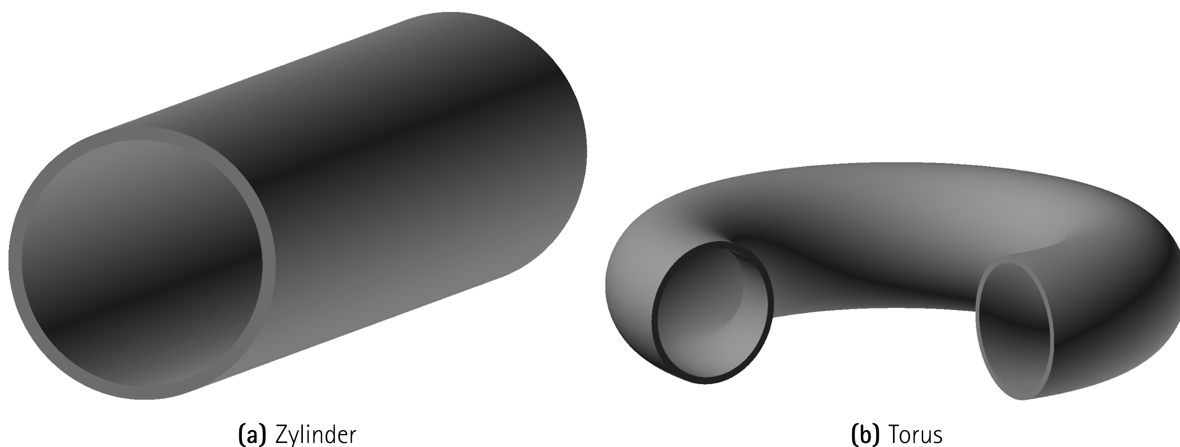
3 Verfahren zur Herstellung gekrümmter Eisenbleche

[A toroidal magnetic core's] practical construction is only possible with ferrite core materials[...], because of the lack of practical laminations effictive in two perpendicular directions and the otherwise excessive eddy-current losses.

– Dr. Eyup Salih Tez

In dieser Arbeit wird das Prinzip der orthogonalen Magnetisierung in aus Blechen geformten magnetischen Kreisen untersucht (siehe Abschnitt 5.2, S. 141). Leider lassen sich nicht alle Blechgeometrien aus einem ebenen Blech zuschneiden, biegen oder falten, ebenso der später zu verwendende Torus. Dieser ist von besonderem Interesse, weil darin zwei homogene, orthogonale Magnetfelder möglich sind, wie bereits in [31] beschrieben.

In *Abb. 3.1* sind beispielhaft zwei Geometrien präsentiert, eine, die sich aus einem ebenen Blech herstellen lässt und eine, für welche die direkte Herstellung von gekrümmtem Blech notwendig ist. Der Unterschied lässt sich so erklären: Wenn ein ebenes Blech aufgerollt wird, erfährt die innere Seite eine Stauchung und die äußere Seite eine Streckung. Weil die Dicke des Bleches gegenüber dem Biegeradius klein ist, können die Streckung und die Stauchung vernachlässigt werden. Sobald das Blech um eine volle Umdrehung aufgerollt ist, erhält man



(a) Zylinder **(b)** Torus

Abb. 3.1: Geometrie aus ebenem und Geometrie aus gekrümmten Blech. Ebenes Blech zu Zylinder aufgerollt (links) und Torus, der nicht aus ebenem Blech hergestellt werden kann (rechts).

einen Zylinder. Theoretisch müsste der Zylinder nun erneut aufgerollt werden, und zwar um eine orthogonal zur ersten stehende Biegeachse. Wenn der zweite Biegeradius groß ist gegenüber dem ersten, erhält man eine Form, die sich als Kreisrand annähern lässt. Bei dem Torus in *Abb. 3.1b* ist der zweite Biegeradius allerdings nur wenig größer als der erste, sodass die Innenseite des Torus gestaucht und die Außenseite gestreckt werden müsste. Im besten Fall resultiert dies in einem erheblichen Unterschied der Blechdicke zwischen der Innen- und der Außenseite des Torus. Praktisch knickt oder reißt das Blech bei dem Versuch. In jedem Fall ist das Ergebnis mangelhaft und es wird ein alternativer Ansatz benötigt, bei welchem der Torus und vergleichbare Geometrien nicht aus ebenem Blecht gefertigt werden.

3.1 Stand der Technik

Es existieren verschiedene Verfahren, um gekrümmte Geometrien aus magnetischen Kernmaterialien herzustellen. In diesem Abschnitt werden einige davon kurz vorgestellt, sie sollen aber nicht vertieft werden.

3D-gedrucktes Eisen wird hergestellt, indem das metallene Filament bei hohen Temeperaturen aufgeschmolzen wird. Der Prozess ist lediglich für grobe Strukturen geeignet, aufwendig in der Durchführung und verursacht sehr unsaubere Oberflächen.

3D-gedrucktes Eisen in Legierungen oder Verbundwerkstoffen existiert von diversen kommerziellen Anbietern, ist allerdings meist stark in der Wahl der Stoffe eingeschränkt und bietet wenig Kontrolle über die Mikrostrukturen.

3D-gedruckte Kunststoffe mit Ferritstaubanteil müssen für die mechanische Stabilität einen erheblichen Anteil von Kunststoff aufweisen. Selbst bei einem Ferrit mit unendlich hoher Permeabilität und einem Kunststoffanteil von lediglich 5 % ist die resultierende relative Permeabilität auf $\mu_\mathrm{r} = 20$ begrenzt.

Ferrithaltige Paste weist das gleiche Problem wie die genannten Kunststoffe auf.

Gesinterte Ferrite sind ein vielversprechender Ansatz und werden auch in [31] als einzige Möglichkeit genannt. Der finanzielle Aufwand übersteigt das Budget für die vorliegende Arbeit bei Weitem.

Aufgewickelter ferromagnetischer Draht hat das Problem, dass er teils ungleichmäßig und teils systematisch verteilte Luftspalte einbringt, welche eine Modellierung der Geometrie nahezu unmöglich machen.[1]

Verschweißte Blechteile sind mit einem hohen Aufwand verbunden weisen schwierig kontrollierbare Inhomogenitäten auf, insbesondere an den und um die Schweißstellen.

Gebogene Bleche eignen sich für Geometrien, deren Netz eine Dimension von nur wenig über zwei aufweist, d. h. es werden leichte „Beulen“ im Resultat vernachlässigt.

[1] Prof. Dr.-Ing. P. Zacharias vom KDEE-EVS hat bereits praktische Teilerfolge mit diesem Ansatz erzielen können, allerdings liegen der Autorin bislang dazu keine Schriftstücke vor.

3.2 Entwicklung eines Verfahrens

Für jede zwei- oder dreidimensionale Blechgeometrie lässt sich ein geschlossenes oder offenes Volumen definieren, dessen Oberfläche identisch zu der Blechfläche ist. Nicht selten dürfte es sich dabei um ein zusammenhängendes Volumen handeln, sodass es sich mithilfe von 3D-Druck oder mit Gussverfahren aus Kunststoff herstellen lässt, ebenso bei einem Torus. Das Volumen oder mindestens eine dünne Schicht an der Oberfläche dessen muss nun elektrisch leitfähig sein, um die Oberfläche galvanisch mit Eisen zu überziehen.

Im Folgenden wird die Entwicklung einen Verfahrens begonnen, allerdings kann es aufgrund der kurzen verfügbaren Zeit nicht fertiggestellt werden, sodass nur die bislang gemachten Fortschritte und Beobachtungen festgehalten werden. Diese Umfassen die Oberflächenbehandlung, sodass ein 3D-gedrucktes Kunststoffteil elektrisch leitfähig wird, sowie erste Erfahrungen mit einem galvanischen Überzug mit Eisen. Schließlich wird ein Ausblick auf ausstehende Arbeiten und den in dieser Arbeit gewählten Workaround gegeben.

3D-Druck

Das zu beschichtende Volumen wird aus Acrylnitril-Butadien-Styrol-Copolymer (ABS) 3D-gedruckt. Der Druck wird beim Mechatronik-Zentrum Hannover in Auftrag gegeben. Das Material ist gewählt, um eine Oberflächenbehandlung mit Aceton zu ermöglichen. Ein Beispiel für einen Halbtorus ist in *Abb. 3.2* gezeigt.

Abb. 3.2: 3D-gedruckter Halbtorus aus ABS. Zwei Hälften können zu einem Torus zusammengesetzt werden. Hohler Stiel (links) zur späteren Befestigung im Galvanisierungsbecken und zum Durchführen von Draht für elektrische Wicklungen.

(a) glatt, 8 mm breit, spezielle Beleuchtung **(b)** roh, 20 mm breit, Umgebungslicht

Abb. 3.3: Graphitisierte Oberfläche. Polierte Oberfläche (links) und rohe Oberfläche (rechts). Beide sind mit graphithaltiger Firnis versehen und anschließend vorsichtig poliert. Die glatte Oberfläche glänzt metallisch und die mit einem Kosmetiktuch(!) verursachten Kratzer sind nur mit einer geeigneten Lichtquelle sichtbar. Die rohe Oberfläche ist rau.

Oberflächenbehandlung

Um eine glatte Oberfläche zu erhalten, wird das Objekt zunächst abgeschliffen und anschließend mit dem Acetondampfverfahren behandelt. [1] Der durch diese Behandlung erzielte Unterschied ist in *Abb. 3.3* deutlich sichtbar.

Anschließend wird die Oberfläche mit Graphit benetzt. Eine Mischung aus 30 % Graphit, 20 % Polyurethan-Glanzfirnis und 50 % Wasser nach Masse kann mit einer Airbrushpistole aufgetragen werden und ergibt eine mechanisch robuste Oberfläche mit guter Leitfähigkeit. Sie wird nach dem Trocken mit Schleifpapier (1000-er Körnung) und Kosmetiktüchern poliert. Zwei ebene Oberflächen sind in *Abb. 3.3* festgehalten. Darin ist eine Oberfläche poliert, die andere ist direkt dem 3D-Druckverfahren entnommen. Beide sind anschließend mit der graphithaltigen Firnis behandelt.

Es existieren Alternativen zu der o. g. graphithaltigen Firnis. Sie Materialien werden als 50 mm langer und 10 mm breiter Streifen auf einer ebenen Oberfläche aus ABS aufgetragen und der Widerstand des Streifens wird gemessen. Das Ergebnis ist in *Tab. 3.1* festgehalten. Daraus wird ersichtlich, dass der Leitlack auf Silberbasis mit Abstand die besten elektrischen Eigenschaften hat. Leider ist für dessen Anwendung keine Zeit im Rahmen der vorliegenden Arbeit, sodass im Folgenden die Graphitfirnis verwendet wird.

Direkte galvanische Beschichtung mit Eisen

Die Oberfläche ist durch die zuvor genannte Behandlung mit Graphitfirnis glatt und elektrisch leitfähig. Nun wird versucht, sie galvanisch mit Eisen zu überziehen. Dazu wird das Objekt an die Kathode einer Galvanisie-

Tab. 3.1: Vergleich verschiedener Oberflächenbeschichtungen. Hersteller oder Korngröße in Klammern angegeben. Widerstand an einem 50 mm langen und 10 mm breiten Streifen aufgenommen. Große Qualitätsunterschiede zwischen den Beschichtungsmaterialien, ebenfalls große Kostenunterschiede.

Oberflächenbeschichtung	Widerstand
Leitlack auf Silberbasis (Kemo Electronic)	5 Ω
Graphitfirnis (20 µm), gepresst, poliert	80 Ω
Graphitpulver (20 µm), mehrere Lagen, gepresst, poliert	200 Ω
Graphitpulver (20 µm), gepresst, poliert	400 Ω
Leitlack auf Graphitbasis (Cramolin), gepresst, poliert	1000 Ω

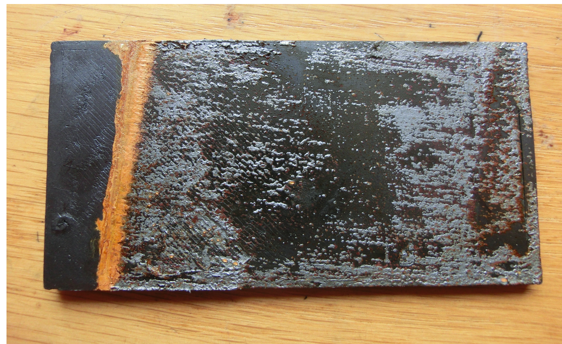
Abb. 3.4: Fotografie vom Galvanisierungsbecken. Spannungsquelle (rechts), Kathode (schwarz), Anode (rot), Polyethyleneimer mit Elektrolytlösung und anodisch verwendetem Eisenblech (mittig). Stange als Halterung der Kathode. Rötliche Färbung der Lösung durch feinen Rost. Kein galvanisch zu überziehendes Objekt an die Kathode angeklemmt.

rungsanlage geklemmt. Der Aufbau ist in *Abb. 3.4* abgebildet. Als Anode wird einfaches Eisenblech aus dem Baumarkt und als Elektrolyt wird Eisen(II)-Sulfat verwendet, sodass sich die folgenden Reaktionsgleichungen ergeben.

Das Eisen(II) wird zu Eisen(III) oxidiert und metallisches Eisen wird an der Kathode abgeschieden:

$$\begin{aligned} \text{Ox: } Fe^{2+} &\longrightarrow Fe^{3+} + e^{-} \\ \text{Red: } Fe^{2+} + 2e^{-} &\longrightarrow Fe \\ \text{Redox: } 3Fe^{2+} &\longrightarrow Fe + 2Fe^{3+} \end{aligned} \tag{3.1}$$

Eisen(II) wird an der Kathode reduziert und metallisches Eisen wird abgeschieden. An der Anode bildet sich

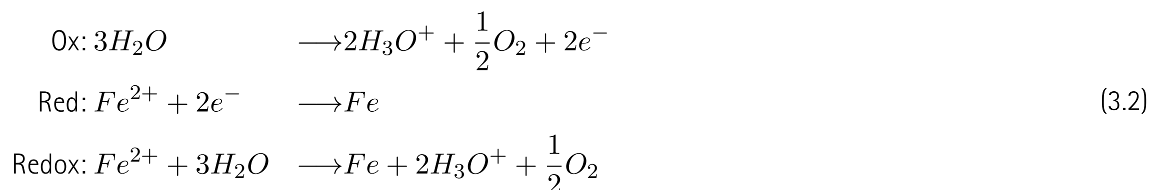


(a) gesamtes Objekt, 100 mm breit

(b) Großaufnahme, 10 mm breit

Abb. 3.5: Galvanisiertes Objekt. Gesamtaufnahme einer zu Testzwecken verwendeten Geometrie mit nicht ins Elektrolytbecken eingetauchtem linken Rand und Rostablagerungen (links), Großaufnahme von Eisen-„Flakes" (rechts). Nur stückweise erfolgte Galvanisierung deutlich erkennbar. Roter Rost. Objekt zeitnah mit reiner Polyurethanfirnis konserviert um weitere Korrosion zu vermeiden.

Sauerstoff:



$$\begin{aligned}
\text{Ox: } 3H_2O &\longrightarrow 2H_3O^+ + \frac{1}{2}O_2 + 2e^- \\
\text{Red: } Fe^{2+} + 2e^- &\longrightarrow Fe \\
\text{Redox: } Fe^{2+} + 3H_2O &\longrightarrow Fe + 2H_3O^+ + \frac{1}{2}O_2
\end{aligned} \tag{3.2}$$

Das Sulfation wird nicht verbraucht:

$$\begin{aligned}
SO_4^{2-} + 2e^- &\longrightarrow SO_3 + \frac{1}{2}O_2 \\
SO_3 + H_2O &\longrightarrow H_2SO_4 \\
H_2SO_4 + 2H_2O &\longrightarrow 2H_3O^+ + SO_4^{2-}
\end{aligned} \tag{3.3}$$

Aus den Reaktionsgleichungen und hier nicht aufgeführten Nebenreaktionen wird ersichtlich, dass lokale pH-Schwankungen nahe den Elektroden auftreten und die Lösung insgesamt saurer wird. Diese können durch die Zugabe von Puffersalzen abgemildert werden, worauf hier nicht weiter eingegangen werden soll.

Das Ergebnis einer solchen Galvanischen Beschichtung ist am Beispiel einer ebenen Oberfläche in *Abb. 3.5* dargestellt. Es ist deutlich sichtbar, wie sich das Eisen nur stellenweise abscheidet. Dafür werden die geringe Leitfähigkeit der Graphitfirnis und die schlechte chemische Kompatibilität verantwortlich vermutet, worauf leider nicht weiter eingegangen werden kann.

Ausblick

Es wird vermutet, dass eine Beschichtung des Graphits mit Kupfer bessere Ergebnisse hervorbringt als mit Eisen. Dieses müsste dann in einem zweiten Schritt mit Eisen überzogen werden. Außerdem könnte die Graphitfirnis durch den Silberleitlack ersetzt werden (s. o.).

Dafür ist leider keine Bearbeitungszeit vorgesehen, sodass die Entwicklung des Verfahrens zur Herstellung gekrümmter Eisenbleche an dieser Stelle abgebrochen wird.

Im Versuch kann entsprechend keine Geometrie verwendet werden, in der zwei annähernd homogene, orthogonale Magnetfelder möglich sind (siehe Abschnitt 5.2.2, S. 144). Dafür wäre der bereit erwähnte Torus notwendig. Stattdessen wird eine Feldsimulation durchgeführt (siehe Kapitel 5, S. 119, siehe Abschnitt 5.2, S. 141), welche zumindest innerhalb von sehr kleinen Raumgebieten homogene Feldverteilungen annimmt und somit ebenfalls einen Zusammenhang zwischen Materialmodell und Klemmenmodell herstellt.

4 Software zur Netzwerk- und Feldsimulation

Der Entwurf von objektorientierter Software ist schwer. Noch schwerer aber ist der Entwurf wiederverwendbarer objektorientierter Software.

– Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides in der Einleitung zu [15]

Später durchgeführte Messungen an realen Aufbauten sollen mithilfe von Simulationen nachvollzogen werden. Bspw. kann ein Materialmodell entwickelt sein, welches nur den intrinsischen Zusammenhang $\boldsymbol{B}(\boldsymbol{H})$ beschreibt, doch mithilfe einer Feldsimulation kann es für ein magnetisches Netzwerkelement verwendet werden. Auf diese Weise wird bereits ein Zusammenhang zwischen beliebigen intrinsischen Modellen und Netzwerkmodellen hergestellt, wie er in der Aufgabenstellung gefordert ist. Zusätzlich können Netzwerkelemente in ganze Netzwerke eingebunden sein, was die vollständige Nachbildung eines Versuchsaufbaus ermöglicht.

Den Lesenden fällt sicherlich auf, dass einige andere Software zur Netzwerk- und Feldsimulation vielfach kommerziell und frei verfügbar ist und es stellt sich die Frage nach der Notwendigkeit der Entwicklung einer weiteren Software, wie sie in diesem Kapitel vorgestellt ist. Der Unzulänglichkeiten verfügbarer Software lauten:

1. Es ist keine Rücksicht auf Felder mit wechselnder Orientierung genommen, sodass die Materialmodelle wahrscheinlich unzweckmäßig sind.
2. Die Materialmodelle lassen sich häufig nur in begrenztem Umfang anpassen.
3. Selbst mit individuell eingestellten Materialmodellen sind die Lösungsmethoden darauf ggf. nicht ausgelegt und verursachen Artefakte.

„Viele höhere Programmiersprachen unterstützen das Programmierparadigma der Objektorientierung. Dabei soll die Struktur der Software die Struktur der Wirklichkeit wiedergeben, welche sie verwaltet, sodass der Programmierer Modelle erschafft, mit welchen er möglichst natürlich und intuitiv weiterarbeiten kann." [26, S. 5][1] [20] Das Paradigma ist in diesem Kapitel von besonderem Interesse, weil die Software reale Netzwerke und Felder simuliert und entsprechend werden sehr anschauliche Klassen entwickelt.

Die folgenden Erläuterungen sollen lediglich einen Überblick über die entwickelte Software geben und ihren Mehrwert für die vorliegende Arbeit hervorheben. Für eine ausführliche Dokumentation sei auf das Handbuch verwiesen (siehe Kapitel 11, S. 249).

[1] Die Autorin entschuldigt sich für die anlasslose und entsprechend unangebrachte geschlechtsspezifische Wortwahl für die Programmierenden in ihrer hier zitierten Bachelorarbeit.

4.1 Struktur

„Abstraktion ist der Schlüssel, um Komplexität zu verwalten." [28, S. 34] Die Berechnung von Netzwerken und Feldern ist komplex und wird erheblich anspruchsvoller, wenn sie auch noch erweiterbar sein soll. Deshalb widmet sich dieser Abschnitt ausschließlich den Abstraktionsebenen, welche die Struktur der Software bestimmen. Es wird eine ausgesprochen abstrakte Problemrepräsentation entworfen, welche sich in einen problemartunabhängigen Teil sowie einen für die Feldberechnung und einen für die Netzwerkberechnung separiert. Bereits hier wird intensiver Gebrauch von objektorientierter Programmierung gemacht, sodass mit Hinblick auf die Implementierung von rechenintensiven Lösungsalgorithmen de facto ausschließlich die Sprache C++ in Frage kommt.

4.1.1 Problemrepräsentation

Die Software benötigt eine innere Struktur und insbesondere eine zur Repräsentation einer lösbaren Netzwerk- oder Feldberechnungsaufgabe. Beide Problemarten werden zunächst auf Gemeinsamkeiten und Unterschiede untersucht, um daraus einen problemartunabhängigen Aufbau zu generieren. Anschließend werden die spezifischen Strukturen zur Feld- und Netzwerkrepräsentation entwickelt. Schließlich werden beide Problemarten so miteinander verknüpft, dass auch gemischte Problemarten mit der Software lösbar sind.

Problemartunabhängiger Aufbau

Die Software muss zwei verschiedene Arten von Problemen behandeln, die Netzwerk- und die Feldberechnung. *Tab. 4.1* stellt die Natur dieser beiden Problemarten gegenüber. Deren Eigenschaften lassen sich abstrahieren:

1. Die Raumdiskretisierung geschieht in Elementen.
2. Es existieren zwei Zustandsgrößen je Element.
3. Die Elemente können verschiedenes Verhalten bei dem Zusammenhang der beiden Zustandsgrößen aufweisen (verschiedene Bauteile bei Netzwerken, inhomogene Materialien bei Feldern).

Tab. 4.1: Vergleich der Strukturen von Netzwerken und Feldern

	Netzwerke	Felder
zugrundeliegende Gesetze	Kirchhoff'sche Gesetze	PDGL aus den Maxwellgleichungen
Zustandsgrößen	U, I	V, Φ
Raumdiskretisierung	Netzwerkelemente	Stützstellen(-elemente)
Zusammenhang zwischen den Zustandsgrößen	Elemente mit spezifischem Verhalten	Elemente mit spezifischem Verhalten
Bedingungen	Maschen, Knoten	Randbedingungen
Zeitabhängigkeit	elementspezifisch	stützstellen- / elementspezifisch

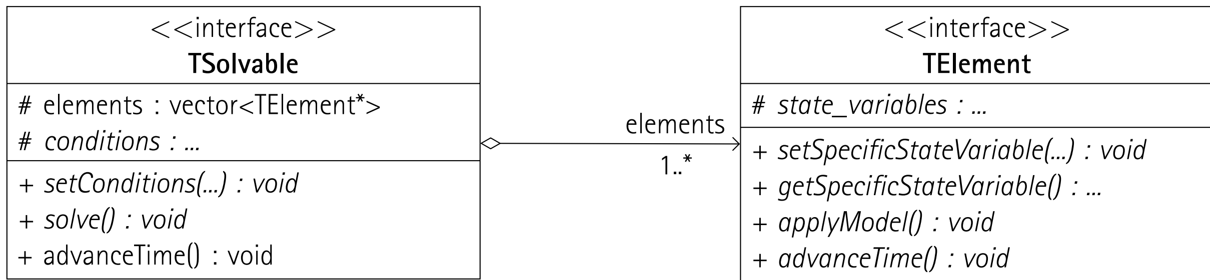


Abb. 4.1: Klassendiagramm der Problemrepräsentation. *TSolvable* hat Übersicht über Elemente und weitere Bedingungen und ist deshalb durch Aufruf von *solve() : void* eindeutig lösbar. *TElement* speichert den Zustand an einem Ort und kann aus gegebenen Zustandsvariablen durch Aufruf von *applyModel() : void* die anderen berechnen (bspw. Strom aus gegebener Spannung).

4. Es existieren Gesetze (Kirchhoff'sche Gesetze bei Netzwerken und PDGL bei Feldern), deren Natur von der Problemart abhängt.
5. Für die Anwendung der Gesetze sind problemspezifische Informationen notwendig (Maschen und Knoten bei Netzwerken und Randbedingungen bei Feldern).
6. Die Zeitabhängigkeit ist spezifisch für jedes einzelne Element.

Obwohl diese Erkenntnisse sehr abstrakt sind, reichen sie aus, um darauf aufbauend eine gemeinsame Struktur für Netzwerk- und Feldsimulationen zu entwickeln. Die Spezialisierung auf die beiden Problemarten geschieht durch Vererbung (siehe Abschnitt 4.1.2, S. 125, siehe Abschnitt 4.1.3, S. 127).

Für die Repräsentation eines Problems wird eine Klasse *TElement* definiert, welche die einzelnen Elemente darstellt. Sie kennt deren Zustandsgrößen und legt den Zusammenhang dazwischen fest, bspw. durch die charakteristische Kennlinie. Außerdem wird eine Klasse *TSolvable* entworfen, die einen Überblick über das Problem hat. Sie beinhaltet sämtliche Elemente und zusätzliche Bedingungen (Maschen und Knoten bei Netzwerken, Randbedingungen bei Feldern). Beide Klasse verfügen über die Methode *advanceTime() : void*. Einem Element wird so mitgeteilt, dass nun ein neuer Zeitpuntk betrachtet wird. Ein Object der Klasse *TSolvable* gibt bei einem Aufruf der Methode die Information an alle seine Elemente weiter. Dieser Aufbau ist in *Abb. 4.1* illustriert. Im Folgenden werden die beiden Klassen präsentiert.

Die Klasse *TElement*

Wie bereits in *Abb. 4.1* abgedruckt, weisen sowohl Netzwerkelemente als auch die Stützstellen der FDM einige Gemeinsamkeiten auf. Die Klasse *TElement*, deren Instanzen als Elemente bezeichnet werden, implementiert diese Gemeinsamkeiten. Das Klassendiagramm ist in *Abb. 4.2* präsentiert. Der Datentyp *T* ist dabei als Templateparameter spezifiziert und sollte skalare Dezimalzahlen repräsentieren können.[2] Die Implementierung umfasst

[2]Auf die Definition besonderer Datentypen wird in dieser Arbeit verzichtet, zumal moderne Consumer-Prozessoren von Comptutern, engl. Central Processing Units (CPUs) Hardwareunterstützung für 64-bit-Fließkommazahlenberechnung bieten.

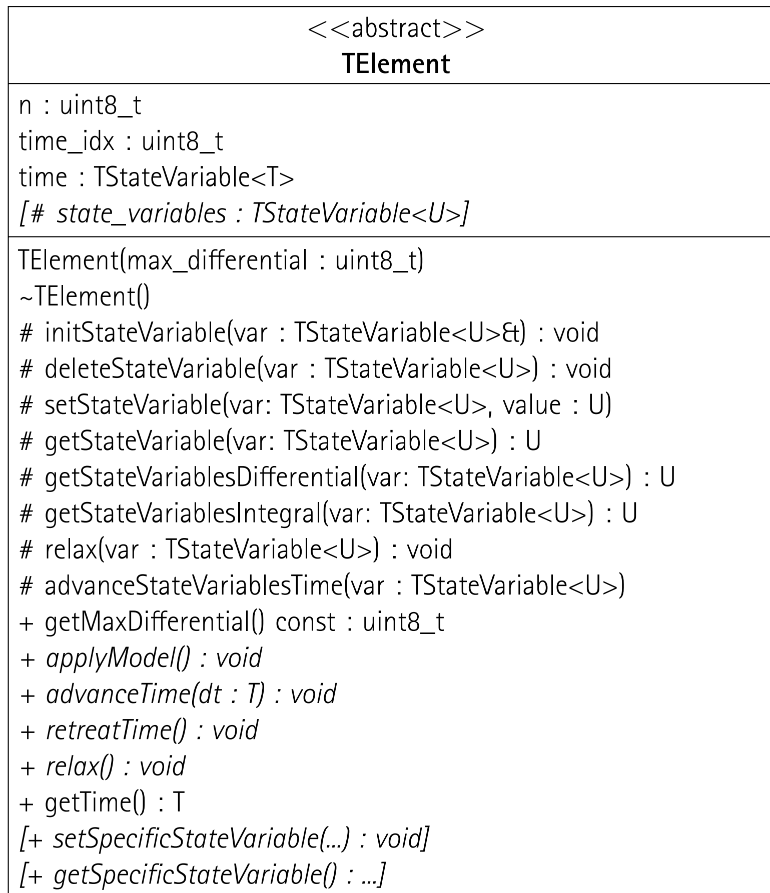


Abb. 4.2: Klassendiagramm von *TElement*. Attribute zur Verwaltung aller Zustandsvariablen und die Zustandvariable *time*, erweiterbar durch abgeleitete Klassen. Konstruktor zur Speicherverwaltung der Zustandsvariablen. Geschützte Methoden als Quasi-Methoden der fiktiven Klasse *TStateVariable*. Öffentliche Methoden zur Veränderung der Zeit und Anwendung eines elementspezifischen Modells. Erweiterbar um *get*- und *set*-Elementfunktionen der abgeleiteten Klassen.

die Bereiche:

1. Zustandsvariablen und Zeit
2. Anwendung eines (elementspezifischen) Modells

Zustandsvariablen sind zunächst als gewöhnliche Attribute von Klassen zu verstehen, die von *TElement* erben. Die Methoden *initStateVariable* und *deleteStateVariable* reservieren Speicher oder geben ihn wieder frei und müssen entsprechend nur im Konstruktor bzw. Destruktor aufgerufen werden. Mit den Methoden *getStateVariable* und *setStateVariable* kann der Wert der Variablen ausgelesen und manipuliert werden. Während diese Funktionalität für gewöhnlich in jeder Klasse einzeln implementiert würde, können von Zustandsvariablen außerdem mit *getStateVariablesDifferential* und *getStateVariablesIntegral* das zeitliche Differential und das zeitliche Inkrement bestimmt werden.

Die Implementierung ist denkbar einfach: Der Templatetyp *TStateVariable* wird als lediglich ein Zeiger auf einen als Templateparameter festgelegten Typ definiert:

```
template<typename U>
using TStateVariable = U*;
```

Für jede Zustandsvariable werden bei der Initialisierung genügend Felder für einen Ringspeicher reserviert, um ihre Werte auch an einem oder mehreren vergangenen Zeitpunkten zu kennen. Die Zustandsvariable *time* speichert die Zeitpunkte in Sekunden. Mit jedem Aufruf von *advanceTime)* werden die Ringspeicher aller Zustandsvariablen um ein Element „weitergedreht". Damit auch bspw. zentrale Differenzenquotienten berechnet werden können, kann auch durch den Aufruf von *retreatTime* einen (oder mehrere) Zeitschritte zurückgegangen werden. Der Index des aktuellen ausgewählten Feldes aller Zustandsvariablen ist in *time_idx* festgehalten. Mit *relax* werden alle Zustandsvariablen auf null gesetzt und deren Historie vergessen.

Abgeleitete Klassen müssen festlegen, wie viele Zeitpunkte für ihre Zustandsvariablen benötigt werden. Dazu reicht, dass sie dem Konstruktor von *TElement* mitteilen, welches die höchste Ableitung bzw. das höchste Integral ist (bspw. zwei für Ableitungen zweiter Ordnung), die sie von ihren Zustandsvariablen berechnen möchten. Zusammen mit dem Templateparameter *method* wird daraus die Anzahl der benötigten Felder jedes Ringspeichers bestimmt (siehe Abschnitt 1.3.1, S. 35).

Weil auf ein Register für alle Zustandsvariablen eines Elements verzichtet wird, muss jede abgeleitete Klasse, die eigene Zustandsvariablen definiert, die Methoden *advanceTime*, *retreatTime* und *relax* reimplementieren und *initStateVariable* und *deleteStateVariable* im Konstruktor bzw. Destruktor aufrufen.

Zuletzt soll noch kurz erwähnt werden, warum keine eigene Klasse für *TStateVariable* entwickelt ist, obwohl sämtliche darauf anwendbaren Funktionen den Charakter von Elementfunktionen haben.[3] Jede Zustandsvariable greift auf Eigenschaften des Elements zu, der sie angehört (Zeit, Größe des Ringspeichers). Die Eigenschaften können entweder bei der Konstruktion der Instanzen übergeben werden oder jede Instanz kann einen Zeiger auf ihr zugehöriges Element besitzen. Beide Implementierungen erhöhen den Speicher- und Rechenaufwand bei zeittransienten Berechnungen nicht unerheblich, weshalb hier eine funktionale statt einer objektorientierten Implementierung gewählt ist. Eine Umstellung des Codes wäre aber mit vertretbarem Aufwand möglich.

Die **Anwendung eines Modells** geschieht durch die von *TElement* abgeleiteten Klassen in drei Schritten:

1. Festlegung mancher Zustandsvariablen
2. Berechnung der weiteren Zustandsvariablen durch ein elementspezifisches Modell
3. Bereitstellung eines konsistenten Zustandes

Wie bereits zuvor erläutert, können abgeleitete Klassen die Zustandsvariablen deklarieren. Sie sind außerdem dazu angehalten, *get*- und *set*-Methoden zu definieren, um umgeleiteten Klassen von *TSolvable* ein einfaches

[3] Elementfunktionen bzw. Methoden nehmen als ersten (in Programmiersprachen häufig verdeckten) Parameter einen Zeiger auf die Klasse, auf deren Instanzen sie angewandt werden. Danach folgt die reguläre Parameterliste. Die hier implementierten Funktionen nehmen als ersten Parameter den Zeiger (bzw. *initStateVariable* nimmt eine Referenz des Zeigers) entgegen, womit sie sich nur geringfügig von Methoden einer fiktiven Klasse *TStateVariable* unterscheiden.

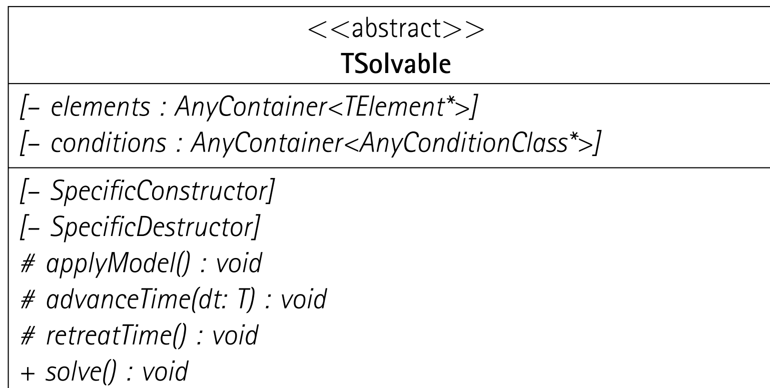


Abb. 4.3: Klassendiagramm von *TSolvable*. Attribute und Konstruktor/Destruktor werden erst in abgeleiteten Klassen deklariert. Geschützte Methoden zur Verwaltung der Elemente. Öffentliche Methode *solve* zur Bestimmung der Lösung.

Interface zu bieten. Weil diese Variablen allerdings sehr problemartspezifisch sind, können keine rein virtuellen Methoden deklariert werden.[4]

Die Klasse *TSolvable*

Die Komposition von Elementen und Bedingungen geschieht in der Klasse *TSolvable*. Diese Informationen lassen sich eindeutig als ein physikalisches Problem interpretieren, weshalb die Klasse außerdem einen Algorithmus zur Bestimmung der Lösung beinhaltet. Das Klassendiagramm ist in *Abb. 4.3* gezeigt.

Die Bedingungen sind so stark problemartspezifisch, weshalb sie und ihre Container von abgeleiteten Klassen deklariert werden. Die Entscheidung, die Elemente und ihre Container ebenfalls vollständig in abgeleitete Klassen zu verlegen, ist dem Umstand geschuldet, dass die Software auch Feldsimulationen beherrschen soll. Die dafür benötigte Rechenleistung kann ggf. monumental werden, sodass die Wahl einer geeigneten Speicherverwaltung eine bedeutsame Einsparung von Zeit und Energie gegenüber einer abstrakteren Implementierung in der Basisklasse bedeutet. Weil die Elemente in den abgeleiteten Klassen deklariert sind, müssen die Funktionen zur Verwaltung ebenfalls dort definiert werden. Sie müssen bspw. im Konstruktor erzeugt und im Destruktor freigegeben werden. Später wird auch eine flexiblere Alternative vorgestellt (siehe Abschnitt 4.1.3, S. 127). Zur Verwaltung der Elemente müssen außerdem die Methoden *applyModel*, *advanceTime* und *retreatTime* implementiert werden, welche lediglich die gleichnamigen Funktionen auf allen ihren Elementen aufrufen.

Ein großer Anteil des Codes entfällt in abgeleiteten Klassen auf die Implementierung der Methode *solve*. Sie bestimmt aus den gegebenen Bedingungen und dem Verhalten der Elemente eine (eindeutige) Lösung. Die Verfahren sind problemartspezifisch und werden sind zu umfangreich, um im Rahmen dieses Dokument beschrieben zu werden. Der Quellcode der Software ist entsprechend dokumentiert (siehe Kapitel 10, S. 173).

[4] Wie in *Tab. 4.1* festgestellt, weisen die Elemente in Netzwerken und Feldern je zwei Zustandsvariablen auf. Diese Gemeinsamkeit wird aber als zu abstrakt betrachtet, um sinnvoll virtuelle Methoden zu deklarieren.

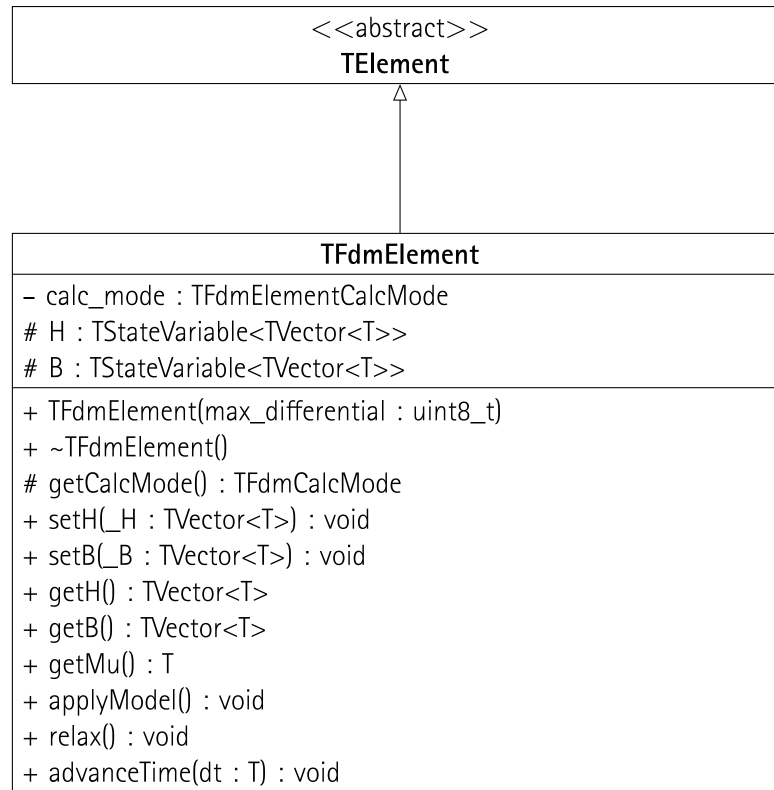


Abb. 4.4: Klassendiagramm von *TFdmElement*. Feldgrößen als Zustandsvariablen. *calc_mode* erinnert, welche davon festgelegt ist und welche berechnet wird. Konstruktor und Destruktor zur Verwaltung der Zustandsvariablen. Öffentliche *get*- und *set*-Methoden, auch zur Berechnung der Permeabilität. Öffentliche Implementierung der virtuellen Funktionen aus *TElement*.

4.1.2 Felder

Die Simulation von magnetischen Feldverteilungen innerhalb einer Anordnung ist einfach aufgebaut: Die Klasse *TFdm* erzeugt ein gleichmäßiges Netz, welches an seinen Knoten mit Objekten der Klasse *TFdmElement* populiert wird. Die Elemente legen das intrinsische Verhalten des Materials an jedem Knotenpunkt fest und die Netzklasse berechnet anhand dessen die gesamte Feldverteilung. Weil die FDM einfacher zu implementieren ist als die FEM und weil sie bei den später untersuchten Geometrien (siehe Abschnitt 5.2.2, S. 144) sogar einen Vorteil beim Rechenaufwand bietet, wird hier auf die FEM verzichtet.

Die Klasse *TFdmElement*

Jede Stützstelle einer Feldsimulation mit der FDM wird durch ein Objekt der abstrakten Klasse *TFdmElement* repräsentiert. Ihre Zustandsvariablen sind die Feldgrößen und abgeleitete Klassen implementieren den Zusammehang zwischen ihnen. Die Klasse stellt ein einheitliches Interface für die das Problem überblickende Klasse bereit. Das Klassendiagramm ist in *Abb. 4.4* festgehalten.

Die Zustandsgrößen *H* und *B* sind Elemente der Klasse *TStateVariable<TVector>* und speichern jeweils den Wert des H- und des B-Feldes an der durch das Element repräsentierten Stützstelle. *TVector* kann mehrdimensionale Größen speichern, wobei der Datentyp der einzelnen Einträge als Templateparameter festgelegt werden kann. *TFdmElement* gibt dafür den Templateparameter *T* von *TElement* einfach weiter.

Die Zugriffsfunktionen sind als öffentlich markiert, um der Überblicksklasse (hier *TFdm*, abgeleitet von *TSolvable*, s. u.) vollen Zugriff auf den Zustand zu gewähren. Wie bereits bei der Vorstellung von *TElement* (s. o.) erwähnt, legt die Überblicksklasse eine der beiden Zustandsvariablen fest, bevor die Methode *applyModel()* die andere Zustandsgröße so berechnen muss, dass ein konsistenter Zustand erreicht ist. Bei jedem Aufruf von *setH* oder *setB* wird deshalb dass Attribut *calc_mode* des Aufzählungsdatentyps *TFdmElementCalcMode* manipuliert, sodass beim Aufruf von *applyModel* bekannt ist, welche Zustandsvariable (zuletzt) vorgegeben ist und entsprechend die andere berechnet werden kann. Gemischte Berechnungen, bei denen bspw. eine Komponente des H-Feldes und die anderen Komponenten des B-Feldes vorgegeben sind, sind nicht implementiert.

Als Besonderheit bei der Feldberechnung wird außerdem die Permeabilität μ benötigt (siehe Abschnitt 1.2.1, S. 31), bereitgestellt durch die öffentliche Elementfunktion *getMu*. Obwohl sie sich aus den Feldgrößen eindeutig bestimmen lässt, ist eine Berechnung nahe der Koordinatenachsen, insbesondere nahe des Ursprungs, der Magnetisierungskennlinie, mit einem Programmieraufwand verbunden, der sich geschickterweise in einer Methode von *TFdmElement* unterbringen lässt. Sowohl *getMu* als auch *applyModel* sind bereits in *TFdmElement* implementiert und greifen bei der Ausführung auf die jeweils andere Funktion zu. Abgeleitete Klassen müssen also nur eine dieser beiden Funktionen implementieren, allerdings wird zum Sparen von Rechenleistung empfohlen, beiden Funktion spezifisch umzusetzen.

Bei einem Vergleich mit der zugrundeliegenden DGL (siehe Abschnitt 1.2.1, S. 31) fällt auf, dass mit der Implementierung von *getMu* lediglich nichtlineare, anisotrope Materialien modelliert werden können, solange sie keine Verluste verursachen. In dem Fall müsste der B-Achsenabschnitt ebenfalls definiert sein. Im Rahmen der Simulationen, die in dieser Arbeit durchgefürt werden (siehe Abschnitt 5.2.4, S. 149), sind keine verlustbehafteten Materialen verwendet, sodass die Implementierung wegen der knappen Bearbeitungszeit aufgeschoben wird.

Obwohl bei der Herleitung der zugrundeliegenden DGL für die Feldberechnung (siehe Abschnitt 1.2.1, S. 31) das magnetische Skalarpotential ϕ_m verwendet wird, handelt es sich dabei um keine Zustandsvariable der Elemente, sondern um eine fiktive Rechengröße. Außerdem unterscheidet sich das Netz aus Elementen zur Diskretisierung des Raumes vom Netz aus Stellen, an welchen ϕ_m berechnet wird. Die Details würden an dieser Stelle zu weit fh�ren und sind den Kommentaren des Quellcodes zu entnehmen (siehe Kapitel 9, S. 171).

Die Klasse *TFdm*

Ein Überblick über die Elemente und Randbedingungen und ein Lösungsverfahren, wie sie in abstrakter Form von *TSolvable* gefordert sind, werden in der Klasse *TFdm* implementiert. Sie enthält außerdem weitere Attribute

und Methoden, um die Eindeutigkeit der Lösung zu gewährleisten, Templateparameter zur Festlegung der Diskretisierungsfeinheit, Hilfsattribute und Methoden zur Speicherung des Ergebnis. Konkrete Problemstellungen werden von abgeleiteten Klassen implementiert. Das Klassendiagramm ist in *Abb. 4.5* abgebildet.

Die Klasse betrachtet eine rechteckige, ebene Fläche, die in $nW \times nH$ Elemente unterteilt und $(nW \cdot dx) \times (nH \cdot dy)$ Quadratmeter groß ist. Die Anzahlen *nW* und *nH* sind Templateparameter, die Differenzen *dx* und *dy* sind einmalig vom Konstruktor festgelegte Attribute. Die Elemente sind als zweidimensionales Array von Zeigern auf *TFdmElement* implementiert, sodass abgeleitete Klassen beliebige Elemente einsetzen können. Die Hilfsarrays *phim*, *new_phim* und *mu_cache* werden nur bei der Anwendung des Lösungsalgorithmus benötigt, sind aber trotzdem als Attribute deklariert, um Compileroptimierungen zu nutzen und Rechenleistung zu sparen.

Die Randbedingungen sind als vier eindimensionale Arrays der Klasse *TFdmBoundaryCondition* umgesetzt. Die Methode *bc(y,x)* gibt eine Referenz auf die zu einem Randelement an der Koordinate (y|x) zurück. Falls die Koordinate nicht auf dem Rand des zu betrachtenden Gebiets liegt, wird eine Ausnahme geworfen. Randbedingungen sind im Allgemeinen von der (abgeleiteten) Klasse zu setzen, die eine konkrete Problemstellung implementiert und können nach jeder Lösung geändert werden.

Die Methode *solve* löst das Problem eindeutig. In folgenden Fällen wirft sie eine Ausnahme:

1. Es existiert keine eindeutige Lösung.
2. Das Lösungsverfahren konvergiert nicht.
3. Das Lösungsverfahren konvergiert nicht in absehbarer Zeit.
4. Ein Zwischenergebnis ist extrem unplausibel.
5. Eine berechnete Permeabilität ist extrem unplausibel.

Die Strategie der Methode und ihrer Hilfsfunktion *solveRecursively* ist aufgrund zahlreicher Optimierungen kompliziert und nicht Thema dieses Dokuments. Der Code ist kommentiert. (siehe Kapitel 9, S. 171).

Das Ergebnis der Feldberechnung kann entweder von anderen (abgeleiteten) Klassen interpretiert oder einfach gespeichert werden. Die Methoden *printBmp*, *printTex* und *saveCsv* bieten diese Funktionalität. Ihre Implementierungen sind nicht Gegenstand dieses Kapitels und können im kommentierten Code nachgeschlagen werden (siehe Kapitel 9, S. 171).

4.1.3 Netzwerke

Sinnvollerweise enthält die Software Strukturen und Algorithmen zur Simulation von Netzwerken. Diese beinhalten insbesondere die Klasse *TSpice*, welche ein komplettes Netzwerk inklusive seiner Elemente und deren Verbindungen (Knoten, Maschen) repräsentiert. Ihre Instanzen sind mit Objekten der Klasse *TSpiceSingleElement* populiert, welche elektrische oder magnetische Zweipole darstellen. Außerdem ist eine Klasse *TSpiceMultiElement* entwickelt, welche $2b$-Pole repräsentiert (siehe Abschnitt 1.1, S. 6).

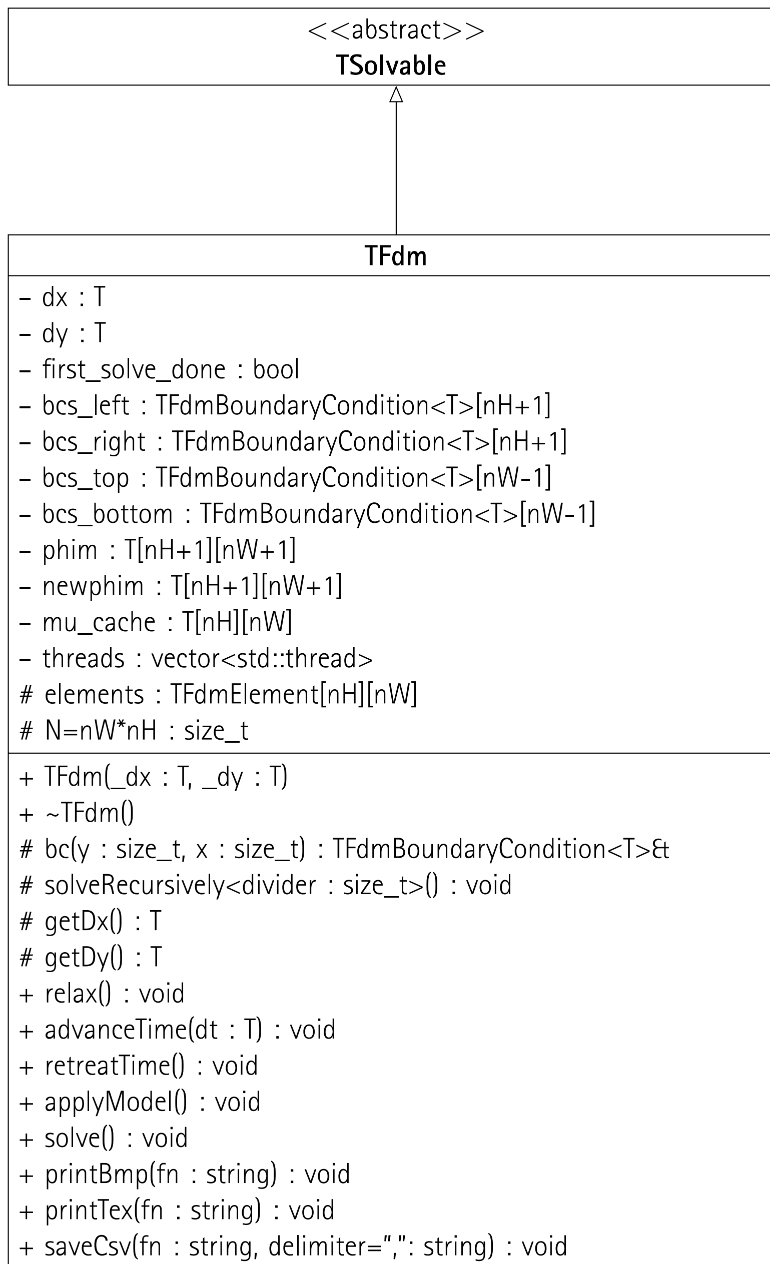


Abb. 4.5: Klassendiagramm von *TFdm*. Templateparameter definieren Anzahl Elemente in Höhe und Breite (*nH*, *nW*). Attribute legen die Größe, Randbedingungen und Elemente fest. Zweidimensionale Hilfsarrays aus Performanzgründen als Attribute statt lokalen Variablen deklariert. Container für Multithreading. Konstruktor und Destruktor zur Elemente-, Randbedingungen- und Speicherverwaltung. Geschützte Zugriffsfunktionen. Öffentliche Implementierungen von virtuellen Methoden aus *TSolvable* und geschützte Hilfsfunktion *solveR-ecursively*. Öffentliche Methoden zum Speichern des Ergebnis als CSV, Windows Bitmap (BMP), etc.

Im Folgenden sei nicht weiter auf die Netzwerkrepräsentation eingegangen, weil diese im Rahmen des Versuchs (siehe Kapitel 6, S. 139) keine Anwendung findet. Die Dokumentation des bereits fertiggestellten Quellcodes ist dem dazugehörigen Handbuch zu entnehmen (siehe Kapitel 11, S. 249).

4.2 Modelle

Im Rahmen dieser Arbeit wird nur ein Bruchteil der implementierten Funktionalität verwendet (siehe Abschnitt 4.3, S. 133). Entsprechend werden auch nur wenige, grundlegende Modelle entwickelt. Diese umfassen Netzwerkelemente, Geometrien und Materialien und sind in *Tab. 4.2* dargestellt. Für die später durchgeführten Versuche (siehe Kapitel 6, S. 139) sind keine Modelle von kompletten Netzwerken notwendig, weshalb auch keine implementiert sind. Der Verständlichkeit halber weist die Autorin darauf hin, dass sogar von den Modellen nicht alle im Rahmen dieser Arbeit verwendet werden. Insbesondere können in der kurzen Bearbeitungszeit keine Netzwerk-, sondern nur Feldsimulationen fertiggestellt werden.

Weil die meisten Modelle selbsterklärend sind, sei an dieser Stelle lediglich auf die beiden nichtlinearen Materialmodelle eingegangen. Die mit ihnen modellierten Magnetisierungskennlinien sind beide in *Abb. 5.9c* illustriert.

Nichtlineares, isotropes, verlustfreies magnetisches Material mit Umsetzungstabelle

Die Klasse *TFdmElementNonlinearIsotropicLossfreeLut* modelliert ein nichtlineares, isotropes, verlustfreies magnetisches Material, indem es einige Punkte auf der charakteristischen Kennlinie in einer LUT speichert und zwischen den Punkten eine lineare Interpolation durchführt. Es greift auf die gleiche Funktion *interpolate* zurück, welche intern auch vom Werkzeug zur Glättung der Quantisierung verwendet wird (siehe Abschnitt 2.3.2, S. 94). Allerdings wird außerhalb des von der LUT abgedeckten Bereichs ein konstanter Zusammenhang $B(H) = \text{const.}$ angenommen.

Bei der Implementierung bestehen zwei verschiedene Möglichkeiten, die LUT zu bilden. Deshalb weist die Klasse einen Templateparameter vom Typ *TFdmElementNonlinearIsotropicLossfreeLutType* auf, der festlegt, ob die Daten für die Tabelle unverändert aus einer Datei gelesen werden sollen (*luttMagCurve*) oder ob ein Algorithmus eine Messung analysieren soll (luttMeasurement). Die Klasse weist außerdem einen Templateparameter auf, welcher festlegt, in welcher Datei die Tabelle oder die zu analysierenden Daten gespeichert sind.

Falls die LUT aus einer Datei gelesen werden soll (*luttMagCurve*), erwartet der Konstruktor der Klasse CSV in der Datei, welche zu verschiedenen Beträgen des H-Feldes die entsprechenden Beträge des B-Feldes zuordnen. Die Bildung der LUT ist trivial.

Die Generierung der LUT aus einer Messung (luttMeasurement) ist komplizierter. Der Konstruktor erwartet erneut CSV in der spezifizierten Datei. Die Spalten „h“ und „b“ werden eingelesen, allerdings können diese inkonsistente Daten enthalten. Dies ist insbesondere dann der Fall, wenn mehrere Perioden einer zeitlichen Schwingung aufgenommen sind und diese aufgrund von Messfehlern geringfügig voneinander abweichen. Eine ggf. vorhandene Hysterese erschwert die Auswertung ebenfalls. Der Algorithmus soll an dieser Stelle nicht thematisiert werden, er ist im Quellcode dokumentiert.

Die LUT ist ein statisches Element der Klasse, sodass für jede Templateinstanziierung von *TFdmElementNonlinearIsotropicLossfreeLut* nur eine LUT angelegt wird, die von sämtlichen Objekten (FDM-Elementen) geteilt

Tab. 4.2: Implementierte Modelle. Modelle von Zweipolen für die Netzwerkberechnung, Geometrien und Materialien für die Feldberechnung. Liste der Modelle ist durch Erbschaft von den genannten Basisklassen (und von *TSpice*) erweiterbar.

Bezeichnung	Typ (Basisklasse)	Umgebung	Inhalt und Zweck
TSpiceSingleElementU	Zweipol (*TSpiceSingleElement*)	Netzwerke	Konstantspannungsquelle
TSpiceSingleElementUAC	Zweipol (*TSpiceSingleElement*)	Netzwerke	Wechselspannungsquelle
TSpiceSingleElementR	Zweipol (*TSpiceSingleElement*)	Netzwerke	linearer Widerstand
TSpiceSingleElementC	Zweipol (*TSpiceSingleElement*)	Netzwerke	lineare Kapazität
TSpiceSingleElementL	Zweipol (*TSpiceSingleElement*)	Netzwerke	lineare Induktivität
TFdmSquare	Geometrie (*TFdm*)	Felder	Quadrat
TFdmSc	Geometrie (*TFdm*)	Felder	„Single-Cross" (siehe Abschnitt 5.2.2, S. 145)
TFdmElementAir	Material (*TFdmElement*)	Felder	idealisierte Luft bzw. Vakuum
TFdmElementLinearIsotropicLossfree	Material (*TFdmElement*)	Felder	lineares, isotropes, verlustfreies magnetisches Material
TFdmElementNonlinearIsotropicLossfree	Material (*TFdmElement*)	Felder	nichtlineares, isotropes, verlustbehaftetes magnetisches Material (analytisch)
TFdmElementNonlinearIsotropicLossfreeLut	Material (*TFdmElement*)	Felder	nichtlineares, isotropes, verlustbehaftetes magnetisches Material (LUT)

wird.

Nichtlineares, isotropes, verlustfreies magnetisches Material mit analytischer Kennlinie

Die Nutzung einer LUT (s. o.) erfordert einen erheblich größeren Rechenaufwand als die Nutzung einer analytischen Funktion, welche die Magnetisierungskennlinie annähert. In [31] ist ein Überblick über einige verschiedene solcher Funktionen gegeben.

Die Klasse *TFdmElementNonlinearIsotropicLossfree* implementiert die folgende Funktion:

$$|\boldsymbol{B}| = B_{\text{sat}} \cdot \left(1 - e^{-\frac{\mu_0(\mu_{\text{r}} - \mu_{\text{r,sat}})}{B_{\text{sat}}} \cdot |\boldsymbol{H}|}\right) + \mu_0 \mu_{\text{r,sat}} |\boldsymbol{H}|; \tag{4.1}$$

Dabei wird der Ansatz verfolgt, dass die differentielle relative Permeabilität von μ_{r} nahe des Koordinatenursprungs der Magnetisierungskennlinie auf $\mu_{\text{r,sat}}$ im Bereich der Sättigung abfällt und dass dieser Vorgang exponentiell geschieht. Falls $\mu_{\text{r,sat}}$ viel kleiner ist als μ_{r}, was häufig der Fall ist, lässt sich deutlich eine Sättigungsflussdichte von B_{sat} beobachten.

Selbstverständlich zeigt der Vektor des B-Feldes dabei stets in die gleiche Richtung wie der des H-Feldes (siehe Abschnitt 1.1.2, S. 11).

4.3 Erweiterbarkeit und Wiederverwendbarkeit

Wie bereits eingangs des Kapitels erwähnt, soll die Software erweiterbar und wiederverwendbar sein. So können spezifische Probleme nicht nur gelöst werden, sondern die Lösungsstrategien können festgehalten und flexibel angepasst werden. Bspw. kann ein Element in einer bestehenden Netzwerksimulation einfach gegen eines ausgetauscht werden, dessen Verhalten durch eine Feldberechnung determiniert ist.

In diesem Abschnitt wird zunächst veranschaulicht, wie diese Erweiterbarkeit und Wiederverwendbarkeit mithilfe der Designmaxime der „strukturellen Kompatibilität" erreicht ist. Anschließend wird die Klasse *TMagneticComponent* beschrieben, welche speziell für den Versuch (siehe Kapitel 6, S. 139) entwickelt ist. Schließlich werden einige Werkzeuge entwickelt, welche auf die zuvor beschriebene Software zugreifen. Das Kapitel ist damit abgeschlossen und die Werkzeuge finden später Anwendung (siehe Kapitel 6, S. 139).

4.3.1 Strukturelle Kompatibilität

Die entwickelten Problemrepräsentation weisen eine hohe Kompatibilität auf, welche die die Verknüpfung von verschiedenartigen Simulationen sehr einfach gestaltet (bspw. Netzwerke mit Feldern oder Abhängigkeiten von Systemen aus anderen Domänen der Physik). Mehrere problemrepräsentierende Klassen können von einer weiteren Klasse geerbt werden,[5] sodass diese ein Problem in ein anderes umformuliert und es somit lösen kann, ohne dass der aufrufende Code die Natur des Problems mit einer geeigneten Lösungsstrategie assoziieren muss.

Bspw. kann eine Klasse von einem Netzwerkelement (*TSpiceSingleElement*) und von einer Feldsimulation (*TFdm*) erben. Wenn es als ersteres verwendet wird, ist die Charakteristik des Netzwerkelements durch die Feldsimulation bestimmt. Diese Strategie ist in der Klasse *TOddGeometryInductance* verfolgt, welche in *Abb. 4.6* abgedruckt ist. In dem Beispiel verhält sich die Klasse gegenüber dem Netzwerk, in welches sie eingebunden ist (Klasse *TSpice*), wie ein gewöhnlicher Zweipol, indem sie die Methoden der Klasse *TSpiceSingleElement* erbt. Ihre Implementierung der Methode *applyModel* beinhaltet aber keine konkrete Berechnungsvorschrift, sondern sie nutzt die festgelegte Netzwerkgröße (im Beispiel der elektrische Strom), um daraus die Randbedingungen für eine Feldsimulation zu definieren, welche sie ebenfalls beinhaltet, weil sie von der Klasse *TFdm* erbt. Sie überlässt das Lösen der Methode *solve* der Feldberechnung und identifiziert schließlich die zu berechnende Größe (im Beispiel die elektrische Spannung) anhand der berechneten Feldgrößen.

Ein weiterer Vorteil der Kompatibilität ist die Möglichkeit, große Teile von Code wiederzuverwenden, ohne dabei versehentliche Fehler zu riskieren, die beim Kopieren von Code möglich wären.

[5] Es sei angemerkt, dass solche programmiertechnischen „Kunststücke" nicht in jeder Maschinensprache implementierbar sind. C++ stellt dafür eine ausgezeichnete Wahl dar.

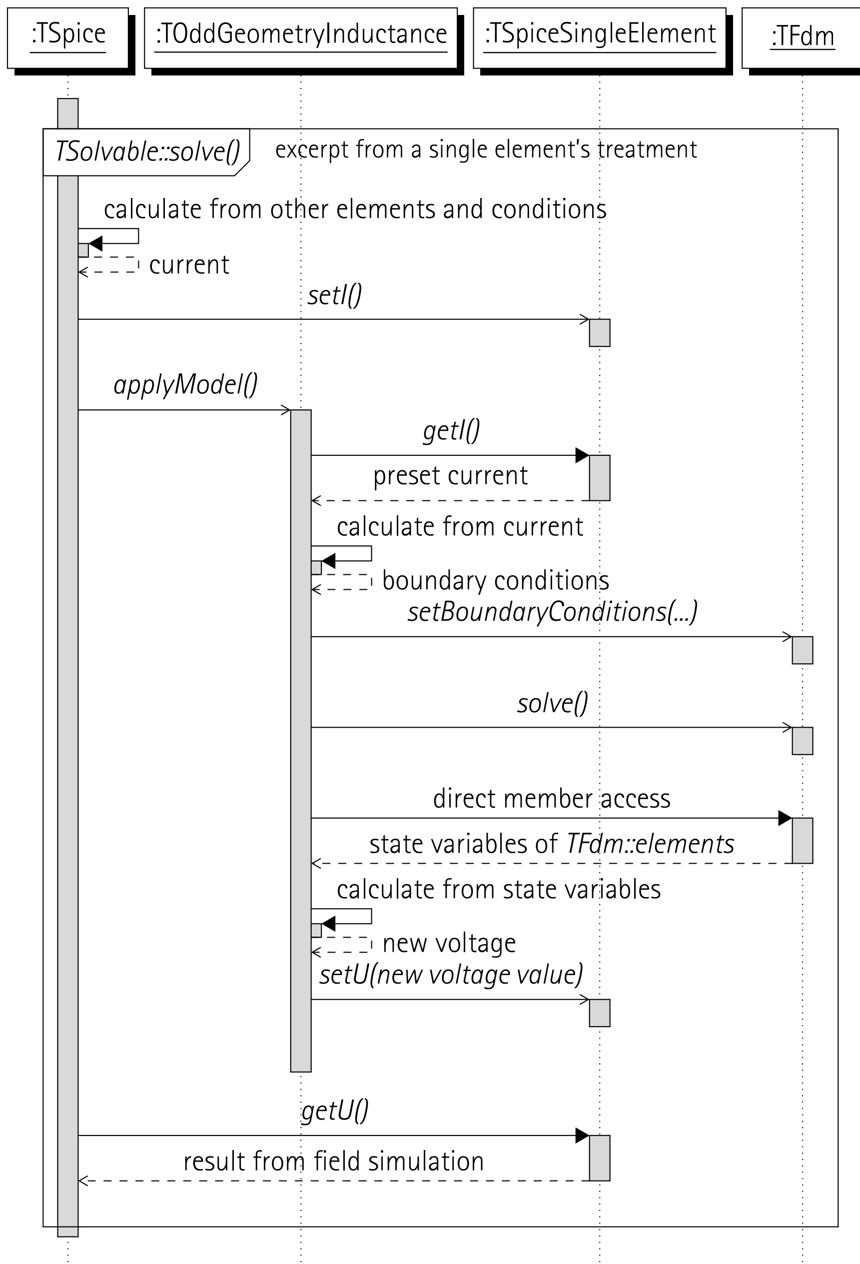


Abb. 4.6: Berechnung eines Netzwerkelementes durch angebundene Feldsimulation. Behandlung eines einzigen Netzwerkelementes in einem einzigen Iterationsschritt zu einem einzigen Zeitpunkt. *TOddGeometryInductance* erbt von *TSpiceSingleElement* und *TFdm*, *TSpice* komponiert Instanzen von *TOddGeometryInductance* (und weiteren Netzwerkelementklassen). Lösungsalgorithmus setzt zunächst Bedingungen, ruft dann *TSpiceSingleElement::applyModel()* auf und ruft Ergebnis ab. *TSpiceSingleElement::applyModel()* wendet die selbe Strategie an und ruft dafür *TFdm::solve()* auf. Zugriff von *TFdm::solve()* auf noch feinerere Objekte (*TElement*,...) hier nicht mit abgebildet.

4.3.2 Die Klasse *TMagneticComponent*

Die Software ist aufgrund ihrer strukturellen Kompatibilität sehr vielseitig einsetzbar. Weil die durchzuführenden Versuche sehr einfach und sehr ähnlich aufgebaut sind (siehe Abschnitt 5.2, S. 141), wird eine Klasse entworfen, welche speziell für diese Versuche geeignet ist. Das Klassendiagramm ist in *Abb. 4.7* präsentiert. Ihr Aufbau ist an den von Netzwerkelementen der Klasse *TSpiceSingleElement* angelehnt und ihr Verhalten ist identisch. Die Methode *applyModel* muss von abgeleiteten Klassen implementiert werden und soll die nicht vorgegebene Zustandsvariable anhand der vorgegebenen (und ggf. zeitlichen Differentialen und Integralen beider) berechnen. Welche Zustandsvariable festgelegt ist und welche berechnet wird, ist durch das Attribut *calc_mode* definiert.

Es wird außerdem eine Klasse *TMagneticComponentOrtho* abgeleitet, welche lediglich einen weiteren Zweipol hinzufügt. Ihr Name deutet bereits auf die Verwendung bei der orthogonalen Magnetisierung in, allerdings ist sie für beliebige magnetische Vierpole verwendbar.

4.3.3 Werkzeuge

Aufgrund der hohen strukturellen Kompatibilität (s. o.) der einzelnen Softwarekomponenten sind die Möglichkeiten zur Kombination nahezu unbegrenzt, obgleich in der späteren Verwendung (siehe Kapitel 6, S. 139) lediglich ein Bruchteil der implementierten Funktionalität benötigt wird. Dabei werden einzelne Komponenten der Software so miteinander kombiniert, dass sich ein Werkzeug ergibt, mithilfe dessen spezifische Simulationsaufgaben erfüllen lassen.

Im Folgenden werden knapp die Werkzeuge vorgestellt, welche im Rahmen des Versuchs (siehe Kapitel 6, S. 139) und für nachfolgende Arbeiten besondere Bedeutung haben.

Magnetisierungskennlinie von Materialien

Die globale Funktion *MagCurve* erwartet als Parameter einen Materialtyp und einen Bereich an H-Feldamplituden, innerhalb dessen die Magnetisierungskennlinie des Materials aufgenommen werden soll. Es speichert die Kennlinie als CSV in einer spezifizierbaren Datei. Die benötigte Ausführungsdauer ist maßgeblich durch die Geschwindigkeit des Dateizugriffs bestimmt.

Ein Beispiel für eine aufgenommene Mangetisierungskennlinie ist in *Abb. 5.9c* gezeigt.

Klemmenverhalten magnetischer Komponenten

Wie bereits zuvor beschrieben, enthält nicht jede Anordnung von magnetischem Material ein homogenes Feld, sodass eine kompliziertere Modellierung notwendig ist, bspw. als Netzwerk- oder als Feldsimulation (siehe Abschnitt 1.1, S. 6). Weil das Klemmenverhalten einer solchen Modellierung in einer von *TMagneticComponent*

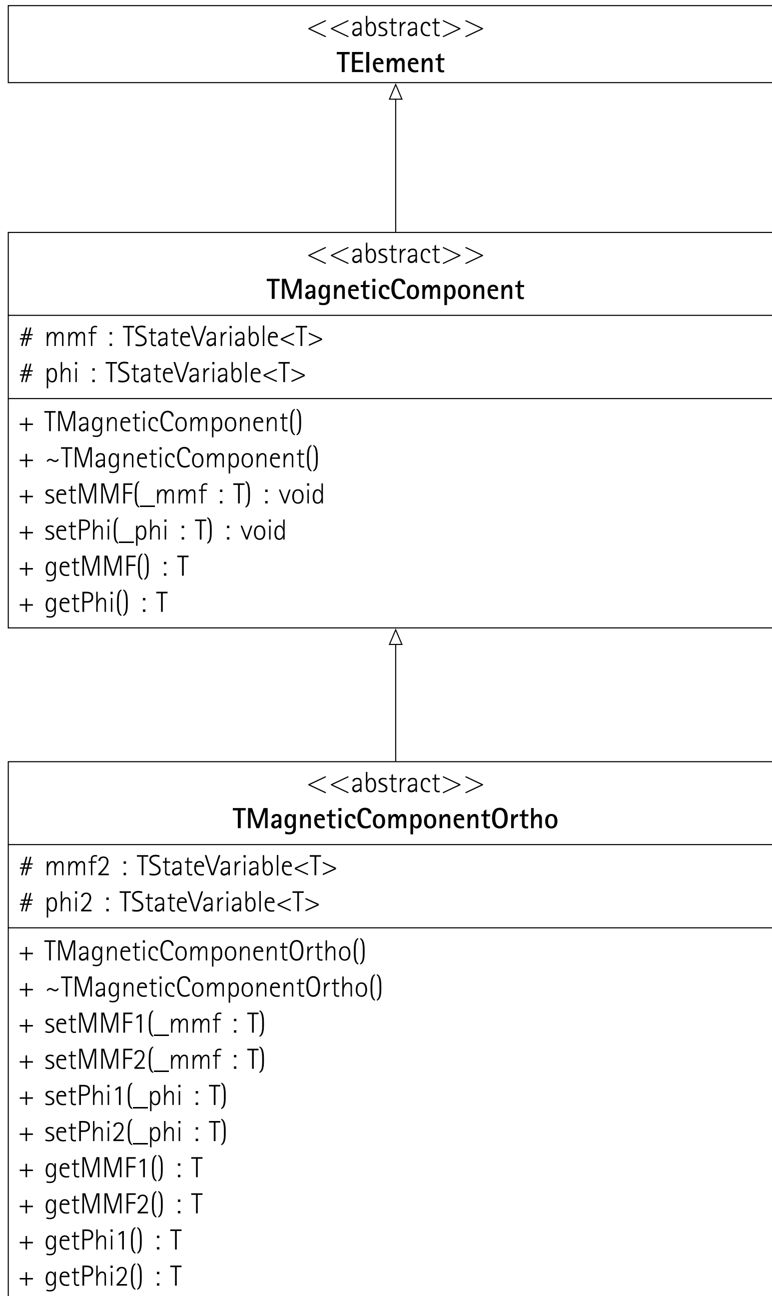


Abb. 4.7: Klassendiagramm von *TMagneticComponent* und *TMagneticComponentOrtho*. Magnetische Spannung und Fluss als geschützte Zustandsvariablen, Konstruktor und Destruktor zur Verwaltung der selben. Öffentliche *get*- und *set*-Methoden. Vererbung in „hat-ein“-Beziehung, *setMMF1* greift auf *setMMF* zurück, etc. Keine Implementierung der rein virtuellen Funktionen von *TElement*

abgeleiteten Klasse möglich ist, kann ein ähnliches Werkzeug wie das zur Bestimmung der Magnetisierungskennlinien von Materialien (s. o.) entwickelt werden.

Die Funktion *MagCurveGeo* erhält als Parameter ein Objekt, welches einen magnetischen Zweipol repräsentiert sowie den Bereich der magnetischen Spannung, welcher durchfahren werden soll. Es speichert die Wertepaare, bestehend aus der magnetischen Spannung und dem Fluss, als CSV in einer Datei. Die für die Ausführung benötigte Dauer ist hauptsächlich von der Performanz der Methode *applyModel* des übergebenen Objektes abhängig.

Zwei Beispiele für eine aufgenommene charakteristische Funktion des Klemmenverhaltens eines sog. „Single-Crosses" (siehe Abschnitt 5.2.2, S. 145) sind in *Abb. 5.12* festgehalten. Darin wird außerdem die Polymorphie der Klasse *TMagneticComponent* genutzt und es wird dem Werkzeug ein Objekt der abgeleiteten Klasse *TMagneticComponentOrtho* übergeben (siehe Abschnitt 4.3.2, S. 135). Die weitere einstellbare magnetische Spannung dieses Objekts wird vor dem Aufruf des Werkzeuges unterschiedlich eingestellt, wie in der Legende der genannten Abbildung festgehalten ist.

Nachbildung von Messungen an magnetischen Vierpolen

Ein spezifisches Problem, welches im Rahmen der durchgeführten Versuche auftritt (siehe Kapitel 6, S. 139), besteht in der Nachbildung einer Messung an einem magnetischen Vierpol, wenn die an den Klemmen angelegten magnetischen Spannung nichttriviale zeitliche Signalformen aufweisen. Die Funktion *MagneticComponentReplay* liest die elektrischen Ströme einer Messung sowie die dazugehörigen Zeitstempel aus einer Datei mit CSV ein und simuliert damit einen magnetischen Vierpol. Die elektrischen Spannungen und ihre zeitlichen Integrale werden als CSV in einer Datei gespeichert. Für die Wandlung zwischen elektrischen und magnetischen Größen werden die Windungszahlen der bei der Messung verwendeten Wicklungen benötigt (siehe Abschnitt 1.1.1, S. 6). Die benötigte Rechendauer ist maßgeblich von der Performanz der Methode *applyModel* des übergebenen Objektes abhängig, welches den magnetischen Vierpol repräsentiert.

Ein Beispiel für die Anwendung des Werkzeugs ist in *Abb. 5.13* abgebildet. Die Ströme und die Windungszahlen stammen aus einer Messung (siehe Abschnitt 5.2, S. 141), die Spannungen und ihre zeitlichen Integrale sind simuliert. Die Dauer für die gesamte darin gezeigte Berechnung beträgt etwa 20 min auf einem handelsüblichen PC.

Allgemeiner zeittransienter Solver

Alle Objekte, deren Klasse von *TSolvable* erbt, lassen sich lösen (siehe Abschnitt 4.1.1, S. 120). Die Klasse *TTimetransientSolver* erbt von einer beliebigen lösbaren Klassen und überschreibt die virtuelle Methode *solve* mit einer eigenen, welche eine zeittransiente Simulation durchführt. Zu jedem Zeitpunkt ruft sie die Methode *solve* der vererbten Klasse auf und schreitet anschließend einen Zeitschritt fort.

Anschaulich bedeutet dies, dass bspw. stets im Wechsel eine Feldsimulation durchgeführt und der Zeitpunkt inkrementiert wird. Durch das Fortschreiten der Zeit können sich Randbedingungen oder die diskretisierten Elemente vom Typ *TFdmElement* ihr Verhalten ändern.

Das Werkzeug ist im Allgemeinen sehr mächtig, wird im Rahmen der vorliegenden Arbeit aber deshalb nicht benötigt, weil der einzige Anwendungsfall, die Nachbildung von Messungen an magnetischen Vierpolen (s. o.), zu einfach zu implementieren ist, als dass sie von der Nutzung des allgemeinen zeittransienten Solvers profitieren würde. In nachfolgenden Arbeiten kann dieses Werkzeug allerdings eine große Hilfe darstellen.

5 Versuch

Prepare for unforseen consequences.

– G-Man

Die zuvor entwickelten Werkzeuge werden in diesem Kapitel verwendet, um einen Versuch aufzubauen und durchzuführen und um Ergebnisse zu erhalten und auszuwerten.

Zuerst werden einige magnetische Kerne aus verschiedenen Geometrien und verschiedenen Materialien vorbereitet. Es wird darauf eingegangen, welche Zwecke die Formen und Stoffe erfüllen und welche davon im Rahmen dieser Arbeit leider nicht verwendet werden können. Anschließend werden sie über einen magnetischen Rückschluss mit einer magnetischen Quelle verbunden und sind bereit zur Vermessung.

Im vorgestellten Versuch werden die magnetischen Aufbauten mithilfe des Messaufbaus (siehe Kapitel 3, S. 51) charakterisiert. Parallel wird ihr Verhalten mit der Simulationssoftware (siehe Kapitel 5, S. 119) berechnet. Abweichungen zwischen der Messung und der Simulation können verwendet werden, um das Simulationsmodell anzupassen. Dabei ist das Ziel gestellt, ein so gutes Modell zu gewinnen, dass das Verhalten des Versuchsaufbaus präzise vorausgesagt werden kann. Der kurzen Bearbeitungszeit geschuldet wird dabei auf den Ansatz von Tez (siehe Abschnitt 1.4.3, S. 44) zurückgegriffen und es werden keine komplizierteren Materialmodelle entwickelt. Der Schluss vom Materialmodell auf das Klemmenmodell geschieht vollständig durch die Simulationssoftware.

Der wissenschaftlichen Gründlichkeit halber sind der Aufbau, die Durchführung und die Auswertung in jeweils einzelne Abschnitte eingeteilt. Entsprechend wird beim Versuchsaufbau bereits auf den Aufbau der Simulation eingegangen und diese erst in der Durchführung gestartet. Die zunächst ungewöhnlich scheinende Aufteilung ist lediglich dem Umstand geschuldet, dass die Auswertung der Daten in dieser Arbeit nicht manuell, sondern teilweise durch die Simulation geschieht.

Schließlich werden die Ergebnisse der Messung und der Simulation ausgewertet und es wird eine Aussage über die Gültigkeit der Modellierung getroffen.

5.1 Vorabversuche

Bevor die Versuche aufgebaut werden können, müssen alle Materialien, Werkzeuge und Methoden verfügbar sein.

Die Messungen am realen Aufbau geschehen mithilfe der Hardware und Software, welche zuvor beschrieben ist (siehe Kapitel 3, S. 51). Ihre Charakterisierung ist zusammen mit ihrem Aufbau festgehalten, sodass keine weiteren Vorabversuche dazu notwendig sind.

Einige Versuchsgegenstände sollen aus galvanisch abgeschiedenem Material hergestellt werden. Der bisherige Stand bei der Entwicklung des Verfahrens ist bereits zuvor ausführlich beschrieben (siehe Kapitel 4, S. 111). Experimente zur Charakterisierung des Herstellungsverfahrens können nicht durchgeführt, weil das Verfahren nicht abschließend entwickelt ist.

Die computergestützten Simulationen werden mithilfe einer eigenen Software duchgeführt. Deren Aufbau bis hin zur Implementierung ist bereits an anderer Stelle dokumentiert (siehe Kapitel 5, S. 119, siehe Kapitel 11, S. 249). Bevor die in diesem Kapitel genannten Berechnungen damit durchgeführt werden können, müssen Ermittlungen zur Charakterisierung der Algorithmen umgesetzt werden. Diese würden inhaltlich zu weit führen, weshalb sich die Autorin auf sorgfältiges Debugging beschränkt.

5.2 Aufbau

Der Versuchsaufbau umfasst einen physischen Aufbau und eine numerische Simulation.

Für den physischen Aufbau wird der Messaufbau (siehe Kapitel 3, S. 51) an verschiedene Last angeschlossen, um diese zu vermessen. Es handelt sich ausschließlich um induktive Lasten, die aus magnetischen Rückschlüssen und den zu untersuchenden magnetischen Netzwerkelementen bestehen.

Zunächst werden die magnetischen Rückschlüsse vorgestellt, anschließend die zu vermessenden Objekte aus verschieden Geometrien und Materialien. Teils werden Zweipole zur Messung vorbereitet, um die Materialien zu charakterisieren und teils Vierpole, um das Phänomen der orthogonalen Magnetisierung zu beobachten.

Für die später daraus gewonnen Messergebnisse wird außerdem eine Simulation vorbereitet, damit die Messergebnisse mit den Simulationsergebnissen verglichen werden können.

5.2.1 Magnetischer Rückschluss

Einige später verwendete Geometrien (siehe Abschnitt 5.2.2, S. 144) sind magnetische n-Pole mit offenen Klemmen, insbesondere die „Singe-Sheets" und „Single-Crosses". Diese Klemmen werden mit einem niederreluktanten Bauteil kurzgeschlossen, einem sog. magnetischen Rückschluss. Auf die genauen Schaltungen wird bei der Vorstellung der Geometrien eingegangen. In diesem Abschnitt wird das Bauteil vorgestellt, mit welchem alle magnetischen Rückschlüsse geschehen.

Ferrit-U-Kern

Als Rückschluss eignet sich ein U-Kern aus Ferrit (N27) mit einer Distanz von 37 mm zwischen den beiden magnetischen Klemmen, wie in *Abb. 5.1* dargestellt. Die Reluktanz lässt sich aus dem Datenblatt entnehmen:

$$\begin{aligned} \frac{1}{2}A_L &= 5400\,\mathrm{nH} = \frac{L}{N^2} \\ \Rightarrow R_\mathrm{m} &= 92\,592\,\mathrm{A\,V^{-1}\,s^{-1}} \end{aligned} \tag{5.1}$$

Der Einfachheit halber wird eine Wicklung mit einer Windungszahl von $N = 300$ und einem Drahtdurchmesser von $d = 0{,}5\,\mathrm{mm}$ auf einen 3D-gedruckten Coilformer um den U-Kern gebracht. Die alternative Platzierung von Wicklungen um die zu vermessenden Geometrien ist wegen der hohen erforderlichen Wicklungszahl und des dicken Drahtes anspruchsvoller. Weil die Streureluktanz der Wicklung parallel zu der zu vermessenden Reluktanz geschaltet ist, lässt sich mithilfe einer Netzwerktransformation (siehe Abschnitt 1.1.4, S. 19) zeigen, dass dieses Phänomen äquivalent zu einer Induktivität im elektrischen Netzwerk in Reihe zur zu vermessenden Induktivität ist. Es ergeben sich die folgenden Effekte:

1. Die magnetische Spannung, welche über dem Spezimen abfällt, lässt sich aus dem Strom durch die

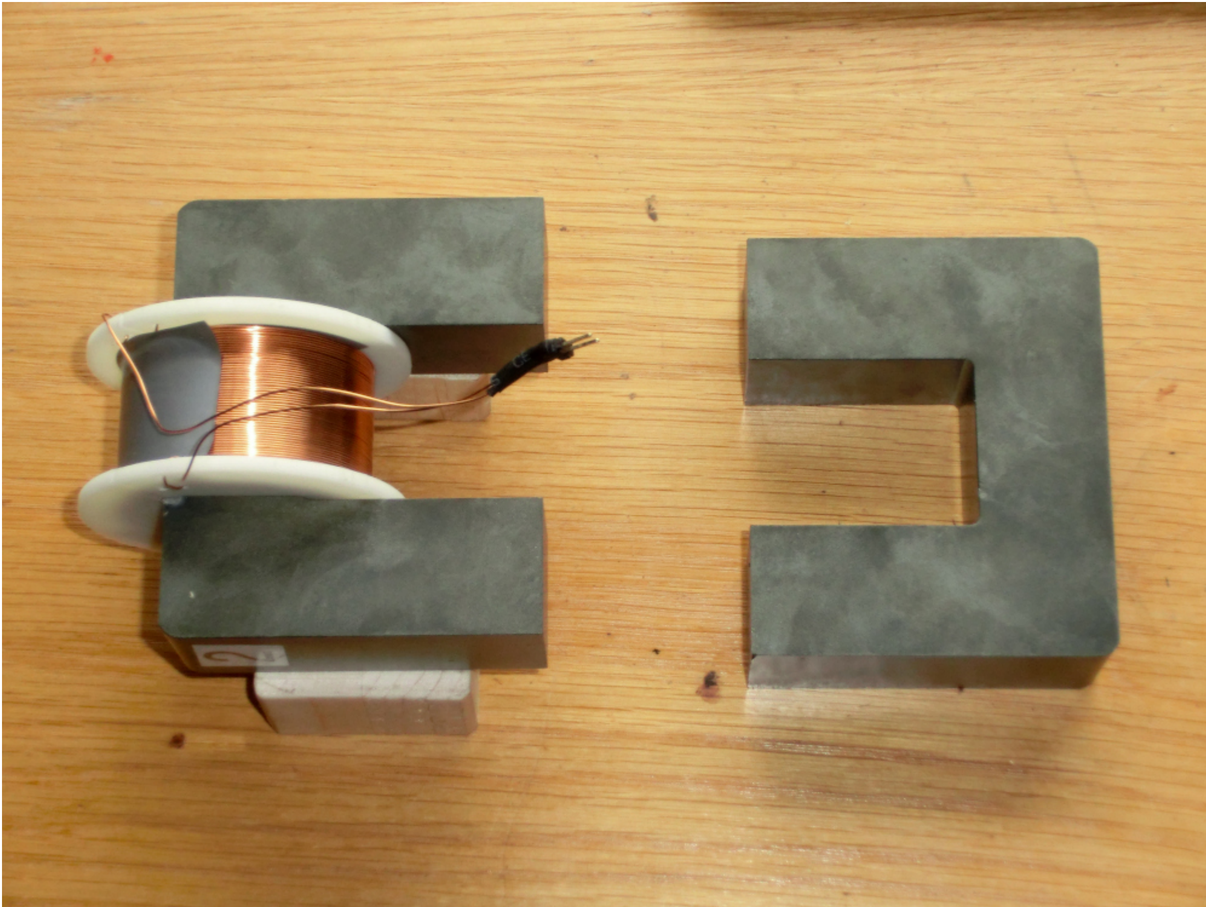


Abb. 5.1: Fotografie des Ferrit-U-Kerns. Mit Wicklung auf 3D-gedrucktem Coilformer (Windungszahl $N = 300$, Durchmesser $d = 0{,}5\,\mathrm{mm}$, links) und ohne (rechts). Mit Stützklötzen zum Anheben (links) und ohne (rechts). Material N27. Datenblatt in [13].

Wicklung und ihrer Windungszahl berechnen und es ist keine weitere Messtechnik notwendig.

2. Einige Messungen belegen, dass die Streureluktanz um ein Vielfaches kleiner ist als die typischen zu vermessenden. Entsprechend kann mit einer elektrischen Gleichspannung ein nahezu konstanter elektrischer Strom und damit eine nahezu konstante magnetische Spannung erreicht werden.
3. Wegen der geringen Streureluktanz lässt eine elektrische Spannungsmessung an der Wicklung auf falsche magnetische Flüsse schließen. Eine weitere Wicklung nahe am Spezimen ist für die Flussmessung notwendig.

Die an den Klemmen der Wicklung messbare Impedanz bei offenen magnetischen Klemmen des Kerns ist in *Abb. 5.2* illustriert und zeigt ein nahezu ideales induktives Verhalten zwischen 200 Hz und 52 kHz.[1]

Weil mit dem Coilformer auf dem Kern keine stabile waagerechte Ausrichtung mehr möglich ist, werden 30 mm

[1] Messungen zeigen, dass die Reluktanz an den offenen magnetischen Klemmen durch die untersuchten Spezimen kaum verringert wird, sodass das die Impedanz annähernd unabhängig von den Messobjekten ist.

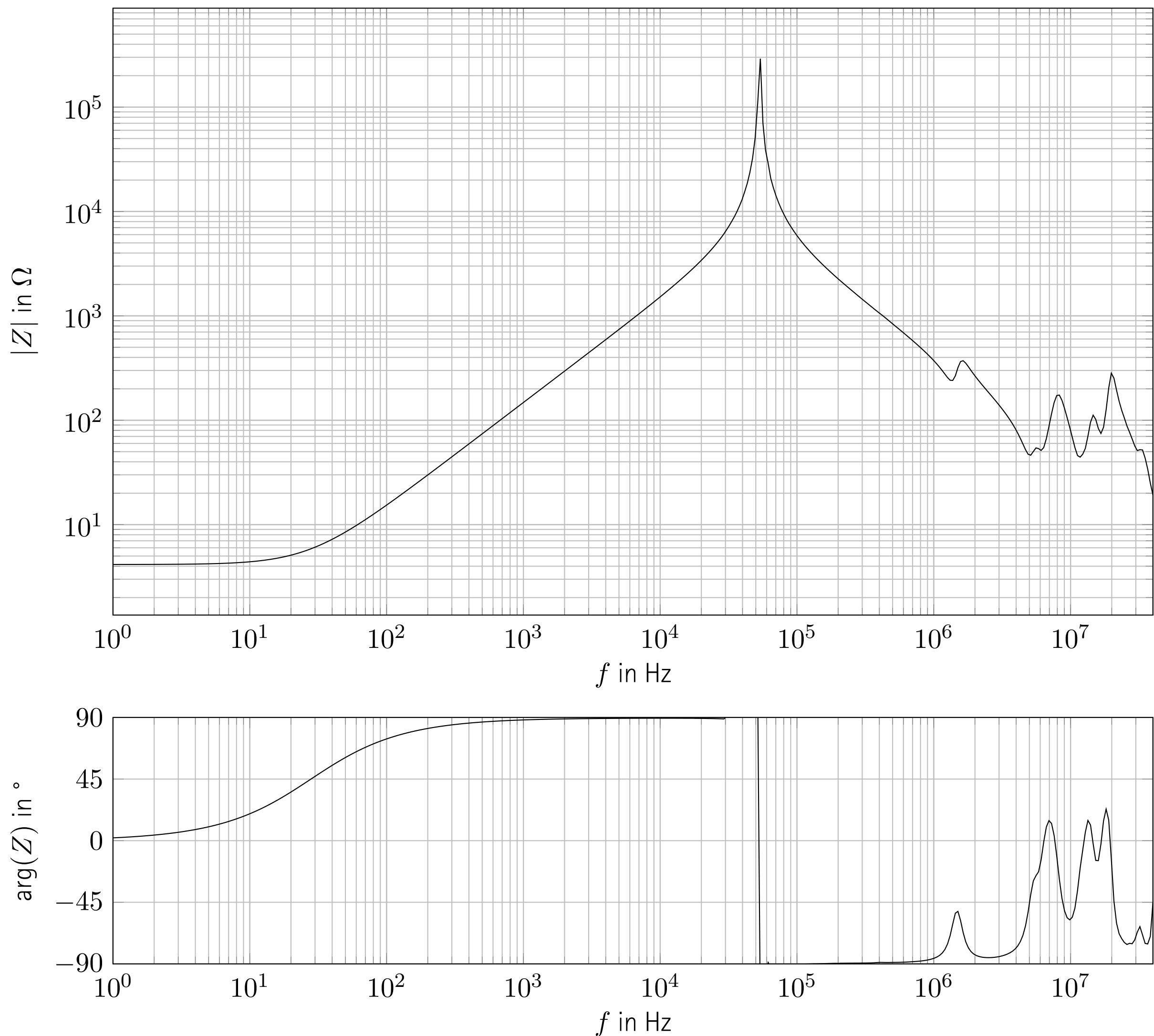


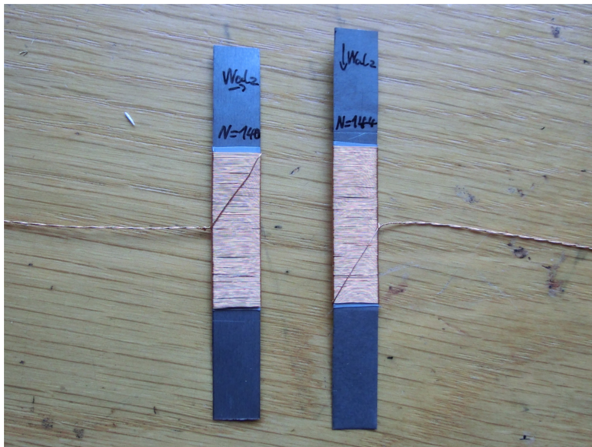
Abb. 5.2: Impedanz der Wicklung auf dem primären Ferrit-U-Kern. Offene magnetische Klemmen. 401 Messpunkte, aufgenommen mit Bode100 (Ein-Port-Messung, Empfängerbandbreite: 1 Hz). Reelle Last durch Wicklungswiderstand bis ca. 10 Hz, rein induktives Verhalten ab ca. 200 Ω, Resonanz bei 52 kHz.

hohe Stützklötze aus Holz zugesägt.

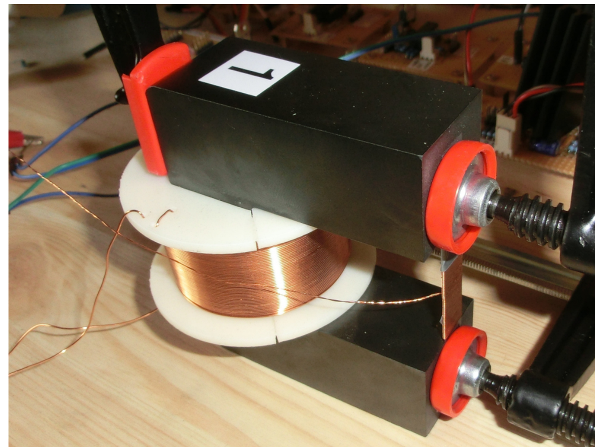
Es werden zwei gleiche Ferrit-U-Kerne bewickelt, weil bei manchen Messungen zwei gleichzeitig benötigt werden.

Ferrit-Ringkern

Zu Vergleichszwecken soll ein Rückschluss aus einem halben toroidalen Kern aus Eisenpulver verwendet werden. Aus zeitlichen Gründen und weil die Messungen mit dem o. g. Ferrit-U-Kern erfolgreich sind, wird auf die



(a) Bleche



(b) eingespanntes Blech

Abb. 5.3: Fotografie des „Single-Sheets“. Bleche mit Markierung der Walzrichtung und Wicklung zur Messung des Flusses mitsamt Markierung (links). Blech an magnetischen Rückschluss (Ferrit-U-Kern mitsamt Wicklung) mit Schraubzwingen eingespannt.

Verwendung des Eisenpulverkerns verzichtet.

5.2.2 Geometrien

Zur Erforschung der orthogonalen Magnetisierung sind Magnetische Kerne mit verschiedenen Geometrien denkbar. Zunächst wird das „Single-Sheet“ vorgestellt, mithilfe dessen die Magnetisierungscharakteristik eines Materials aufgenommen werden kann. Anschließend werden der Torus und das „Single-Cross“ verglichen.

„Single-Sheet“

Zunächst soll eine einfache Geometrie vermessen werden, um die Magnetisierungskennlinien der zu untersuchenden Materialien zu erfahren. Zum einen sind diese durch den Hersteller nicht bekanntgegeben, zum anderen dient diese Geometrie zur Validierung der Sinnhaftigkeit der gewählten Einstellungen am Messaufbau, insbesondere der verschiedenen auswählbaren Signalamplituden und Verstärkungen, wie am großen Impedanzwandler (siehe Abschnitt 2.2.4, S. 73). Dazu dient das so genannte „Single-Sheet“, welches weite Verwendung im Einzelblechtester, engl. Single-Sheet-Tester (SST) findet. Es zeichnet sich durch seine einfache rechteckige Geometrie aus und ist insbesondere im unteren Frequenzbereich dazu geeignet, ein weitestgehend homogenes Magnetfeld zu führen. Das Blech (beispielhaft aus nicht-kornorientiertem Elektroblech) und dessen Einspannung in einen magnetischen Kreis sind in *Abb. 5.3* abgedruckt. Die Querschnittsmaße betragen 10 mm × 0,1 mm. Die effektive Länge erstreckt sich über die Klemmendistanz des verwendeten Rückschlusses, hier 37 mm.

Das Blech wird mit Kupferlackdraht mit einem Durchmesser von $d = 0{,}2$ mm bewickelt, um den Fluss durch

das Blech zu messen. Spannungsmessungen an der Wicklung um den Rückschluss sind ungenau, weil der Spannungsabfall über dem ohmschen Anteil der Impedanz nicht genau bestimmt werden kann.[2] Außerdem zeigen einige Messungen, dass der Streufluss um ein Vielfaches größer ist als der Fluss durch das Spezimen.

Torus

Ein wesentlicher Nachteil des „Single-Sheets" ist die Begrenzung auf eine einzige Richtung des durchströmenden Magnetfeldes. Ein dünner Torus kann zwei orthogonale Magnetfelder leiten und trotzdem Homogenität der Felder in beiden Richtungen gewährleisten. [31] [24] Dieser Vorteil entsteht dadurch, dass ein dünner Torus als ein zweimal aufgerolltes Rechteck betrachtet werden kann. Nach dem ersten Aufrollen erhält man einen Zylinder, nach dem zweiten einen Torus. Damit sind die jeweils gegenüberliegenden Seiten des Rechtecks zusammengeführt und es treten an diesen keine Randeffekte auf. Der Unterschied in der Feldform gegenüber einem rechteckigen Blech ist in [24] veranschaulicht.

Weil das Verfahren zur Herstellung gekrümmter Eisenbleche (siehe Kapitel 4, S. 111) nicht vollständig entwickelt ist, kann im Rahmen dieser Arbeit keine torusförmige Geometrie hergestellt werden. Die dringende Empfehlung zur Erforschung der orthogonalen Magnetisierung anhand eines Torus in [31] wird von der Autorin an nachfolgenden Arbeiten weitergeben.

Eine dreidimensionale Schnittskizze eines dünnen Torus mit der dazugehörigen Bewicklung ist in *Abb. 5.4* präsentiert. Man sieht, dass die verursachten magnetischen Felder nicht homogen im magnetischen Material verteilt sind, solange die Permeabilität dessen nicht als unendlich größer als die des umgebenden Mediums, bspw. Luft, angenommen werden kann. Als Gegenmaßnahme können die Windungen gleichmäßger verteilt werden. Dazu wird für die innenliegende Wicklung eine inerte Stützstruktur benötigt, bspw. aus Kunststoff 3D-gedruckt. Auf weitere Probleme, wie den Proximity-Effekt, geeignete Coilformer, Dimensionierung und parasitäre Größen soll hier nicht weiter eingegangen werden, sie können in nachfolgenden Arbeiten behandelt werden.

„Single-Cross"

Die wichtigste Geometrie zur Erforschung der orthogonalen Magnetisierung ist das sog. „Single-Cross". Im Gegensatz zum Single-Sheet ermöglicht es eine orthogonale Magnetisierung und anders als der Torus ist es einfach zu fertigen. Es handelt sich dabei um ein rechteckiges (oder ggf. quadratisches) Eisenblech, welches an allen vier dünnen Flächen über Blech einem magnetischen Rückschluss verbunden ist. Dadurch, dass diese Verbindungen aus dem gleichen Material bestehen können wie der rechteckige Kern und entsprechend eine Fertigung in einem Stück möglich ist, lässt sich eine Übergangsreluktanz vermeiden.[3] Ein Beispiel für ein „Single-

[2] Bei einigen Messungen ist der Spannungsabfall über dem ohmschen Wicklungswiderstand deutlich größer als der Spannungsabfall über der Induktivität, sodass erhebliche Fehler bei der Quantisierung entstehen.

[3] Die Vermeidung einer Übergangsreluktanz ist insbesondere wichtig, um ein definiertes Feld zu erhalten. Die in dem Fall unvermeidbaren Übergangsreluktanzen zwischen dem Blech und den Rückschlüssen liegen außerhalb des Raumgebiets, in welchem die orthogonale Magnetisierung stattfindet.

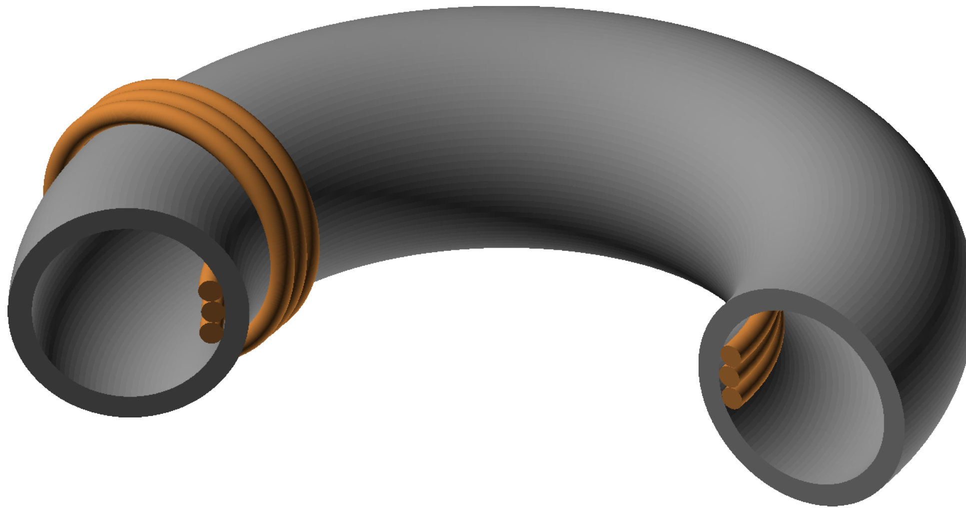
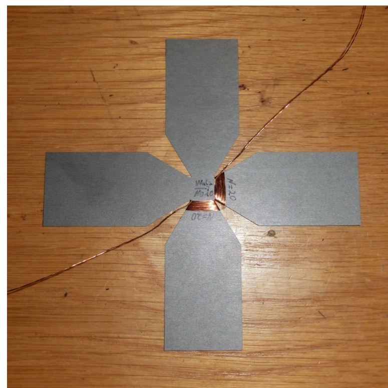
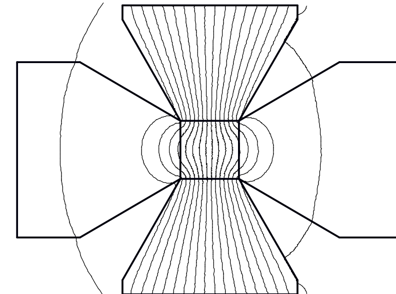


Abb. 5.4: Schnittskizze eines torusförmigen Kerns. Innere und äußere Wicklung angedeutet, nicht gleichmäßig verteilt. Zwei annähernd homogene Felder in orthogonaler Orientierung zueinander möglich. [24]

(a) dünne Anschlüsse



(b) breite Anschlüsse



(c) Feldsimulation [24]

Abb. 5.5: „Single-Cross". Nicht-kornorientiertes Blech. Version mit dünnen magnetischen Anschlüssen (links), Version mit gleicher Quadratgröße in der Mitte und breiten magnetischen Anschlüssen (mittig) und Feldsimulation von letzterer (rechts). Bleche mit Wicklungen zur Flussmessung bewickelt und gekennzeichnet, Walzrichtung und Material gekennzeichnet. Feldsimulation mit Software „FEMM" weit im Bereich der Sättigung für vertikales Feld, kein horizontales Feld.

Cross" ist in *Abb. 5.5a* gezeigt. Die darin offensichtliche hohe Flussdichte in den magnetischen Anschlüssen beeinflusst erheblich die Reluktanz, welche die Messergebnisse beeinflusst.[4] Um diesem Effekt vorzubeugen, kann die Geometrie der Anschlüsse variiert werden, im einfachsten Fall werden sie breiter, wie in *Abb. 5.5b* und *Abb. 5.5c* festgehalten, in dieser Arbeit aber nicht weiter verfolgt. Ein weiterer Ansatz, bei dem „Längsluftspalte" parallel zur Flussrichtung in die Anschlüsse des „Single-Crosses" eingebracht werden, ist in [24] diskutiert und spielt in dieser Arbeit ebenfalls keine Rolle.

[4] Durch die geringe Reluktanz des Rückschlusses wird das Messergebnis hauptsächlich durch die Reluktanz des eingespannten Blechs beeinflusst.

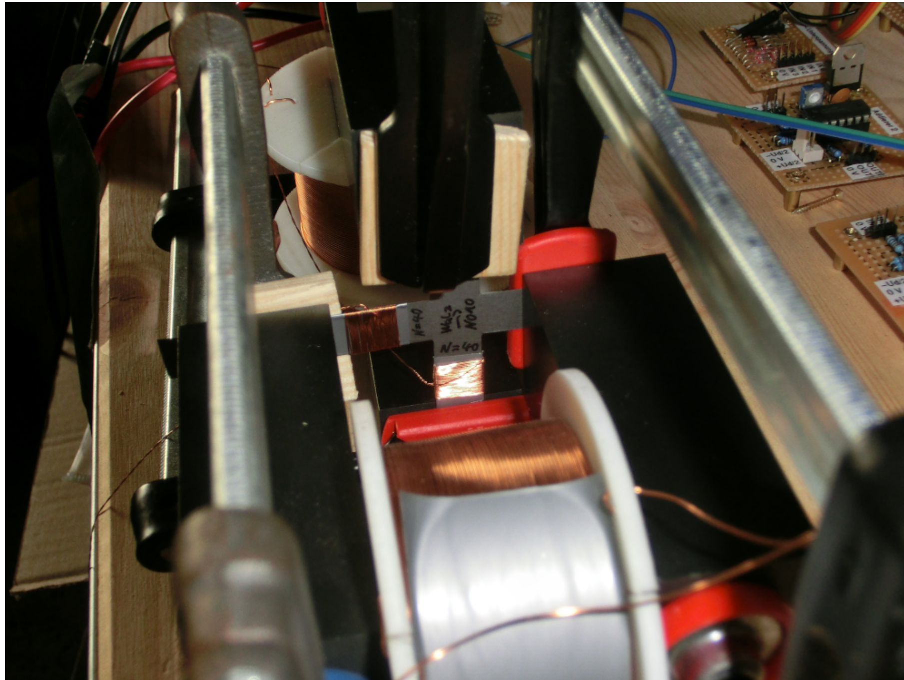


Abb. 5.6: Fotografie des „Single-Crosses", an Rückschlüsse angeklemmt. Sekundärer Rückschluss mit Wicklung (vorn), primärer Rückschluss mit Wicklung (hinten), „Single-Cross" mit zwei Messwicklungen und Kennzeichnungen (mittig). Schraubzwingen mit Unterlegern aus Holz zur Fixierung.

In [31] ist erläutert, dass zur Erzwingung orthogonaler Magnetisierung zwei einzelne Rückschlüsse notwendig sind, die höchstens über eine hohe parasitäre Reluktanz miteinander verbunden sind. Die jeweils gegenüberliegenden magnetischen Kontakte eines „Single-Crosses" werden deshalb mit einem Rückschluss verbunden. Die nebeneinander liegenden Enden der Geometrie sind nicht miteinander verbunden, außer über das in der Mitte liegende Blechquadrat. Das „Single-Cross" ist in *Abb. 5.6* an die beiden Rückschlüsse angeklemmt abgebildet.

5.2.3 Materialien

Sämtliche der zuvor genannten Geometrien sollen mit sämtlichen der folgend vorgestellten Materialien hergestellt werden, um eine breite Übersicht über das Phänomen der orthogonalen Magnetisierung zu erhalten. Davon ist die Geometrie des Torus ausgenommen (die ohnehin nicht im Rahmen dieser Arbeit gefertigt werden kann), weil ihre Herstellung gemäß des aktuellen Wissensstandes nicht aus beliebigen Materialien möglich ist.

Galvanisches Eisen

Die torusförmige Geometrie kann nach dem aktuellen Wissenstand nur aus galvanisch abgeschiedenem Eisen oder alternativ mit einem hohen finanziellen Aufwand aus Ferrit hergestellt werden (siehe Kapitel 4, S. 111). Weil letzteres aus dem Rahmen der Arbeit fällt, bleibt nur galvanisch abgeschiedenes Eisen. Damit die Ergebnisse

dieser Geometrie mit den der anderen verglichen werden können, müsse alle Geometrien einmal aus galvanisch abgeschiedenem Eisen bereitstehen.

Weil das Verfahren zur Herstellung von Blechen aus galvanischem Eisen nicht abgeschlossen ist (siehe Kapitel 4, S. 111), kann das Material in dieser Arbeit nicht untersucht werden.

Nicht-kornorientiertes Elektroblech

Sowohl „Single-Sheets" als auch „Single-Crosses" lassen sich aus dem Material „NO10" der LCD LaserCut AG herstellen. Es steht in dieser Arbeit in einer Materialstärke von 0,1 mm zur Verfügung und kann entsprechend mit einer Haushaltsschere zugeschnitten werden. [5] Obwohl das Blech keine Kornorientierung aufweist, ist es möglicherweise nicht magnetisch isotrop und wird deshalb nach der Walzrichtung unterschieden. Die Walzrichtung ist auf allen Blechzuschnitten handschriftlich markiert. Die Materialeigenschaften sind unbekannt.

Kornorientiertes Elektroblech

Kornorientierte Elektrobleche sind magnetisch anisotrop, weshalb die orthogonale Magnetisierung auch in ihnen untersucht werden soll. Für die Bearbeitung der vorliegenden Arbeit ist die Beschaffung dieses Materials leider nicht möglich, weshalb es in zukünftigen Ausarbeitungen behandelt werden soll.

Verlustleistungsreiches Blech

Messungen zeigen, dass das o. g. Blech (siehe Nicht-kornorientiertes Elektroblech) eine geringe Verlustleistung aufweist. Um den Einfluss der orthogonalen Magnetisierung auf die Breite der Hysterese der Magnetisierungskurve zu untersuchen, ist ein Material notwendig, welches hohe Ummagnetisierungsverluste bewirkt. Der Umfang solcher Messungen würde den Rahmen der Arbeit überschreiten, weshalb ein solches Material erst in anschließenden Arbeiten untersucht werden soll.

Ferrit

Bei den genutzten Blechgeometrien (siehe Abschnitt 5.2.2, S. 144) können nahe der Grenzen zum magnetischen Rückschluss Wirbelströme auftreten. Außerdem ist die Wahl von Elektroblech im Bereich der Leistungselektronik unüblich. Aus diesen Gründen sollten Ferritwerkstoffe untersucht werden. Wegen der hohen Fertigungskosten für die spezifischen Geometrien sind für die vorliegende Arbeit keine Messungen an Ferriten geplant.

[5] Versuche, das Material mit einer Laubsäge und einem Metallsägeblatt zu sägen schlagen fehl.

5.2.4 Simulation

Die später aus dem physischen Versuch gewonnenen Ergebnisse sollen mit einer Simulation verglichen werden. Sie dient dazu, ein intrinsisches Materialmodell zu validieren, ohne ein analytisches extrinsisches Modell zu benötigen. Dieses Feature der Simulation ist zuvor detailliert beschrieben (siehe Kapitel 5, S. 119). Im Folgenden werden die Geometrie und das zu überprüfende Materialmodell vorgestellt. Letzteres besteht aus einer verlustfreien 1D-Magnetisierungskennlinie $B(H)$ und einer Erweiterung zu einem vollständigen Materialmodell $\boldsymbol{B}(\boldsymbol{H})$. Schließlich wird auf die Vorbereitung des Werkzeugs zur Nachbildung von Messungen eingegangen.

Geometrien

Das „Single-Sheet" wird nicht in einer Feldsimulation nachgebildet, weil das Feld darin homogen ist. Entsprechend wird je Materialtyp ein einzelnes (FDM-)Element vom Werkzeug zur Abtastung von Materialkennlinien (siehe Abschnitt 4.3.3, S. 135) erzeugt. Das Werkzeug bekommt die zu untersuchenden Materialmodelle (s. u.) und den zu betrachtenden Bereich den H-Feldes (hier: $-5000\,\mathrm{A\,m^{-1}}$ bis $5000\,\mathrm{A\,m^{-1}}$) übergeben und speichert das Ergebnis selbstständig.

Im Single-Cross ist das magnetische Feld nicht homogen und entsprechend wird das Werkzeug zur Abtastung von magnetischen Netzwerkkomponenten verwendet (siehe Abschnitt 4.3.3, S. 135). Die Netzwerkkomponente wird außerhalb des Funktionsbauches des Werkzeuges erzeugt, um wiederverwendet werden zu können. Sie ist vom Typ *TFdmSc* (siehe Abschnitt 4.2, S. 130) und bekommt die genannten Parameter übergeben:

```
TFdmSc<                  // Single Cross
    ELEMENT_PARAMS,
    100,                 // 100 elements in height
    100,                 // 100 elements in width
5   TFdmElementNonlinearIsotropicLossfreeApproximation   // material type
> setup(0.03,0.03);    // 3 cm in width and height, therefore 1 cm wide magnetic paths
```

Verlustfreie 1D-Magnetisierungskennlinie

Bevor ein vollständiges Materialmodell abgeleitet werden kann, ist eine verlustfreie 1D-Magnetisierungskennlinie $B(H)$ benötigt. In der Simulation werden zwei verschiedene Annäherungen an die gemessene Hystereseschleife verwendet.

Die Annhäherung auf Basis einer **LUT** verwendet ist vom Typ *TFdmElementNonlinearIsotropicLossfreeLut* (siehe Abschnitt 4.2, S. 130) und liest die Daten direkt aus der Messung mit $\hat{u}_{\mathrm{q},1} = 10\,\mathrm{V}$; $u_{1,2} = 0\,\mathrm{V}$ ein (*TFdmElementNonlinearIsotropicLossfreeLutType lut_type = luttMeasurement*, siehe Abschnitt 4.2, S. 130). Diese Näherung für die Magnetisierungskennlinie wird für die Feldsimulationen nicht verwendet, weil dabei einige numerische Probleme auftreten (siehe Abschnitt 5.4, S. 154).

Die **analytische** Näherungskennlinie ist im Materialmodell *TFdmElementNonlinearIsotropicLossfree* (siehe Abschnitt 4.2, S. 130) umgesetzt und wird hier mit den folgenden Templateparametern verwendet:

```
typedef TFdmElementNonlinearIsotropicLossfree<
    ELEMENT_PARAMS,
    4000,                          // mu_r is 4000 near H=0
    1200000,                       // B_sat = 1.2 T
    60                             // mu_r is 60 near B_sat
> TFdmElementNonlinearIsotropicLossfreeApproximation;
```

Die hohe differentielle Permeabilität im Bereich der Sättigung in Kombination mit der geringen Sättigungsflussdichte ist gewählt, weil dies die Messungen am besten annähert.[6]

Vollständiges Materialmodell

Das vollständige Materialmodell ist im Allgemeinen in einer Klasse implementiert, die von der Klasse erbt, welche die verlustfreie 1D-Magnetisierungskennlinie (s. o.) beinhaltet. Im Rahmen der vorliegenden Arbeit wird lediglich das Modell von Tez (siehe Abschnitt 1.4.3, S. 44) angewandt, welches so einfach aufgebaut ist, dass die zuvor beschriebenen Materialmodelle dieses bereits korrekt und vollständig implementieren. Die zusammen mit den Geometrien (s. o.) genannten Werkzeuge werden deshalb einfach mit dem verlustfreien 1D-Materialmodell *TFdmElementNonlinearIsotropicLossfreeApproximation* (s. o.) verwendet.

Nachbildung von Messungen

Die zusammen mit den Geometrien genannten Werkzeuge durchlaufen einen magnetischen Feldstärkebereich (s. o.), variieren während dieses Durchlaufs aber keine weiteren Parameter. Es ist davon auszugehen, dass der elektrische Strom durch die Sekundärwicklung und damit die sekundäre magnetische Feldstärke bei den Messungen nicht konstant gehalten wird. Die gemessenen Magnetisierungskennlinien sind in dem Fall nicht mit den simulierten vergleichbar.

Um diese Vergleichbarkeit herzustellen, wird das Werkzeug zur Messwiederholung an magnetischen Komponenten genutzt (siehe Abschnitt 4.3.3, S. 135). Dabei werden die elektrischen Ströme aus der Messung exakt als Parameter für die Feldsimulation am „Single-Cross“ genutzt. Die resultierenden elektrischen Spannungen und deren zeitliche Integrale werden als CSV in einer Datei gespeichert.

Die Parameter für das Werkzeug umfassen das bereits zuvor vorgestellte „Single-Cross“ aus dem bereits zuvor genannten Materialmodell. Für die Wandlung zwischen elektrischen und magetischen Netzwerkgrößen sind die Wicklungen für den elektrischen Strom mit einer Wicklungszahl von $N_{\mathrm{i},1} = N_{\mathrm{i},2} = 300$ und die Wicklungen für

[6] Das betreuende Institut stellt eine grob aufgelöste Messung der Magnetisierungskennlinie des Materials „NO20“ bereit, die evtl. ähnlich zu der des verwendeten Materials „NO10“ ist. Darin wird deutlich, dass die Sättigungsflussdichte von etwa 2 T erst bei sehr hohen Feldstärken erreicht wird, die relative differentielle Permeabilität aber bereits zuvor auf $\mu_{\mathrm{r,diff}} \approx 60$ absinkt. Die selbe Beobachtung wird bei den Messungen am „Single-Sheet“ gemacht.

die elektrische Spannungsmessung mit einer Wicklungszahl von $N_{\mathrm{u},1} = N_{\mathrm{u},2} = 40$ spezifiziert. Die Blechstärke des simuliertes „Single-Crosses“ beträgt $d = 0{,}1$ mm.

5.3 Durchführung

Dank des gut automatisierten Messaufbaus ist die Versuchsdurchführung einfach.

Messungen

Zunächst werden die Verstärkungsfaktoren der großen Impedanzwandler (siehe Abschnitt 2.2.4, S. 73) auf 11 für die primäre und 5.7 für die sekundäre Wicklung gestellt. Anschließend ist eine Kalibrierung notwendig, die mit dem Kalibrierungswerkzeug (siehe Abschnitt 2.3.2, S. 94) durchgeführt wird.

Die Werkstücke werden einzeln mit Schraubzwingen an die magnetischen Rückschlüsse geklemmt und die Messung geschieht mit dem Werkzeug zur improvisierten Vierpolcharakterisierung (siehe Abschnitt 2.3.2, S. 94), welchem die in *Tab. 5.1* aufgelisteten Einstellungen mitgeteilt werden. Anschließend wird die grobe Quantisierung des Oszilloskops mithilfe des dafür vorgesehenen Werkzeugs geglättet (siehe Abschnitt 2.3.2, S. 94) und die Spannungsintegrale mit dem speziellen Werkzeug dafür berechnet (siehe Abschnitt 2.3.2, S. 94). Der Vorgang ist mit dem zuvor vorgestellten Bash-Skript voll automatisiert (siehe Abschnitt 2.4, S. 108). Nachdem das Spezimen an den Rückschluss angeklemmnt ist, dauert die vollautomatische Aufnahme und Nachbearbeitung der Daten rund 20 min. Die Daten bedürfen keiner weiteren Bearbeitung, sondern können sofort bspw. mit Ti*k*z dargestellt werden.

Es sind weitere Parametervariationen denkbar, als die in *Tab. 5.1* genannten an den Signalamplituden. In nachfolgenden Arbeiten können auch Frequenz, Gleichspannungsoffset der Wechselspannung und Signalform für beide Wicklungen geändert werden, um ein umfassenderes Modell der orthogonalen Magnetisierung zu ent-

Tab. 5.1: Parameter für die improvisierte Vierpolmessung. Wechselspannung an der Primärwicklung, kleine Gleichspannung an der Sekundärwicklung.

Parameter	Wert
Ausgang (pimär)	*RP_CH_2*, kalibriert als „u1hs“
Signalform (primär)	Sinus, 100 Hz
Amplitude (primär)	1 V bis 10 V in Schritten von 1 V
Ruhezustand (primär)	0 V
Ausgang (sekundär)	*RP_CH_1*, kalibriert als „u2hs“
Signalform (sekundär)	Gleichspannung
Amplitude (sekundär)	0 V bis 2 V in Schritten von 0,2 V
Ruhezustand (sekundär)	0 V
Sicherheitswartezeit	0 s
Einschwingzeit	0,25 s
Zeiteinheit	ms
Dateinamen	test-%N1-%n2.csv, mit %N1: Amplitude der Primärspannung in V, mit %n2: Sekundärspannung in 0,2 V

wickeln und um praktische Anwendungen zu erforschen.

Simulationen

Die Simulation wird auf einem handelsüblichen PC gestartet und ist in Abhängigkeit der Detailtiefe trotz des Verzichts auf Multi-Threading innerhalb von einigen Sekunden bis einigen Minuten abgeschlossen. Die Bedienung ist trivial. Der vollständige Code der *main*-Funktion ist der Arbeit angehängt (siehe Code, S. 247).

5.4 Ergebnisse

Bei den gemessenen Größen handelt es sich um Ströme und Spannungen (siehe Abschnitt 5.2, S. 141, siehe Kapitel 3, S. 51), von welchen sich auf die magnetischen Spannungen und Flüsse, sowie auf die magnetischen Feldstärken und Flussdichten schließen lässt. Zunächst werden die gemessenen Größen in Abhängigkeit der Zeit dargestellt, um die Natur der Messergebnisse zu veranschaulichen. Anschließend ist das Phänomen der orthogonalen Magnetisierung mithilfe der Magnetisierungskennlinien $B_1(H_1, H_2)$ der untersuchten Materialien illustriert.

Nachdem die Messergebnisse des physischen Aufbaus erläutert sind, werden die Ergebnisse der Simulation vorgestellt. Sie umfassen die Annäherung der Magnetisierungskennlinien als isotrope, verlustfreie Kennlinien, wie sie für die Feldsimulation verwendet werden, die Magnetisierungskennlinien $B_1(H_1, H_2)$ aus stationären Feldsimulationen und eine zeittransiente simulative Nachbildung der Messung.

Weil der automatisierte Messaufbau mehrere hundert verschiedene Messungen durchführt und die Simulation ebenfalls weitestgehend automatisiert ist, kann in diesem Abschnitt nur ein winziger Bruchteil der Ergebnisse vorgestellt werden. Alle Daten sind der Arbeit angehängt (siehe Kapitel 9, S. 171).

5.4.1 Zeitveräufe der Netzwerkgrößen

Bei jeder Messung sind der Strom durch die Wicklung um den Rückschluss und die Spannung an der Wicklung um das Spezimen in Abhängigkeit der Zeit festgehalten. Der zeitliche Messbereich umfasst 16 ms und ist in 2000 äquidistante Messzeitpunkte unterteilt. Bei den magnetischen Vierpolen („Single-Cross") enthalten die Messdaten außerdem die Spannung und den Strom auf der sekundären Seite.

Die Messdaten sowie das berechnete Spannungsintegral sind beispielhaft für das „Single-Sheet" aus dem Material „NO10" mit dem Fluss parallel zur Walzrichtung in *Abb. 5.7* abgedruckt. Es ist deutlich erkennbar, wie der Strom wegen der sinusförmigen Quellspannung und der hohen Streuinduktivität annähernd sinusförmig ist (siehe Abschnitt 5.2.1, S. 141). Die elektrische Spannung zeigt die für magnetische Sättigung typischen Spitzen und im unteren Messbereich einiges Rauschen. Das zeitliche Integral der Spannung ist durch die filternde Wirkung des Integraloperators geglättet.[7] Die Form der Magnetisierungskennlinie $\left(\int u \,\mathrm{d}\, t\right)(i)$ kann bereits erahnt werden und ist in *Abb. 5.9a* präsentiert.

Die selbe Messung wie für das „Single-Sheet" ist für das „Single-Cross" in der schmalen Version durchgeführt, wobei eine variierende Gleichspannungen an die Klemmen der Wicklung auf dem sekundären Rückschluss angelegt ist. In *Abb. 5.8* sind die Netzwerkgrößen bei der Messung des „Single-Crosses" aus dem Material „NO10" mit dem primären Fluss orthogonal zur Walzrichtung und dem sekundären Fluss entsprechend parallel zur Walzrichtung gezeigt. Es ist deutlich erkennbar, wie ein sekundäres magnetisches Feld die Spannungsspitzen

[7] Die „Transformation des Integrals" lautet $\left(\mathcal{L} \int_0^t f(\tau)\,\mathrm{d}\,\tau\right)(s) = \frac{1}{s}(\mathcal{L}f)(s)$. Entsprechend dämpft die Integration mit einer Dekade pro Dekade.

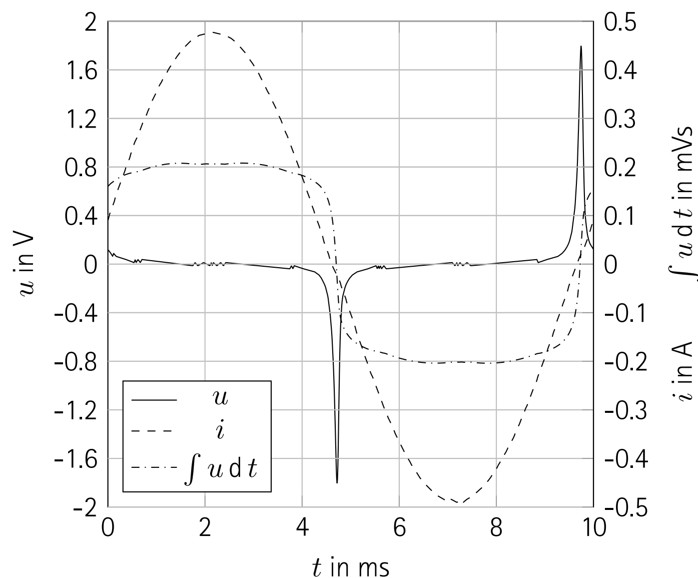


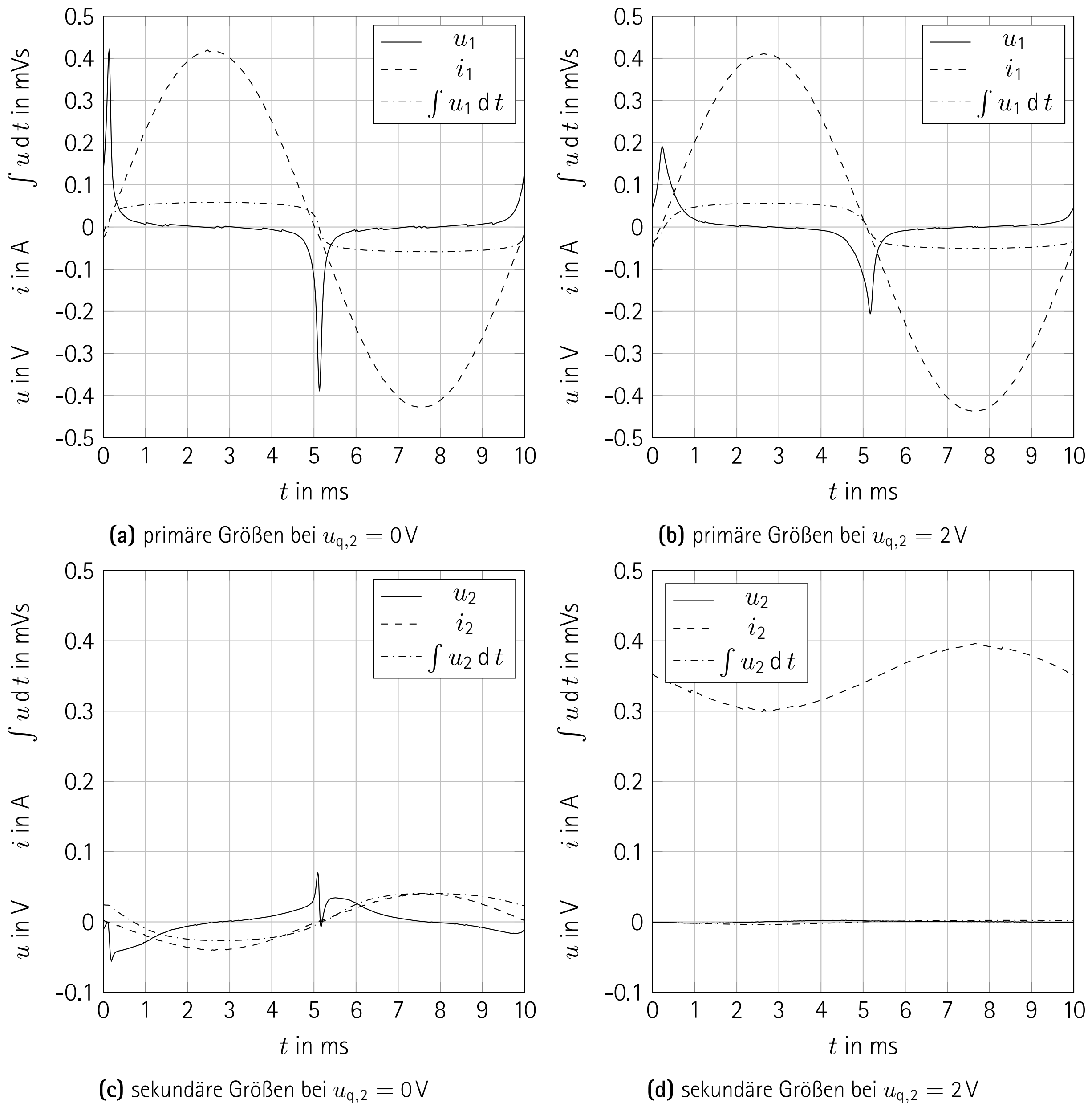
Abb. 5.7: Messung am „Single-Sheet" aus NO10 in Walzrichtung. Annähernd sinusförmiger Strom. Spannungsspitzen und Plateaus beim zeitlichen Spannungsintegral durch Aussteuerung weit in den Sättigungsbereich des Materials.

und damit die Steilheit des zeitlichen Spannungsintegrals auf der Primärseite reduziert, und einen geringen Einfluss auf die Amplitude des primären Spannungsintegrals hat. Das Übersprechen der primären Wechselspannung auf die Sekundärseite ist in erster Linie auf parasitäre Effekte zurückzuführen. Geschicktere Wicklungsdesigns zur Unterdrückung dieses Effekts können in zukünftigen Arbeiten umgesetzt werden. Man sieht, dass der sekundäre Strom nur in erster Näherung konstant ist und durch das Primärsignal beeinflusst wird. Ein wahrscheinlich vorhandener Gleichanteil des sekundären Spannungsintegrals kann messtechnisch nicht erfasst werden, weil die Messwicklung dieses kurzschließt. In *Abb. 5.11* sind die bereits zuvor ablesbaren Magnetisierungskennlinien festgehalten.

5.4.2 Magnetisierungskennlinien

Die Magnetsierungskennlinien sind bislang nur als Netzwerkgrößen bekannt. Für das „Single-Sheet" kann ein homogenes Feld angenommen werden, sodass die Kennlinie des Materials „NO10" für Felder parallel zur Walzrichtung in *Abb. 5.9b* und für Felder orthogonal zur Walzrichtung in *Abb. 5.10b* abgebildet sind. Die geringfügigen Abweichungen lassen sich nicht zuverlässig durch die Walzrichtung begründen, sondern können auch durch Abweichungen bei der Einspannung des Bleches verursacht sein.

Beim „Single-Cross" lässt sich nicht auf die Magnetisierungskennlinie des Materials schließen, weil kein homogenes Feld vorliegt. Die scheinbare Kennlinie ist in *Abb. 5.11* dargestellt. Die bereits beschriebene Verringerung der Spannungsintegralsteigungen durch einen orthogonalen Fluss sind deutlich sichtbar. Zusätzlich ist die Amplitude des Spannungsintegrals etwas verringert.



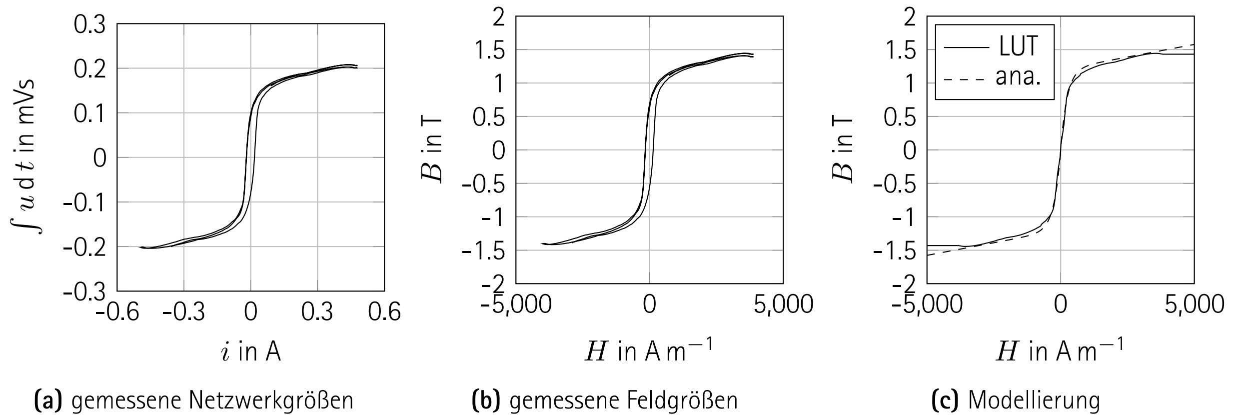
(a) primäre Größen bei $u_{\mathrm{q},2} = 0\,\mathrm{V}$

(b) primäre Größen bei $u_{\mathrm{q},2} = 2\,\mathrm{V}$

(c) sekundäre Größen bei $u_{\mathrm{q},2} = 0\,\mathrm{V}$

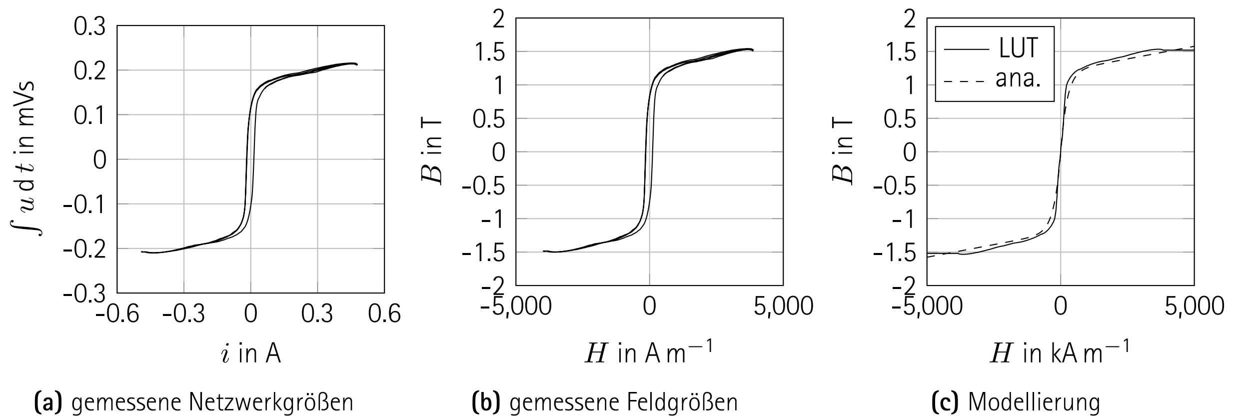
(d) sekundäre Größen bei $u_{\mathrm{q},2} = 2\,\mathrm{V}$

Abb. 5.8: Messung am „Single-Cross" (schmal) aus NO10 mit primärem Fluss orthogonal zur Walzrichtung. Ohne sekundäres Feld (links) und mit konstanter sekundärer Spannungsquelle, entsprechend annähernd konstantem sekundärem Strom (rechts). Orthogonaler Fluss reduziert nur unerheblich die primäre Flussamplitude, aber deutlich die primäre Flussänderung. Primärer Strom ist unverändert. Leichtes Übersprechen der primären Signale auf die sekundäre Seite (unten links), welches durch besseres Messwicklungsdesign verringerbar ist. Geringeres Übersprechen auf die sekundäre Spannung, aber Überlagerung von sekundärem DC-Strom mit übertragenem AC-Strom bei orthogonaler Magnetisierung. Wahrscheinlich vorhandener Gleichanteil des sekundären Spannungsintegrals messtechnisch nicht erfassbar.



(a) gemessene Netzwerkgrößen **(b)** gemessene Feldgrößen **(c)** Modellierung

Abb. 5.9: Magnetisierungskennlinie des „Single-Sheets" aus NO10 in Walzrichtung. Gemessene Netzwerkgrößen am „Single-Sheet" (links), korrespondierende homogene Feldgrößen (mittig) und verlustfreie Annäherung (rechts). Deutlich erkennbare Sättigung, schmale Hysterese, gute Modellierung über den gesamten gemessenen Bereich mit LUT (rechts, durchgezogen), befriedigende analytische Näherung (rechts, gestrichelt).



(a) gemessene Netzwerkgrößen **(b)** gemessene Feldgrößen **(c)** Modellierung

Abb. 5.10: Magnetisierungskennlinie des „Single-Sheets" aus NO10 orthogonal zur Walzrichtung. Gemessene Netzwerkgrößen am „Single-Sheet" (links), korrespondierende homogene Feldgrößen (mittig) und verlustfreie Annäherung (rechts). Annähernd identisch zu Magetnetisierungskennlinie und Näherungsmodellierung wie parallel zur Walzrichtung (siehe Abb. 5.9).

5.4.3 Modellierte Magnetisierungskennlinien

Das in der Simulation verwendete und zu evaluierende Materialmodell benötigt eine verlustfreie und isotrop geltende Magnetisierungskennlinie $B_1(H_1)$. Zu diesem Zweck wird das Material der Klasse *TFdmElementNonlinearIsotropicLossfreeLut* verwendet (siehe Abschnitt 4.2, S. 130). Es erzeugt eine LUT, in der es die Mittelwerte zwischen den beiden Kurven bildet, aus denen die Hystereseschleife besteht. Seine Interpretation der Messdaten ist für das Material „NO10" in Walzrichtung in *Abb. 5.9c* und orthogonal zur Walzrichtung in *Abb. 5.10c* illustriert. Es ist bemerkenswert, dass tatsächlich geringe Unterschiede deutlich werden, obwohl das Material

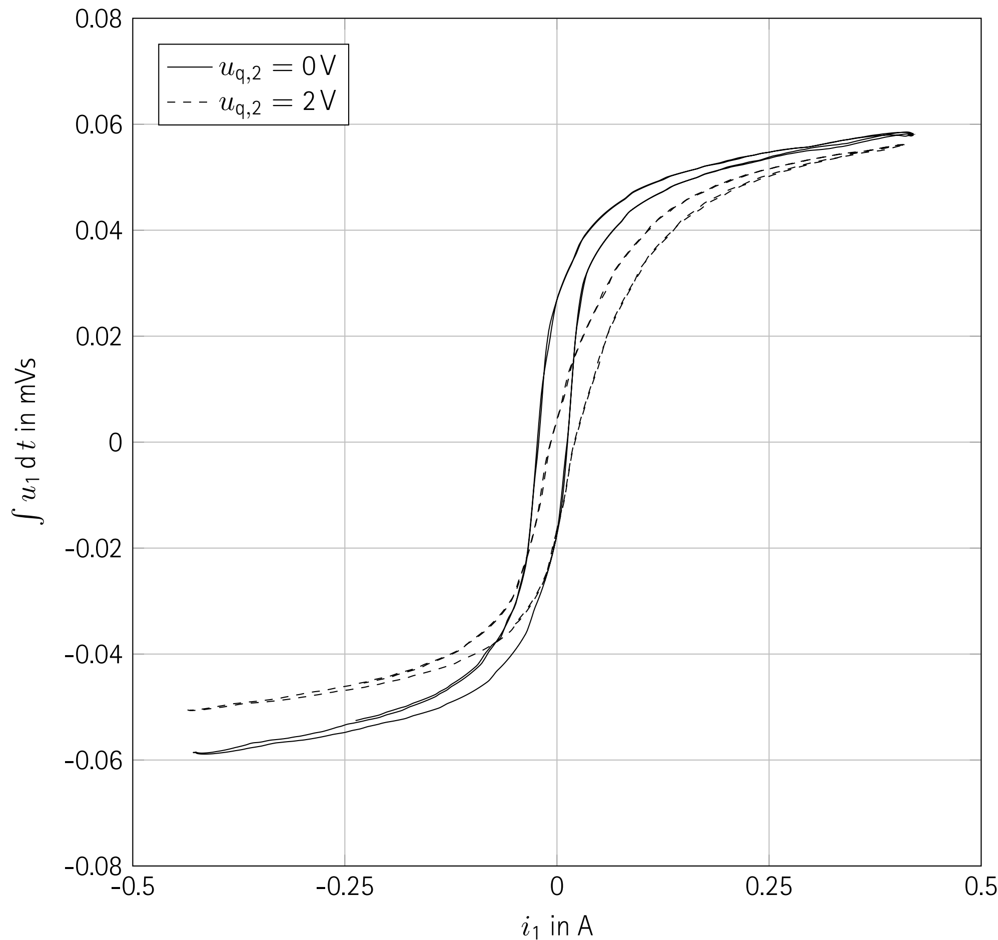


Abb. 5.11: Scheinbare Magnetisierungskennlinie des „Single-Crosses" (schmal) aus NO10 orthogonal zur Walzrichtung. Kein linearer Rückschluss auf Feldgrößen möglich wegen mangelnder Homogenität der Felder. Verringerung der Permeabilität und der Flussamplitude durch orthogonales Feld.

annähernd isotrop sein soll. Diese sind aber nur begrenzt reproduzierbar.[8] Bei den vorgestellten Messungen ist die Permeabilität für kleine Flussdichten parallel zur Walzrichtung größer, die Sättigungsflussdichte[9] ist orthogonal zur Walzrichtung größer. Mit Ausnahme der Hysterese ist die Modellierung augenscheinlich gut zutreffend. Weil keines der zu überprüfenden Modelle für orthogonale Magnetisierung für verlustbehaftete Materialien gilt und weil die Untersuchung der Verluste den Umfang der Arbeit übersteigen würde, ist diese Vereinfachung bei der Magnetisierungskennlinie unproblematisch.

Die Feldsimulationen (s. u.) funktionieren mit dem Materialmodell auf Basis einer LUT nur schlecht und teil-

[8] Die Unterschiede bleiben bestehen, wenn die Messung an einem eingespannten Blech wiederholt wird. Wenn das Blech entfernt und neu eingespannt wird, sind andere Unterschiede beobachtbar. Der Grund scheint in schwierig kontrollierbaren parasitären Reluktanzen zu liegen, auf die hier nicht weiter eingegangen werden soll.

[9] Das betreuende Institut stellt gering aufgelöste Messdaten für die Magnetisierungskennlinie des Materials „NO20" bereit, welches evtl. mit dem hier verwendeten „NO10" vergleichbar ist. Darin liegt die Sättigungsflussdichte bei etwa 2 T, die Kennlinie weist aber den gleichen charakteristischen Knick nahe 1,1 T auf.

weise gar nicht. Bei den durch verrauschte Messungen „eckige" Kurven konvergiert der Algorithmus nicht. Bei gering aufgelösten Tabellen, wie bspw. der vom betreuenden Institut für das Material „NO20" bereitgestellten, konvergiert das Lösungsverfahren nur sehr langsam und ist unbrauchbar.

Die Probleme bei der Feldsimulation mit einem Materialmodell mit einer LUT treten nicht bei dem analytischen Materialmodell auf. Das Modell ist in *Abb. 5.9c* abgedruckt. Der Einfachheit macht es bei der Annäherung einen Kompromiss zwischen der Kennlinie in den beiden Richtungen (parallel und orthogonal zur Walzrichtung), weshalb es sich in *Abb. 5.10c* um das selbe Modell handelt. Die Kennlinie ist sehr ähnlich zur der des Modells auf Basis einer LUT, weist aber alle Vorteile einer analytischen Magnetisierungskennlinie auf (siehe Abschnitt 4.2, S. 130).

5.4.4 Stationäre Simulation

Das analytische Materialmodell (s. o.) wird für die Simulation eines „Single-Crosses" genutzt. Dabei wird der gleiche Bereich für die primäre magnetische Spannung verwendet, wie bei der Messung am „Single-Cross", die oben diskutiert ist. In *Abb. 5.12* ist das simulierte Klemmenverhalten auf der Primärseite präsentiert, wobei für die magnetische Spannung in orthogonaler Richtung die gleichen Werte eingesetzt werden, wie bei der zuvor durchgeführten Messung. Allerdings ist die orthogonale magnetische Spannung (anders als bei der Messung) hier nicht annähernd, sondern exakt konstant. Die Form der Kurven weist Augenscheinlich eine hohe Ähnlichkeit mit der Messung auf.

5.4.5 Zeittransiente Simulation

Bei der Messung am „Single-Cross" kann der Strom durch die Sekundärwicklung nicht exakt konstant gehalten werden, wie bereits zuvor in *Abb. 5.8* gezeigt, weshalb für eine detailgetreue simulative Nachbildung der Messung zeittransiente Simulationen durchgeführt sind. Bei der Simulation sind die elektrischen Ströme aus den Messdaten übernommen und die elektrischen Spannungen und ihre zeitlichen Integrale sind simuliert. Es ist das Modell mit der zuvor gezeigten analytischen Magnetisierungskennlinie verwendet. Die Ergebnisse sind in *Abb. 5.13* festgehalten. Beim Vergleich mit der Messung in *Abb. 5.8* fällt die gute Übereinstimmung auf. Lediglich der Gleichanteil des zeitlichen Integrals der elektrischen Spannung an der sekundären Messwicklung ist in der Simulation korrekt berechnet, während er bei der Messung nicht erfasst werden kann.

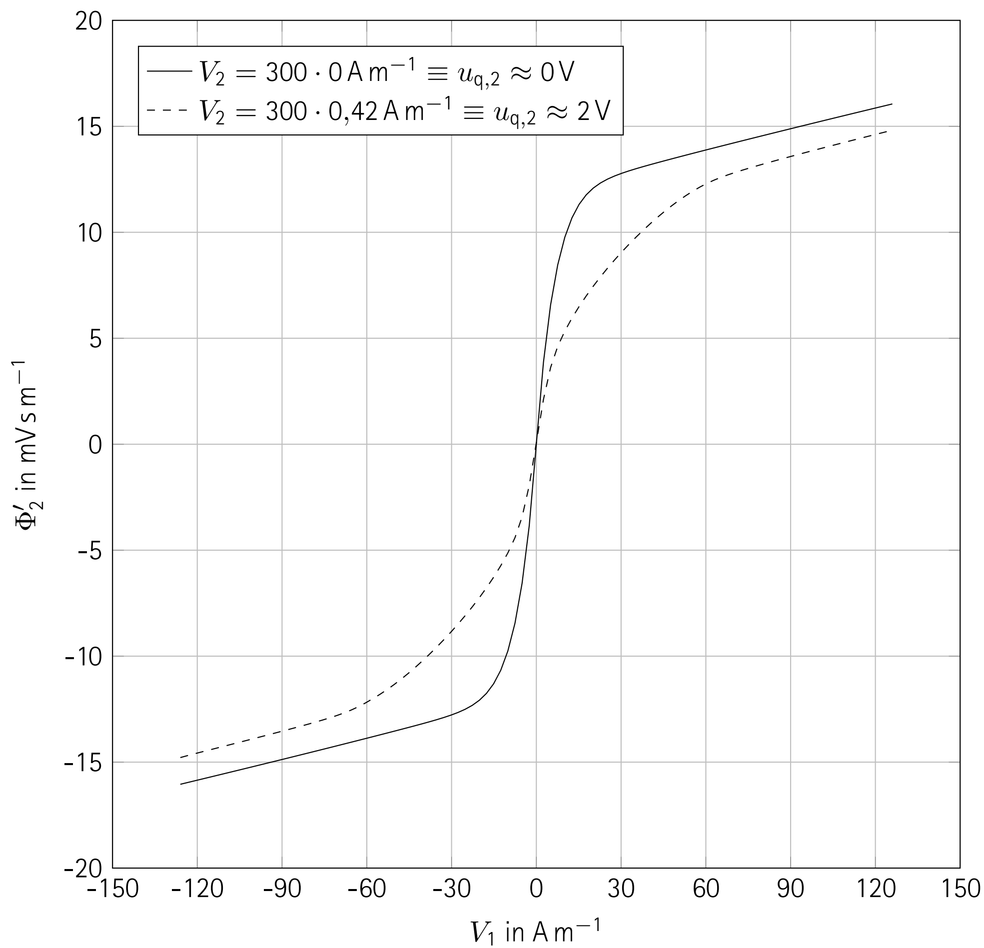
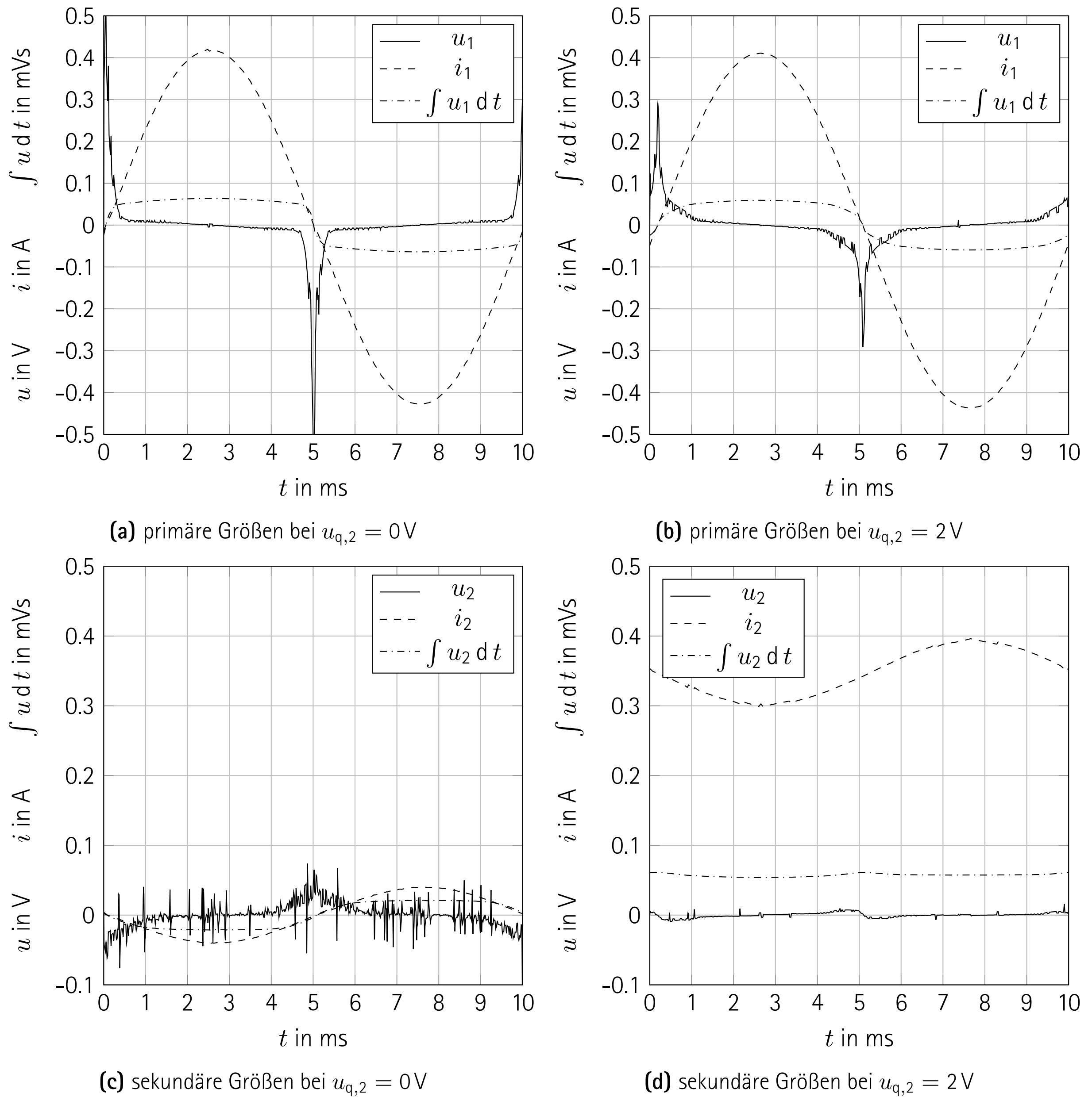


Abb. 5.12: Simuliertes Klemmenverhalten auf der Primärseite des „Single-Crosses" (schmal) aus NO10. Isotrope Kennlinie aus analytischer Näherung (fig:Versuch-Ergebnisse-mag-NO10-SST-orth-hbsim) für Materialmodell genutzt. Simulation ohne orthogonales Feld (durchgezogen) und mit vergleichbarer orthogonaler magnetischer Konstantspannungsquelle wie bei Messung (siehe Abb. 5.11). Geringere Flusssteigung und geringere Flussamplitude durch orthogonales Feld, außer nahe Koordinatenursprung. Geringe Schwankungen von sekundärem Strom **nicht** mitsimuliert.



(a) primäre Größen bei $u_{\mathrm{q},2} = 0\,\mathrm{V}$

(b) primäre Größen bei $u_{\mathrm{q},2} = 2\,\mathrm{V}$

(c) sekundäre Größen bei $u_{\mathrm{q},2} = 0\,\mathrm{V}$

(d) sekundäre Größen bei $u_{\mathrm{q},2} = 2\,\mathrm{V}$

Abb. 5.13: Simulative Nachbildung der Messung am „Single-Cross" (schmal) aus NO10 mit primärem Fluss orthogonal zur Walzrichtung. Aus der Messung übernommene Ströme und simulierte Spannungen und Spannungsintegrale. Ohne sekundäres Feld (links) und mit konstanter sekundärer Spannungsquelle, entsprechend annähernd konstantem sekundärem Strom (rechts). Gute Übereinstimmung mit der Messung (siehe Abb. 5.8). Gleichanteil des sekundären Spannungsintegrals (unten rechts) korrekt simuliert, welcher bei der Messung verloren geht.

5.5 Auswertung

Im Zentrum des Versuches steht die Aufgabe, ein Materialmodell zu entwickeln, welches der orthogonalen Magnetisierung gerecht wird. Dafür wird das Modell von Tez (siehe Abschnitt 1.4.3, S. 44) genutzt. Die Feldsimulationen ermöglichen den Vergleich der Simulationsergebnisse mit den Messergebnissen.

Beim Vergleich der Messung in *Abb. 5.8* und der Simulation in *Abb. 5.13* fallen die folgenden Abweichungen der Simulation gegenüber der Messung auf:

1. Die primäre Spannung weist eine etwas höhere Amplitude auf, sowohl mit als auch ohne orthogonalem Feld. Der Grund dafür ist unbekannt.
2. Sowohl die primäre als auch die sekundäre Spannung sind verrauscht, was auf die numerische Differentiation des zeitlichen Spannungsintegrals zurückzuführen ist.
3. Die Amplituden der Spannungsintegrale sind in sämtlichen Fällen korrekt berechnet.
4. Die Amplitude des sekundären Spannungsintegrals weist bei der orthogonalen Magnetisierung einen Gleichanteil auf, welcher korrekt simuliert und falsch gemessen ist, denn der Gleichanteil geht bei der Messung mit einer Wicklung verloren.
5. Der sekundäre Spannungsverlauf hat ohne orthogonale Magnetisierung einen nachvollziehbaren Verlauf, der von der Messung abweicht. Die bei den Extrema auftretenden Schwingungen in der Messung sind erklärbar.[10]
6. Die sekundäre Spannung weist bei orthogonaler Magnetisierung einen Verlauf mit einer geringfügig höheren Amplitude und einem wohldefinierten Verlauf auf. Die Messung versagt hier aufgrund der kleinen Signale.
7. Der Strom und das Spannungsintegral sind erwartungsgemäß in Phase. Bei der Messung besteht eine geringfügige Phasenverschiebung, die nicht erklärbar ist.

Eine Analyse des gemessenen Klemmenverhaltens der primären Reluktanz des „Single-Crosses" in *Abb. 5.11* und der Simulation in *Abb. 5.12* ergibt:

1. Bei der Messung beeinflusst eine orthogonales Feld die Steigung der Magnetiserungskennlinie auch nahe des Koordinatenursprungs, bei der Simulation nicht.
2. Die Hysterese ist bei der Messung unerklärlicherweise geringfügig an der i_1-Achse verschoben, wofür keine Erklärung bekannt ist. Die Simulation verwendet ein verlustfreies Materialmodell und weist entsprechend keine Hysterese auf.
3. Bei einem orthogonalen Feld ist die Kennlinie bei der Messung etwas an der Spannungsintegralachse

[10] Parasitäre Kapazitäten, bspw. zwischen den Windungen oder zwischen den Windungen und dem Blech bilden zusammen mit der Induktivität der Messwicklung einen Schwingkreis, welcher an den Stellen der plötzlichen, schnellen Flussänderung und dem damit einhergehenden Spannungsimpuls angeregt wird.

verschoben, bei der Simulation selbstverständlich nicht. Das Phänomen kann nicht erklärt werden.

4. In der Simulation sind ohne orthogonales Feld zwei Knicke und mit orthogonaler Magnetisierung vier Knicke in der Kennlinie beobachtbar. Diese sind in der Messung nur mit „genauem Hinschauen" sichtbar.
5. Sowohl in der Simulation als auch in der Messung ist die differentielle Permeabilität bzw. die differentielle Permeanz bei großen Feldstärken unabhängig von einem orthogonalen Feld.
6. Die gemessenen und simulierten Kennlinien sind insgesamt sehr ähnlich.

Sowohl die zeittransienten als auch die Simulationen des Klemmenverhaltens weisen eine sehr gute Übereinstimmung mit den Messungen auf. Lediglich die Modellierung von Verlusten in Form der Hysterese fehlt. Es sei darauf hingewiesen, dass hier ein verlustarmes, nahezu isotropes Material vermessen ist. Die Validierung des Modells für weitere Materialien steht aus.

6 Schlussbetrachtung

Science without results is just witchcraft.

– Cave Johnson

Im Rahmen der Arbeit sind eine Vielzahl an verschiedenen Aufgaben erfüllt.

Die Autorin hat sich intensiv mit den Grundlagen magnetischer Netzwerke beschäftigt und nützliche Werkzeuge identifiziert. Den Lesenden ist außerdem ein Einstieg in die zum Verständnis der Arbeit hilfreichen Themen gegeben. Es ist das ernüchternd umfangarme Ergebnis einer Literaturrecherche zur Modellierung orthogonaler Magnetisierung festgehalten.

Anschließend ist ein Messaufbau zur Vermessung von Vierpolen entwickelt, welcher aufgrund von Unzulänglichkeiten der verwendeten Hardware (insbesondere einer Falschangabe in den technischen Spezifikationen des „RedPitaya") improvisiert umgebaut wird. Mithilfe eines externen Oszilloskops nimmt der Aufbau vollautomatisch(!) Messungen auf und bearbeitet sie so nach, dass sie anschließend ausgewertet werden können. Der Umfang an verschiedenen Software-Werkzeugen kann einfach erweitert werden, weil eine Bibliothek zum Zugriff auf die Ein- und Ausgänge mitsamt abstrakter und konkreter Features implementiert ist.

Ein Verfahren zur Herstellung gekrümmter Eisenbleche kann nicht entwickelt werden, weil dafür die Bearbeitungszeit nicht ausreicht. Es ist allerdings ein vielversprechender Ansatz gefunden, der in späteren Arbeiten weiterverfolgt werden kann. Außerdem ist als Ersatz für den Torus, welcher zwei homogene, orthogonale Magnetfelder erlaubt, mit der Simulationssoftware ein Werkzeug geschaffen, welches die Aufgabe der Verknüpfung eines Materialmodells mit einem Klemmenmodell auch ohne homogene Felder erfüllt.

Eine mächtige Software zur Netzwerk- und Feldsimulation bietet umfangreiche Möglichkeiten zur Entwicklung von Simulationswerkzeugen und stellt bereits einige fertige Implementierungen bereit. Es ist bemerkenswert, welches hohe Maß an Wiederverwendbarkeit, Erweiterbarkeit und Kompatibilität erreicht ist, sodass beliebige Simulationen durchgeführt werden können. Bspw. können Netzwerk- und Feldsimulationen miteinander gekoppelt werden, es ist aber auch die Erweiterung um zuvor nicht verwendete Domänen der Physik möglich. Die Software dient insbesondere dem Zweck, eine Schnittstelle zwischen einem intrinsischen Materialmodell und dem extrinsischen Verhalten eines gesamten Aufbaus bereitzustellen und erfüllt diesen im Rahmen des Versuchs einwandfrei.

Im Versuch werden die zuvor erarbeiteten Ressourcen genutzt. Dabei ist natürlich auf die gekrümmten Geometrien, insbesondere den Torus, verzichtet, weil die Erarbeitung des dafür notwendigen Herstellungsverfahrens

nicht abgeschlossen ist. Es werden magnetische Spannungen und Flüsse durch verschiedene Geometrien mithilfe des Messaufbaus erzeugt und gemessen. Unter Zuhilfenahme der magnetischen Netzwerkanalyse und der Simulationssoftware sind die Ergebnisse verarbeitet, sodass ein Vergleich zwischen dem Modell von Tez und dem untersuchten Material „NO10" angestellt werden kann.

Das Ergebnis ist überraschend: Mit Ausnahme der Hysterese und kleinen Abweichungen beschreibt das Modell von Tez die physikalischen Vorgänge im Versuch qualitativ korrekt und sogar mit einer unerwarteten quantitativen Richtigkeit und Präzision. Die Übertragung des intrinsischen Modells auf Netzwerkgrößen mithilfe der Simulationssoftware funktioniert einwandfrei und mit überschaubarem Rechenaufwand. Das Modell gilt allerdings lediglich für das verlustarme, nahezu isotrope „NO10", weil im Rahmen der Arbeit keine weiteren Materialien zur Verfügung stehen.

Die nicht erfüllte Aufgabe, verlustbehaftete und anisotrope (ferro-)magnetische Materialien auf das Prinzip der orthogonalen Magnetisierung zu untersuchen, wird an nachfolgende Arbeiten weitergegeben. Dazu ist ein umfangreiches Repertoire an Werkzeugen entwickelt, welches wiederverwendet und erweitert werden kann.

Ihnen bleiben nach dem Lesen noch Fragen zur vorliegenden Arbeit? Kontaktieren Sie die Autorin unter:
ortho-mag@schulz.com.de

Literatur

[1] 3Druck.com: *Aceton Dampf sorgt für glatte Oberfläche bei ABS Objekte (Update)*. https://3druck.com/objects/aceton-dampf-sorgt-fuer-glatte-oberflaechen-bei-abs-objekte-129150/, abgerufen am 15. Juni 2020.

[2] S. D. Wanlass et al: *The Paraformer, A New Passive Power Conversion Device*. IEEE, Wescon Technical Papers Vol. 12, No. 2, 1968.

[3] F. S. Attia und F. Leydecker: *Mathematik III für Ingenieure*. Skript zur Vorlesung, 2015.

[4] F. S. Attia und F. Leydecker: *Mathematik IV für Ingenieure*. Skript zur Vorlesung, 2016.

[5] B. D. Bedford: *Magnetic Saturation Device*. US Patentantrag 2 636 158, 1948.

[6] M. Binnewies, M. Jäckel, H. Willner und G. Rayner-Canham: *Allgemeine und Anorganische Chemie*. Spektrum Akademischer Verlag, 2004.

[7] Wikimedia Commons: *File:VFPt magnets BHM.svg*. https://en.wikipedia.org/wiki/File:VFPt_magnets_BHM.svg, abgerufen am 13. Juni 2020.

[8] Red Pitaya d.d.: *Red Pitaya 0.97 documentation*. https://redpitaya.readthedocs.io/en/latest/developerGuide/125-14/extent.html#auxiliary-analog-input-channels, abgerufen am 15. Juni 2020.

[9] Red Pitaya d.d.: *Red Pitaya STEMlab board*. https://www.redpitaya.com/f130/STEMlab-board, abgerufen am 04. Mai 2020.

[10] Endeavor Business Media Machine Design: *What's The Difference Between FEM, FDM, and FVM?* https://www.machinedesign.com/3d-printing-cad/fea-and-simulation/article/21832072/whats-the-difference-between-fem-fdm-and-fvm, abgerufen am 13. Juni 2020.

[11] H. P. Diaz, J. M. Codes, M. Saliternig und P. Maisel: *PCIM Europe Messe-Guide – Wärmeabfuhr bei induktiven Komponenten*.

[12] Git user „RedPitaya“: *RedPitaya/Makefile at master – RedPitaya/RedPitaya – GitHub*. https://github.com/RedPitaya/RedPitaya/blob/master/Examples/C/Makefile, abgerufen am 04. Mai 2020.

[13] EPCOS AG: *Ferrites and accessories – U 93/76/30 with I 93/28/30Cores*, 2017. https://www.tdk-electronics.tdk.com/inf/80/db/fer/u_93_76_30.pdf, abgerufen am 09. Juni 2020.

[14] J. Friebe: *Permanentmagnetische Vormagnetisierung von Speicherdrosseln in Stromrichtern.* kassel university press GmbH, 2014.

[15] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Entwurfsmuster.* Addison-Wesley Verlag, 2011.

[16] Galliot Lehrmittel GbR: *Stabmagnet, rund.* http://shop.galiot-lehrmittel.de/Stabmagnet-rund, abgerufen am 13. Juni 2020.

[17] W. A. Geyger: *Application of magnetic amplifiers in industrial instrumentation and control.* IRE Transactions on Industrial Electronics, vol. PGIE-5, 1958.

[18] H. Haase, H. Garbe und H. Gerth: *Grundlagen der Elektrotechnik – 4. erweiterte Auflage.* Schöneworth Verlag, 2018.

[19] D. C. Jiles und D. L. Atherton: *Theory of ferromagnetic hysteresis.* Journal of Applied Physics 55, 1984.

[20] R. Kaiser: *C++ mit dem Borland C++ Builder 2007: Einführung in den C++-Standard und die objektorientierte Windows-Programmierung.* Xpert.press. Springer Berlin Heidelberg, 2007.

[21] E. S. Kuh und I. N. Hajj: *Nonlinear circuit theory: Resistive networks.* Proceedings of the IEEE, 59(3):340–355, March 1971.

[22] K. Küpfmüller, W. Mathis und A. Reibiger: *Theoretische Elektrotechnik : Eine Einführung – 19., aktualisierte Auflage.* Springer Vieweg, 2013.

[23] T. Lilge: *Regelungstheorie: Nichtlineare Systeme.* Skript zur Vorlesung, 2015.

[24] J. Pfeiffer, P. Küster, I. Schulz, J. Friebe und P. Zacharias: *Review of flux interaction of differently aligned magnetic fields in inductors and tansformers.* unveröffentlicht, 2020.

[25] D. W. Ver Planck, L. A. Finzi und D. C. Beaumariage: *Analytical determination of characteristics of magnetic amplifiers with feedback.* Transactions of the American Institute of Electrical Engineers, vol. 68, no. 1, 1949.

[26] I. E. M. Schulz: *Entwurf und Implementierung eines Speicher- und Energiemanagementsystems für einen Schwungradspeicher.* Bachelorarbeit, unveröffentlich, 2017.

[27] I. E. M. Schulz: *Hiwi-Tätigkeitsbericht: Preliminary studies about orthogonal magnetization.* unveröffentlicht, 2018.

[28] A. S. Tanenbaum: *Moderne Betriebssysteme.* Pearson Studium – IT. Pearson Deutschland, 2009.

[29] Texas Instruments: *OPA548 High-Voltage, High-Current Operational Amplifier*, 1997. https://www.ti.com/lit/ds/symlink/opa548.pdf, abgerufen am 04. Mai 2020.

[30] Texas Instruments: *LMx24,LMx24x,LMx24xx,LM2902,LM2902x,LM2902xx,LM2902xxx – QuadrupleOperationalAmplifiers*, 2015. http://www.ti.com/lit/ds/symlink/lm324.pdf, abgerufen am 04. Mai 2020.

[31] E. S. Tez: *The Parametric Transformer.* E. S. Tez, 1977.

[32] Prof. M. Welzl: *Socketprogrammierung mit ANSI – C und LINUX.*

http://heim.ifi.uio.no/michawe//teaching/cn-alt/unterlagen/csockets/index.html, abgerufen am 04. Juni 2020.

A Daten-CD

B Programmcode

B.1 Verlustreluktanz (MATLAB) . . . 174
B.2 Transformierte Kapatität (MATLAB) . . . 176
B.3 Zerlegte Reluktanz (MATLAB) . . . 179
B.4 Clientseitige Backups vom RedPitaya (Bash) . . . 180
B.5 Makefile für den RedPitaya (Makefile) . . . 181
B.6 Kalibrierungswerkzeug (C++) . . . 183
B.7 Kalibrierungsübersicht (Bash) . . . 185
B.8 Kalibrierungsaneignung (Bash) . . . 186
B.9 Funktionsgenerator (C++) . . . 187
B.10 Messgerät (C++) . . . 193
B.11 Zweipolcharakterisierung (C++) . . . 197
B.12 intelligente Vierpolcharakterisierung (nicht fertiggestellt) (C++) . . . 206
B.13 primitive Vierpolcharakterisierung (C++) . . . 215
B.14 improvisierte Vierpolcharakterisierung (C++) . . . 221
B.15 Spannungsintegralberechnung (C++) . . . 232
B.16 Nachträgliche Kalibrierung (C++) . . . 237
B.17 Glättung der Quantisierung (C++) . . . 241
B.18 Simulation (C++) . . . 245

Programmcode B.1: Verlustreluktanz (MATLAB)

```
tic;
f = 50;              % Netzfrequenz
w = 2*pi*f;          % Kreiszahl omega
T = 1/f;             % Periodendauer
t_offset = 50 * T; % Berechnungsdauer vorm betrachteten Zeitraum, um
                     % stationären Zustand zu erreichen
N = 200;             % Windungszahl der Wicklung
R = 20;              % Wicklungswiderstand
Rm = 150000;         % Reluktanz des gesamten Kreises (linear)
eps_t = 1*10^-6;     % 1..10 µs eignen sich
t = [-t_offset:eps_t:T];
u = 1*cos(w*t);      % Spannungsquelle

% Startvektor in der Zeitdimension alle Größen gleich null (stimmt für
% wL >> R). Hier ist wL > R, daher stellt sich erst nach ein paar
% Perioden der stationäre Zustand ein.
i    = NaN(size(t)); i    (1) = 0; % Strom durch die Wicklung
v    = NaN(size(t)); v    (1) = 0; % magnetische Spannung
psi  = NaN(size(t)); psi  (1) = 0; % realer Fluss
psi0 = NaN(size(t)); psi0 (1) = 0; % gesamter Fluss bei Verwendung einer
                                   % Ersatzreluktanz (identisch zu realem
                                   % Fluss bei R=0)
for j=2:size(t,2)
    i(j)    = i(j-1);
    v(j)    = v(j-1);
    for(k=1:10) % iterative Lösung, bei mehr als ca. 5 Schritten kann das
                % Ergebnis durch FLOPs weniger genau werden.
        psi(j) = psi(j-1) + 1/N * (u(j)-R*i(j)) * eps_t;
        psi0(j) = psi0(j-1) + 1/N * u(j) * eps_t;
        v(j) = psi(j) * Rm;
        i(j) = 1/N * v(j);
    end
end

% nur die letzte exakt eine Periode anzeigen, der Rest zählt als
% Einschwingvorgang
u = u(size(t,2)-T/eps_t:size(t,2));
i = i(size(t,2)-T/eps_t:size(t,2));
v = v(size(t,2)-T/eps_t:size(t,2));
psi = psi(size(t,2)-T/eps_t:size(t,2));
psi0 = psi0(size(t,2)-T/eps_t:size(t,2));
t = t(size(t,2)-T/eps_t:size(t,2));

psiV = psi0-psi;

t_RmV = t; % plot wird erst später erstellt und t wird noch bearbeitet.
RmVsim = v./(psiV); % theoretische Verlustreluktanz, wie in Simulation gemessen
RmVsim = RmVsim(2:end);
RmV = (N*N/R*diff(psiV)./eps_t) ./ (psiV(2:end)); % theoretische Verlustreluktanz, wie mit
    Formel berechnet
fprintf('maximale Abweichung Formel vs. Messung: %f ppm\n',max(abs(RmV./RmVsim-1))*10^6);
```

```
n = round(10^-4 / eps_t);
t = t(1:n:end);
u = u(1:n:end);
55 i = i(1:n:end);
v = v(1:n:end);
psi = psi(1:n:end);
psi0 = psi0(1:n:end);
psiV = psiV(1:n:end);
60
% Zeitlicher Verlauf der Netzwerkgrößen
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Zeitlicher Verlauf');
    hold on;
65  plot(t,u,'k-');
    plot(t,i*100,'k--');
    plot(t,v,'k:');
    plot(t,psi*10^5,'k-.');
    legend('u','i \cdot 10^2','v','\psi \cdot 10^5');
70  grid on;
    writetable(array2table([t.',u.',i.'*100,v.',psi.'*10^5,psi0.'*10^5,(psi0.'-psi.')*10^5],
        ...
        'VariableNames',{'t','u','iE2','v','psiE5','psi0E5','psiVE5'}),'t.csv','Delimiter','\t');

% Kennlinie der Reluktanz, Kennschleife der theoretischen Verlustreuktanz
75 % und Summe beider
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Kennlinien');
    hold on;
    plot(v,psi,'k:');
80  plot(v,psiV,'k--');
    plot(v,psi0,'k-');
    legend('physikalische Reluktanz','Verlustreluktanz', ...
        'Parallelschaltung','Location','Northwest');
    xlabel('V in A');
85  ylabel('\Psi in Vs');
    grid on;
    writetable(array2table([v.',psi.',psiV.',psi0.'], ...
        'VariableNames', {'v','psi','psiV','psi0'}),'Rm.csv','Delimiter','\t');

90 % Abweichung der Verlustreluktanz (v/(psi_V)) von der berechneten
% Verlustreluktanz (N^2/R * d/dt)
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Abweichung Simulation vs. Formel');
    hold on;
95  plot(t_RmV(2:end),(RmV./RmVsim-1)*10^6,'k-');
    xlabel('t in s');
    ylabel('Abweichung in ppm');
    grid on;
    writetable(array2table([t_RmV(2:end).',(RmV./RmVsim-1).'*10^6], ...
100     'VariableNames', {'t','err'}), 'Rmerr.csv','Delimiter','\t');
toc;
```

Programmcode B.2: Transformierte Kapatität (MATLAB)

```
tic;
f = 50;             % Netzfrequenz
w = 2*pi*f;         % Kreiszahl omega
T = 1/f;            % Periodendauer
t_offset = 10 * T;  % Berechnungsdauer vorm betrachteten Zeitraum, um
                    % stationären Zustand zu erreichen
N = 200;            % Windungszahl der Wicklung
C = 0.00015;        % Reihenkapazität
Rm = 150000;        % Reluktanz des gesamten Kreises (linear)
eps_t = 1*10^-8;    % 10n s eignen sich bei t_offset = 10*T
t = [-t_offset:eps_t:T];
u = 1*cos(w*t);     % Spannungsquelle

% Startvektor in der Zeitdimension alle Größen gleich null (stimmt für
% wL >> 1/(wC)). Hier ist wL > 1/(wC), daher stellt sich erst nach ein paar
% Perioden der stationäre Zustand ein. Nur das Stromzeitintegral ist
% schlecht geschätzt.
i    = NaN(size(t));  i   (1) = 0;  % Strom durch die Wicklung
v    = NaN(size(t));  v   (1) = 0;  % magnetische Spannung
psi  = NaN(size(t));  psi (1) = 0;  % realer Fluss
psi0 = NaN(size(t));  psi0(1) = 0;  % gesamter Fluss bei Verwendung einer
                                    % Ersatzreluktanz (identisch zu realem
                                    % Fluss bei R=0)
idt  = NaN(size(t));  idt (1) = - max(u) / ((w*N*N/Rm-1/(w*C))*w);
                                    % Stromzeitintegral
                                    % Alternative zum Startwert: Verwendung
                                    % eines Reihenwiderstandes mit 1 Ohm
for j=2:size(t,2)
    i(j)    = i(j-1);
    v(j)    = v(j-1);
    for(k=1:10)  % iterative Lösung, bei mehr als ca. 5 Schritten kann das
                 % Ergebnis durch FLOPs weniger genau werden.
        idt(j) = idt(j-1) + i(j)*eps_t;
        psi(j) = psi(j-1) + 1/N * (u(j)-1/C *idt(j)) * eps_t;
        psi0(j) = psi0(j-1) + 1/N * u(j) * eps_t;
        v(j) = psi(j) * Rm;
        i(j) = 1/N * v(j);
    end
end

% nur die letzte exakt eine Periode anzeigen, der Rest zählt als
% Einschwingvorgang
u = u(size(t,2)-T/eps_t:size(t,2));
i = i(size(t,2)-T/eps_t:size(t,2));
v = v(size(t,2)-T/eps_t:size(t,2));
psi = psi(size(t,2)-T/eps_t:size(t,2));
psi0 = psi0(size(t,2)-T/eps_t:size(t,2));
idt = idt(size(t,2)-T/eps_t:size(t,2));
t = t(size(t,2)-T/eps_t:size(t,2));

psiV = psi0-psi;
```

```
t_RmC = t;  % plot wird erst später erstellt und t wird noch bearbeitet.
RmCsim = v./(psiV);  % theoretische Transformierte Kapazität, wie in Simulation gemessen
RmCsim = RmCsim(2:end);
RmC = -N*N*w*w*C;  % theoretische Transformierte Kapazität, wie mit Formel berechnet  % TODO:
    d²/dt² statt -w²
fprintf('maximale Abweichung Formel vs. Messung: %f ppm\n',max(abs(RmC./RmCsim-1))*10^6);

n = round(10^-4 / eps_t);
t = t(1:n:end);
u = u(1:n:end);
i = i(1:n:end);
v = v(1:n:end);
psi = psi(1:n:end);
psi0 = psi0(1:n:end);
psiV = psiV(1:n:end);
idt  = idt(1:n:end);

% Zeitlicher Verlauf der Netzwerkgrößen
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Zeitlicher Verlauf');
    hold on;
    plot(t,u,'k-');
    plot(t,i*100,'k--');
    plot(t,v,'k:');
    plot(t,psi*10^5,'k-.');
    plot(t,idt*5*10^4);
    legend('u','i \cdot 10^2','v','\psi \cdot 10^5','idt\cdot5\cdot10^4');
    grid on;
    writetable(array2table([t.',u.',i.'*100,v.',psi.'*10^5,psi0.'*10^5,(psi0.'-psi.')*10^5],
        ...
        'VariableNames',{'t','u','iE2','v','psiE5','psi0E5','psiVE5'}),'t.csv','Delimiter','\t');

% Kennlinie der Reluktanz, Kennschleife der theoretischen Transformierten Kapazität
% und Summe beider. Nur zwischen wt=pi/2 und wt=3pi/2, um  hübsche Graphik
% zu erhalten - Achtung: Fälschliche Hysterese könnte übersehen werden
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Kennlinien');
    hold on;
    plot(v(round(size(v,2)/4):round(size(v,2)*3/4)),psi(round(size(v,2)/4):round(size(v,2)*3/4)),'k:');
    plot(v(round(size(v,2)/4):round(size(v,2)*3/4)),psiV(round(size(v,2)/4):round(size(v,2)*3/4)),'k--');
    plot(v(round(size(v,2)/4):round(size(v,2)*3/4)),psi0(round(size(v,2)/4):round(size(v,2)*3/4)),'k-');
    legend('physikalische Reluktanz','Transformierte Kapazität', ...
           'Parallelschaltung','Location','Northwest');
    xlabel('V in A');
    ylabel('\Psi in Vs');
    grid on;
    writetable(array2table([v.',psi.',psiV.',psi0.'], ...
        'VariableNames', {'v','psi','psiV','psi0'}),'Rm.csv','Delimiter','\t');

% Abweichung der Transformierten Kapazität (v/(psi_V)) von der berechneten
% Transformierten Kapazität (-N^2*w^2*C)
figure; % nur Veranschaulichung - finale Abb. in TikZ
    title('Abweichung Simulation vs. Formel');
    hold on;
```

```
105     plot(t_RmC(2:end),(RmC./RmCsim-1)*10^6,'k-');
        xlabel('t in s');
        ylabel('Abweichung in ppm');
        ylim([-1E2 1E2]);
        grid on;
110     writetable(array2table([t_RmC(2:end).',(RmC./RmCsim-1).'*10^6], ...
            'VariableNames', {'t','err'}), 'Rmerr.csv','Delimiter','\t');
    toc;
```

Programmcode B.3: Zerlegte Reluktanz (MATLAB)

```
v = [-1:0.01:1];
psi_lin  = v/1;                        % linearer Teil
a=1;
b=1;
psi_ges  = ((b.^3./(27.*a.^3) + v.^2./(4.*a.^2)).^(1./2) + v./(2.*a)).^(1./3) - ...
    b./(3.*a.*((b.^3./(27.*a.^3) + v.^2./(4.*a.^2)).^(1./2) + v./(2.*a)).^(1./3));
                                       % Inverse von v = a*psi.^3 + b*psi
                                       % linearer + nichtlinearer Teil
psi_nl   = psi_ges - psi_lin;          % nichtlinearer Teil
k=1;
psi_hysp = +0.05*(sqrt(1-k*v.^2).^5); % potenzierte Ellipse, willkürlich
                                       % oberer Hystereseteil
psi_hysn = -0.05*(sqrt(1-k*v.^2).^5); % unterer Hystereseteil
psi_gesp = psi_ges+psi_hysp;           % oberer gesamter Verlauf
psi_gesn = psi_ges+psi_hysn;           % unterer gesamter Verlauf

figure;
    hold on;
    plot(v,psi_lin,'k--');
    plot(v,psi_nl,'k:');
    plot(v,psi_hysp,'k-.');
    plot(v,psi_gesp,'k-');

    plot(v,psi_hysn,'k-.');
    plot(v,psi_gesn,'k-');
%     plot(v,psi_ges ,'k-');
    legend('lin','nl','hys','ges','Location','Northwest');
    grid on;

    writetable(array2table([v.',psi_lin.',psi_nl.',psi_hysp.',psi_hysn.',...
        psi_gesp.',psi_gesn.',psi_ges.'], ...
        'VariableNames',{'v','psi_lin','psi_nl','psi_hysp','psi_hysn',...
        'psi_gesp','psi_gesn','psi_ges'}),'Rm.csv','Delimiter','\t');
```

Programmcode B.4: Clientseitige Backups vom RedPitaya (Bash)

```bash
#!/bin/bash

#tmpdir=$(mktemp --directory -p /home/ilka/backup_rp)
tmpdir="/home/ilka/backup_rp/$(date +'%Y-%m-%d - %H:%M')"
mkdir "$tmpdir"

cd "$tmpdir" && sftp ilka@redpitaya:/home/ilka/OrthoMeasure <<EOF
get -r *
bye
EOF

ls -lah "$tmpdir"
du -sh "$tmpdir"
chmod u-w -R "$tmpdir"
#rm -r "$tmpdir"
```

Programmcode B.5: Makefile für den RedPitaya (Makefile)

```
CXXFLAGS  = -std=c++17 -g -Wall -Werror
CXXFLAGS += -I/opt/redpitaya/include
CXXFLAGS += -I/home/ilka/OrthoMeasure
LDFLAGS   = -L/opt/redpitaya/lib
LDLIBS    = -lm -lpthread -lrp -lstdc++fs

#SRCS=$(wildcard *.cpp)
#OBJS=$(SRCS:.cpp=.o)

#$(info OBJS is [${OBJS}])

# IMPORTANT: run 'make clean' after making changes to the librarie's classes' data structures


all: calib fungen meter impmsr orthomsr orthomsr-simple orthomsr-scope udt smth converter

%.o: %.cpp %.hpp
	$(CXX) -c $(CXXFLAGS) $< -o $@

clean:
	$(RM) *.o
	$(RM) src/*.o
	$(RM) src/interface/*.o
	$(RM) calib fungen meter impmsr orthomsr orthomsr-simple orthomsr-scope udt smth converter

#test: test.o test2.o
#	$(CXX) $(LDFLAGS) $^ -o $@

calib:  calib.cpp \
	src/util.o \
	src/interface/interface.o \
	src/interface/input.o \
	src/interface/output.o \
	src/interface/impedancemeasuresetup.o
		$(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

fungen: fungen.cpp \
	src/util.o \
	src/interface/interface.o \
	src/interface/input.o \
	src/interface/output.o \
	src/interface/impedancemeasuresetup.o
		$(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

meter: meter.cpp \
	src/util.o \
	src/interface/interface.o \
	src/interface/input.o \
	src/interface/output.o \
	src/interface/impedancemeasuresetup.o
		$(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

```
impmsr: impmsr.cpp \
    src/util.o \
    src/interface/interface.o \
    src/interface/input.o \
    src/interface/output.o \
    src/interface/impedancemeasuresetup.o
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

orthomsr: orthomsr.cpp \
        src/util.o \
        src/interface/interface.o \
        src/interface/input.o \
        src/interface/output.o \
        src/interface/impedancemeasuresetup.o
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

orthomsr-simple: orthomsr-simple.cpp \
        src/util.o \
        src/interface/interface.o \
        src/interface/input.o \
        src/interface/output.o \
        src/interface/impedancemeasuresetup.o
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

orthomsr-scope: orthomsr-scope.cpp \
        src/util.o \
        src/interface/interface.o \
        src/interface/output.o \
        src/scope.o
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@

udt:  udt.cpp \
    src/util.o

smth: smth.cpp \
    src/util.o

converter: converter.cpp \
        src/util.o \
        src/interface/interface.o \
        src/interface/input.o \
        src/interface/output.o \
        src/interface/impedancemeasuresetup.o
        $(CXX) $(CXXFLAGS) $(LDFLAGS) $^ $(LDLIBS) -o $@
```

Programmcode B.6: Kalibrierungswerkzeug (C++)

```cpp
#include <iostream>

#include "src/util.hpp"
#include "src/interface/input.hpp"
#include "src/interface/output.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;

int main(int argc, char **argv) {
    int retval=2;
    cout << std::scientific;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) ) {
            cerr << "usage: calib --help"<<endl
                 << "usage: calib [OPTIONS] CLASS1 NAME1 [CLASS2 NAME2 [...] ]"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                    prints this message"<<endl
                 << " -v --verbose                 verbose output"<<endl
                 << endl
                 << "classes:"<<endl
                 << " -i --input                   analog input pin"<<endl
                 << " -o --output                  analog output pin"<<endl
                 << " -z --impedancemeasuresetup  impedance measure setup"<<endl
                 << endl
                 << "names:"<<endl
                 << " * run ./lscalib to list available names sorted by class"<<endl
                 << " * OR specify an existing name to edit that interface"<<endl
                 << " * OR specify an non-existent name to create that interface"<<endl
                 << endl
                 << "For full documentation, see master thesis by Ilka Schulz or
                    contact:"<<endl
                 << "Leibniz Universität Hannover"<<endl
                 << "Institut für Antriebssystem und Leistungselektronik"<<endl
                 << "z. Hd. Ilka Schulz"<<endl
                 << "Welfengarten 1"<<endl
                 << "30167 Hannover"<<endl
                 << "Germany"<<endl;
        }
        else { // actual program
            initHardwareAccess();
            string sw=""; // command line switch
            for(int i=1; i<argc; i++) {
                string a = argv[i]; // currently processed argument
                if(sw=="-i" || sw=="--input") {
                    TInput(a,false).guidedSetup(false);
                    sw = "";
                }
                else if(sw=="-o" || sw=="--output") {
                    TOutput(a,false).guidedSetup(false);
                    sw = "";
                }
```

```
                else if(sw=="-z" || sw=="--impedancemeasuresetup") {
                    TImpedanceMeasureSetup(a,false).guidedSetup(false);
                    sw = "";
55              }
                else {
                    if(!a.empty() && a[0]=='-') { // argument is a switch
                        if(a=="-v" || a=="--verbose")
                            verbose = true;
60                      else if(sw=="")
                            sw = a;
                        else
                            ERROR("bad command line arguments. Check your syntax around
                                arguments >"+sw+"< and >"+a+"<.");
                    }
65                  else { // targets
                        ERROR("bad command line arguments. Do not provide targets");
                    }
                }
            }
70      }
        retval=0;
    }
    catch(TException &e) {
        e.show();
75      retval=1;
    }
    catch(...) {
        TException("unknown error").show();
        retval=1;
80  }

    rp_Release();
    return retval;
}
```

Programmcode B.7: Kalibrierungsübersicht (Bash)

```
#!/bin/bash

echo "available configurations: "
cd conf && tree --noreport *
```

Programmcode B.8: Kalibrierungsaneignung (Bash)

```
#!/bin/bash

sudo chown "$UID:$UID" conf -R
```

Programmcode B.9: Funktionsgenerator (C++)

```
#include <iostream>
#include <chrono>
#include <math.h>

#include "src/util.hpp"
#include "src/interface/output.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
using namespace std::chrono;


#define SET_TO_DEFAULT()  {                   \
    if(default_pwm_set)                       \
        output->putRaw(default_pwm);          \
    else                                      \
        output->putValue(default_value);      \
}

int main(int argc, char **argv)  {
    int retval=2;
    //cout << std::scientific;
    //cerr << std::scientific;
    // parameters
    TOutput *output=nullptr;
    double frequency=0;
    double amplitude=0;
    TWaveform waveform=wfSin;
    bool frequency_set=false,
         amplitude_set=false;

    int safety_seconds=5;
    int duration_seconds=1;
    double default_value=0;
    double default_pwm=0;
    bool default_value_set=false,
         default_pwm_set=false;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) )  {
            cerr << "usage: fungen --help"<<endl
                 << "usage: fungen OPTIONS"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                   prints this message"<<endl
                 << " -v --verbose                verbose output"<<endl
                 << " -f --frequency  FREQ        set frequency to FREQ hertz. 0 or 'DC' for
                     DC"<<endl
                 << " -a --amplitude  AMP         set amplitude to AMP volts or amps"<<endl
                 << " -w --waveform   WAVEFORM    will create the waveform WAVEFORM (see
                     below for valid values, default:"<<waveformToString(waveform)<<")"<<endl
                 << " -o --output     OUTNAME     use output with name OUTNAME"<<endl
                 << " -z              IMPNAME     use output specified in impedance measure
                     setup with name IMPNAME"<<endl
```

```
50                << " -s --safety        TIME          wait for TIME seconds before enabling
                     output (default: "<<safety_seconds<<")"<<endl
                  << " -T --duration      TIME          generate output for TIME seconds (default:
                     "<<duration_seconds<<")"<<endl
                  << " -d --default       VALUE         set output to constant (calibrated
                     physical) VALUE when ideling, e.g. after ending (default:
                     "<<default_value<<")"<<endl
                  << " -da --default-analog VALUE   set output to constant (PWM) VALUE when
                     ideling, e.g. after ending (default: using --default)"<<endl
                  << endl
55                << "waveforms (case-insensitive):"<<endl
                  << "  dc                          direct current"<<endl // @todo use
                     waveformToString
                  << "  sin                         sine wave"<<endl
                  << "  cos                         cosine wave"<<endl
                  << "  square                      square"<<endl
60                << endl
                  << "For full documentation, see master thesis by Ilka Schulz or
                     contact:"<<endl
                  << "Leibniz Universität Hannover"<<endl
                  << "Institut für Antriebssystem und Leistungselektronik"<<endl
                  << "z. Hd. Ilka Schulz"<<endl
65                << "Welfengarten 1"<<endl
                  << "30167 Hannover"<<endl
                  << "Germany"<<endl;
          }
          else { // actual program
70            // parse arguments
              string sw="";
              for(int i=1; i<argc; i++) {
                  string a = argv[i];
                  if(sw=="-f" || sw=="--frequency") {
75                    frequency = stod(a);
                      frequency_set = true;
                      sw = "";
                  }
                  else if(sw=="-a" || sw=="--amplitude") {
80                    amplitude = stod(a);
                      amplitude_set = true;
                      sw = "";
                  }
                  else if(sw=="-w" || sw=="--waveform") {
85                    waveform = stringToWaveform(a);
                      sw = "";
                  }
                  else if(sw=="-o" || sw=="--output") {
                      output = new TOutput(a,true);
90                    sw = "";
                  }
                  else if(sw=="-z") {
                      output = new
                          TOutput(TImpedanceMeasureSetup(a,true).getOutput()->getName(), true);
                           // TImpedanceMeasureSetup will delete its interfaces, so another
                          output is created with the same name. force_load is actived in both
```

```
                        cases to ensure loading configuration from file.
                    sw = "";
95              }
                else if(sw=="-s" || sw=="--safety") {
                    safety_seconds = stoi(a);
                    sw = "";
                }
100             else if(sw=="-T" || sw=="--duration") {
                    duration_seconds = stoi(a);
                    sw = "";
                }
                else if(sw=="-d" || sw=="--default") {
105                 default_value = stod(a);
                    default_value_set = true;
                    sw = "";
                }
                else if(sw=="-da" || sw=="--default-analog") {
110                 default_pwm = stod(a);  /// @todo catch errors in all stod and stoi
                        functions
                    default_pwm_set = true;
                    sw = "";
                }
                else {
115                 if(!a.empty() && a[0]=='-') {  // argument is a switch
                        if(a=="-v" || a=="--verbose")
                            verbose = true;
                        else if(sw=="")
                            sw = a;
120                     else
                            ERROR("bad command line arguments. Check your syntax around
                                arguments >"+sw+"< and >"+a+"<.");
                    }
                    else {  // argument is a target
                        ERROR("bad command line arguments. Do not provide targets (here:
                            >"+a+"<)");
125                 }
                }
            }

            // process frequency / DC
130         if(frequency==0)  // this is technically not needed, if the default waveform is
                cos. However, it is cleaner this way
                waveform = wfDc;
            if(waveform==wfDc) {
                frequency = 0;
                frequency_set = true;
135         }

            // check arguments for validity
            if(output==nullptr)
                ERROR("please specify an output (other than null)");
140         if(!frequency_set)
                ERROR("please specify a frequency");
            if(!amplitude_set)
```

```
        ERROR("please specify an amplitude");
    //if(!waveform_set)
145 //     ERROR("please specify a waveform");
    if(safety_seconds<0)
        ERROR("negative safety time");
    if(duration_seconds<=0)
        ERROR("duration must be at least 1 second");
150 if(default_pwm_set && default_value_set)
        ERROR("exclusive options -d and -da both selected");

    // summarize settings
    cerr << "ATTENTION! will generate function:"<<endl
155      << "output:                "<<output->getName()<<endl
         << "frequency:             "<<frequency<<" hertz"<<endl
         << "amplitude:             "<<amplitude<<" volts or amps"<<endl
         << "waveform:              "<<waveformToString(waveform)<<endl
         << "duration:              "<<duration_seconds<<"
             second"<<(duration_seconds!=1?"s":"")<<endl
160      << "default "<<(default_pwm_set?"pwm":"phys.")<<" output:
             "<<(default_pwm_set?"
             ":"")<<(default_pwm_set?default_pwm:default_value)<<(default_pwm_set?"
             pwm":" volts or amps")<<endl;
    initHardwareAccess();

    // safety period
    if(safety_seconds==0)
165     cerr << "starting immediately!"<<endl;
    else {
        cerr << "starting in "<<safety_seconds<<" seconds: ";
        SET_TO_DEFAULT();
        for(int i=safety_seconds; i>0; i--) {
170         cerr << i << " ";
            cerr.flush();
            sleep(1);
        }
        cerr << endl;
175 }
    cerr << "enabled now!"<<endl;
    cerr << "Generating function... ";
    cerr.flush();

180 // generate
    output->putSignal(frequency, amplitude, waveform);
    if(output->isSelfRefreshing() ) {
        sleep(duration_seconds);
        cerr << "done"<<endl;
185 }
    else {
        high_resolution_clock::time_point t_started = high_resolution_clock::now();
        high_resolution_clock::time_point t_now;
        duration<double> T_passed;
190     uint32_t counter=0;
        do {
            t_now = high_resolution_clock::now();
```

```
                T_passed = duration_cast<duration<double> >(t_now-t_started);
                counter++;
195             output->refreshSignal();
                /// @todo sleep to reduce sample rate?
            }
            while(T_passed.count()<duration_seconds);
            cerr << "done"<<endl;
200         cerr << "ticks generated: "<<counter<<"
                ("<<uint32_t(counter/T_passed.count()+0.5)<<" per second)"<<endl;
        }

        /*if(waveform==wfDc) { // it appeas that not updating the output value adds 4
            bits(!) of additional resolution...
            output->putValue(amplitude);
205         sleep(duration_seconds);
            cerr << "done"<<endl;
        }
        else {
            high_resolution_clock::time_point t_started = high_resolution_clock::now();
210         high_resolution_clock::time_point t_now;
            duration<double> T_passed;
            uint32_t counter=0;
            do {
                t_now = high_resolution_clock::now();
215             T_passed = duration_cast<duration<double> >(t_now-t_started);
                counter++;
                switch(waveform) {
                    //case wfDc: already implemented above
                    case wfSin:
220                     output->putValue(amplitude *
                            sin(2*M_PI*frequency*T_passed.count()));
                        break;
                    case wfCos:
                        output->putValue(amplitude *
                            cos(2*M_PI*frequency*T_passed.count()));
                        break;
225                 case wfSquare:
                        NOT_IMPLEMENTED();
                    default:
                        ERROR("generator: unknown waveform");
                }
230             /// @todo sleep to reduce sample rate?
            }
            while(T_passed.count()<duration_seconds);
            cerr << "done"<<endl;
            cerr << "ticks generated: "<<counter<<"
                ("<<uint32_t(counter/T_passed.count()+0.5)<<" per second)"<<endl;
235     }*/

        // turn off
        SET_TO_DEFAULT();
        cerr << "output disabled again"<<endl;
240 }
    retval = 0;
```

```
    }
    catch(TException &e) {
        e.show();
245     retval = 1;
    }
    catch(...) {
        TException("unknown error").show();
        retval = 1;
250 }
    //rp_ApinReset
    rp_Release();
    if(output!=nullptr)
        delete output;
255 return retval;
}
```

Programmcode B.10: Messgerät (C++)

```cpp
#include <iostream>
#include <chrono>
#include <math.h>
#include <vector>

#include "src/util.hpp"
#include "src/interface/input.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
using namespace std::chrono;


int main(int argc, char **argv) {
    int retval=2;
    cout << std::scientific;
    // parameters
    vector<TInterface*> interfaces;
    double delay_seconds=0.5;
    string delimiter="\t";

    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) ) {
            cerr << "usage: meter --help"<<endl
                 << "usage: meter OPTIONS"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                    prints this message"<<endl
                 << " -v --verbose                 verbose output"<<endl
                 << " -i --input      INNAME       use input with name INNAME"<<endl
                 << " -z --setup      IMPNAME      use inputs specified in impedance measure setup with name IMPNAME"<<endl
                 << " -d --delay      TIME         delay for TIME seconds between two measurements (default: "<<delay_seconds<<" seconds)"<<endl
                 << " -c --delimiter  STRING       use STRING as table column delimiter"<<endl
                 << endl
                 << "For full documentation, see master thesis by Ilka Schulz or contact:"<<endl
                 << "Leibniz Universität Hannover"<<endl
                 << "Institut für Antriebssystem und Leistungselektronik"<<endl
                 << "z. Hd. Ilka Schulz"<<endl
                 << "Welfengarten 1"<<endl
                 << "30167 Hannover"<<endl
                 << "Germany"<<endl;
        }
        else { // actual program
            // parse arguments
            string sw="";
            for(int i=1; i<argc; i++) {
                string a = argv[i];
                if(sw=="-i" || sw=="--input") {
                    interfaces.push_back(new TInput(a,true));
                    sw = "";
```

```
        }
        else if(sw=="-z" || sw=="--setup") {
            interfaces.push_back(new TImpedanceMeasureSetup(a,true));
            sw = "";
        }
        else if(sw=="-d" || sw=="--delay") {
            delay_seconds = stod(a);
            sw = "";
        }
        else if(sw=="-c" || sw=="--delimiter") {
            delimiter = a;
            sw = "";
        }
        else {
            if(!a.empty() && a[0]=='-') { // argument is a switch
                if(a=="-v" || a=="--verbose") {
                    verbose = true;
                    sw = "";
                }
                else if(sw=="")
                    sw = a;
                else
                    ERROR("bad command line arguments. Check your syntax around
                        arguments >"+sw+"< and >"+a+"<.");
            }
            else if(sw!="")
                ERROR("bad switch: >"+sw+"<");
            else { // argument is a target
                ERROR("bad command line arguments. Do not provide targets (here:
                    >"+a+"<)");
            }
        }
    }
    if(sw!="")
        ERROR("bad switch: >"+sw+"<");

    // check arguments for validity
    if(interfaces.empty())
        ERROR("please specify at least one input");
    if(delay_seconds < 0)
        ERROR("delay time cannot be negative");
    for(TInterface *i : interfaces) {
        if(i == nullptr)
            ERROR("null interface specified");
        TImpedanceMeasureSetup *s = dynamic_cast<TImpedanceMeasureSetup*>(i);
        if(s!=nullptr && s->getInputU()==nullptr)
            ERROR("impedance measure setup \'"+s->getName()+"\' has invalid voltage
                input");
        if(s!=nullptr && s->getInputI()==nullptr)
            ERROR("impedance measure setup \'"+s->getName()+"\' has invalid current
                input");
    }
    if(delimiter.empty())
        ERROR("empty delimiter string");
```

```
100         if(delimiter.find('\n') != string::npos)
                ERROR("delimiter must not contain line breaks");

            // init hardware
            cerr << "reading from input..." << endl;
105         initHardwareAccess();

            // table header
            string table_header="idx"+delimiter+"t"+delimiter;
            for(TInterface *i : interfaces) {
110             TInput *input = dynamic_cast<TInput*>(i);
                if(input != nullptr) {
                    table_header += input->getName() + delimiter;
                    continue;
                }
115             TImpedanceMeasureSetup *ims = dynamic_cast<TImpedanceMeasureSetup*>(i);
                if(ims != nullptr) {
                    table_header += ims->getName() + "_u" + delimiter + ims->getName() + "_i"
                        + delimiter;
                    continue;
                }
120             UNREACHABLE();
            }
            table_header.erase(table_header.size()-delimiter.size());
            cout << table_header << endl;

125         // read from hardware
            high_resolution_clock::time_point t_started = high_resolution_clock::now();
            uint32_t counter=0;
            while(true) {
                high_resolution_clock::time_point t_now = high_resolution_clock::now();
130             duration<double> T_passed = duration_cast<duration<double> >(t_now-t_started);
                cout << counter << delimiter
                     << T_passed.count() << delimiter;
                for(TInterface *i : interfaces) {
                    TInput *input = dynamic_cast<TInput*>(i);
135                 if(input != nullptr)
                        cout << input->measureValue()<<delimiter;
                    else {
                        TImpedanceMeasureSetup *ims =
                            dynamic_cast<TImpedanceMeasureSetup*>(i);
                        if(ims != nullptr)
140                         cout << ims->getInputU()->measureValue() << delimiter <<
                                ims->getInputI()->measureValue() << delimiter;
                    }
                }
                counter++;

145             // @todo sleep to reduce sample rate?
                while(true) { // wait for next point in time
                    double diff = duration_cast<duration<double>
                        >(t_started-high_resolution_clock::now()).count() +
                        (delay_seconds*counter); // time still to wait in seconds
                    if( diff <= 1E-6)
```

```
                    break;
150             usleep(diff*1E6*0.99);
            }
            usleep(delay_seconds*1E6);

            cout << endl;  // this comes after the wait, so in case of an abort signal
                there will be no extra new line at the end of the file
155         cout.flush();
        }
    }
    retval = 0;
  }
160 catch(TException &e) {
    e.show();
    retval = 1;
  }
  catch(...)  {
165   TException("unknown error").show();
    retval = 1;
  }
  //rp_ApinReset
  rp_Release();
170 for(TInterface *i : interfaces)
    if(i!=nullptr)
      delete i;
  return retval;
}
```

Programmcode B.11: Zweipolcharakterisierung (C++)

```
#include <iostream>
#include <chrono>
#include <math.h>
#include <vector>
#include <fstream>

#include "src/util.hpp"
#include "src/interface/input.hpp"
#include "src/interface/output.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
using namespace std::chrono;


#define SET_TO_DEFAULT() {                    \
    if(default_pwm_set)                       \
        setup->getOutput()->putRaw(default_pwm);          \
    else                                      \
        setup->getOutput()->putValue(default_value);    \
}

// ----------------------------------------------------------------------------
/**
 * Hold the result from a measurement (time stamp, both voltages, both currents)
 * @details This class uses vector<double> to hold the data because this may need less CPU
     usage and might be easier to use for future developers than a multimap.
 * There are no mechanism in place to keep consistancy or protect data in any way. Use
     responsibly! This structure or a similar one could be implemented to be used
 * by the class TImpedanceMeasureSetup, but that code may never be (re-)used...
 */
struct TMeasureResult {
public:
    vector<double> t,                        ///< time stamp
                   u,                        ///< primary voltage
                   udt,                      ///< primary voltage integral
                   i;                        ///< primary current
public:
    void clear();                            ///< clears all containers and therefore ALL
        stored data
    double getMax(vector<double> &vec);      ///< return maximum from a vector, e.g. u1
    void saveToCsv(string fn, string delimiter);   ///< saves the stored data to a csv file
        @param fn filename to write to
};

void TMeasureResult::clear() {
    t.clear();
    u.clear();
    i.clear();
}
double TMeasureResult::getMax(vector<double> &vec) {
    if(vec.empty())
        ERROR("empty vector");
```

```
    double retval = vec[0];
50  for(size_t idx=1; idx<vec.size(); idx++)
        if(vec[idx] > retval)
            retval = vec[idx];
    return retval;
}
55 void TMeasureResult::saveToCsv(string fn, string delimiter) {
    fstream f(fn.c_str(), ios::out|ios::binary);
        f << "t"<<delimiter<<"u"<<delimiter<<"i"<<delimiter<<"udt"<<endl;
        for(size_t idx=0; idx<t.size(); idx++)
            f << t[idx]<<delimiter<<u[idx]<<delimiter<<i[idx]<<delimiter<<udt[idx] << endl;
60  f.close();
}
// ------------------------------------------------------------------------------
/**
 * generates the specified signals and takes measurements
65 * @details This function creates the specified signals on the impedance measure setup's
     output port and takes measurements on voltage and current after prime_periods cycles
     have passed. There are no safety features, which must be implemented by the caller
     function! The measured data are not automatically saved to disc but kept in RAM.
 * @param setup impedance measure setup
 * @param frequency output signal frequency
 * @param amplitude  output signal amplitude
 * @param waveform   output signal waveform
70 * @param prime_periods cycles (respective to frequency) in which the output is active but no
     measurement is taken
 * @param measure_periods cycles (respective to frequency) in which the inputs will be
     monitored
 */
TMeasureResult measure(TImpedanceMeasureSetup *setup,
                       double frequency,
75                     double amplitude,
                       TWaveform waveform,
                       double prime_periods,
                       double measure_periods) {
    TMeasureResult retval;
80
    // commented-out code in this function is an attempt to read from the "fast" analog input
    /*uint32_t buff_size = 16384;
    float *buff = new float[buff_size];
    bool triggered = false;
85
    rp_AcqReset();
    rp_AcqSetDecimation(RP_DEC_1024);  double divider=1024.0;//  double n=488;
    rp_AcqSetTriggerDelay(0);
    rp_AcqStart();*/
90
    high_resolution_clock::time_point t_started = high_resolution_clock::now();
    high_resolution_clock::time_point t_now;
    double T_passed;
    uint32_t counter=0;
95  setup->getOutput()->putSignal(frequency, amplitude, waveform);
    do {
        t_now = high_resolution_clock::now();
```

```
        T_passed = duration_cast<duration<double> >(t_now-t_started).count(); // - 0.004;
        counter++;
100     setup->getOutput()->refreshSignal();
        if(T_passed >= prime_periods/frequency) {//1024.0*1.0/125E6 * n) {
            /*if(!triggered) {
                rp_AcqSetTriggerLevel(RP_CH_2,0.0);
                rp_AcqSetTriggerSrc(RP_TRIG_SRC_AWG_PE);
105             //rp_AcqSetTriggerSrc(RP_TRIG_SRC_NOW);
                triggered = true;
            }
            rp_acq_trig_state_t state = RP_TRIG_STATE_TRIGGERED;
            rp_AcqGetTriggerState(&state);
110         if(state == RP_TRIG_STATE_TRIGGERED) {
                static bool msg_showed=false;
                if(!msg_showed) {
                    cerr << "triggered: "<<T_passed<<endl;
                    msg_showed = true;
115             }
            }*/
            retval.t.push_back(T_passed);
            retval.u.push_back(setup->getInputU()->measureValue());
            retval.i.push_back(setup->getInputI()->measureValue());
120     }
    }
    while(T_passed < (prime_periods+measure_periods)/frequency);
    /*cerr << "Test\n";

125 rp_AcqGetOldestDataV(RP_CH_2, &buff_size, buff);
    cerr << "Test2\n";

    for(size_t i = 0; i < buff_size; i++){
        double timestamp = divider*1.0/125E6 * i + prime_periods/frequency;
130     if(timestamp > (prime_periods+measure_periods)/frequency)
            break;
        retval.t.push_back(timestamp);
        retval.u.push_back(setup->getOutput()->reconstructSignalValue(timestamp));
        retval.i.push_back(buff[i]);
135 }*/

    // calculate voltage integral with simpson rule
    retval.udt.push_back(0);
    for(size_t idx=1; idx<retval.t.size()-1; idx++)
140     retval.udt.push_back( retval.udt[idx-1] + (retval.t[idx+1]-retval.t[idx-1])/12 *
            ( retval.u[idx-1] + 4.0*retval.u[idx] + retval.u[idx+1] ) );
    retval.udt.push_back( (retval.t[retval.t.size()-1]-retval.t[retval.t.size()-2]) *
        (retval.u[retval.t.size()-2]+retval.u[retval.t.size()-1]) * 0.5 );
    // set mean voltage integral to zero
    double mean_udt=0;
    for(size_t idx=0; idx<retval.t.size(); idx++)
145     mean_udt += retval.udt[idx];
    mean_udt /= double(retval.t.size());
    for(size_t idx=0; idx<retval.t.size(); idx++)
        retval.udt[idx] -= mean_udt;
```

```
    cerr << "done"<<endl;

    //delete[] buff;
    return retval;
}
//---------------------------------- global settings ----------------------------------
const double min_freq = 1E-3;  // minimum frequency for ouput


TImpedanceMeasureSetup *setup=nullptr;
double frequency=0;
double amplitude=0;
TWaveform waveform=wfSin;
bool frequency_set=false,
     amplitude_set=false;

int safety_seconds=5;
int prime_periods=20,
    measure_periods=1;
double default_value=0;
double default_pwm=0;
bool default_value_set=false,
     default_pwm_set=false;

string delimiter="\t";        // for csv files
bool pretty_timestamp=false;  // whether to start time stamp at zero
int time_factor=1;            // factor to multiply time output with, useful for importing milliseconds into TeX
int current_factor=1;         // factor to multiply current output with, useful for importing milliamps into TeX
int udt_factor=1;             // factor to multiply voltage integral output with, useful for importing millivoltseconds into TeX
string fn_raw="";             // fn for raw output
string fn_char="";            // fn to save evaluated characteristic to

//---------------------------------- main ----------------------------------
int main(int argc, char **argv)  {
    int retval=2;
    cout << std::scientific;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) )  {
            cerr << "usage: impmsr --help"<<endl
                 << "usage: impmsr OPTIONS"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                  prints this message"<<endl
                 << " -v --verbose               verbose output"<<endl
                 << " -f --frequency  FREQ       set frequency to FREQ hertz. 0 or 'DC' for DC"<<endl
                 << " -a --amplitude  AMP        set amplitude to AMP volts or amps"<<endl
                 << " -w --waveform   WAVEFORM   will create the waveform WAVEFORM (see below for valid values)"<<endl
                 << " -z --setup      IMPNAME    use output specified in impedance measure
```

```
                    setup with name IMPNAME"<<endl
                << " -s --safety      TIME         wait for TIME seconds before enabling
                    output (default: "<<safety_seconds<<")"<<endl
200             << " -Np              CYCLES       prime output for CYCLES cycles (default:
                    "<<prime_periods<<")"<<endl
                << " -Nm              CYCLES       measure from inputs for CYCLES cycles after
                    priming outputs (default: "<<measure_periods<<")"<<endl
                << " -d --default     VALUE        set output to constant (calibrated
                    physical) VALUE when ideling, e.g. after ending (default:
                    "<<default_value<<")"<<endl
                << " -da --default-analog VALUE   set output to constant (PWM) VALUE when
                    ideling, e.g. after ending (default: using --default)"<<endl
                << endl
205             << "data output settings:"<<endl
                << " -c --delimiter   STRING       use STRING as table column delimiter"<<endl
                << " -pt --pretty-timestamp        let timestamp start with zero, regardless
                    of priming time (see option -Np)"<<endl
                << " -tf --time-factor FACTOR     multiply time with FACTOR"<<endl
                << " -if --current-factor FACTOR multiply current with FACTOR"<<endl
210             << " -udtf --udt-factor FACTOR    multiply voltage integral with FACTOR"<<endl
                << " -or --output-raw FILE        save raw measurements to csv file
                    FILE"<<endl
                << " -oc --output-characteristic FILE  save evaluated data to csv file
                    FILE"<<endl
                << "waveforms (case-insensitive):"<<endl
                << " dc                            direct current"<<endl /// @todo use
                    waveformToString
215             << " sin                           sine wave"<<endl
                << " cos                           cosine wave"<<endl
                << " square                        square"<<endl
                << endl
                << "For full documentation, see master thesis by Ilka Schulz or
                    contact:"<<endl
220             << "Leibniz Universität Hannover"<<endl
                << "Institut für Antriebssystem und Leistungselektronik"<<endl
                << "z. Hd. Ilka Schulz"<<endl
                << "Welfengarten 1"<<endl
                << "30167 Hannover"<<endl
225             << "Germany"<<endl;
        }
        else { // actual program
            // parse arguments
            string sw="";
230         for(int i=1; i<argc; i++) {
                string a = argv[i];
                if(sw=="-f" || sw=="--frequency") {
                    frequency = stod(a);
                    frequency_set = true;
235                 sw = "";
                }
                else if(sw=="-a" || sw=="--amplitude") {
                    amplitude = stod(a);
                    amplitude_set = true;
240                 sw = "";
```

```
            }
            else if(sw=="-w" || sw=="--waveform") {
                waveform = stringToWaveform(a);
                sw = "";
            }
            else if(sw=="-z" || sw=="--setup") {
                setup = new TImpedanceMeasureSetup(a,true);
                sw = "";
            }
            else if(sw=="-s" || sw=="--safety") {
                safety_seconds = stoi(a);
                sw = "";
            }
            else if(sw=="-Np") {
                prime_periods = stod(a);
                sw = "";
            }
            else if(sw=="-Nm") {
                measure_periods = stod(a);
                sw = "";
            }
            else if(sw=="-d" || sw=="--default") {
                default_value = stod(a);
                default_value_set = true;
                sw = "";
            }
            else if(sw=="-da" || sw=="--default-analog") {
                default_pwm = stod(a);  /// @todo catch errors in all stod and stoi
                    functions
                default_pwm_set = true;
                sw = "";
            }
            else if(sw=="-c" || sw=="--delimiter") {
                delimiter = a;
                sw = "";
            }
            else if(sw=="-tf" || sw=="--time-factor") {
                time_factor = stoi(a);
                sw = "";
            }
            else if(sw=="-if" || sw=="--current-factor") {
                current_factor = stoi(a);
                sw = "";
            }
            else if(sw=="-udtf" || sw=="--udt-factor") {
                udt_factor = stoi(a);
                sw = "";
            }
            else if(sw=="-or" || sw=="--output-raw") {
                fn_raw = a;
                sw = "";
            }
            else if(sw=="-oc" || sw=="--output-characteristic") {
                fn_char = a;
```

```
                sw = "";
295         }
            else {
                if(!a.empty() && a[0]=='-') {  // argument is a switch
                    if(a=="-v" || a=="--verbose")
                        verbose = true;
300                 else if(a=="-pt" || a=="--pretty-timestamp")
                        pretty_timestamp = true;
                    else if(sw=="")
                        sw = a;
                    else
305                     ERROR("bad command line arguments. Check your syntax around
                            arguments >"+sw+"< and >"+a+"<.");
                }
                else if(sw!="")
                    ERROR("bad switch: >"+sw+"<");
                else {  // argument is a target
310                 ERROR("bad command line arguments. Do not provide targets (here:
                        >"+a+"<)");
                }
            }
        }
        if(sw!="")
315         ERROR("bad switch: >"+sw+"<");

        // check arguments for validity
        if(setup==nullptr)
            ERROR("please specify a setup (other than null)");
320     if(setup->getOutput()==nullptr)
            ERROR("setup does not have an output configured");
        if(setup->getInputU()==nullptr)
            ERROR("setup does not have a voltage input configured");
        if(setup->getInputI()==nullptr)
325         ERROR("setup does not have a current input configured");
        if(!frequency_set)
            ERROR("please specify a frequency");
        if(frequency <= min_freq)
            ERROR("frequency must be at least "+to_string(min_freq));
330     if(!amplitude_set)
            ERROR("please specify an amplitude");
        //if(!waveform_set)
        //    ERROR("please specify a waveform");
        if(safety_seconds<0)
335         ERROR("negative safety time");
        if(prime_periods<0)
            ERROR("prime period cannot be negative");
        if(measure_periods<=0)
            ERROR("measure period must be at least one period");
340     if(default_pwm_set && default_value_set)
            ERROR("exclusive options -d and -da both selected");
        if(delimiter.empty())
            ERROR("empty delimiter string");
        if(delimiter.find('\n') != string::npos)
345         ERROR("delimiter must not contain line breaks");
```

```
    // summarize settings
    cerr << "ATTENTION! will generate function:"<<endl
         << "setup:                 "<<setup->getName()<<endl
350      << "frequency:             "<<frequency<<" hertz"<<endl
         << "amplitude:             "<<amplitude<<" volts or amps"<<endl
         << "waveform:              "<<waveformToString(waveform)<<endl
         << "duration:              "<<(prime_periods+measure_periods)/frequency<<"
             seconds"<<endl
         << "default "<<(default_pwm_set?"pwm":"phys.")<<" output:
             "<<(default_pwm_set?"
             ":"")<<(default_pwm_set?default_pwm:default_value)<<(default_pwm_set?"
             pwm":" volts or amps")<<endl;
355 initHardwareAccess();
    // safety period
    if(safety_seconds==0)
        cerr << "starting immediately!"<<endl;
    else {
360     cerr << "starting in "<<safety_seconds<<" seconds: ";
        SET_TO_DEFAULT();
        for(int i=safety_seconds; i>0; i--) {
            cerr << i << " ";
            cerr.flush();
365         sleep(1);
        }
        cerr << endl;
    }
    cerr << "enabled now!"<<endl;
370 cerr << "Generating function... ";
    cerr.flush();
    // generate
    TMeasureResult result = measure(setup,
                                    frequency,
375                                 amplitude,
                                    waveform,
                                    prime_periods,
                                    measure_periods);
    // turn off
380 SET_TO_DEFAULT();
    cerr << "output disabled again"<<endl;

    // raw output
    if(! fn_raw.empty())
385     result.saveToCsv(fn_raw, delimiter);

    // pretty output
    if(! fn_char.empty()) {
        if(pretty_timestamp)
390         for(size_t i=0; i<result.t.size(); i++)
                result.t[i] -= prime_periods/frequency;
        for(size_t i=0; i<result.t.size(); i++) {
            result.t  [i] *= time_factor;
            result.i  [i] *= current_factor;
395         result.udt[i] *= udt_factor;
```

```
                }
                result.saveToCsv(fn_char, delimiter);
            }

400         /// @todo evaluate and save
        }
        retval = 0;
    }
    catch(TException &e) {
405     e.show();
        retval = 1;
    }
    catch (...)  {
        TException("unknown error").show();
410     retval = 1;
    }
    //rp_ApinReset
    rp_Release();
    if(setup!=nullptr)
415     delete setup;
    return retval;
}
```

Programmcode B.12: intelligente Vierpolcharakterisierung (nicht fertiggestellt) (C++)

```cpp
/**
 * This is just a prototype and no done software!
 */


#include <iostream>
#include <math.h>
#include <chrono>
#include <fstream>
#include <vector>
#include "src/util.hpp"
#include "src/interface/interface.hpp"
#include "src/interface/input.hpp"
#include "src/interface/output.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
using namespace std::chrono;




// ------------------------------------------------------------------------
/**
 * Hold the result from a measurement (time stamp, both voltages, both currents)
 * @details This class uses vector<double> to hold the data because this may need less CPU usage and might be easier to use for future developers than a multimap.
 * There are no mechanism in place to keep consistancy or protect data in any way. Use responsibly! This structure or a similar one could be implemented to be used
 * by the class TImpedanceMeasureSetup, but that code may never be (re-)used...
 */
struct TMeasureResult {
public:
    vector<double> t,                           ///< time stamp
                   u1,                          ///< primary voltage
                   i1,                          ///< primary current
                   u2,                          ///< secondary voltage
                   i2;                          ///< secondary current
public:
    void clear();                               ///< clears all containers and therefore ALL stored data
    vector<double>& getU(int idx);              ///< returns u1 for idx==0 and u2 for idx==1; undefined behaviour for other values
    vector<double>& getI(int idx);              ///< returns i1 for idx==0 and i2 for idx==1; undefined behaviour for other values
    double getMax(vector<double> &vec);         ///< return maximum from a vector, e.g. u1
    void saveToCsv(string fn, char delimiter);  ///< saves the stored data to a csv file @param fn filename to write to
};

void TMeasureResult::clear() {
    t.clear();
    u1.clear();
```

```
    i1.clear();
    u2.clear();
    i2.clear();
}
vector<double>& TMeasureResult::getU(int idx)  {
    if(idx==0)
        return u1;
    if(idx==1)
        return u2;
    ERROR("invalid idx");
}
vector<double>& TMeasureResult::getI(int idx)  {
    if(idx==0)
        return i1;
    if(idx==1)
        return i2;
    ERROR("invalid idx");
}
double TMeasureResult::getMax(vector<double> &vec)  {
    if(vec.empty())
        ERROR("empty vector");
    double retval = vec[0];
    for(size_t i=1; i<vec.size(); i++)
        if(vec[i] > retval)
            retval = vec[i];
    return retval;
}
void TMeasureResult::saveToCsv(string fn, char delimiter)  {
    fstream f(fn.c_str(), ios::out|ios::binary);
        f << "t"<<delimiter<<"u1"<<delimiter<<"i1"<<delimiter<<"u2"<<delimiter<<"i2"<<endl;
        for(size_t i=0; i<t.size(); i++)
            f << t[i]<<delimiter<<u1[i]<<delimiter<<i1[i]<<delimiter<<u2[i]<<delimiter<<i2[i]
                << endl;
    f.close();
}
// ----------------------------------------------------------------------------
#define SET_TO_DEFAULT()  cout << "set to default not implemented."<<endl  /// @todo implement

/**
 * generates the specified signals and takes measurements
 * @details This function creates the specified signals on two ports and takes measurements
     on voltage and current after prime_seconds seconds have passed. There are no safety
     features, which must be implemented by the caller function! The measured data are not
     automatically saved to disc but kept in RAM.
 * @param setups primary and secondary impedance measure setups
 * @param frequencies primary and secondary output signal frequencies
 * @param amplitudes  primary and secondary output signal amplitudes
 * @param waveforms   primary and secondary output signal waveforms
 * @param prime_seconds time in seconds in which the outputs are active but no measurement is
     taken
 * @param measure_seconds time in seconds n which the inputs will be monitored
 */
TMeasureResult measure(TImpedanceMeasureSetup *setups[2],
                       double frequencies[2],
```

```
                          double amplitudes[2],
                          TWaveform waveforms[2],
                          double prime_seconds,
                          double measure_seconds) {
100     TMeasureResult retval;
        high_resolution_clock::time_point t_started = high_resolution_clock::now();
        high_resolution_clock::time_point t_now;
        duration<double> T_passed;
        uint32_t counter=0;
105     do {
            t_now = high_resolution_clock::now();
            T_passed = duration_cast<duration<double> >(t_now-t_started);
            counter++;
            for(int i=0; i<2; i++) { // write to outputs
110             switch(waveforms[i]) {
                    case wfDc:
                        setups[i]->getOutput()->putValue(amplitudes[i] * 1);
                    case wfSin:
                        setups[i]->getOutput()->putValue(amplitudes[i] *
                            sin(2*M_PI*frequencies[i]*T_passed.count()));
115                     break;
                    case wfCos:
                        setups[i]->getOutput()->putValue(amplitudes[i] *
                            cos(2*M_PI*frequencies[i]*T_passed.count()));
                        break;
                    case wfSquare:
120                     NOT_IMPLEMENTED();
                    default:
                        ERROR("generator: unknown waveform");
                }
            }
125         if(T_passed.count() >= prime_seconds) {
                retval.t.push_back(T_passed.count());
                retval.u1.push_back(setups[0]->getInputU()->measureValue());
                retval.i1.push_back(setups[0]->getInputI()->measureValue());
                retval.u2.push_back(setups[1]->getInputU()->measureValue());
130             retval.i2.push_back(setups[1]->getInputI()->measureValue());
            }
        }
        while(T_passed.count() < prime_seconds+measure_seconds);
        cerr << "done"<<endl;
135     //cout << "ticks generated: "<<counter<<" ("<<uint32_t(counter/T_passed.count()+0.5)<<"
            per second)"<<endl; // not precise as reading from inputs slows down loop
        return retval;
    }
    // ------------------------------------------------------------------------------
    // global settings
140 TImpedanceMeasureSetup *setups[2] = {nullptr, nullptr};
    double frequencies[2] = {100, 0};
    double amplitudes[2];
    bool amplitudes_set[2] = {false, false};
    TWaveform waveforms[2] = {wfSin, wfDc};
145 double safety_seconds=5;
    double prime_seconds=0.1; // duration to prime the output and achieve a steady state
```

```
double measure_seconds=0.01;  // duration to measure from inputs
char delimiter = '\t'; /// @todo set via parameter?
vector<string> save_fns;
//------------------------------------------------------------------------------
int main(int argc, char **argv)  {
    int retval=2;
    cout << std::scientific;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) )  {
            cerr << "usage: orthomsr --help"<<endl
                 << "usage: orthomsr OPTIONS"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                      prints this message"<<endl
                 << " -v --verbose                   verbose output"<<endl
                 << " -z1            SETUP           use impedance measure setup SETUP for
                    primary winding"<<endl
                 << " -f1            FREQ            set frequency of primary signal to FREQ
                    hertz. 0 or 'DC' for DC (default: "<<frequencies[0]<<")"<<endl
                 << " -a1            AMP             set maximum amplitude of primary signal to
                    AMP volts or amps"<<endl
                 << " -w1            WAVEFORM        will create the waveform WAVEFORM on
                    primary output (see below for valid values, default:
                    "<<waveformToString(waveforms[0])<<")"<<endl
                 << " -z2            SETUP           use impedance measure setup SETUP for
                    secondary winding"<<endl
                 << " -f2            FREQ            set frequency of secondary signal to FREQ
                    hertz. 0 or 'DC' for DC (default: "<<frequencies[1]<<")"<<endl
                 << " -a2            AMP             set maximum amplitude of secondary signal
                    to AMP volts or amps"<<endl
                 << " -w2            WAVEFORM        will create the waveform WAVEFORM on
                    secondary output (see below for valid values, default:
                    "<<waveformToString(waveforms[1])<<")"<<endl
                 << " -o             FILE            print results to FILE (see below for
                    variables; specify this option multiple times to save the same data to
                    different files)"<<endl
                 << " -s --safety    TIME            wait for TIME seconds before enabling
                    outputs (default: "<<safety_seconds<<")"<<endl
                 << " -Tp            TIME            prime output for TIME seconds (default:
                    "<<prime_seconds<<")"<<endl
                 << " -Tm            TIME            measure from inputs for TIME seconds after
                    priming outputs (default: "<<measure_seconds<<")"<<endl
                 //<< " -d --default     VALUE       set outputs to constant (calibrated
                    physical) VALUE when ideling, e.g. after ending (default:
                    "<<default_value<<")"<<endl
                 //<< " -da --default-analog VALUE  set outputs to constant (PWM) VALUE when
                    ideling, e.g. after ending (default: using --default)"<<endl
                 << endl
                 << "waveforms (case-insensitive):"<<endl
                 << " dc                             direct current"<<endl /// @todo use
                    waveformToString
                 << " sin                            sine wave"<<endl
                 << " cos                            cosine wave"<<endl
                 << " square                         square"<<endl
```

```
                 << endl
                 << "filename variables:"<<endl  ///< @todo to be continued
                 << " %u1                         primary voltage"<<endl
185              << " %i1                         primary current"<<endl
                 << " %u2                         secondary voltage"<<endl
                 << " %i2                         secondary current"<<endl
                 << " %n                          measurement counting index"<<endl
                 << " %n1                         primary measurement counting index"<<endl
190              << " %n2                         secondary measurement counting index"<<endl
                 << endl
                 << "For full documentation, see master thesis by Ilka Schulz or
                    contact:"<<endl
                 << "Leibniz Universität Hannover"<<endl
                 << "Institut für Antriebssystem und Leistungselektronik"<<endl
195              << "z. Hd. Ilka Schulz"<<endl
                 << "Welfengarten 1"<<endl
                 << "30167 Hannover"<<endl
                 << "Germany"<<endl;
        }
200     else  {  // actual program
            // parse arguments
            string sw="";
            for(int i=1; i<argc; i++)  {
                string a = argv[i];
205             if(sw=="-z1")  {
                    setups[0] = new TImpedanceMeasureSetup(a,true);
                    sw = "";
                }
                else if(sw=="-z2")  {
210                 setups[1] = new TImpedanceMeasureSetup(a,true);
                    sw = "";
                }
                else if(sw=="-f1")  {
                    frequencies[0] = stod(a);
215                 sw = "";
                }
                else if(sw=="-f2")  {
                    frequencies[1] = stod(a);
                    sw = "";
220             }
                else if(sw=="-a1")  {
                    amplitudes[0] = stod(a);
                    amplitudes_set[0] = true;
                    sw = "";
225             }
                else if(sw=="-a2")  {
                    amplitudes[1] = stod(a);
                    amplitudes_set[1] = true;
                    sw = "";
230             }
                else if(sw=="-w1")  {
                    waveforms[0] = stringToWaveform(a);
                    sw = "";
                }
```

```
235                 else if(sw=="-w2") {
                        waveforms[0] = stringToWaveform(a);
                        sw = "";
                    }
                    else if(sw=="-o") {
240                     save_fns.push_back(a);
                        sw = "";
                    }
                    else if(sw=="-s" || sw=="--safety") {
                        safety_seconds = stoi(a);
245                     sw = "";
                    }
                    else if(sw=="-Tp") {
                        prime_seconds = stod(a);
                        sw = "";
250                 }
                    else if(sw=="-Tm") {
                        measure_seconds = stod(a);
                        sw = "";
                    }
255                 /*else if(sw=="-d" || sw=="--default") {
                        default_value = stod(a);
                        default_value_set = true;
                        sw = "";
                    }
260                 else if(sw=="-da" || sw=="--default-analog") {
                        default_pwm = stod(a); /// @todo catch errors in all stod and stoi
                            functions
                        default_pwm_set = true;
                        sw = "";
                    }*/
265                 else {
                        if(!a.empty() && a[0]=='-') { // argument is a switch
                            if(a=="-v" || a=="--verbose")
                                verbose = true;
                            else if(sw=="")
270                             sw = a;
                            else
                                ERROR("bad command line arguments. Check your syntax around
                                    arguments >"+sw+"< and >"+a+"<.");
                        }
                        else { // argument is a target
275                         ERROR("bad command line arguments. Do not provide targets (here:
                                >"+a+"<)");
                        }
                    }
                }
                if(!sw.empty())
280                 ERROR("invalid last switch in command line arguments");
                // check arguments for validity
                if(setups[0] == nullptr)
                    ERROR("please specify a setup for the primary winding");
                if(setups[1] == nullptr)
285                 ERROR("please specify a setup for the secondary winding");
```

```
        for(int i=0; i<2; i++)
            if(setups[i]->getInputI() == nullptr || setups[i]->getInputU() == nullptr)
                ERROR("setup "+setups[i]->getName()+" not properly set up");
        /// @todo check frequencies for valid ranges
290     /// @todo check amplitudes for valid ranges
        if(!amplitudes_set[0])
            ERROR("please specify a primary amplitude");
        if(!amplitudes_set[1])
            ERROR("please specify a secondary amplitude");
295     if(waveforms[0] == waveforms[1] && waveforms[0]!=wfDc)
            cerr << "WARNING: equal waveforms for both outputs, but not DC"<<endl;
        if(safety_seconds<0)
            ERROR("negative safety time");
        if(prime_seconds<0)
300         ERROR("priming time must be a non-negative value");
        if(measure_seconds<=0)
            ERROR("measuring time must be a positive value");
        //if(default_pwm_set && default_value_set)
        //    ERROR("exclusive options -d and -da both selected");
305     // summarize settings
        cerr << "ATTENTION! will generate output."<<endl
             /// @todo summary of output
             /*<< "output:                "<<output->getName()<<endl
             << "frequency:             "<<frequency<<" hertz"<<endl
310          << "amplitude:             "<<amplitude<<" volts or amps"<<endl
             << "waveform:              "<<waveformToString(waveform)<<endl
             << "duration:              "<<duration_seconds<<"
                 second"<<(duration_seconds!=1?"s":"")<<endl
             << "default "<<(default_pwm_set?"pwm":"phys.")<<" output:
                 "<<(default_pwm_set?"
                 ":"")<<(default_pwm_set?default_pwm:default_value)<<(default_pwm_set?"
                 pwm":" volts or amps")<<endl*/;
        initHardwareAccess();
315     // safety period
        if(safety_seconds==0)
            cerr << "starting immediately!"<<endl;
        else {
            cerr << "starting in "<<safety_seconds<<" seconds: ";
320         SET_TO_DEFAULT();
            for(int i=safety_seconds; i>0; i--) {
                cerr << i << " ";
                cerr.flush();
                sleep(1);
325         }
            cerr << endl;
        }
        cerr << "enabled now!"<<endl;

330     // find out "saturation voltage" for both setups
        cerr << "Searching for saturation levels... "<<endl;
        cerr.flush();
        double ampmins[2] = {0, 0},
               ampmaxs[2] = {amplitudes[0], amplitudes[1]};
335     for(int i=0; i<2; i++) {
```

```
            double Zmax = 0,
                   amp  = ampmaxs[i]/1000;
            while(amp<ampmaxs[i]) {
                double freqs[2] ={100,100};
340             double amps[2] = {i==0?amp:0, i==1?amp:0};
                TWaveform wavs[2] = {wfSin,wfSin};
                TMeasureResult result = measure(setups,
                                                freqs,
                                                amps,
345                                             wavs,
                                                prime_seconds,
                                                measure_seconds);
                double Z = result.getMax(result.getU(i)) / result.getMax(result.getI(i));
                if(Z > Zmax)
350                 Zmax = Z;
                if(Z < Zmax*0.9) {
                    ampmaxs[i] = amp;
                    break;
                }
355             amp *= 1.2;
            }
            ampmaxs[i] = min(ampmaxs[i]*5, amplitudes[i]); // depth into saturation, but
                do not overshoot user-defined limit
        }
        cerr << "ampmins = {"<<ampmins[0]<<", "<<ampmins[1]<<"}"<<endl
360          << "ampmaxs = {"<<ampmaxs[0]<<", "<<ampmaxs[1]<<"}"<<endl;

        // run test
        cerr << "Running test..."<<endl;
        cerr.flush();
365     int ns[2]   = {1, 10};
        for(int i=0; i<2; i++)
            if(ns[i]>1)
                ns[i] ++;
        int idxs[2];
370     int n=0;
        for(idxs[0]=0; idxs[0]<ns[0]; idxs[0]++)
            for(idxs[1]=0; idxs[1]<ns[1]; idxs[0]++,n++) {
                double amps[2];
                for(int i=0; i<2; i++)
375                 if(ns[i] == 1)
                        amps[i] = ampmaxs[i];
                    else
                        amps[i] = ampmins[i] + (ampmaxs[i]-ampmins[i])/ns[i] * idxs[i];
                TMeasureResult result = measure(setups,
380                                             frequencies,
                                                amps,
                                                waveforms,
                                                prime_seconds,
                                                measure_seconds);
385             SET_TO_DEFAULT();
                for(string fn : save_fns) {
                    ReplaceAll(fn, "%u1", to_string(result.getMax(result.u1)));
                    ReplaceAll(fn, "%i1", to_string(result.getMax(result.i1)));
```

```
                        ReplaceAll(fn, "%u2", to_string(result.getMax(result.u2)));
390                     ReplaceAll(fn, "%i2", to_string(result.getMax(result.i2)));
                        ReplaceAll(fn, "%n",  to_string(n));
                        ReplaceAll(fn, "%n1", to_string(idxs[0]));
                        ReplaceAll(fn, "%n2", to_string(idxs[1]));
                        result.saveToCsv(fn,delimiter);
395                 }
                    //// @todo sleep to cool down
                }

            /// @todo continue here
400         // turn off
            SET_TO_DEFAULT();
            cerr << "output disabled again"<<endl;
        }
        retval = 0;
405 }
    catch(TException &e) {
        e.show();
        retval = 1;
    }
410 catch (...)  {
        TException("unknown error").show();
        retval = 1;
    }
    //rp_ApinReset
415 rp_Release();
    for(int i=0; i<2; i++)
        if(setups[i]!=nullptr)
            delete setups[i];
    return retval;
420 }
```

Programmcode B.13: primitive Vierpolcharakterisierung (C++)

```
#include <iostream>
#include <chrono>
#include <math.h>
#include <vector>
#include <fstream>

#include "src/util.hpp"
#include "src/interface/input.hpp"
#include "src/interface/output.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
using namespace std::chrono;


#define SET_TO_DEFAULT() {                  \
    if(default_pwm_set)                     \
        setup->getOutput()->putRaw(default_pwm);        \
    else                                    \
        setup->getOutput()->putValue(default_value);    \
    if(setup2!=nullptr)                     \
        setup2->getOutput()->putValue(default_value_2);  \
}
// ---------------------------- global settings --------------------------------
const double min_freq = 1E-3;  // minimum frequency for ouput


TImpedanceMeasureSetup *setup=nullptr;
double frequency=0;
double amplitude=0;
TWaveform waveform=wfSin;
bool frequency_set=false,
     amplitude_set=false;

TImpedanceMeasureSetup *setup2=nullptr;
double amplitude_2=0;
double default_value_2=0;

int safety_seconds=5;
double duration_seconds;
double default_value=0;
double default_pwm=0;
bool default_value_set=false,
     default_pwm_set=false;

// ---------------------------- main ------------------------------------------
int main(int argc, char **argv) {
    int retval=2;
    cout << std::scientific;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) ) {
            cerr << "usage: orthomsr-simple --help"<<endl
                 << "usage: orthomsr-simple OPTIONS"<<endl
```

```
                << endl
                << "options:"<<endl
                << " -h --help                  prints this message"<<endl
                << " -v --verbose               verbose output"<<endl
                << " -f --frequency  FREQ       set frequency to FREQ hertz. 0 or 'DC' for DC"<<endl
                << " -a --amplitude  AMP        set amplitude to AMP volts or amps"<<endl
                << " -w --waveform   WAVEFORM   will create the waveform WAVEFORM (see below for valid values)"<<endl
                << " -z --setup      IMPNAME    use output specified in impedance measure setup with name IMPNAME"<<endl
                << " -s --safety     TIME       wait for TIME seconds before enabling output (default: "<<safety_seconds<<")"<<endl
                << " -T --duration   TIME       generate output for TIME seconds (default: "<<duration_seconds<<")"<<endl
                << " -d --default    VALUE      set output to constant (calibrated physical) VALUE when ideling, e.g. after ending (default: "<<default_value<<")"<<endl
                << " -da --default-analog VALUE  set output to constant (PWM) VALUE when ideling, e.g. after ending (default: using --default)"<<endl
                << " -z2             IMPNAME    use output specified in impedance measure setup with name IMPNAME as secondary dc source"<<endl
                << " -a2             AMP        set secondary amplitude to AMP volts or amps (default: "<<amplitude_2<<")"<<endl
                << " -d2             VALUE      set secondary output to constant (calibrated physical) VALUE when ideling, e.g. after ending (default: "<<default_value_2<<")"<<endl
                << endl
                << "waveforms (case-insensitive):"<<endl
                << " dc                         direct current"<<endl /// @todo use waveformToString
                << " sin                        sine wave"<<endl
                << " cos                        cosine wave"<<endl
                << " square                     square"<<endl
                << endl
                << "For full documentation, see master thesis by Ilka Schulz or contact:"<<endl
                << "Leibniz Universität Hannover"<<endl
                << "Institut für Antriebssystem und Leistungselektronik"<<endl
                << "z. Hd. Ilka Schulz"<<endl
                << "Welfengarten 1"<<endl
                << "30167 Hannover"<<endl
                << "Germany"<<endl;
        }
        else { // actual program
            // parse arguments
            string sw="";
            for(int i=1; i<argc; i++) {
                string a = argv[i];
                if(sw=="-f" || sw=="--frequency") {
                    frequency = stod(a);
                    frequency_set = true;
                    sw = "";
                }
```

```
            else if(sw=="-a" || sw=="--amplitude") {
                amplitude = stod(a);
95              amplitude_set = true;
                sw = "";
            }
            else if(sw=="-w" || sw=="--waveform") {
                waveform = stringToWaveform(a);
100             sw = "";
            }
            else if(sw=="-z" || sw=="--setup") {
                setup = new TImpedanceMeasureSetup(a,true);
                sw = "";
105         }
            else if(sw=="-s" || sw=="--safety") {
                safety_seconds = stoi(a);
                sw = "";
            }
110         else if(sw=="-T" || sw=="--duration") {
                duration_seconds = stod(a);
                sw = "";
            }
            else if(sw=="-d" || sw=="--default") {
115             default_value = stod(a);
                default_value_set = true;
                sw = "";
            }
            else if(sw=="-da" || sw=="--default-analog") {
120             default_pwm = stod(a); /// @todo catch errors in all stod and stoi
                    functions
                default_pwm_set = true;
                sw = "";
            }
            else if(sw=="-z2") {
125             setup2 = new TImpedanceMeasureSetup(a,true);
                sw = "";
            }
            else if(sw=="-a2") {
                amplitude_2 = stod(a);
130             sw = "";
            }
            else if(sw=="-d2") {
                default_value_2 = stod(a);
                sw = "";
135         }
            else {
                if(!a.empty() && a[0]=='-') { // argument is a switch
                    if(a=="-v" || a=="--verbose")
                        verbose = true;
140                 else if(sw=="")
                        sw = a;
                    else
                        ERROR("bad command line arguments. Check your syntax around
                            arguments >"+sw+"< and >"+a+"<.");
                }
```

```
145                 else if(sw!="")
                        ERROR("bad switch: >"+sw+"<");
                    else {  // argument is a target
                        ERROR("bad command line arguments. Do not provide targets (here:
                            >"+a+"<)");
                    }
150             }
            }
            if(sw!="")
                ERROR("bad switch: >"+sw+"<");
            // check arguments for validity
155         if(setup==nullptr)
                ERROR("please specify a setup (other than null)");
            if(setup->getOutput()==nullptr)
                ERROR("setup does not have an output configured");
            if(!frequency_set)
160             ERROR("please specify a frequency");
            if(frequency <= min_freq)
                ERROR("frequency must be at least "+to_string(min_freq));
            if(!amplitude_set)
                ERROR("please specify an amplitude");
165         if(setup2!=nullptr && setup2->getOutput()==nullptr)
                ERROR("secondary setup does not have an output configured");
            //if(!waveform_set)
            //    ERROR("please specify a waveform");
            if(safety_seconds<0)
170             ERROR("negative safety time");
            if(duration_seconds<=0)
                ERROR("duration must be a postive value in seconds");
            if(default_pwm_set && default_value_set)
                ERROR("exclusive options -d and -da both selected");
175
            // summarize settings
            cerr << "ATTENTION! will generate function:"<<endl
                 << "setup:                  "<<setup->getName()<<endl
                 << "frequency:              "<<frequency<<" hertz"<<endl
180              << "amplitude:              "<<amplitude<<" volts or amps"<<endl
                 << "waveform:               "<<waveformToString(waveform)<<endl
                 << "duration:               "<<duration_seconds<<" seconds"<<endl
                 << "default "<<(default_pwm_set?"pwm":"phys.")<<" output:
                     "<<(default_pwm_set?"
                     ":"")<<(default_pwm_set?default_pwm:default_value)<<(default_pwm_set?"
                    pwm":" volts or amps")<<endl
                 << "secondary setup:
                     "<<(setup2!=nullptr?setup2->getName():"null")<<endl;
185         if(setup2 != nullptr) {
                 cerr << "secondary dc:          "<<amplitude_2<<" volts or amps"<<endl
                      << "secondary default:     "<<default_value_2<<" volts or amps"<<endl;
            }

190         initHardwareAccess();

            // safety period
            if(safety_seconds==0)
```

```
                cerr << "starting immediately!"<<endl;
195         else {
                cerr << "starting in "<<safety_seconds<<" seconds: ";
                SET_TO_DEFAULT();
                for(int i=safety_seconds; i>0; i--) {
                    cerr << i << " ";
200                 cerr.flush();
                    sleep(1);
                }
                cerr << endl;
            }
205         cerr << "enabled now!"<<endl;
            cerr << "Generating function... ";
            cerr.flush();

            // generate
210         if(setup2 != nullptr) {
                setup2->getOutput()->putValue(amplitude_2);
            }
            setup->getOutput()->putSignal(frequency, amplitude, waveform);

215         if(setup->getOutput()->isSelfRefreshing() && (setup2==nullptr ||
                setup2->getOutput()->isSelfRefreshing())) {
                usleep(duration_seconds*1E6);
                cerr << "done"<<endl;
            }
            else {
220             high_resolution_clock::time_point t_started = high_resolution_clock::now();
                high_resolution_clock::time_point t_now;
                duration<double> T_passed;
                uint32_t counter=0;
                do {
225                 t_now = high_resolution_clock::now();
                    T_passed = duration_cast<duration<double> >(t_now-t_started);
                    counter++;
                    setup->getOutput()->refreshSignal();
                    if(setup2!=nullptr)
230                     setup2->getOutput()->refreshSignal();
                    /// @todo sleep to reduce sample rate?
                }
                while(T_passed.count()<duration_seconds);
                cerr << "done"<<endl;
235             cerr << "ticks generated: "<<counter<<"
                    ("<<uint32_t(counter/T_passed.count()+0.5)<<" per second)"<<endl;
            }

            // turn off
            SET_TO_DEFAULT();
240         cerr << "output disabled again"<<endl;

        }
        retval = 0;
    }
245 catch(TException &e) {
```

```
            e.show();
            retval = 1;
        }
        catch(...) {
250         TException("unknown error").show();
            retval = 1;
        }
        //rp_ApinReset
        rp_Release();
255     if(setup!=nullptr)
            delete setup;
        return retval;
    }
```

Programmcode B.14: improvisierte Vierpolcharakterisierung (C++)

```cpp
#include <iostream>
#include <math.h>
#include <chrono>
#include <fstream>
#include <vector>
#include "src/util.hpp"
#include "src/interface/interface.hpp"
#include "src/interface/output.hpp"
#include "src/scope.hpp"
using namespace std;
using namespace std::chrono;

void SET_TO_DEFAULT();
// ------------------------------------------------------------------------------------------------
/**
 * Hold the result from a measurement (time stamp, both voltages, both currents)
 * @details This class uses vector<double> to hold the data because this may need less CPU
     usage and might be easier to use for future developers than a multimap.
 * There are no mechanism in place to keep consistancy or protect data in any way. Use
     responsibly! @todo This structure or a similar one could be implemented to be used
 * by the class TImpedanceMeasureSetup, but that code may never be (re-)used...
 */
struct TMeasureResult  {
public:
    vector<double> t,                         ///< time stamp
                   u1,                        ///< primary voltage
                   i1,                        ///< primary current
                   u2,                        ///< secondary voltage
                   i2;                        ///< secondary current
public:
    void clear();                             ///< clears all containers and therefore ALL
        stored data
    vector<double>& getU(int idx);            ///< returns u1 for idx==0 and u2 for
        idx==1; undefined behaviour for other values
    vector<double>& getI(int idx);            ///< returns i1 for idx==0 and i2 for
        idx==1; undefined behaviour for other values
    double getMax(vector<double> &vec);       ///< return maximum from a vector, e.g. u1
    void saveToCsv(string fn, char delimiter);  ///< saves the stored data to a csv file
        @param fn filename to write to
};

void TMeasureResult::clear()  {
    t.clear();
    u1.clear();
    i1.clear();
    u2.clear();
    i2.clear();
}
vector<double>& TMeasureResult::getU(int idx)  {
    if(idx==0)
        return u1;
    if(idx==1)
```

```
        return u2;
    ERROR("invalid idx");
}
50 vector<double>& TMeasureResult::getI(int idx) {
    if(idx==0)
        return i1;
    if(idx==1)
        return i2;
55  ERROR("invalid idx");
}
double TMeasureResult::getMax(vector<double> &vec) {
    if(vec.empty())
        ERROR("empty vector");
60  double retval = vec[0];
    for(size_t i=1; i<vec.size(); i++)
        if(vec[i] > retval)
            retval = vec[i];
    return retval;
65 }
void TMeasureResult::saveToCsv(string fn, char delimiter) {
    fstream f(fn.c_str(), ios::out|ios::binary);
        if(!f)
            ERROR("cannot save to file");
70      f << "t"<<delimiter<<"u1"<<delimiter<<"i1"<<delimiter<<"u2"<<delimiter<<"i2"<<endl;
        for(size_t i=0; i<t.size(); i++)
            f << t[i]<<delimiter<<u1[i]<<delimiter<<i1[i]<<delimiter<<u2[i]<<delimiter<<i2[i]
                << endl;
    f.close();
}
75 // ------------------------------------------------------------------------------------------
/**
 * generates the specified signals and takes measurements
 * @details This function creates the specified signals on two ports and takes measurements
   on voltage and current after prime_seconds seconds have passed. There are no safety
   features, which must be implemented by the caller function! The measured data are not
   automatically saved to disc but kept in RAM.
 * @param outputs primary and secondary outputs
80 * @param frequencies primary and secondary output signal frequencies
 * @param amplitudes  primary and secondary output signal amplitudes
 * @param waveforms   primary and secondary output signal waveforms
 * @param prime_seconds time in seconds in which the outputs are active but no measurement is
   taken
 * @param scope_addr  string containing the IPV4 address of the Tektronix scope, e.g.
   192.168.1.103. Host names are not supported.
85 * @param shunts      primary and secondary shunt resistors
 * @todo implement calibration
 */
TMeasureResult measure(TOutput *outputs[2],
                       double frequencies[2],
90                     double amplitudes[2],
                       TWaveform waveforms[2],
                       double prime_seconds,
                       string scope_addr,
                       double shunts[2]) {
```

```
    // turn off signal for scope setup time
    SET_TO_DEFAULT();

    // setup scope
    TScope *scope = new TScope(scope_addr);
    try {
        cerr << "measuring... ";
        // generate output
        for(size_t i=0; i<2; i++) { /// @todo is this really synchronized?
            outputs[i]->putSignal(frequencies[i], amplitudes[i], waveforms[i]);
            if(! outputs[i]->isSelfRefreshing() )
                ERROR("can only work with self-refreshing outputs, e.g. fast RedPitaya's
                    outputs");
        }

        // prime output...
        while(prime_seconds>1) { // on some systems, usleep cannot take more than 1E6
            microseconds per call
            usleep(1E6);
            prime_seconds--;
        }
        if(prime_seconds>=1E-6)
            usleep(prime_seconds*1E6);

        ///@todo turn outputs off after acquisition (happens inside TScope::measure()
        vector<vector<double> > measured = scope->measure();

        // turn off signal
        SET_TO_DEFAULT();


        TMeasureResult retval;
        if(measured.size() != 5)
            ERROR("unexpected number of columns: "+to_string(measured.size()));
        retval.t  = measured[0];
        retval.u1 = measured[1];
        retval.i1 = measured[2];
        retval.u2 = measured[3];
        retval.i2 = measured[4];

        // apply shunt resistors
        /// @todo implement proper(!) calibration
        for(size_t i=0; i<retval.t.size(); i++) {
            retval.i1[i] /= shunts[0];
            //retval.u1[i] -= retval.i1[i]*(shunts[0]+6.3); /// @todo needed if correctly
                measuring current @todo remove winding resistance for debug
            retval.i2[i] /= shunts[1];
            //retval.u2[i] -= retval.i2[i]*shunts[1];
        }

        // correcting AC voltages. This takes a periods mean value and subtracts it from the
            voltage if the voltage is meant to be an AC voltage
        /// @todo give options to turn of AC voltage correction
```

```
145         {
                for(size_t i=0; i<2; i++) {
                    if(waveforms[i] != wfDc) {  // of course, DC voltages are just what they are
                        vector<double> &u = (i==0) ? retval.u1 : retval.u2;  // just for
                            readability
                        double t_begin = (retval.t[retval.t.size()-1] + retval.t[0])/2 -
                            0.5/frequencies[i];  /// @todo This only works for equally-distanced
                            time steps
150                     double t_end   = (retval.t[retval.t.size()-1] + retval.t[0])/2 +
                            0.5/frequencies[i];  /// @todo This only works for equally-distanced
                            time steps
                        double mean=0;
                        size_t count=0;
                        for(size_t i=0; i<retval.t.size(); i++)  // intentionally using a
                            variable with the same name as outer loop in order to shadow the
                            outer loop's one
                            if(retval.t[i]>=t_begin && retval.t[i]<=t_end) {
155                             mean += u[i];
                                count++;
                            }
                        if(count == 0)
                            ERROR("bad time interval, too large");
160                     if(count < 10)
                            cerr << endl << "WARNING: no proper AC voltage correction"<<endl;
                        mean /= count;
                        for(size_t i=0; i<u.size(); i++)
                            u[i] -= mean;
165                     //cerr << "[ac voltage corrected] ";
                    }
                }
            }

170         cerr << "done";
            return retval;
        }
        catch(TException &e) {
            delete scope;
175         throw e;
        }
        catch(...) {
            delete scope;
            ERROR("unknown error");
180     }
    }
    // -----------------------------------------------------------------------------
    // global settings
    TOutput *outputs[2] = {nullptr, nullptr};
185 double frequencies[2] = {100, 0};
    double amplitudes[2];
    bool amplitudes_set[2] = {false, false};
    TWaveform waveforms[2] = {wfSin, wfDc};
    double default_values[2] = {0,0};
190 bool default_values_set[2];
    double safety_seconds=5;
```

```
double prime_seconds=0.1;  // duration to prime the output and achieve a steady state
char delimiter = ';'; /// @todo set via parameter?
vector<string> save_fns;
195 int n_measurements_1=1,
        n_measurements_2=1;
string scope_addr="192.168.1.103";
double shunts[2] = {1.131,1.133};

200 double time_factor=1;

void SET_TO_DEFAULT()  {
    for(int i=0; i<2; i++)
        outputs[i]->putValue(default_values[i]);
205 }

// ------------------------------------------------------------------------------
int main(int argc, char **argv)  {
    int retval=2;
210     cout << std::scientific;
    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) )  {
            cerr << "usage: orthomsr-scope --help"<<endl
                 << "usage: orthomsr-scope OPTIONS"<<endl
215              << endl
                 << "options:"<<endl
                 << " -h --help                     prints this message"<<endl
                 << " -v --verbose                  verbose output"<<endl
                 << " -o1             ONAME         use high speed output ONAME for primary
                   signal"<<endl
220              << " -f1             FREQ          set frequency of primary signal to FREQ
                   hertz. 0 or 'DC' for DC (default: "<<frequencies[0]<<")"<<endl
                 << " -a1             AMP           set maximum amplitude of primary signal to
                  AMP volts or amps"<<endl
                 << " -w1             WAVEFORM      will create the waveform WAVEFORM on
                   primary output (see below for valid values, default:
                   "<<waveformToString(waveforms[0])<<")"<<endl
                 << " -d1             VALUE         set primary output to constant (calibrated
                   physical) VALUE when ideling, e.g. after ending"<<endl
                 << " -o2             ONAME         use high speed output ONAME for secondary
                   signal"<<endl
225              << " -f2             FREQ          set frequency of secondary signal to FREQ
                   hertz. 0 or 'DC' for DC (default: "<<frequencies[1]<<")"<<endl
                 << " -a2             AMP           set maximum amplitude of secondary signal
                   to AMP volts or amps"<<endl
                 << " -w2             WAVEFORM      will create the waveform WAVEFORM on
                   secondary output (see below for valid values, default:
                   "<<waveformToString(waveforms[1])<<")"<<endl
                 << " -d2             VALUE         set secondary output to constant
                   (calibrated physical) VALUE when ideling, e.g. after ending"<<endl
                 << " -o              FILE          print results to FILE (see below for
                   variables; specify this option multiple times to save the same data to
                   different files)"<<endl
230              << " -s --safety     TIME          wait for TIME seconds before enabling
                   outputs (default: "<<safety_seconds<<" seconds)"<<endl
```

```
                << " -Tp                TIME          prime output for TIME seconds (default:
                    "<<prime_seconds<<" seconds)"<<endl
                //<< " -Tm              TIME          measure from inputs for TIME seconds
                    after priming outputs (default: "<<measure_seconds<<")"<<endl
                << " -N1                NUMBER        measure NUMBER different values for primary
                    output (default: "<<n_measurements_1<<" ohms)"<<endl
                << " -N2                NUMBER        measure NUMBER different values for
                    secondary output (default: "<<n_measurements_2<<" ohms)"<<endl
235             << " --scope-addr       NETADDR       scope must be reachable under IPv4 address
                    NETADDR (default"<<scope_addr<<")"<<endl
                << " -r1                RES           assume RES ohms for primary shunt (default:
                    "<<shunts[0]<<endl
                << " -r2                RES           assume RES ohms for primary shunt (default:
                    "<<shunts[1]<<endl
                << " -tf                FACTOR        multiply time values with FACTOR when
                    saving to file (default: "<<time_factor<<")"<<endl
                //<< " -d --default     VALUE         set outputs to constant (calibrated
                    physical) VALUE when ideling, e.g. after ending (default:
                    "<<default_value<<")"<<endl
240             //<< " -da --default-analog VALUE   set outputs to constant (PWM) VALUE when
                    ideling, e.g. after ending (default: using --default)"<<endl
                << endl
                << "waveforms (case-insensitive):"<<endl
                << " dc                               direct current"<<endl /// @todo use
                    waveformToString
                << " sin                              sine wave"<<endl
245             << " cos                              cosine wave"<<endl
                << " square                           square"<<endl
                << endl
                << "filename variables:"<<endl  ///< @todo to be continued
                << " %u1                              primary voltage"<<endl
250             << " %i1                              primary current"<<endl
                << " %u2                              secondary voltage"<<endl
                << " %i2                              secondary current"<<endl
                << " %n                               measurement counting index"<<endl
                << " %N                               measurement counting index plus one"<<endl
255             << " %n1                              primary measurement counting index"<<endl
                << " %n2                              secondary measurement counting index"<<endl
                << " %N1                              primary measurement counting index plus
                    one"<<endl
                << " %N2                              secondary measurement counting index plus
                    one"<<endl
                << endl
260             << "For full documentation, see master thesis by Ilka Schulz or
                    contact:"<<endl
                << "Leibniz Universität Hannover"<<endl
                << "Institut für Antriebssystem und Leistungselektronik"<<endl
                << "z. Hd. Ilka Schulz"<<endl
                << "Welfengarten 1"<<endl
265             << "30167 Hannover"<<endl
                << "Germany"<<endl;
        }
        else {  // actual program
            // parse arguments
```

```
270         string sw="";
            for(int i=1; i<argc; i++) {
                string a = argv[i];
                if(sw=="-o1") {
                    outputs[0] = new TOutput(a,true);
275                 sw = "";
                }
                else if(sw=="-o2") {
                    outputs[1] = new TOutput(a,true);
                    sw = "";
280             }
                else if(sw=="-f1") {
                    frequencies[0] = stod(a);
                    sw = "";
                }
285             else if(sw=="-f2") {
                    frequencies[1] = stod(a);
                    sw = "";
                }
                else if(sw=="-a1") {
290                 amplitudes[0] = stod(a);
                    amplitudes_set[0] = true;
                    sw = "";
                }
                else if(sw=="-a2") {
295                 amplitudes[1] = stod(a);
                    amplitudes_set[1] = true;
                    sw = "";
                }
                else if(sw=="-w1") {
300                 waveforms[0] = stringToWaveform(a);
                    sw = "";
                }
                else if(sw=="-w2") {
                    waveforms[0] = stringToWaveform(a);
305                 sw = "";
                }
                else if(sw=="-o") {
                    save_fns.push_back(a);
                    sw = "";
310             }
                else if(sw=="-s" || sw=="--safety") {
                    safety_seconds = stoi(a);
                    sw = "";
                }
315             else if(sw=="-Tp") {
                    prime_seconds = stod(a);
                    sw = "";
                }
                else if(sw=="-N1") {
320                 n_measurements_1 = stoi(a);
                    sw = "";
                }
                else if(sw=="-N2") {
```

```
                        n_measurements_2 = stoi(a);
325                     sw = "";
                    }
                    else if(sw=="--scope-addr") {
                        scope_addr = a;
                        sw = "";
330                 }
                    else if(sw=="-r1") {
                        shunts[0] = stod(a);
                        sw = "";
                    }
335                 else if(sw=="-r2") {
                        shunts[1] = stod(a);
                        sw = "";
                    }
                    else if(sw=="-d1") {
340                     default_values[0] = stod(a);
                        default_values_set[0] = true;
                        sw = "";
                    }
                    else if(sw=="-d2") {
345                     default_values[1] = stod(a);
                        default_values_set[1] = true;
                        sw = "";
                    }
                    else if(sw=="-tf") {
350                     time_factor = stod(a);
                        sw = "";
                    }
                    else {
                        if(!a.empty() && a[0]=='-') { // argument is a switch
355                         if(a=="-v" || a=="--verbose")
                                verbose = true;
                            else if(sw=="")
                                sw = a;
                            else
360                             ERROR("bad command line arguments. Check your syntax around
                                    arguments >"+sw+"< and >"+a+"<.");
                        }
                        else { // argument is a target
                            ERROR("bad command line arguments. Do not provide targets (here:
                                >"+a+"<)");
                        }
365                 }
                }
                if(!sw.empty())
                    ERROR("invalid last switch in command line arguments");
                // check arguments for validity
370             if(outputs[0] == nullptr)
                    ERROR("please specify an output for the primary winding");
                if(outputs[1] == nullptr)
                    ERROR("please specify an output for the secondary winding");
                /// @todo check frequencies for valid ranges
375             /// @todo check amplitudes for valid ranges
```

```
        if(!amplitudes_set[0])
            ERROR("please specify a primary amplitude");
        if(!amplitudes_set[1])
            ERROR("please specify a secondary amplitude");
380     if(waveforms[0] == waveforms[1] && waveforms[0]!=wfDc)
            cerr << "WARNING: equal waveforms for both outputs, but not DC"<<endl;
        if(!default_values_set[0] || !default_values_set[1])
            ERROR("default values not set");
        if(safety_seconds<0)
385         ERROR("negative safety time");
        if(prime_seconds<0)
            ERROR("priming time must be a non-negative value");
        if(n_measurements_1<=0)
            ERROR("number of measurements for primary output must be a positive value");
390     if(n_measurements_2<=0)
            ERROR("number of measurements for secondary output must be a positive value");
        if(save_fns.empty())
            cerr << "WARNING: no filenames specified to save to"<<endl;

395     //if(default_pwm_set && default_value_set)
        //    ERROR("exclusive options -d and -da both selected");
        // summarize settings
        cerr << "ATTENTION! will generate output."<<endl
             /// @todo summary of output
400          /*<< "output:              "<<output->getName()<<endl
             << "frequency:           "<<frequency<<" hertz"<<endl
             << "amplitude:           "<<amplitude<<" volts or amps"<<endl
             << "waveform:            "<<waveformToString(waveform)<<endl
             << "duration:            "<<duration_seconds<<"
                 second"<<(duration_seconds!=1?"s":"")<<endl
405          << "default "<<(default_pwm_set?"pwm":"phys.")<<" output:
                 "<<(default_pwm_set?"
                 ":"")<<(default_pwm_set?default_pwm:default_value)<<(default_pwm_set?"
                 pwm":" volts or amps")<<endl*/;
        initHardwareAccess();
        // safety period
        if(safety_seconds==0)
            cerr << "starting immediately!"<<endl;
410     else {
            cerr << "starting in "<<safety_seconds<<" seconds: ";
            SET_TO_DEFAULT();
            for(int i=safety_seconds; i>0; i--) {
                cerr << i << " ";
415             cerr.flush();
                sleep(1);
            }
            cerr << endl;
        }
420     cerr << "enabled now!"<<endl;

        // run test
        cerr << "Running test..."<<endl;
        cerr.flush();
425     double ampmins[2] = {0, 0},
```

```
        ampmaxs[2] = {amplitudes[0], amplitudes[1]};
    int ns[2] = {n_measurements_1, n_measurements_2};
    int idxs[2];
    int n=0;
    for(idxs[0]=0; idxs[0]<ns[0]; idxs[0]++)
        for(idxs[1]=0; idxs[1]<ns[1]; idxs[1]++,n++) {
            double amps[2];
            for(int i=0; i<2; i++)
                if(ns[i] == 1)
                    amps[i] = ampmaxs[i];
                else {
                    if(i==0)
                        amps[i] = ampmins[i] + (ampmaxs[i]-ampmins[i])/ns[i] *
                            (idxs[i]+1);
                    else
                        amps[i] = ampmins[i] + (ampmaxs[i]-ampmins[i])/(ns[i]-1) *
                            idxs[i];
                }
            TMeasureResult result = measure(outputs,
                                            frequencies,
                                            amps,
                                            waveforms,
                                            prime_seconds,
                                            scope_addr,
                                            shunts
                                           );
            SET_TO_DEFAULT();
            for(size_t idx=0; idx<result.t.size(); idx++) {
                result.t[idx] *= time_factor;
                if(abs(result.u1[idx]) < 1E-20)    /// @todo make pretty
                    result.u1[idx] = 0;
                if(abs(result.i1[idx]) < 1E-20)
                    result.i1[idx] = 0;
                if(abs(result.u2[idx]) < 1E-20)
                    result.u2[idx] = 0;
                if(abs(result.i2[idx]) < 1E-20)
                    result.i2[idx] = 0;

            }
            for(string fn : save_fns) {
                fn = ReplaceAll(fn, "%u1", to_string(result.getMax(result.u1)));
                fn = ReplaceAll(fn, "%i1", to_string(result.getMax(result.i1)));
                fn = ReplaceAll(fn, "%u2", to_string(result.getMax(result.u2)));
                fn = ReplaceAll(fn, "%i2", to_string(result.getMax(result.i2)));
                fn = ReplaceAll(fn, "%n1", to_string(idxs[0]));
                fn = ReplaceAll(fn, "%n2", to_string(idxs[1]));
                fn = ReplaceAll(fn, "%n", to_string(n));
                fn = ReplaceAll(fn, "%N1", to_string(idxs[0]+1));
                fn = ReplaceAll(fn, "%N2", to_string(idxs[1]+1));
                fn = ReplaceAll(fn, "%N", to_string(n+1));
                cout << ", saving to \""<<fn<<"\""<<endl;
                result.saveToCsv(fn,delimiter);
            }
            /// @todo sleep to cool down
```

```
                    }
                }
480             retval = 0;
            }
            catch(TException &e) {
                e.show();
                retval = 1;
485         }
            catch (...)   {
                TException("unknown error").show();
                retval = 1;
            }
490         //rp_ApinReset
            rp_Release();
            for(int i=0; i<2; i++)
                if(outputs[i]!=nullptr)
                    delete outputs[i];
495         return retval;
        }
```

Programmcode B.15: Spannungsintegralberechnung (C++)

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>
#include <math.h>

#include "src/util.hpp"
using namespace std;
// ————————————————————— main ——————————————————————————————
int main(int argc, char **argv) {
    int retval=2;
    cout << std::scientific;

    vector<string> fns;
    string rowt="t";
    string rowu="u";
    string rowudt="udt";
    string delimiter=";";
    double frequency=NAN; // for udt correction

    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) ) {
            cerr << "usage: udt --help"<<endl
                 << "usage: udt OPTIONS FILE1 [FILE2 [FILE3 [...]]]"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                  prints this message"<<endl
                 << " -v --verbose               verbose output"<<endl
                 << " -t              NAME       column name containing times (default:
                    "<<rowt<<")"<<endl
                 << " -u              NAME       column name containing voltages (default:
                    "<<rowu<<")"<<endl
                 << " -udt            NAME       column name to contain voltage integrals
                    (default: "<<rowudt<<")"<<endl
                 << " -f              FREQ       frequency of signal for voltage correction,
                    use NaN for none (default: "<<frequency<<")"<<endl
                 << " -c              STRING     use STRING as delimiter (default:
                    "<<delimiter<<")"<<endl
                 // TODO: backup option?
                 << endl
                 << "For full documentation, see master thesis by Ilka Schulz or
                    contact:"<<endl
                 << "Leibniz Universität Hannover"<<endl
                 << "Institut für Antriebssystem und Leistungselektronik"<<endl
                 << "z. Hd. Ilka Schulz"<<endl
                 << "Welfengarten 1"<<endl
                 << "30167 Hannover"<<endl
                 << "Germany"<<endl;
        }
        else { // actual program
            // parse arguments
            string sw="";
```

```
    for(int i=1; i<argc; i++) {
        string a = argv[i];
        if(sw=="-t") {
50          rowt = a;
            sw = "";
        }
        else if(sw=="-u") {
            rowu = a;
55          sw = "";
        }
        else if(sw=="-udt") {
            rowudt = a;
            sw = "";
60      }
        else if(sw=="-f") {
            frequency = stod(a);
            sw = "";
        }
65      else if(sw=="-c") {
            delimiter = a;
            sw = "";
        }
        else {
70          if(!a.empty() && a[0]=='-') { // argument is a switch
                if(a=="-v" || a=="--verbose")
                    verbose = true;
                else if(sw=="")
                    sw = a;
75              else
                    ERROR("bad command line arguments. Check your syntax around
                        arguments >"+sw+"< and >"+a+"<.");
            }
            else if(sw!="")
                ERROR("bad switch: >"+sw+"<");
80          else { // argument is a target
                fns.push_back(a);
            }
        }
    }
85  if(sw!="")
        ERROR("bad switch: >"+sw+"<");

    // check arguments for validity
    if(fns.empty())
90      ERROR("not target specified");

    for(string fn : fns) {
        // read data
        vector<string>colnames;          // corrent order (left to right) of column
            names
95      map<string,vector<double> > data;  // associates column name with content
        {
            fstream f(fn.c_str(), ios::binary|ios::in);
                if(!f)
```

```
                    ERROR("cannot read from file \""+fn+"\"");
100             string line;
                getline(f,line);
                if(!f)
                    ERROR("cannot read columns");
                colnames = splitline(line,delimiter);
105             if(colnames.empty())
                    ERROR("file does not contain any columns");
                for(size_t line_idx=2; true; line_idx++) {  // looping through
                    lines. note: starting with line 2, because line 1 holds the
                    column names and the topmost line has the index 1 for some
                    reason...
                    getline(f,line);
                    if(!f)
110                     break;
                    if(line.empty())
                        break;  // it would be possible to throw an exception, but
                            this shall end gracefully
                    vector<string> literals = splitline(line,delimiter);
                    if(literals.size() != colnames.size())
115                     ERROR("in line "+to_string(line_idx)+": wrong number of
                            columns");
                    for(size_t col_idx=0; col_idx<colnames.size(); col_idx++)
                        data[colnames[col_idx]].push_back(stod(literals[col_idx]));
                }
            f.close();
120     }

        // manipulate data
        {
            vector<double> &t = data[rowt];
125         vector<double> &u = data[rowu];
            vector<double> &udt = data[rowudt];  // creates the row, if it does not
                exist, gets it otherwise
            if(t.size() != u.size())
                ERROR("time and voltage column are not equally long");
            if(t.empty())
130             ERROR("table is empty");
            if(! udt.empty())
                ERROR("voltage integral row already exists");

            // calculate voltage integral with simpson rule
135         udt.clear();
            udt.push_back(0);  // first value
            for(size_t idx=1; idx<t.size()-1; idx++)
                udt.push_back( udt[idx-1] + (t[idx+1]-t[idx-1])/12 * ( u[idx-1]
                    + 4.0*u[idx] + u[idx+1] ) );
            udt.push_back( udt[udt.size()-1] + (t[t.size()-1]-t[t.size()-2]) *
                (u[t.size()-2]+u[t.size()-1]) * 0.5 );  // last value
140         // set mean voltage integral to zero  TODO: make this work for unevenly
                spaced time points
            // correcting AC voltages. This takes a periods mean value and subtracts
                it from the voltage if the voltage is meant to be an AC voltage
            /// @todo give options to turn off AC voltage integral correction
```

```
                {
                    if(!isnan(frequency)) { /// @todo better floating point comparison
145                     double t_begin = frequency==0 ? t[0] : (t[t.size()-1] + t[0])/2 -
                            0.5/frequency;  /// @todo This only works for
                            equally-distanced time steps
                        double t_end   = frequency==0 ? t[t.size()-1] : (t[t.size()-1] +
                            t[0])/2 + 0.5/frequency;  /// @todo This only works for
                            equally-distanced time steps
                        double mean=0;
                        size_t count=0;
                        for(size_t i=0; i<t.size(); i++)  // intentionally using a
                            variable with the same name as outer loop in order to shadow
                            the outer loop's one
150                         if(t[i]>=t_begin && t[i]<=t_end) {
                                mean += udt[i];
                                count++;
                            }
                        mean /= count;
155                     //cerr << "t_begin="<<t_begin<<";  t_end="<<t_end<<";
                            mean="<<mean<<";  count="<<count<<endl;
                        if(count == 0)
                            ERROR("bad time interval, too large (file: "+fn+")");
                        for(size_t i=0; i<udt.size(); i++)
                            udt[i] -= mean;
160                     if(count < 10)
                            cerr << "WARNING: no proper voltage integral correction
                                (file: "<<fn<<", f: "<<frequency<<")"<<endl;
                        else
                            ;//cerr << "[voltage integral corrected by "<<mean<<"]"<<endl;
                    }
165             }
                // old algorithm for udt correction
                /*double mean_udt=0;
                for(size_t idx=0; idx<t.size(); idx++)
                    mean_udt += udt[idx];
170             mean_udt /= double(t.size());
                for(size_t idx=0; idx<t.size(); idx++)
                    udt[idx] -= mean_udt;*/
            }

175         // write data
            {
                fstream f(fn.c_str(), ios::binary|ios::out);
                    if(!f)
                        ERROR("cannot write to file");
180
                    string line="";
                    for(auto itr : data)
                        line += itr.first+delimiter;
                    line.erase(line.size()-1);
185                 f << line <<endl;
                    for(size_t i=0; i<data.begin()->second.size(); i++) {
                        line = "";
                        for(auto itr : data) {
```

```
                                double towrite = itr.second[i];
190                             if(abs(towrite) < 1E-20)   /// @todo make pretty
                                    towrite = 0;
                                line += dtos(towrite)+delimiter;
                            }
                            line.erase(line.size()-1);
195                         f << line << endl;
                        }
                        //f << endl;
                    if(!f)
                        ERROR("cannot save file");
200             }
            }
        }
        retval = 0;
    }
205 catch(TException &e) {
        e.show();
        retval = 1;
    }
    /*catch(...)  {
210     TException("unknown error").show();
        retval = 1;
    }*/
    return retval;
}
```

Programmcode B.16: Nachträgliche Kalibrierung (C++)

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>

#include "src/util.hpp"
#include "src/interface/input.hpp"
#include "src/interface/impedancemeasuresetup.hpp"
using namespace std;
// ——————————————————— main ———————————————————————————————
int main(int argc, char **argv)  {
    int retval=2;
    cout << std::scientific;

    TInput *input=nullptr;
    string fn="";
    string rowraw="";
    string rowval="";
    string delimiter=";";

    try {
        if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) )  {
            cerr << "usage: converter --help"<<endl
                 << "usage: converter OPTIONS FILE"<<endl
                 << endl
                 << "options:"<<endl
                 << " -h --help                   prints this message"<<endl
                 << " -v --verbose                verbose output"<<endl
                 << " -i --input       INAME      name of the input containing
                   calibration"<<endl
                 << " -zu              SETUP      setup containing the voltage input
                   containing calibration"<<endl
                 << " -zi              SETUP      setup containing the current input
                   containing calibration"<<endl
                 << " -r --raw-col     NAME       column name containing raw measurements
                   (default: "<<rowraw<<")"<<endl
                 << " -v --val-col     NAME       column name to contain converted values
                   (default: "<<rowval<<")"<<endl
                 << " -c               STRING     use STRING as delimiter (default:
                   "<<delimiter<<")"<<endl
                 // TODO: backup option?
                 << endl
                 << "For full documentation, see master thesis by Ilka Schulz or
                   contact:"<<endl
                 << "Leibniz Universität Hannover"<<endl
                 << "Institut für Antriebssystem und Leistungselektronik"<<endl
                 << "z. Hd. Ilka Schulz"<<endl
                 << "Welfengarten 1"<<endl
                 << "30167 Hannover"<<endl
                 << "Germany"<<endl;
        }
        else  {  // actual program
```

```
        // parse arguments
        string sw="";
        for(int i=1; i<argc; i++) {
            string a = argv[i];
50          if(sw=="-i" || sw=="--input") {
                input = new TInput(a,true);
                sw = "";
            }
            else if(sw=="-zu") {
55              TImpedanceMeasureSetup *setup = new TImpedanceMeasureSetup(a,true); //
                    keeps setup in heap and destructor will not be executed on program
                    exit...
                input = setup->getInputU();
            }
            else if(sw=="-zi") {
                TImpedanceMeasureSetup *setup = new TImpedanceMeasureSetup(a,true); //
                    keeps setup in heap and destructor will not be executed on program
                    exit...
60              input = setup->getInputI();
            }
            else if(sw=="-r" || sw=="-raw-col") {
                rowraw = a;
                sw = "";
65          }
            else if(sw=="-v" || sw=="--val-col") {
                rowval = a;
                sw = "";
            }
70          else {
                if(!a.empty() && a[0]=='-') { // argument is a switch
                    if(a=="-v" || a=="--verbose")
                        verbose = true;
                    else if(sw=="")
75                      sw = a;
                    else
                        ERROR("bad command line arguments. Check your syntax around
                            arguments >"+sw+"< and >"+a+"<.");
                }
                else if(sw!="")
80                  ERROR("bad switch: >"+sw+"<");
                else { // argument is a target
                    if(fn == "")
                        fn = a;
                    else
85                      ERROR("specify only one target!");
                }
            }
        }
        if(sw!="")
90          ERROR("bad switch: >"+sw+"<");

        // check arguments for validity
        if(fn.empty())
            ERROR("not target specified");
```

```
95
    // read data
    vector<string>colnames;            // corrent order (left to right) of column
        names
    map<string,vector<double> > data;  // associates column name with content
    {
100     fstream f(fn.c_str(), ios::binary|ios::in);
            if(!f)
                ERROR("cannot read from file");
            string line;
            getline(f,line);
105         if(!f)
                ERROR("cannot read columns");
            colnames = splitline(line,delimiter);
            if(colnames.empty())
                ERROR("file does not contain any columns");
110         for(size_t line_idx=2; true; line_idx++)  {  // looping through lines.
                note: starting with line 2, because line 1 holds the column names and
                the topmost line has the index 1 for some reason...
                getline(f,line);
                if(!f)
                    break;
                if(line.empty())
115                 break;  // it would be possible to throw an exception, but this
                        shall end gracefully
                vector<string> literals = splitline(line,delimiter);
                if(literals.size() != colnames.size())
                    ERROR("in line "+to_string(line_idx)+": wrong number of columns");
                for(size_t col_idx=0; col_idx<colnames.size(); col_idx++)
120                 data[colnames[col_idx]].push_back(stod(literals[col_idx]));
            }
        f.close();
    }

125 // manipulate data
    {
        // TODO: test algorithm
        vector<double> &raw = data[rowraw];
        vector<double> &val = data[rowval];  // creates the row, if it does not
            exist, gets it otherwise
130     if(raw.empty())
            ERROR("table is empty");
        if(! val.empty())
            ERROR("converted values row already exists");

135     // convert
        for(size_t i=0; i<raw.size(); i++)
            val.push_back(input->convertMeasurementToValue(raw[i]));
    }

140 // write data
    {
        fstream f(fn.c_str(), ios::binary|ios::out);
            if(!f)
```

```
                    ERROR("cannot write to file");
145
                string line="";
                for(auto itr : data)
                    line += itr.first+delimiter;
                line.erase(line.size()-1);
150             f << line <<endl;
                for(size_t i=0; i<data.begin()->second.size(); i++) {
                    line = "";
                    for(auto itr : data)
                        line += dtos(itr.second[i])+delimiter;
155                 line.erase(line.size()-1);
                    f << line << endl;
                }
                //f << endl;
            if(!f)
160             ERROR("cannot save file");
        }
    }
    retval = 0;
  }
165 catch(TException &e) {
    e.show();
    retval = 1;
  }
  catch (...) {
170   TException("unknown error").show();
    retval = 1;
  }
  if(input!=nullptr)
    delete input;
175 return retval;
}
```

Programmcode B.17: Glättung der Quantisierung (C++)

```
#include <iostream>
#include <map>
#include <vector>
#include <fstream>
#include <math.h>

#include "src/util.hpp"
using namespace std;

map<double,double> smoothen(vector<double> t, vector<double> val) {
    map<double,double> retval;
    if(t.empty())
        return retval;
    //retval[t[0]] = val[0];
    if(t.size()==1)
        return retval;
    size_t last_commit=-1;
    for(size_t i=1; i<t.size(); i++) {
        if(val[last_commit+1] != val[i]) {
            retval[(t[last_commit+1]+t[i-1])*0.5] = val[i-1];
            last_commit = i-1;
        }
    }
    if(val[last_commit] != val[val.size()-1])
        retval[ (t[(last_commit+1)] + t[t.size()-1])/2 ] = val[val.size()-1];
    return retval;
}
double interpolate(map<double,double> samples, double t) {
    if(samples.size()<2)
        ERROR("need at least two points for interpolation");
    map<double,double>::iterator itlow = samples.lower_bound(t);// lowerbound(samples,t);
    if(itlow!=samples.begin())
        itlow = std::prev(itlow,1);
    map<double,double>::iterator itup = itlow;
    std::advance(itup,1);
    if(itup==samples.end()) {
        itlow = std::prev(itlow,1);
        itup  = std::prev(itup,1);
    }
    return itlow->second + (itup->second-itlow->second)/(itup->first-itlow->first) *
        (t-itlow->first);
}
// ---------------------------- main ----------------------------
int main(int argc, char **argv) {
    int retval=2;
    cout << std::scientific;

    vector<string> fns;
    string rowt="t";
    string delimiter=";";

    try {
```

```
    if(argc<=1 || (argc==2 && (string(argv[1])=="-h"||string(argv[1])=="--help")) ) {
        cerr << "usage: smth --help"<<endl
            << "usage: smth OPTIONS FILE1 [FILE2 [FILE3 [...]]]"<<endl
55          << endl
            << "options:"<<endl
            << " -h --help                   prints this message"<<endl
            << " -v --verbose                verbose output"<<endl
            << " -t            NAME          column name containing times (default:
               "<<rowt<<")"<<endl
60          << " -c            STRING        use STRING as delimiter (default:
               "<<delimiter<<")"<<endl
            // TODO: backup option?
            << endl
            << "For full documentation, see master thesis by Ilka Schulz or
               contact:"<<endl
            << "Leibniz Universität Hannover"<<endl
65          << "Institut für Antriebssystem und Leistungselektronik"<<endl
            << "z. Hd. Ilka Schulz"<<endl
            << "Welfengarten 1"<<endl
            << "30167 Hannover"<<endl
            << "Germany"<<endl;
70  }
    else { // actual program
        // parse arguments
        string sw="";
        for(int i=1; i<argc; i++) {
75          string a = argv[i];
            if(sw=="-t") {
                rowt = a;
                sw = "";
            }
80          else if(sw=="-c") {
                delimiter = a;
                sw = "";
            }
            else {
85              if(!a.empty() && a[0]=='-') { // argument is a switch
                    if(a=="-v" || a=="--verbose")
                        verbose = true;
                    else if(sw=="")
                        sw = a;
90                  else
                        ERROR("bad command line arguments. Check your syntax around
                            arguments >"+sw+"< and >"+a+"<.");
                }
                else if(sw!="")
                    ERROR("bad switch: >"+sw+"<");
95              else { // argument is a target
                    fns.push_back(a);
                }
            }
        }
100     if(sw!="")
            ERROR("bad switch: >"+sw+"<");
```

```
    // check arguments for validity
    if(fns.empty())
        ERROR("not target specified");

    for(string fn : fns) {
        // read data
        //cout << fn<<endl;
        vector<string>colnames;            // corrent order (left to right) of column
            names
        map<string,vector<double> > data;  // associates column name with content
        {
            fstream f(fn.c_str(), ios::binary|ios::in);
                if(!f)
                    ERROR("cannot read from file \""+fn+"\"");
                string line;
                getline(f,line);
                if(!f)
                    ERROR("cannot read columns");
                colnames = splitline(line,delimiter);
                if(colnames.empty())
                    ERROR("file does not contain any columns");
                for(size_t line_idx=2; true; line_idx++) { // looping through
                    lines. note: starting with line 2, because line 1 holds the
                    column names and the topmost line has the index 1 for some
                    reason...
                    getline(f,line);
                    if(!f)
                        break;
                    if(line.empty())
                        break; // it would be possible to throw an exception, but
                            this shall end gracefully
                    vector<string> literals = splitline(line,delimiter);
                    if(literals.size() != colnames.size())
                        ERROR("in line "+to_string(line_idx)+": wrong number of
                            columns");
                    if(literals.size() != colnames.size())
                        ERROR("line "+to_string(line_idx)+" contains invalid number
                            of columns (file: "+fn+")");
                    //cerr << "line: "<<line_idx<<endl;
                    for(size_t col_idx=0; col_idx<colnames.size(); col_idx++)
                        data[colnames[col_idx]].push_back(stod(literals[col_idx]));
                }
            f.close();
        }
        // manipulate data
        {
            vector<double> &t = data[rowt];
            for(map<string,vector<double> >::iterator itr=data.begin();
                itr!=data.end(); itr++) {
                if(itr->second.begin() == t.begin())
                    continue;
                vector<double> &val = itr->second;
                if(t.size() != val.size())
```

```
                        ERROR("time and value column are not equally long");
                    if(t.empty())
150                     ERROR("table is empty");
                    map<double,double> newtable = smoothen(t,val);

                    //for(map<double,double>::iterator itr=newtable.begin();
                        itr!=newtable.end(); itr++)
                    //     cerr << itr->first<<"\t"<<itr->second<<endl;
155                 for(size_t i=0; i<t.size(); i++)
                        val[i] = interpolate(newtable,t[i]);
                }
            }

160         // write data
            {
                fstream f(fn.c_str(), ios::binary|ios::out);
                    if(!f)
                        ERROR("cannot write to file");
165
                    string line="";
                    for(auto itr : data)
                        line += itr.first+delimiter;
                    line.erase(line.size()-1);
170                 f << line <<endl;
                    for(size_t i=0; i<data.begin()->second.size(); i++) {
                        line = "";
                        for(auto itr : data) {
                            double towrite = itr.second[i];
175                         if(abs(towrite)<1E-20)    /// @todo make pretty
                                towrite = 0;
                            line += dtos(towrite)+delimiter;
                        }
                        line.erase(line.size()-1);
180                     f << line << endl;
                    }
                    //f << endl;
                if(!f)
                    ERROR("cannot save file");
185         }
        }
    }
    retval = 0;
  }
190 catch(TException &e) {
    e.show();
    retval = 1;
  }
  /*catch(...) {
195   TException("unknown error").show();
    retval = 1;
  }*/
  return retval;
}
```

Programmcode B.18: Simulation (C++)

```
/** @file */
#include "src/demos/all.hpp"
#include "src/matmodels/linearisotropiclossfree.hpp"
#include "src/matmodels/nonlinearisotropiclossfree.hpp"
#include "src/matmodels/nonlinearisotropiclossfreelut.hpp"
#include "src/fdmmodels/sc.hpp"
#include "src/tools/magcurve.hpp"
#include "src/tools/magcurvegeo.hpp"
#include "src/tools/magneticcomponentreplay.hpp"

bool VERBOSE = false;

/*void worker_fdm() {
    ///@todo implement
}*/

constexpr char measurement_NO10par_fn[]    =
    "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SST-par.csv";
constexpr char measurement_NO10orth_fn[]   =
    "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SST-orth.csv";
constexpr char measurement_NO20_fn[]       =
    "/mnt/laptop/Documents/Masterarbeit/Messungen/NO20_fine.csv";
constexpr char measurement_delimiter[]     = ";";

/** @todo doc
 */
int main()
{
    vector<std::thread*> threads;
    auto time = std::chrono::high_resolution_clock::now();
    typedef double T;
    constexpr TRungeKuttaMethod method = TRungeKuttaMethod::PseudoEuler;
    try {
        // ---- put main code here ----
        //FemDemo();  currently out of order
        //SpiceDemo("/home/ilka/test/spice");
        //FdmDemo();
        /*MagCurve<ELEMENT_PARAMS, TFdmElementNonlinearIsotropicLossfree<ELEMENT_PARAMS,2200>
            >
                                    ("/home/ilka/test/magcurve.csv", -1428, 1428,200);

        TMagneticComponent<ELEMENT_PARAMS> *sc = new TFdmSc<ELEMENT_PARAMS, 100, 100,
            TFdmElementNonlinearIsotropicLossfree<ELEMENT_PARAMS,2200> >();
        MagCurveGeo<ELEMENT_PARAMS>(sc, "/home/ilka/test/magcurvegeo.csv", -50, 50, 200);
        delete sc;*/

        typedef TFdmElementNonlinearIsotropicLossfree<ELEMENT_PARAMS,4000,1200000,60>
            TFdmElementNonlinearIsotropicLossfreeApproximation;
        /*typedef TFdmElementNonlinearIsotropicLossfreeLut<
                ELEMENT_PARAMS,
                luttMagCurve,
                measurement_NO20_fn,
```

```
        measurement_delimiter
>TFdmElementNonlinearIsotropicLossfreeApproximation;*/

50  // ------------------------------------------------------------ SST
    ------------------------------------------------------------------
// ---------- show mag curve of NO10par for homogeneous flux ----------
MagCurve<
        ELEMENT_PARAMS,
        TFdmElementNonlinearIsotropicLossfreeLut<
55          ELEMENT_PARAMS,
            luttMeasurement,
            measurement_NO10par_fn,
            measurement_delimiter
        >
60  > ("/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-mag-par.csv",-5000,5000);
MagCurve<
        ELEMENT_PARAMS,
        TFdmElementNonlinearIsotropicLossfreeApproximation
>
    ("/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-mag-par-approx.csv",-5000,5000);
65
// ---------- show mag curve of NO10orth for homogeneous flux----------
MagCurve<
        ELEMENT_PARAMS,
        TFdmElementNonlinearIsotropicLossfreeLut<
70          ELEMENT_PARAMS,
            luttMeasurement,
            measurement_NO10orth_fn,
            measurement_delimiter
        >
75  > ("/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-mag-orth.csv",-5000,5000);
MagCurve<
        ELEMENT_PARAMS,
        TFdmElementNonlinearIsotropicLossfreeApproximation
>
    ("/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-mag-orth-approx.csv",-5000,5000);
80


// ------------------------------------------------------------ SCT
    ------------------------------------------------------------------
TFdmSc<
85      ELEMENT_PARAMS,
        100,
        100,
        TFdmElementNonlinearIsotropicLossfreeApproximation
> setup(0.03,0.03);
90
//return 0;
//---------- mag curve of NO10orth ----------
{
    map<string,double> orthmmfs;
95  orthmmfs["orth0V"] = 0;
    orthmmfs["orth2V"] = 300*0.35;  // equals N*i2_DC
```

```
            for(map<string,double>::iterator itr=orthmmfs.begin(); itr!=orthmmfs.end();
                itr++) {
                setup.relax();
                setup.setMMF2(itr->second);
100             MagCurveGeo
                        <ELEMENT_PARAMS>
                        (&setup,
                        "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-mag-orth-SCT-"+itr->first+".csv"
                        -300*0.42,
105                     +300*0.42,
                        100);
            }
        }
        //------------ time-transient replay of measurement on NO10orth ------------
110     MagneticComponentReplay<ELEMENT_PARAMS>(
          &setup,
          0.1E-3,
          "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SCT-slim-orth/test-10-0.csv",
          "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SCT-slim-orth/sim-10-0.csv",
115       ";",
          300, 40, 300, 40);
        MagneticComponentReplay<ELEMENT_PARAMS>(
          &setup,
          0.1E-3,
120       "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SCT-slim-orth/test-10-10.csv",
          "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SCT-slim-orth/sim-10-10.csv",
          ";",
          300, 40, 300, 40);
    } catch (TException &e) {
125     e.show();
        return 1;
    }
    catch(...) {
        TException("Unknown exception").show();
130     return 1;
    }
    cout.flush();
    for(vector<std::thread*>::iterator itr=threads.begin(); itr!=threads.end(); itr++)
        if((*itr)->joinable())
135         (*itr)->join();
    auto duration = std::chrono::high_resolution_clock::now() - time;
    cout << endl<<"all threads done - computing time:
        "<<std::chrono::duration_cast<std::chrono::milliseconds>(duration).count()<<"
        ms"<<endl;
    cout.flush();
    return 0;
140 }
```

C Handbücher

Diese Arbeit wird zusammen mit den folgenden Handbüchern ausgeliefert:

- Softwarepaket „OrthoMeasure“ von Ilka Schulz
- Softwarepaket „OrthoSim“ von Ilka Schulz

Die Handbücher sind Teil der schriftlichen Ausarbeitung und ergänzen dieses Dokument um wichtige Informationen. Wenn Ihnen die Handbücher nicht zur Verfügung stehen, fragen Sie bitte Ihre*n Distributor*in für die vorliegende Arbeit oder kontaktieren Sie die Autorin der Handbücher:

Leibniz Universität Hannover
Institut für Antriebssysteme und Leistungselektronik
z. Hd. Ilka Schulz
Welfengarten 1
30167 Hannover
Germany

Handbook OrthoMeasure

Generated by Doxygen 1.8.13

Contents

1 Todo List 2

2 Hierarchical Index 2

2.1 Class Hierarchy . . . 2

3 Class Index 3

3.1 Class List . . . 3

4 File Index 3

4.1 File List . . . 3

5 Class Documentation 4

5.1 TException Class Reference . . . 4

5.1.1 Constructor & Destructor Documentation . . . 4

5.1.2 Member Function Documentation . . . 4

5.1.3 Member Data Documentation . . . 5

5.2 TImpedanceMeasureSetup Class Reference . . . 5

5.2.1 Detailed Description . . . 7

5.2.2 Constructor & Destructor Documentation . . . 7

5.2.3 Member Function Documentation . . . 8

5.2.4 Member Data Documentation . . . 16

5.3 TInput Class Reference . . . 17

5.3.1 Detailed Description . . . 18

5.3.2 Constructor & Destructor Documentation . . . 18

5.3.3 Member Function Documentation . . . 19

5.3.4 Member Data Documentation . . . 26

5.4 TInterface Class Reference . . . 26

5.4.1 Detailed Description . . . 27

5.4.2 Constructor & Destructor Documentation . . . 28

5.4.3 Member Function Documentation . . . 28

5.4.4 Member Data Documentation . . . 35

5.5 TOutput Class Reference 36
5.5.1 Detailed Description 38
5.5.2 Member Enumeration Documentation 38
5.5.3 Constructor & Destructor Documentation 39
5.5.4 Member Function Documentation 40
5.5.5 Member Data Documentation 53
5.6 TScope Struct Reference 54
5.6.1 Detailed Description 55
5.6.2 Constructor & Destructor Documentation 55
5.6.3 Member Function Documentation 56
5.7 TSocket Class Reference 58
5.7.1 Detailed Description 59
5.7.2 Constructor & Destructor Documentation 59
5.7.3 Member Function Documentation 60
5.7.4 Member Data Documentation 61

6 File Documentation 62
6.1 src/interface/impedancemeasuresetup.cpp File Reference 62
6.2 src/interface/impedancemeasuresetup.hpp File Reference 63
6.2.1 Enumeration Type Documentation 63
6.3 src/interface/input.cpp File Reference 64
6.4 src/interface/input.hpp File Reference 64
6.5 src/interface/interface.cpp File Reference 65
6.6 src/interface/interface.hpp File Reference 65
6.7 src/interface/output.cpp File Reference 66
6.7.1 Function Documentation 67
6.8 src/interface/output.hpp File Reference 67
6.8.1 Enumeration Type Documentation 68
6.8.2 Function Documentation 69
6.9 src/scope.cpp File Reference 69
6.10 src/scope.hpp File Reference 70
6.11 src/util.cpp File Reference 70
6.11.1 Function Documentation 71
6.11.2 Variable Documentation 74
6.12 src/util.hpp File Reference 74
6.12.1 Macro Definition Documentation 75
6.12.2 Function Documentation 75
6.12.3 Variable Documentation 77

Index **79**

1 Todo List

Member TInput::read (p. **25**) **()**
: use double instead of int?

Class TInterface (p. **26**)
: questionable design to keep hardware access and calibration in same class, maybe resulting in too powerful derived classes

Member TInterface::setName (p. **33**) **(string _name)**
: write another method TInterface::migrateName(string) that migrates config file to new name

Class TOutput (p. **36**)
: implement static list for instances to register, so no two instances can collide
: make multiple instances of executables run simultaneously without unpredictable outcome

Member TOutput::convertValueToPWM (p. **40**) **(double value)**
: check range

Member TOutput::getCurrentPwm (p. **41**) **()**
: verify that this is actually correct

Member TOutput::getRangeMax (p. **43**) **()**
: magic number

Member TOutput::getRangeMin (p. **43**) **()**
: magic number

Member TOutput::write (p. **52**) **(double pwm)**
: use double instead of int?

Member TScope::measure (p. **57**) **()**
: get smallest and largest scope values automatically
: check preamble, whether channel is actually activaed
: maybe activate every channel beforehand?

Class TSocket (p. **58**)
: support UTF-8

Member TSocket::TSocket (p. **59**) **(string remote_addr, int port)**
: implement IPv6
: implement hostname resolution

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

TException **4**

TInterface **26**

TImpedanceMeasureSetup 5

TInput 17

TOutput 36

TSocket 58

TScope 54

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

TException 4

TImpedanceMeasureSetup
Output, one voltage input and one current input 5

TInput
Analog input including calibration 17

TInterface
Abstract representation of an input or output or complex hardware interface 26

TOutput
Analog output including calibration 36

TScope
Connected scope (Tektronix TBS2000 series) 54

TSocket
Makes use of Linux' socket libary and provides object-oriented access 58

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

src/ scope.cpp 69

src/ scope.hpp 70

src/ util.cpp 70

src/ util.hpp 74

src/interface/ impedancemeasuresetup.cpp 62

src/interface/ impedancemeasuresetup.hpp 63

src/interface/ input.cpp 64

src/interface/ input.hpp 64

src/interface/ interface.cpp 65

src/interface/ interface.hpp 65

src/interface/ output.cpp 66

src/interface/ output.hpp 67

5 Class Documentation

5.1 TException Class Reference

```
#include <util.hpp>
```

Public Member Functions

- **TException** (string _msg)
- string **getMsg** ()
- void **show** ()

Private Attributes

- string **msg**

5.1.1 Constructor & Destructor Documentation

5.1.1.1 TException()

```
TException::TException (
            string _msg )
```

5.1.2 Member Function Documentation

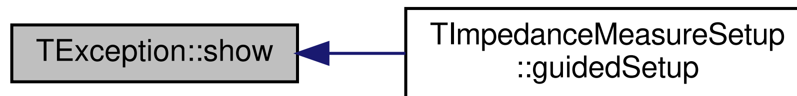
5.1.2.1 getMsg()

```
string TException::getMsg ( )
```

5.1.2.2 show()

```
void TException::show ( )
```

Here is the caller graph for this function:



5.1.3 Member Data Documentation

5.1.3.1 msg

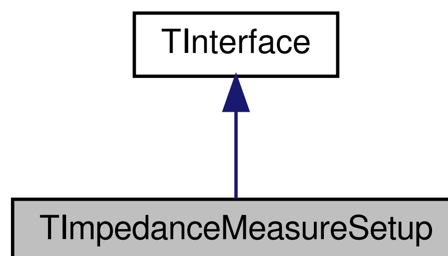
```
string TException::msg [private]
```

5.2 TImpedanceMeasureSetup Class Reference

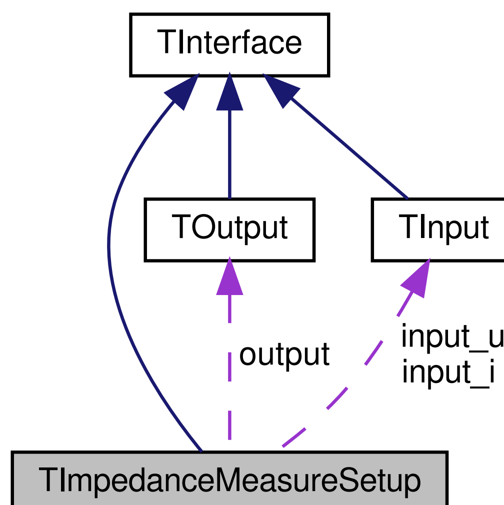
represents an output, one voltage input and one current input

```
#include <impedancemeasuresetup.hpp>
```

Inheritance diagram for TImpedanceMeasureSetup:



Collaboration diagram for TImpedanceMeasureSetup:



Public Member Functions

- **TImpedanceMeasureSetup** (string _name, bool force_load)

    constructor
- virtual ~**TImpedanceMeasureSetup** ()

    destructor
- void **setOutput** (**TOutput** *_ouput)

    sets output
- void **setInputU** (**TInput** *_input_u)

    sets volage input
- void **setInputI** (**TInput** *_input_i)

    sets current input
- void **setOutputByName** (string on)

    sets output by name.
- void **setInputUByName** (string iun)

    sets voltage input by name.
- void **setInputIByName** (string iin)

    sets current input by name.
- void **setOutputType** (**TImpedanceMeasureSetupOutputType** _output_type)

    sets output type
- **TOutput** * **getOutput** () const

    gets output as pointer
- **TInput** * **getInputU** () const

    gets voltage input as pointer
- **TInput** * **getInputI** () const

    gets current input as pointer
- string **getOutputName** () const

    gets output name (convenience method)
- string **getInputUName** () const

    gets voltage input name (convenience method)
- string **getInputIName** () const

    gets current input name (convenience method)
- **TImpedanceMeasureSetupOutputType** **getOutputType** () const

    gets output type
- virtual void **guidedSetup** (bool allow_name_change) override

    *implementation of **TInterface::guidedSetup()** (p. 31)*

Protected Member Functions

- virtual string **getClass** () const override

    *implementation of **TInterface::getClass()** (p. 28)*
- virtual map< string, string > **getProperties** () override

    *implementation of **TInterface::getProperties** (p. 30)*
- virtual void **setProperty** (string id, string val) override

    *implementation of **TInterface::setProperty** (p. 34)*

Static Private Member Functions

- static string **outputTypeToName** (**TImpedanceMeasureSetupOutputType** ot)
  converts a output type to a human-readable name (also see TImpedanceMeasureSetupOutputType). Throws an exception if provided with an invalid parameter.
- static **TImpedanceMeasureSetupOutputType** **nameToOutputType** (string n)
  converts a human-readable name to an output type (also see TImpedanceMeasureSetupOutputType). Throws an exception if provided with an invalid parameter.

Private Attributes

- **TOutput** ∗ **output**
  pointer to an output
- **TInput** ∗ **input_u**
  pointer to an input measuring the voltage over the impedance
- **TInput** ∗ **input_i**
  pointer to an input measuring the current over the impedance
- **TImpedanceMeasureSetupOutputType** **output_type**
  output type

Additional Inherited Members

5.2.1 Detailed Description

represents an output, one voltage input and one current input

This class basically holds an output, one voltage input and one current input and is therefore only for convenience. However, using the TImpedanceMeasureSetupOutputType, it may be extended by complex function that analyze impedances.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 TImpedanceMeasureSetup()

```
TImpedanceMeasureSetup::TImpedanceMeasureSetup (
            string _name,
            bool force_load )
```

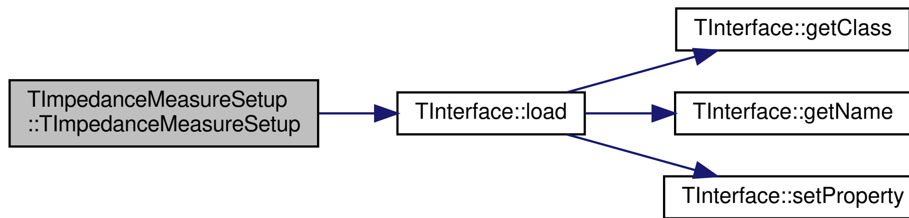
constructor

The constructor will attempt to load the interfaces properties from file according to its name; this behavior can not be turned off. If loading the configuration fails, the parameter force_load determines, whether a warning will be printed to the console or an exception will be thrown. All pointers will be set to nullptr

Parameters

_name	passed to **TInterface::TInterface()** (p. 28)
force_load	passed to **TInterface::load()** (p. 31)

Here is the call graph for this function:



5.2.2.2 ∼TImpedanceMeasureSetup()

```
TImpedanceMeasureSetup::∼TImpedanceMeasureSetup ( )  [virtual]
```

destructor

A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. All objects, that are pointed to by the pointers (except nullptr) will be deleted.

5.2.3 Member Function Documentation

5.2.3.1 getClass()

```
string TImpedanceMeasureSetup::getClass ( ) const  [override], [protected], [virtual]
```

implementation of **TInterface::getClass()** (p. 28)

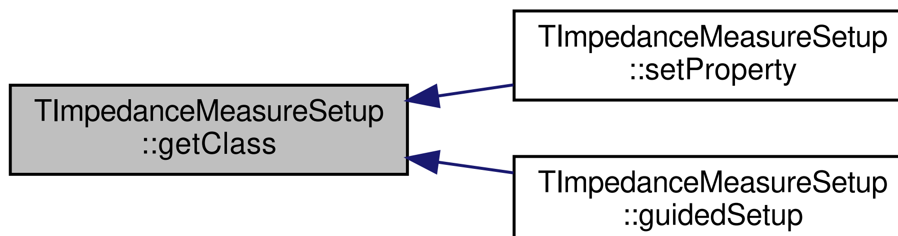
Returns a string unique for this class in order to group all objects of this class, especially for structuring config files or perhaps user interfaces

Returns

string "impedancemeasuresetup"

Implements **TInterface** (p. 28).

Here is the caller graph for this function:

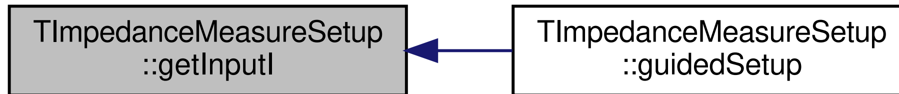


5.2.3.2 getInputI()

```
TInput * TImpedanceMeasureSetup::getInputI ( ) const
```

gets current input as pointer

Here is the caller graph for this function:

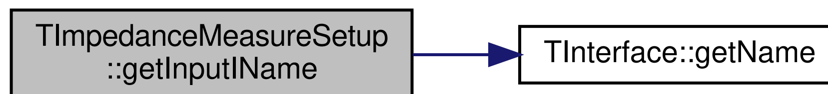


5.2.3.3 getInputIName()

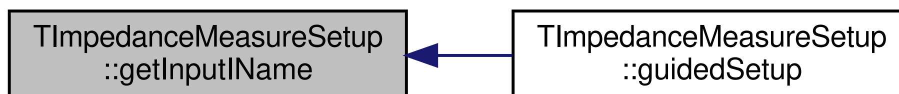
```
string TImpedanceMeasureSetup::getInputIName ( ) const
```

gets current input name (convenience method)

Here is the call graph for this function:



Here is the caller graph for this function:

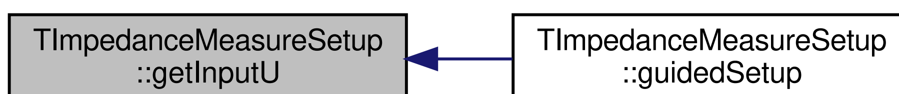


5.2.3.4 getInputU()

```
TInput * TImpedanceMeasureSetup::getInputU ( ) const
```

gets voltage input as pointer

Here is the caller graph for this function:

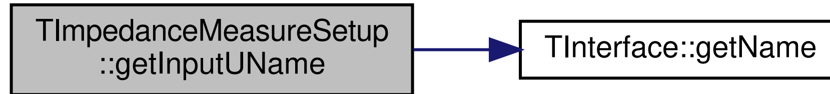


5.2.3.5 getInputUName()

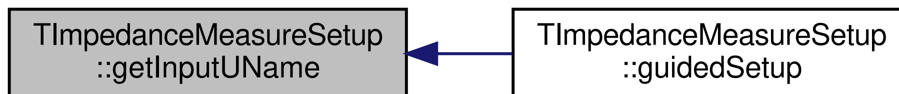
`string TImpedanceMeasureSetup::getInputUName ( ) const`

gets voltage input name (convenience method)

Here is the call graph for this function:



Here is the caller graph for this function:

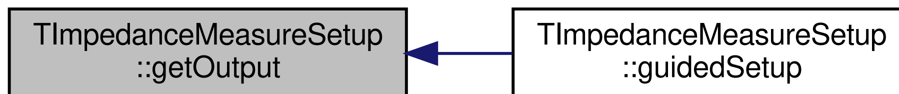


5.2.3.6 getOutput()

`TOutput * TImpedanceMeasureSetup::getOutput ( ) const`

gets output as pointer

Here is the caller graph for this function:

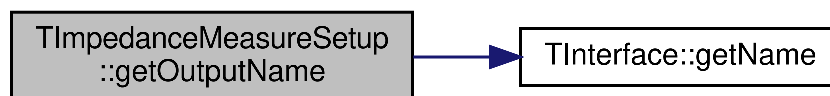


5.2.3.7 getOutputName()

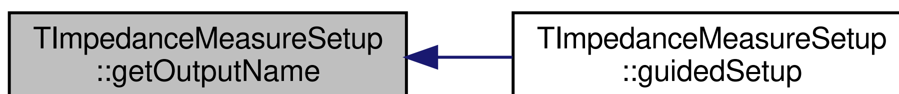
`string TImpedanceMeasureSetup::getOutputName ( ) const`

gets output name (convenience method)

Here is the call graph for this function:



Here is the caller graph for this function:



5.2.3.8 getOutputType()

```
TImpedanceMeasureSetupOutputType TImpedanceMeasureSetup::getOutputType ( ) const
```

gets output type

5.2.3.9 getProperties()

```
map< string, string > TImpedanceMeasureSetup::getProperties ( ) [override], [protected],
[virtual]
```

implementation of **TInterface::getProperties** (p. 30)

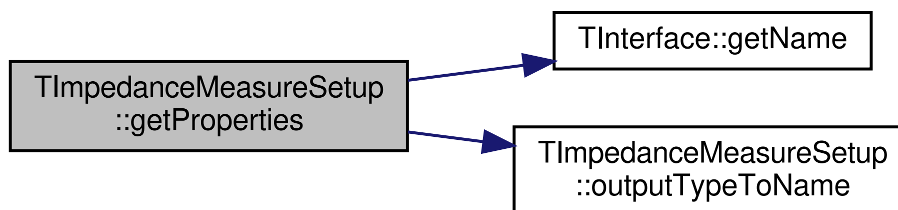
used by parent class, e.g. for configuration files or user interfaces

Returns

map associating property names with properties' values

Implements **TInterface** (p. 30).

Here is the call graph for this function:



5.2.3.10 guidedSetup()

```
void TImpedanceMeasureSetup::guidedSetup (
            bool allow_name_change ) [override], [virtual]
```

implementation of **TInterface::guidedSetup()** (p. 31)

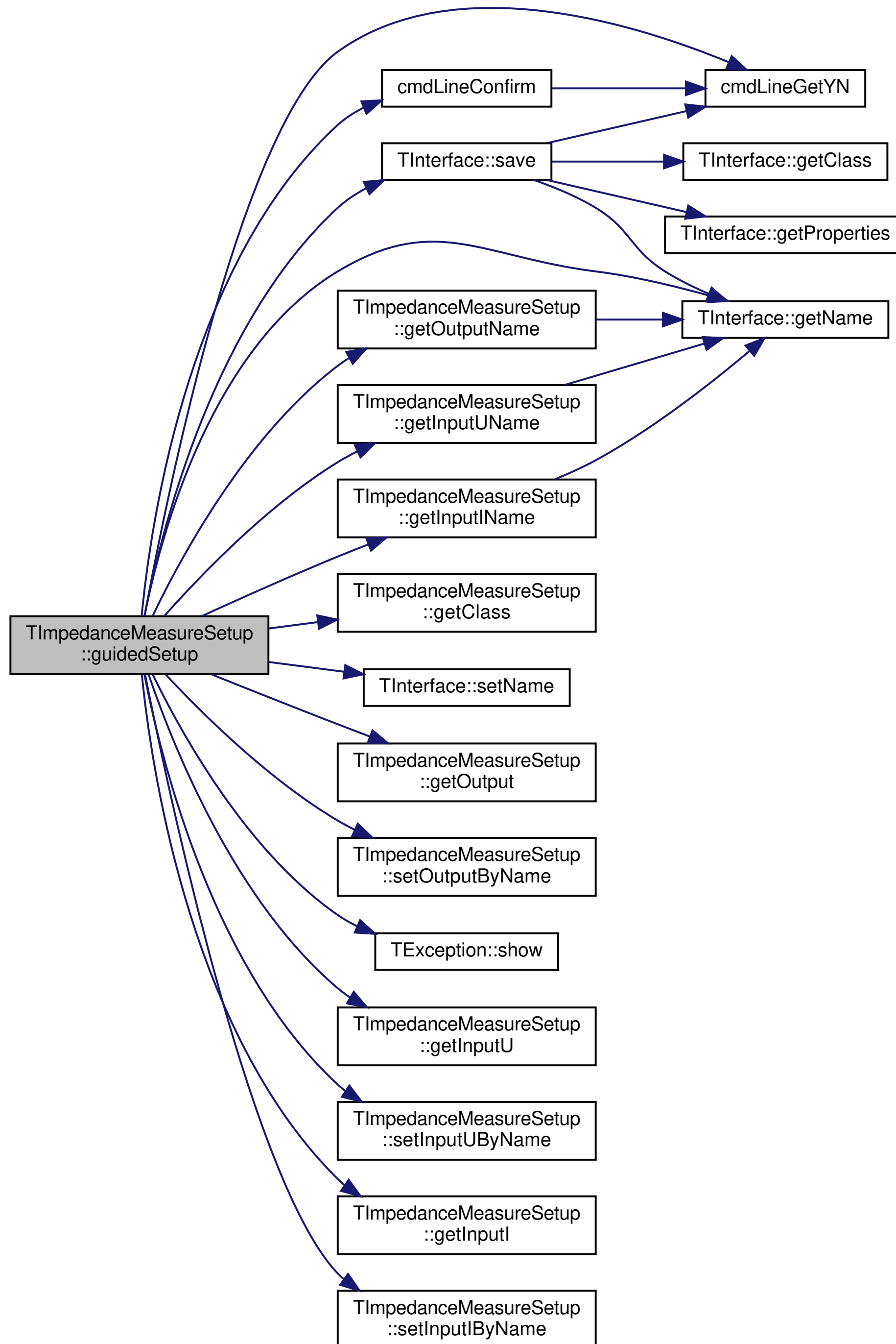
Walks the user through a guided setup and calibration. Starts a textual interface on the terminal to let the user setup the name and interfaces. Finally, the user may decide to save or discard alle changes. If the user is allowed to change the interfaces's name the config file will probably change its name, too, so an old config file may corrupt the config structure.

Parameters

allow_name_change	whether the user is allowed to change the interface's name

Implements **TInterface** (p. 31).

Here is the call graph for this function:



5.2.3.11 nameToOutputType()

```
TImpedanceMeasureSetupOutputType TImpedanceMeasureSetup::nameToOutputType (
            string n ) [static], [private]
```

converts a human-readable name to an output type (also see TImpedanceMeasureSetupOutputType). Throws an exception if provided with an invalid parameter.

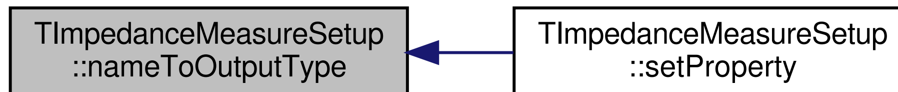
Parameters

n	textual representation of the output type to convert

Returns

the according output type

Here is the caller graph for this function:



5.2.3.12 outputTypeToName()

```
string TImpedanceMeasureSetup::outputTypeToName (
              TImpedanceMeasureSetupOutputType ot )  [static], [private]
```

converts a output type to a human-readable name (also see TImpedanceMeasureSetupOutputType). Throws an exception if provided with an invalid parameter.

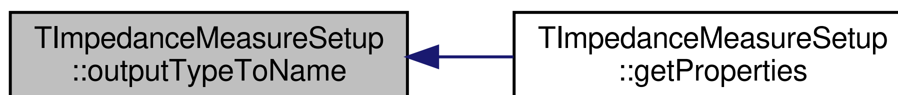
Parameters

ot	output type to convert

Returns

according textual representation

Here is the caller graph for this function:



5.2.3.13 setInputI()

```
void TImpedanceMeasureSetup::setInputI (
              TInput * _input_i )
```

sets current input

This method copies the pointer, not the class!

5.2.3.14 setInputIByName()

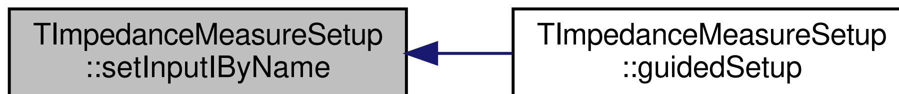
```
void TImpedanceMeasureSetup::setInputIByName (
            string iin )
```

sets current input by name.

Parameters

iin	input name

Here is the caller graph for this function:



5.2.3.15 setInputU()

```
void TImpedanceMeasureSetup::setInputU (
            TInput * _input_u )
```

sets volage input

This method copies the pointer, not the class!

5.2.3.16 setInputUByName()

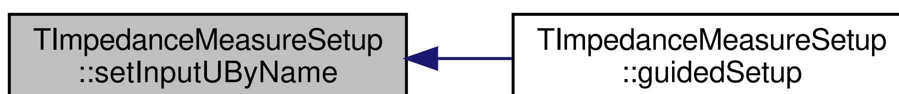
```
void TImpedanceMeasureSetup::setInputUByName (
            string iun )
```

sets voltage input by name.

Parameters

iun	input name

Here is the caller graph for this function:



5.2.3.17 setOutput()

```
void TImpedanceMeasureSetup::setOutput (
              TOutput * _ouput )
```

sets output

This method copies the pointer, not the class!

5.2.3.18 setOutputByName()

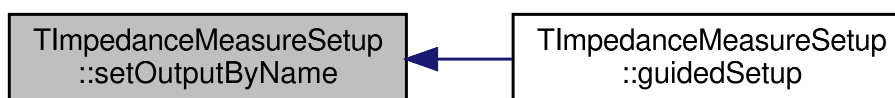
```
void TImpedanceMeasureSetup::setOutputByName (
             string on )
```

sets output by name.

Parameters

on	output name

Here is the caller graph for this function:



5.2.3.19 setOutputType()

```
void TImpedanceMeasureSetup::setOutputType (
              TImpedanceMeasureSetupOutputType _output_type )
```

sets output type

5.2.3.20 setProperty()

```
void TImpedanceMeasureSetup::setProperty (
             string id,
             string val )  [override], [protected], [virtual]
```

implementation of **TInterface::setProperty** (p. 34)

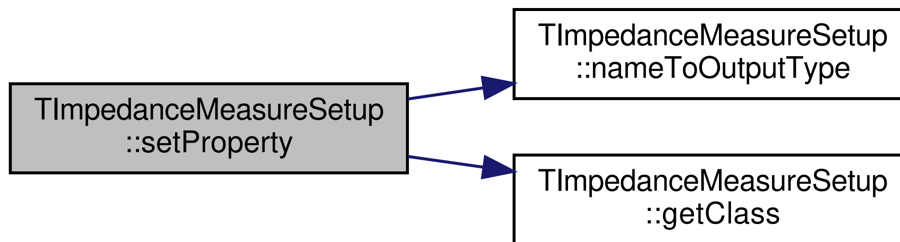
Sets the property with the specified name to the specified value. Specifically, all data other than string may be converted, so calling functions must apply to general standards

Parameters

id	textual name of a property to manipulate
val	textual representation of the data to set

Implements **TInterface** (p. 34).

Here is the call graph for this function:



5.2.4 Member Data Documentation

5.2.4.1 input_i

TInput∗ TImpedanceMeasureSetup::input_i [private]

pointer to an input measuring the current over the impedance

5.2.4.2 input_u

TInput∗ TImpedanceMeasureSetup::input_u [private]

pointer to an input measuring the voltage over the impedance

5.2.4.3 output

TOutput∗ TImpedanceMeasureSetup::output [private]

pointer to an output

That may be any kind of output and it is not necessary for an impedance measurement. However, most measurement methods will need an output.

5.2.4.4 output_type

TImpedanceMeasureSetupOutputType TImpedanceMeasureSetup::output_type [private]

output type

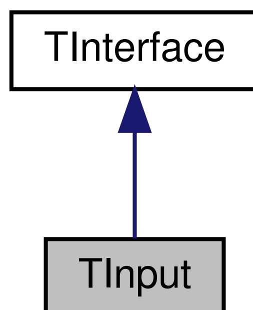
This is not used in any of the tools so far and shall be reviewd before use. See TImpedanceMeasureSetupOutputType for details.

5.3 TInput Class Reference

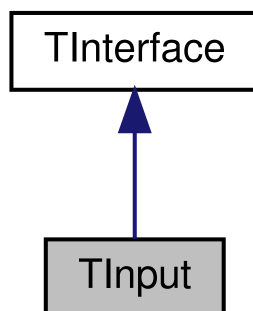
represents an analog input including calibration.

```
#include <input.hpp>
```

Inheritance diagram for TInput:



Collaboration diagram for TInput:



Public Member Functions

- **TInput** (string _name, bool force_load)
  constructor
- virtual ∼**TInput** ()
  destructor
- void **calibrateZero** (double measured)
  changes calibration to understand the measured value as physical zero
- void **calibrateValue** (double val, double measured)
  changes calibration to understand the measured value as the specified physical value
- void **calibrateTwoPoints** (double vala, double measureda, double valb, double measuredb)
  changes calibration to understand the two measured values as the specified physical values
- virtual void **guidedSetup** (bool allow_name_change) override
  implementation of ***TInterface::guidedSetup()*** *(p. 31)*
- double **convertMeasurementToValue** (double measured)
  interpretes a measured raw value as the corresponding physical value
- double **measureValue** ()
  takes a raw measurement and applies calibration to get calibrated physical value

Protected Member Functions

- virtual string **getClass** () const override
  *implementation of **TInterface::getClass()** (p. 28)*
- virtual map< string, string > **getProperties** () override
  *implementation of **TInterface::getProperties** (p. 30)*
- virtual void **setProperty** (string id, string val) override
  *implementation of **TInterface::setProperty** (p. 34)*

Private Member Functions

- double **read** ()
  reads a raw value from the analog input

Static Private Member Functions

- static string **portToName** (uint p)
  converts an enumerated port value to a readable string
- static uint **nameToPort** (string n)
  converts a readable string to an enumerated port value

Private Attributes

- uint **port**
  Red Pitaya's input port.
- double **zero**
  measurement value that corresponds to physical value equal zero
- double **slope**
  physical value per measurement value

Additional Inherited Members

5.3.1 Detailed Description

represents an analog input including calibration.

This class holds the port and calibration for a RedPitaya input (only slow ones) It can read from them. There are serious issues with performance. This does acutally not read faster than approx. 18 kS/s and gets even slower, when reading from multiple objects simultaneously. The 100 kS/s, specified by the manufacturer, cannot be reached.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 TInput()

```
TInput::TInput (
            string _name,
            bool force_load )
```

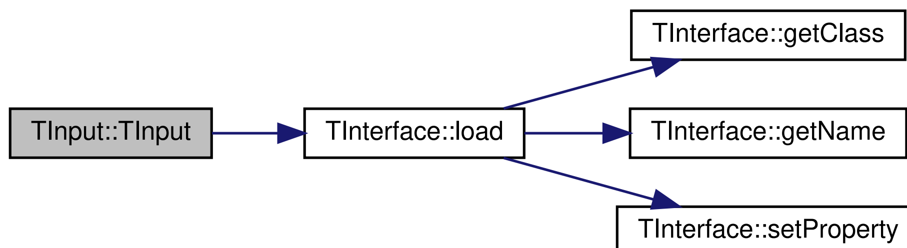
constructor

The constructor will attempt to load the interfaces properties from file according to its name; this behavior can not be turned off. If loading the configuration fails, the parameter force_load determines, whether a warning will be printed to the console or an exception will be thrown.

Parameters

_name	passed to **TInterface::TInterface** (p. 28)
force_load	passed to **TInterface::load()** (p. 31)

Here is the call graph for this function:



5.3.2.2 ∼TInput()

```
TInput::∼TInput ( ) [virtual]
```

destructor

A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. It does not contain code, though.

5.3.3 Member Function Documentation

5.3.3.1 calibrateTwoPoints()

```
void TInput::calibrateTwoPoints (
            double vala,
            double measureda,
            double valb,
            double measuredb )
```

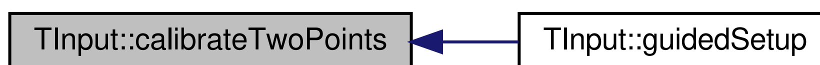
changes calibration to understand the two measured values as the specified physical values

Two measurements with different values shall be taken, usually by using the method **TInput::read()** (p. 25), and be passed to this method. The (possibly dimensionless) measured values will then be intepreted as the specified physical values. The changed calibration is not automatically saved, use **TInterface::save()** (p. 33) for that.

Parameters

vala	the physical value associated with the raw value measureda
measureda	the raw value which is to be interpreted as the physical value vala
valb	the physical value associated with the raw value measuredb
measuredb	the raw value which is to be interpreted as the physical value valb

Here is the caller graph for this function:



5.3.3.2 calibrateValue()

```
void TInput::calibrateValue (
            double val,
            double measured )
```

changes calibration to understand the measured value as the specified physical value

Before this function is called, TInterface::calibrateZero() must be finished. A measurement shall be taken, usually by using the method **TInput::read()** (p. 25), and be passed to this method. The (possibly dimensionless) measured value will then be interpreted as the specified physical value. The changed calibration is not automatically saved, use **TInterface::save()** (p. 33) for that.

Parameters

val	physical value, measured with a calibration tool
measured	raw value, measured with **TInput::read()** (p. 25)

5.3.3.3 calibrateZero()

```
void TInput::calibrateZero (
            double measured )
```

changes calibration to understand the measured value as physical zero

A measurement shall be taken, usually by using the method **TInput::read()** (p. 25), and be passed to this method. The (possibly dimensionless) measured value will then be intepreted as a physical value of zero. The changed calibration is not automatically saved, use **TInterface::save()** (p. 33) for that.

Parameters

measured	the measured value which is to be interpreted as physical zero

5.3.3.4 convertMeasurementToValue()

```
double TInput::convertMeasurementToValue (
            double measured )
```

interpretes a measured raw value as the corresponding physical value

Conversion is based on the object's internal calibration, so that must be available first.

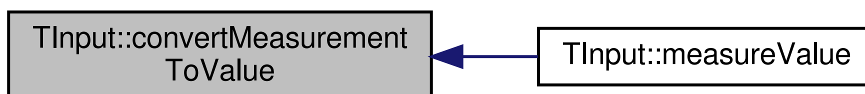
Parameters

measured	measured raw value

Returns

corresponding physical value according to calibration

Here is the caller graph for this function:



5.3.3.5 getClass()

```
string TInput::getClass ( ) const  [override], [protected], [virtual]
```

implementation of **TInterface::getClass()** (p. 28)

Returns a string unique for this class in order to group all objects of this class, especially for structuring config files or perhaps user interfaces

Returns

string "input"

Implements **TInterface** (p. 28).

Here is the caller graph for this function:

5.3.3.6 getProperties()

```
map< string, string > TInput::getProperties ( )  [override], [protected], [virtual]
```

implementation of **TInterface::getProperties** (p. 30)

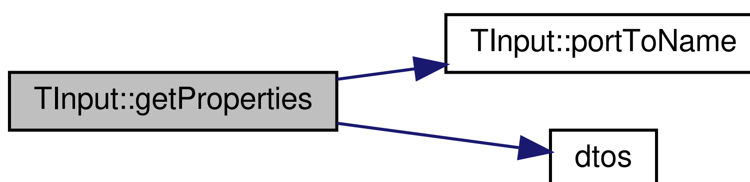
used by parent class, e.g. for configuration files or user interfaces

Returns

map associating property names with properties' values

Implements **TInterface** (p. 30).

Here is the call graph for this function:



5.3.3.7 guidedSetup()

```
void TInput::guidedSetup (
            bool allow_name_change )  [override], [virtual]
```

implementation of **TInterface::guidedSetup()** (p. 31)

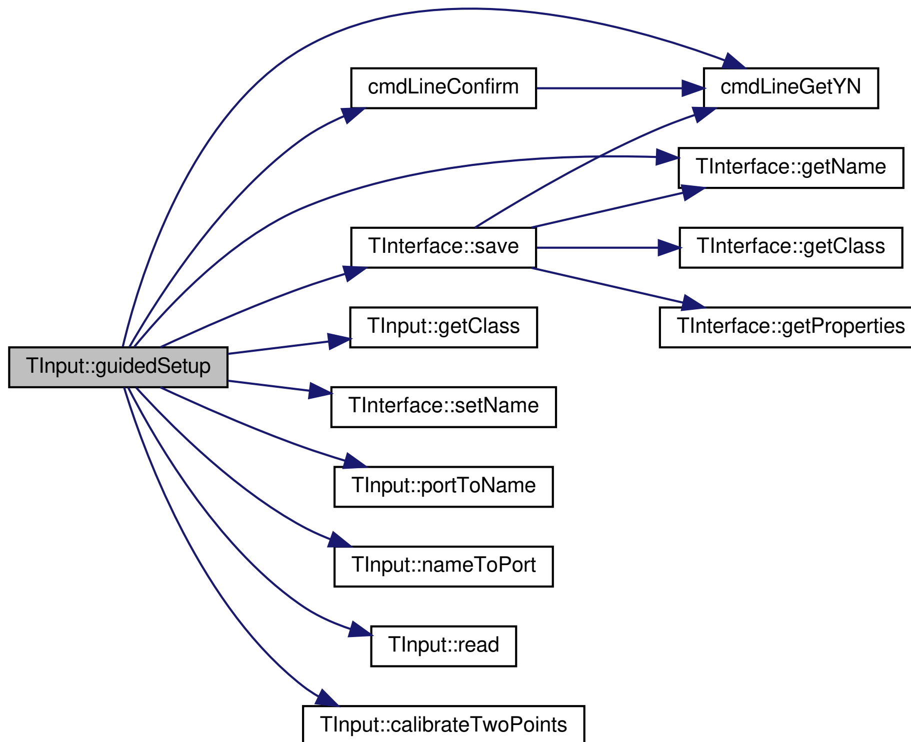
Walks the user through a guided setup and calibration. Starts a textual interface on the terminal to let the user setup the name and port and also provides a convenient wizard for calibration. Finally, the user may decide to save or discard alle changes. If the user is allowed to change the interface's name the config file will probably change its name, too, so an old config file may corrupt the config structure.

Parameters

allow_name_change	whether the user is allowed to change the interface's name

Implements **TInterface** (p. 31).

Here is the call graph for this function:



5.3.3.8 measureValue()

```
double TInput::measureValue ( )
```

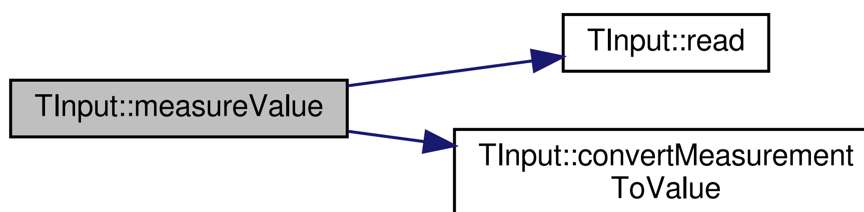
takes a raw measurement and applies calibration to get calibrated physical value

Conversion is based on the object's internal calibration, so that must be available first.

Returns

physical value from RedPitaya's input port regarding the calibration

Here is the call graph for this function:



5.3.3.9 nameToPort()

```
uint TInput::nameToPort (
             string n )  [static], [private]
```

converts a readable string to an enumerated port value

This may be used for user interaction as well as reading configuration from human-readable config files. Throws an exception for invalid parameters.

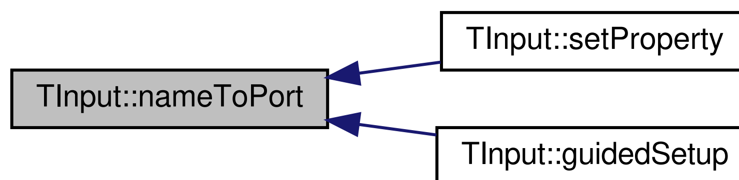
Parameters

n	textual representation of Red Pitaya's input port

Returns

Red Pitaya's input port (enumerated type) matching the specified textual representation

Here is the caller graph for this function:



5.3.3.10 portToName()

```
string TInput::portToName (
             uint p )  [static], [private]
```

converts an enumerated port value to a readable string

This may be used for user interaction as well as writing configuration to human-readable config files. Throws an exception for invalid parameters.

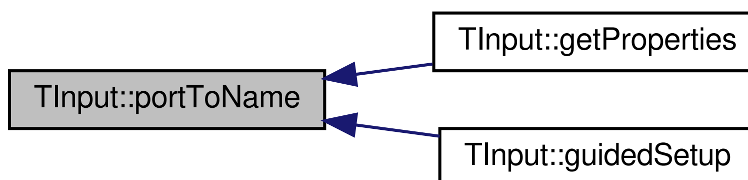
Parameters

p	Red Pitaya's input port (enumerated type)

Returns

textual representation of the specified port

Here is the caller graph for this function:



5.3.3.11 read()

```
double TInput::read ( )  [private]
```

reads a raw value from the analog input

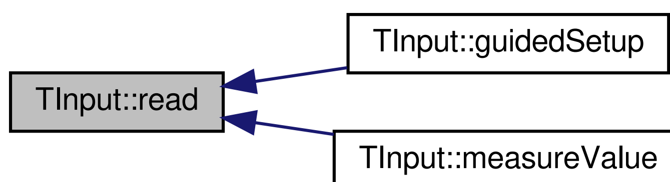
This function shall not be used by outside code as raw values are rather irrelevant, but class member function may want to read raw values. See **TInput::measureValue()** (p. 23) for access to input readings.

Returns

raw value read from input

Todo use double instead of int?

Here is the caller graph for this function:



5.3.3.12 setProperty()

```
void TInput::setProperty (
            string id,
            string val )  [override], [protected], [virtual]
```

implementation of **TInterface::setProperty** (p. 34)

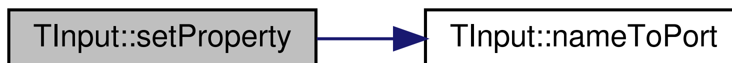
Sets the property with the specified name to the specified value. Specifically, all data other than string may be converted, so calling functions must apply to general standards

Parameters

id	textual name of a property to manipulate
val	textual representation of the data to set

Implements **TInterface** (p. 34).

Here is the call graph for this function:



5.3.4 Member Data Documentation

5.3.4.1 port

`uint TInput::port [private]`

Red Pitaya's input port.

This must be a slow analog input. Fast analog inputs and digital inputs are not supported.

5.3.4.2 slope

`double TInput::slope [private]`

physical value per measurement value

This is part of the internal calibration and will automatically be set by the corresponding calibrating methods.

5.3.4.3 zero

`double TInput::zero [private]`

measurement value that corresponds to physical value equal zero

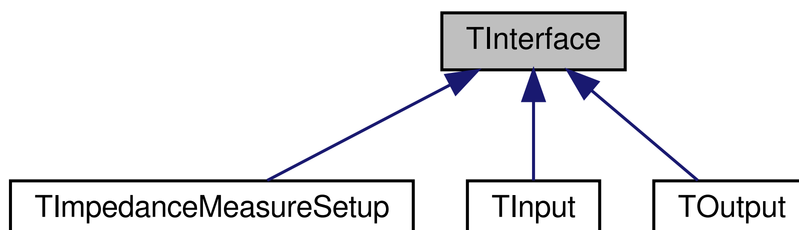
This is part of the internal calibration and will automatically be set by the corresponding calibrating methods.

5.4 TInterface Class Reference

abstract representation of an input or output or complex hardware interface

```
#include <interface.hpp>
```

Inheritance diagram for TInterface:



Public Member Functions

- **TInterface** (string _name)

  constructor
- virtual ~**TInterface** ()

  destructor A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. It does not contain code, though.
- string **getName** ()

  returns interface's name
- virtual void **guidedSetup** (bool allow_name_change)=0

  walks the user through a setup and calibration wizard
- void **save** ()

  saves configuration to file
- void **load** (bool force)

  load configuration from file
- void **load** (string _name, bool force)

  changes name and attempts to load configuration

Protected Member Functions

- void **setName** (string _name)

  sets the name variable
- virtual string **getClass** () const =0

  receives class specifier from derived classes
- virtual map< string, string > **getProperties** ()=0

  receives map of properties from derived classes
- virtual void **setProperty** (string key, string val)=0

  tells derived class to set property

Static Protected Attributes

- static const string **CONFIG_DIR** = "/home/ilka/OrthoMeasure/conf"

  unified configuration directory
- static const string **CONFIG_DELIM** = "\t"

  delimiter between property name and property value in a config file

Private Attributes

- string **name**

  the interface's unique name

5.4.1 Detailed Description

abstract representation of an input or output or complex hardware interface

This abstract class outlines the interface for inputs, outputs and complex interfaces. It also provides functionality to save properties to file and to restore from file.

Todo questionable design to keep hardware access and calibration in same class, maybe resulting in too powerful derived classes

5.4.2 Constructor & Destructor Documentation

5.4.2.1 TInterface()

```
TInterface::TInterface (
            string _name )
```

constructor

Sets the name but will NOT try to load configuration. However, derived classes' constructors may attempt to load configuration.

Parameters

_name	the name to set

5.4.2.2 ∼TInterface()

```
TInterface::∼TInterface ( ) [virtual]
```

destructor A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. It does not contain code, though.

5.4.3 Member Function Documentation

5.4.3.1 getClass()

```
virtual string TInterface::getClass ( ) const [protected], [pure virtual]
```

receives class specifier from derived classes

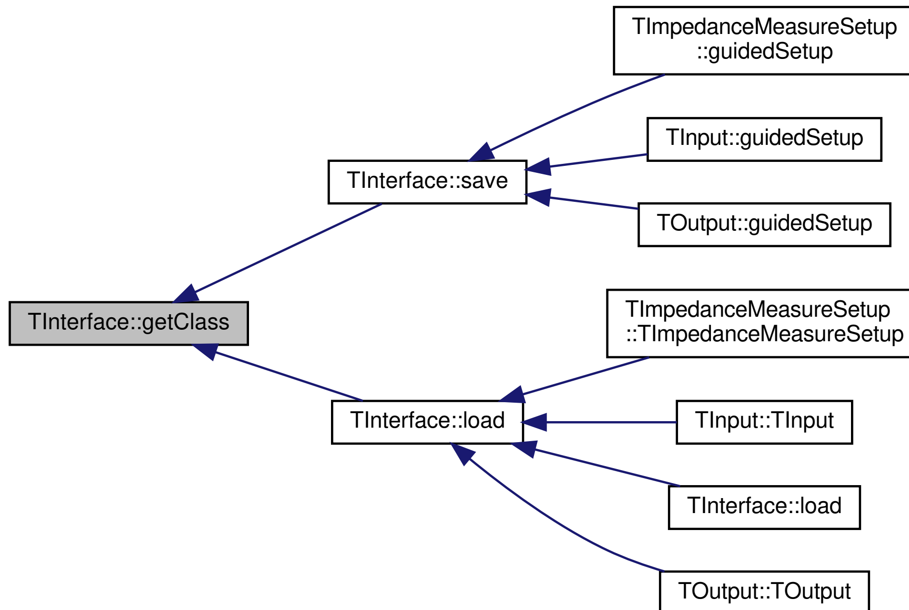
Derived classes MUST implement this method. This class name must be unique for each derived class and must be valid file path characters as the configuration file's exact path depends on it.

Returns

the derived class' identifier

Implemented in **TOutput** (p. 41), **TImpedanceMeasureSetup** (p. 8), and **TInput** (p. 21).

Here is the caller graph for this function:



5.4.3.2 getName()

```
string TInterface::getName ( )
```

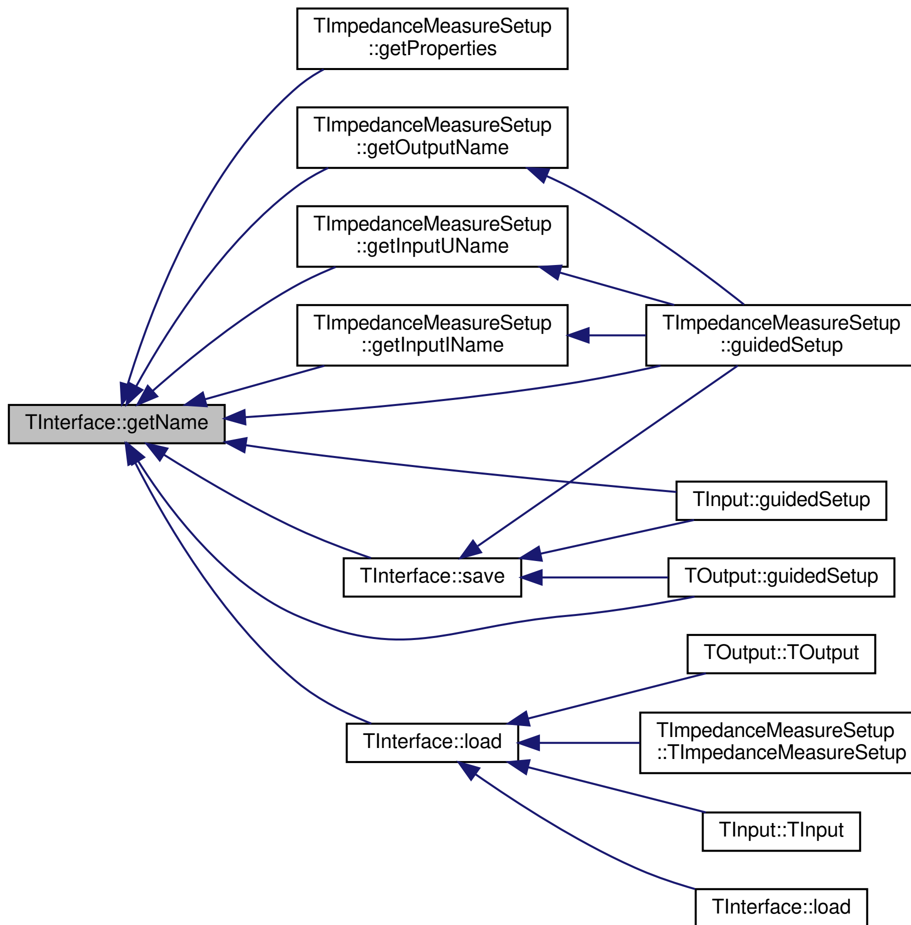
returns interface's name

The interface's name does NOT include its class name.

Returns

interface's name

Here is the caller graph for this function:



5.4.3.3 getProperties()

```
virtual map<string,string> TInterface::getProperties ( )  [protected], [pure virtual]
```

receives map of properties from derived classes

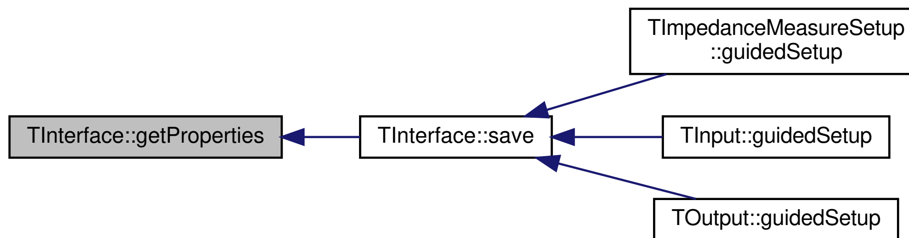
Derived classes MUST implement this method. Maps all property names to property values or textual representation of those. Needed to save configuration to config file.

Returns

map of property names (keys) and property values (values)

Implemented in **TOutput** (p. 42), **TImpedanceMeasureSetup** (p. 11), and **TInput** (p. 21).

Here is the caller graph for this function:



5.4.3.4 guidedSetup()

```
virtual void TInterface::guidedSetup (
             bool allow_name_change )  [pure virtual]
```

walks the user through a setup and calibration wizard

Derived classes MUST implement this method. The wizard may use a textual console interface or a graphical user interface, but it should be compatible to the current session, so usually a console interface MUST be implemented. The wizard should feature the following steps:

1. IF allow_name_change is true, the user may be asked to set a new name, otherwise this MUST be skipped
2. Hardware ports (inputs / outputs) may be changed
3. A convenient(!) calibration wizard may be started that even allows the user to correct mistakes The setup wizard MUST allow to keep old values and SHOULD be tolerant towards user errors and accept revocations from the user.

Parameters

allow_name_change	whether the wizard will allow the user to change the object's name

Implemented in **TOutput** (p. 44), **TImpedanceMeasureSetup** (p. 11), and **TInput** (p. 22).

5.4.3.5 load() [1/2]

```
void TInterface::load (
             bool force )
```

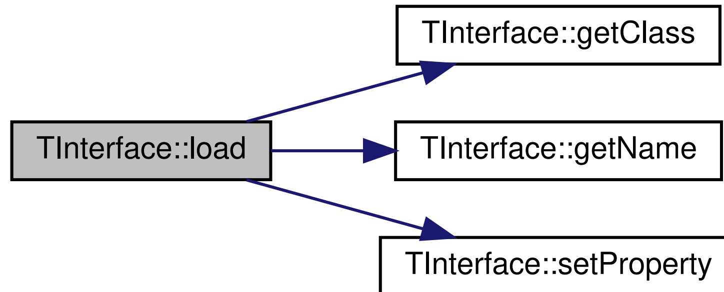
load configuration from file

This function uses **TInterface::setProperty()** (p. 34) to apply values read from file. See **TInterface::save()** (p. 33) for further details.

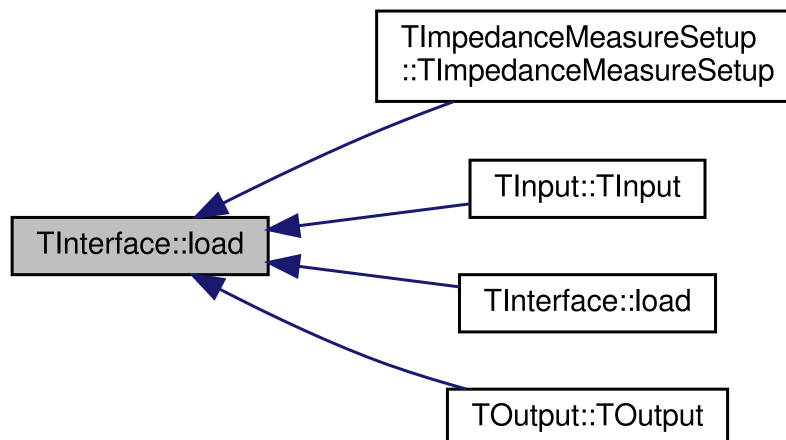
Parameters

force	whether to throw an exception on failure (otherwise just print a warning)

Here is the call graph for this function:



Here is the caller graph for this function:



5.4.3.6 load() [2/2]

```
void TInterface::load (
            string _name,
            bool force )
```

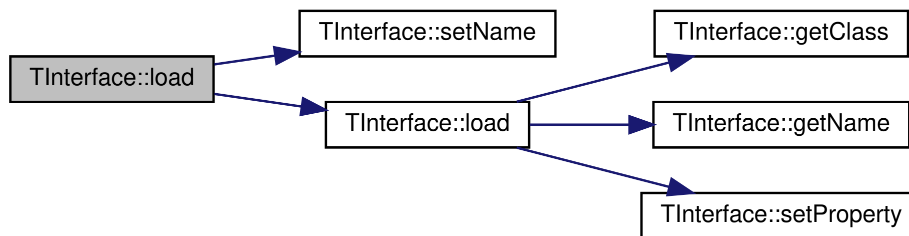
changes name and attempts to load configuration

This function does nothing else than calling **TInterface::setName()** (p. 33) and TInterface::load(force)

Parameters

_name	passed to **TInterface::setName()** (p. 33)
_force	passed to **TInterface::load()** (p. 31)

Here is the call graph for this function:

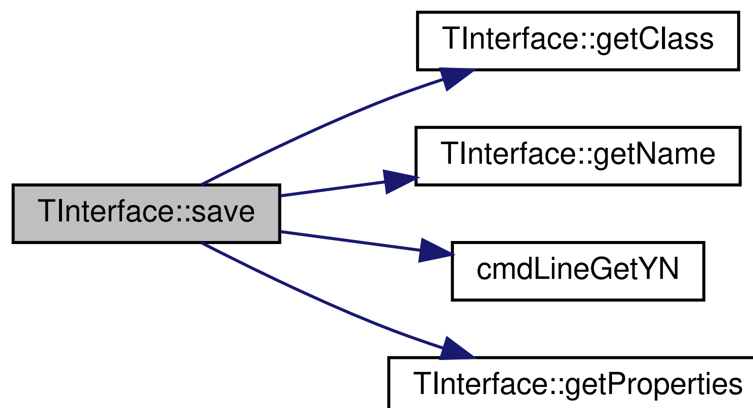


5.4.3.7 save()

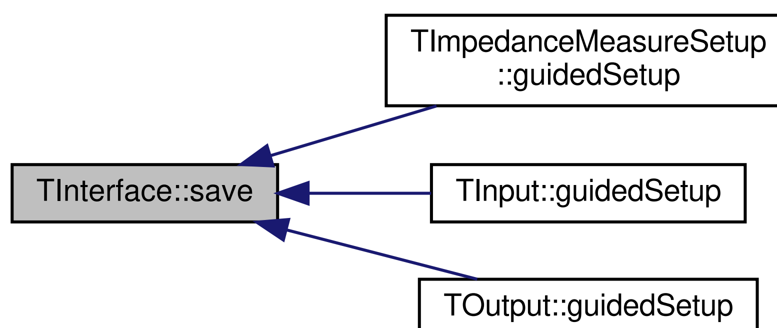
```
void TInterface::save ( )
```

saves configuration to file

This function stores all properties returned by **TInterface::getProperties()** (p. 30) to a file for later (re-)use. Class name and object name are stored indirectly in the file's path. The file name is determined by **TInterface::CON←FIG_DIR** (p. 35), **TInterface::getClass()** (p. 28) and **TInterface::getName()** (p. 29). See code for exact definition. Here is the call graph for this function:



Here is the caller graph for this function:



5.4.3.8 setName()

```
void TInterface::setName (
            string _name )  [protected]
```

sets the name variable

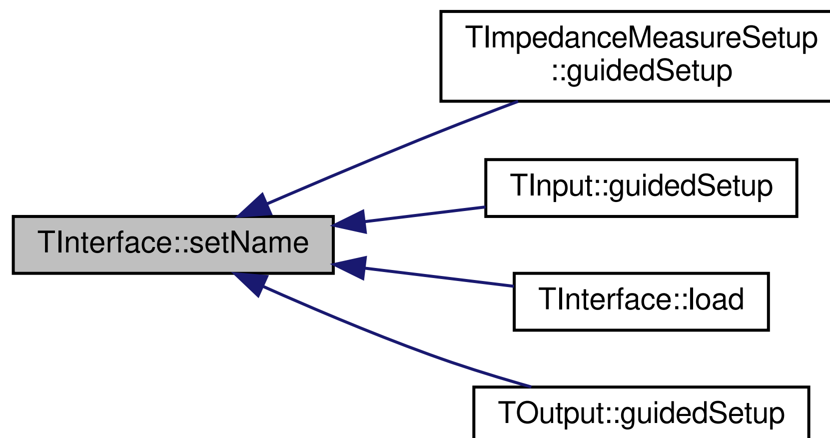
This will NOT affect any other properties and the function will NOT try to load configuration from file. However, configuration files associated with the old name will become mostly unusable after this as they are still associated with the old file name

Parameters

_name	the new name to use

Todo write another method TInterface::migrateName(string) that migrates config file to new name

Here is the caller graph for this function:



5.4.3.9 setProperty()

```
virtual void TInterface::setProperty (
            string key,
            string val )  [protected], [pure virtual]
```

tells derived class to set property

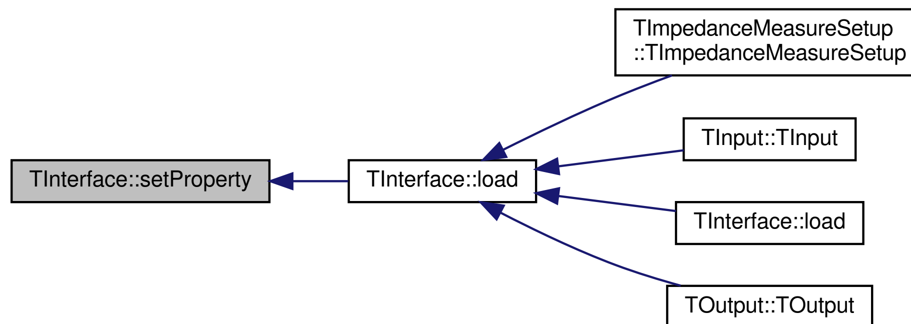
Derived classes MUST implement this method. Uses the exact same characteristic as **TInterface::getProperties()** (p. 30). Needed to load configuration from config file. Implementation SHOULD throw an exception if the property name (key) is unknown.

Parameters

key	a property's name
val	a property's value

Implemented in **TOutput** (p. 52), **TImpedanceMeasureSetup** (p. 15), and **TInput** (p. 25).

Here is the caller graph for this function:



5.4.4 Member Data Documentation

5.4.4.1 CONFIG_DELIM

```
const string TInterface::CONFIG_DELIM = "\t"  [static], [protected]
```

delimiter between property name and property value in a config file

Definition can be found in according translation unit. This is usually just a single character, e.g. '' or a ' '. Be careful, not to use '
', because this seperates different properties.

5.4.4.2 CONFIG_DIR

```
const string TInterface::CONFIG_DIR = "/home/ilka/OrthoMeasure/conf"  [static], [protected]
```

unified configuration directory

Definition can be found in according translation unit and should be an absolute(!) path without trailing '/', e.g. "/etc/OrthoMeasure/conf". Make sure the effective(!) running user has read, write and execution permission on the specified directory (effective user would usually be root, so this is irrelevant)

5.4.4.3 name

```
string TInterface::name  [private]
```

the interface's unique name

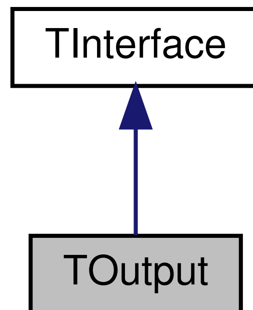
The name must be unique within all interfaces of the same interface class (see **TInterface::getClass()** (p. 28) for details) and must contain only valid file path characters. It will be used to associate a configuration file name with the interface to load from or to save to. If two object of the same interface class and the same name are created within one executable, this results in undefined behavior! Different executables (or different instances of the same executable) which contain one instance of the same interface class and the same name each may not interfere. However, changes that are made (e.g. calibration) will NOT be transfered between running programs instantly, but can be saved to disk for later loading.

5.5 TOutput Class Reference

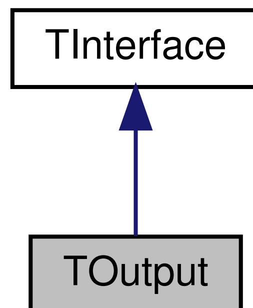
represents an analog output including calibration

`#include <output.hpp>`

Inheritance diagram for TOutput:



Collaboration diagram for TOutput:



Public Types

- enum **TPort** {
  LS0, **LS1**, **LS2**, **LS3**,
  HS0, **HS1** }

  enumerates the different ouput ports of the RedPitaya
- enum **TType** { **LS**, **HS** }

  enumerates the "fast" and the "slow" output type Each value of TPort is either a "fast" or a "slow" output, but TPort does not hold any information about that (except constants' names).

Public Member Functions

- **TOutput** (string _name, bool force_load)

  constructor
- virtual ∼**TOutput** ()

  destructor
- void **calibrateTwoPoints** (double pwma, double vala, double pwmb, double valb)

  changes calibration to understand the two measured values as the specified physical values

- virtual void **guidedSetup** (bool allow_name_change) override
  *implementation of **TInterface::guidedSetup()** (p. 31)*
- double **convertValueToPWM** (double value)
  converts a measured physical value to the corresponding raw (pwm) value
- void **putValue** (double val)
  takes a raw value, applies calibration to get raw (pwm) value and writes it to output port
- void **putSignal** (double _frequency, double _amplitude, **TWaveform** _waveform)
  generates signal on output port If the port is a fast port (TType::HS), then this function tells the built-in FPGA to create a signal. If it is a slow port (TType:LS), then this function will only initialize the necessary private variables, so that refreshSignal can update the signal and the calling code needs to loop through refreshSignal fast!
- bool **isSelfRefreshing** ()
  whether the signal runs without further support
- void **refreshSignal** ()
  refreshes signal if needed
- void **putRaw** (double pwm)
  takes a raw value and writes it to output port
- double **reconstructSignalValue** (double timestamp)
  calculates signal value at point of time

Protected Member Functions

- virtual string **getClass** () const override
  *implementation of **TInterface::getClass()** (p. 28)*
- virtual map< string, string > **getProperties** () override
  *implementation of **TInterface::getProperties** (p. 30)*
- virtual void **setProperty** (string id, string val) override
  *implementation of **TInterface::setProperty** (p. 34)*

Private Member Functions

- double **getRangeMin** ()
  get minimum value for raw output signal
- double **getRangeMax** ()
  get maximum value for raw output signal
- void **write** (double pwm)
  writes a raw value to the analog output
- double **getCurrentPwm** ()
  gets current raw output value

Static Private Member Functions

- static string **portToName** (**TPort** p)
  converts an enumerated port value to a readable string
- static **TPort** **nameToPort** (string n)
  converts a readable string to an enumerated port value
- static string **getPortNameOptions** ()
  creates a human-readable string of port options
- static **TType** **portToType** (**TPort** p)
  gets a port's type
- static rp_apin_t **portToRPLS** (**TPort** p)
  converts a port to RP API slow port
- static rp_channel_t **portToRPHS** (**TPort** p)
  converts a port to RP API fast port

Private Attributes

- **TPort port**
  Red Pitaya's output port.
- double **zero**
  raw value that corresponds to physical value equal zero
- double **slope**
  physical value per raw value
- double **signal_frequency**
  signal frequency
- double **signal_amplitude**
  signal amplitude
- **TWaveform signal_waveform**
  signal waveform
- high_resolution_clock::time_point **signal_started**
  signal starting point in time

Additional Inherited Members

5.5.1 Detailed Description

represents an analog output including calibration

This class holds the port and calibration for a RedPitaya output (both slow and fast ones). It can create signals.

NOTE: There should not be two instances of **TOutput** (p. 36) accessing the same port in a single running executable.

Todo implement static list for instances to register, so no two instances can collide

make multiple instances of executables run simultaneously without unpredictable outcome

5.5.2 Member Enumeration Documentation

5.5.2.1 TPort

```
enum TOutput::TPort
```

enumerates the different ouput ports of the RedPitaya

The RedPitaya's port definitions are a mess, so this type may be converted only for RP API calls. See rp.h for details

Enumerator

LS0	
LS1	
LS2	
LS3	
HS0	
HS1	

5.5.2.2 TType

```
enum TOutput::TType
```

enumerates the "fast" and the "slow" output type Each value of TPort is either a "fast" or a "slow" output, but TPort does not hold any information about that (except constants' names).

Enumerator

LS	
HS	

5.5.3 Constructor & Destructor Documentation

5.5.3.1 TOutput()

```
TOutput::TOutput (
            string _name,
            bool force_load )
```

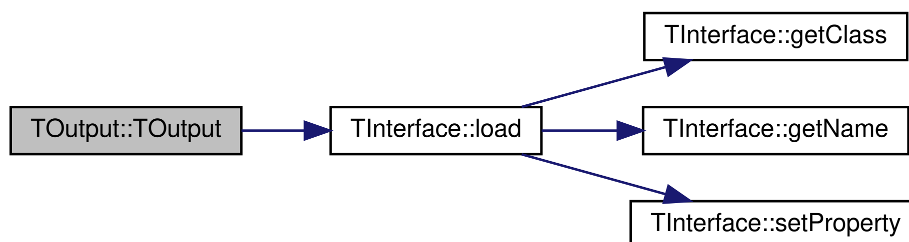
constructor

The constructor will attempt to load the interfaces properties from file according to its name; this behavior can not be turned off. If loading the configuration fails, the parameter force_load determines, whether a warning will be printed to the console or an exception will be thrown.

Parameters

_name	passed to **TInterface::TInterface** (p. 28)
force_load	passed to **TInterface::load()** (p. 31)

Here is the call graph for this function:



5.5.3.2 ∼TOutput()

```
TOutput::∼TOutput ( ) [virtual]
```

destructor

A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. It does not contain code, though.

5.5.4 Member Function Documentation

5.5.4.1 calibrateTwoPoints()

```
void TOutput::calibrateTwoPoints (
            double pwma,
            double vala,
            double pwmb,
            double valb )
```

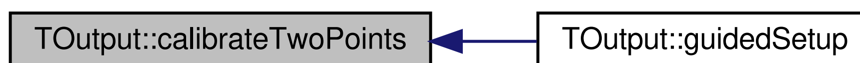
changes calibration to understand the two measured values as the specified physical values

Two measurements with different values written to the output port shall be taken, usually by using the method T←- Output::raw(), and be passed to this method. The (possibly dimensionless) measured values will then be intepreted as the specified physical values. The changed calibration is not automatically saved, use **TInterface::save()** (p. 33) for that.

Parameters

pwma	the raw (pwm) value which is to be interpreted as the physical value vala
vala	the physical value associated with the raw value pwma
pwmb	the raw (pwm) value which is to be interpreted as the physical value valb
valb	the physical value associated with the raw value pwmb

Here is the caller graph for this function:



5.5.4.2 convertValueToPWM()

```
double TOutput::convertValueToPWM (
            double value )
```

converts a measured physical value to the corresponding raw (pwm) value

Conversion is based on the object's internal calibration, so that must be available first.

Parameters

value	physical value to convert

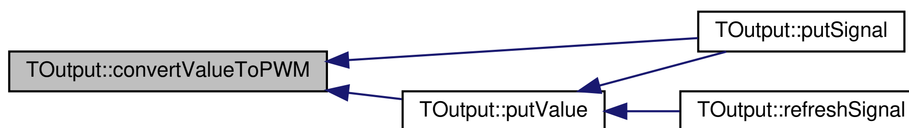
Returns

corresponding raw (pwm) value according to calibration

Todo check range

Here is the call graph for this function:

Here is the caller graph for this function:



5.5.4.3 getClass()

```
string TOutput::getClass ( ) const  [override], [protected], [virtual]
```

implementation of **TInterface::getClass()** (p. 28)

Returns a string unique for this class in order to group all objects of this class, especially for structuring config files or perhaps user interfaces

Returns

string "output"

Implements **TInterface** (p. 28).

Here is the caller graph for this function:

5.5.4.4 getCurrentPwm()

```
double TOutput::getCurrentPwm ( ) [private]
```

gets current raw output value

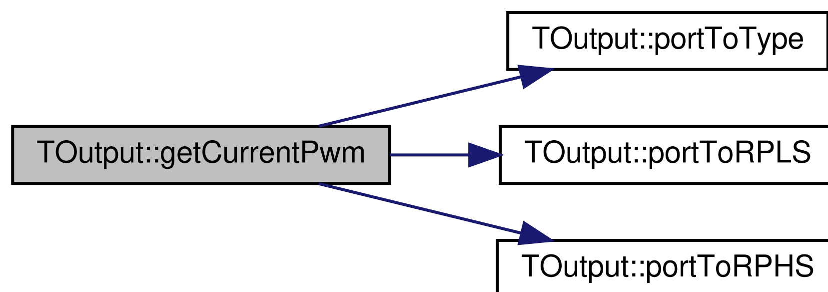
Class methods may want to know the raw value which is currently applied to the output port. Outside function should not need this.

Returns

currently applied raw output value

Todo verify that this is actually correct

Here is the call graph for this function:



5.5.4.5 getPortNameOptions()

```
string TOutput::getPortNameOptions ( ) [static], [private]
```

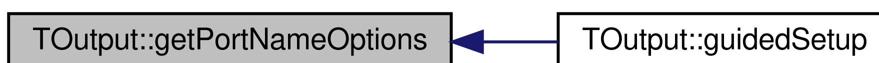
creates a human-readable string of port options

This is especially useful for **TOutput::guidedSetup** (p. 44) or other kinds of calibration, when a user is supposed to choose a port BY NAME.

Returns

string of comma-separated port names that exist for the current hardware

Here is the caller graph for this function:



5.5.4.6 getProperties()

```
map< string, string > TOutput::getProperties ( )  [override], [protected], [virtual]
```

implementation of **TInterface::getProperties** (p. 30)

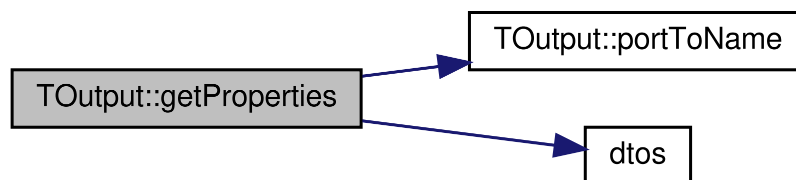
used by parent class, e.g. for configuration files or user interfaces

Returns

map associating property names with properties' values

Implements **TInterface** (p. 30).

Here is the call graph for this function:



5.5.4.7 getRangeMax()

```
double TOutput::getRangeMax ( )  [private]
```

get maximum value for raw output signal

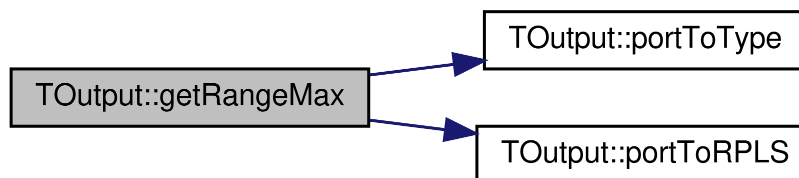
The maximum output voltage differs between slow and fast analog outputs.

Returns

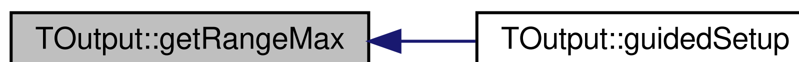
current port's maximum raw output signal

Todo magic number

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.4.8 getRangeMin()

```
double TOutput::getRangeMin ( )  [private]
```

get minimum value for raw output signal

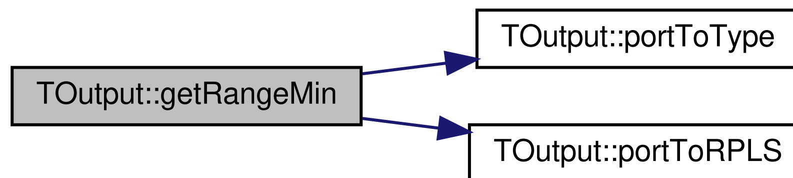
The minimum output voltage differs between slow and fast analog outputs.

Returns

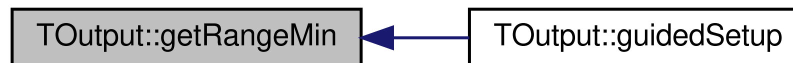
current port's minimum raw output signal

Todo magic number

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.4.9 guidedSetup()

```
void TOutput::guidedSetup (
            bool allow_name_change )  [override], [virtual]
```

implementation of **TInterface::guidedSetup()** (p. 31)

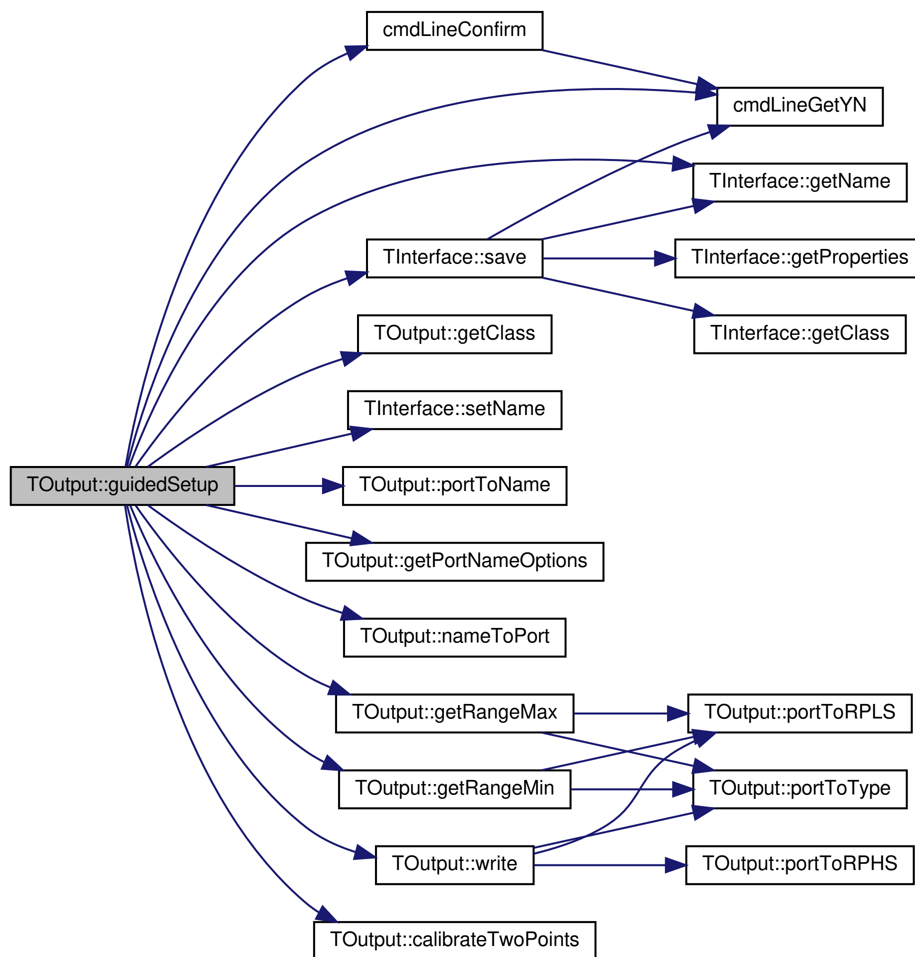
Walks the user through a guided setup and calibration. Starts a textual interface on the terminal to let the user setup the name and port and also provides a convenient wizard for calibration. Finally, the user may decide to save or discard alle changes. If the user is allowed to change the interfaces's name the config file will probably change its name, too, so an old config file may corrupt the config structure.

Parameters

allow_name_change	whether the user is allowed to change the interface's name

Implements **TInterface** (p. 31).

Here is the call graph for this function:



5.5.4.10 isSelfRefreshing()

```
bool TOutput::isSelfRefreshing ( )
```

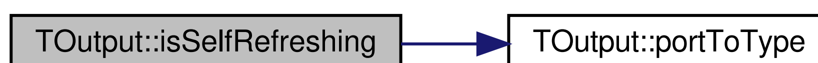
whether the signal runs without further support

This is generally true for fast outputs and false for slow outputs (see TType). If it is false, the calling code needs to loop through refreshSignal fast!

Returns

whether the signal runs without further support

Here is the call graph for this function:



5.5.4.11 nameToPort()

```
TOutput::TPort TOutput::nameToPort (
            string n )  [static], [private]
```

converts a readable string to an enumerated port value

This may be used for user interaction as well as reading configuration from human-readable config files. Throws an exception for invalid parameters.

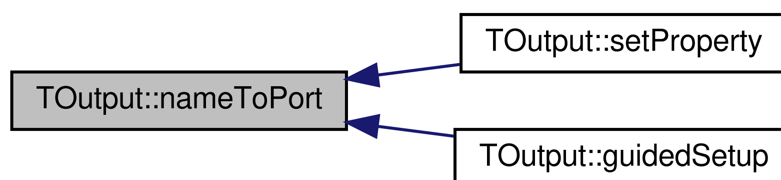
Parameters

n	textual representation of Red Pitaya's output port

Returns

Red Pitaya's output port (enumerated type) matching the specified textual representation

Here is the caller graph for this function:



5.5.4.12 portToName()

```
string TOutput::portToName (
            TPort p )  [static], [private]
```

converts an enumerated port value to a readable string

This may be used for user interaction as well as writing configuration to human-readable config files. Throws an exception for invalid parameters.

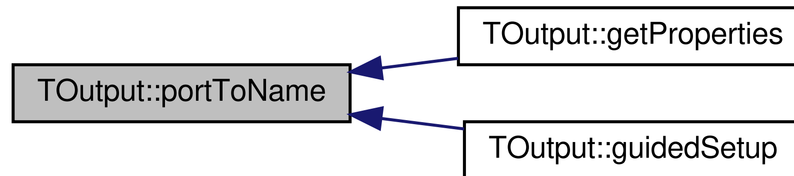
Parameters

p	Red Pitaya's output port (enumerated type)

Returns

textual representation of the specified port

Here is the caller graph for this function:



5.5.4.13 portToRPHS()

```
rp_channel_t TOutput::portToRPHS (
              TPort p )  [static], [private]
```

converts a port to RP API fast port

The RP API defines ports differently. See TPort for details. Throws an exception for invalid parameters.

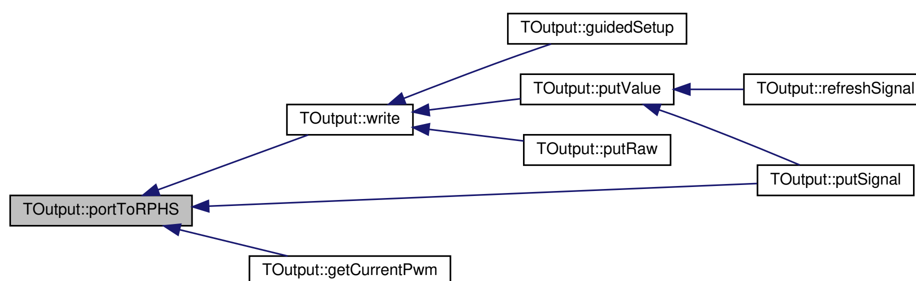
Parameters

p	port to convert

Returns

RP API fast port

Here is the caller graph for this function:



5.5.4.14 portToRPLS()

```
rp_apin_t TOutput::portToRPLS (
              TPort p )  [static], [private]
```

converts a port to RP API slow port

The RP API defines ports differently. See TPort for details. Throws an exception for invalid parameters.

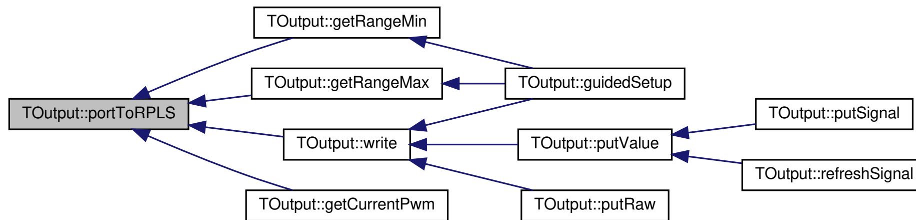
Parameters

p	port to convert

Returns

RP API slow port

Here is the caller graph for this function:



5.5.4.15 portToType()

```
TOutput::TType TOutput::portToType (
            TPort p )  [static], [private]
```

gets a port's type

See TPort and TType for further information.

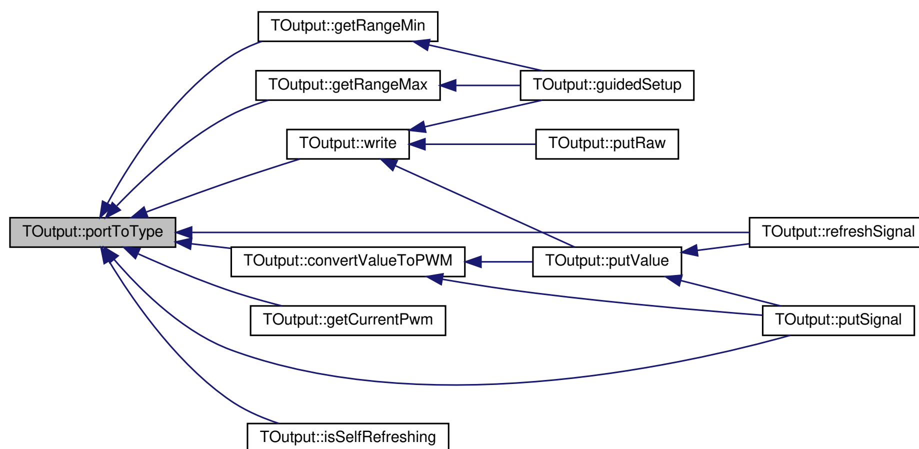
Parameters

p	port to get its type

Returns

port's type

Here is the caller graph for this function:



5.5.4.16 putRaw()

```
void TOutput::putRaw (
            double pwm )
```

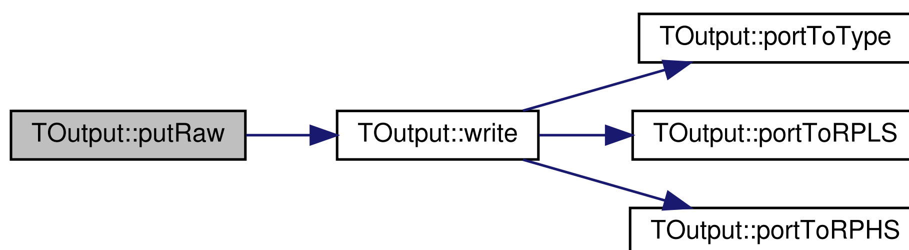
takes a raw value and writes it to output port

This method may or may not be discontinued later, as most external function would only need to access **TOutput::putValue()** (p. 50)

Parameters

pwm	raw (pwm) value for RedPitaya's output port

Here is the call graph for this function:



5.5.4.17 putSignal()

```
void TOutput::putSignal (
            double _frequency,
```

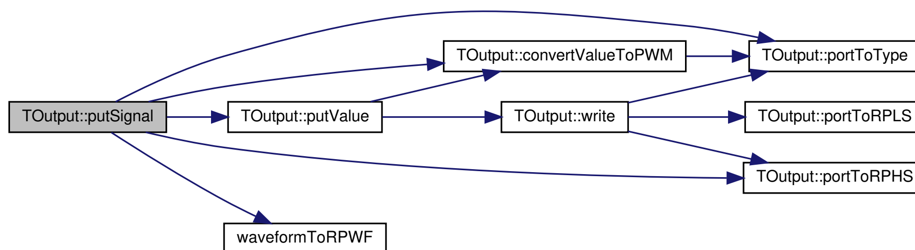
```
            double _amplitude,
             TWaveform _waveform )
```

generates signal on output port If the port is a fast port (TType::HS), then this function tells the built-in FPGA to create a signal. If it is a slow port (TType:LS), then this function will only initialize the necessary private variables, so that refreshSignal can update the signal and the calling code needs to loop through refreshSignal fast!

Parameters

_frequency	signal frequency in hertz
_amplitude	signal amplitude in volts or amps, according to calibration
_waveform	signal waveform

Here is the call graph for this function:



5.5.4.18 putValue()

```
void TOutput::putValue (
            double val )
```

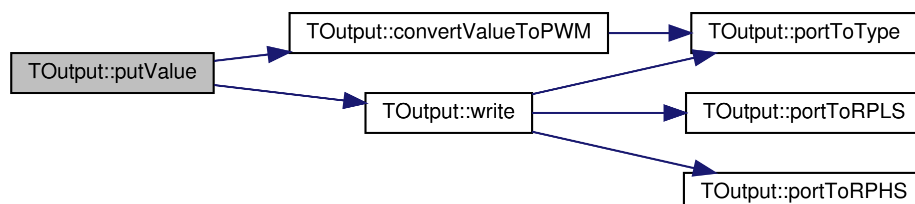
takes a raw value, applies calibration to get raw (pwm) value and writes it to output port

Conversion is based on the object's internal calibration, so that must be available first.

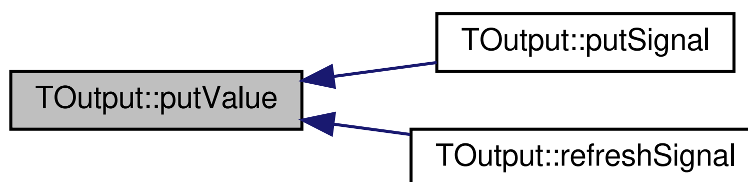
Parameters

val	physical value for RedPitaya's output port regarding the calibration

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.4.19 reconstructSignalValue()

```
double TOutput::reconstructSignalValue (
            double timestamp )
```

calculates signal value at point of time

If a signal is applied (see putSignal), this function calculates the signal's (calibrated, not raw) value at the specified point in time.

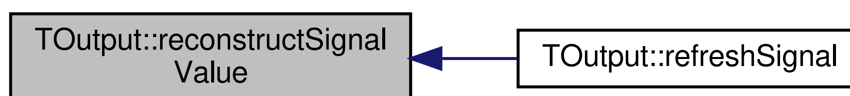
Parameters

timestamp	point in time

Returns

signal's value at specified point in time

Here is the caller graph for this function:

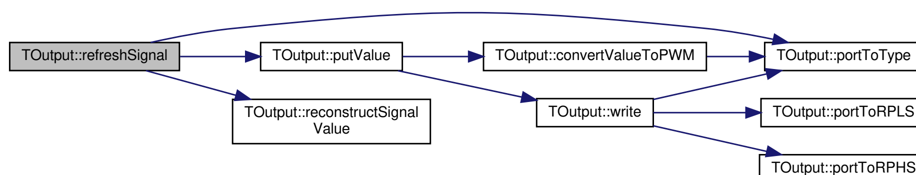


5.5.4.20 refreshSignal()

```
void TOutput::refreshSignal ( )
```

refreshes signal if needed

If the signal needs to be refreshed (see isSelfRefreshing), then this function does so by writing a new analog value to the output. This value will be constant temporarily until the next call of this method. Here is the call graph for this function:



5.5.4.21 setProperty()

```
void TOutput::setProperty (
            string id,
            string val )  [override], [protected], [virtual]
```

implementation of **TInterface::setProperty** (p. 34)

Sets the property with the specified name to the specified value. Specifically, all data other than string may be converted, so calling functions must apply to general standards

Parameters

id	textual name of a property to manipulate
val	textual representation of the data to set

Implements **TInterface** (p. 34).

Here is the call graph for this function:

5.5.4.22 write()

```
void TOutput::write (
            double pwm )  [private]
```

writes a raw value to the analog output

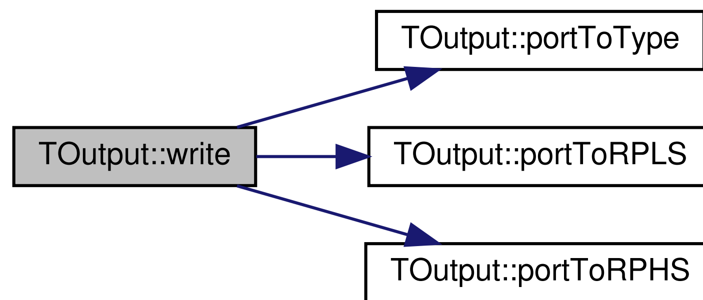
This function shall not be used by outside code as raw values are rather irrelevant, but class member functions may want to write raw values. See TInput::putValue() for access to input readings.

Parameters

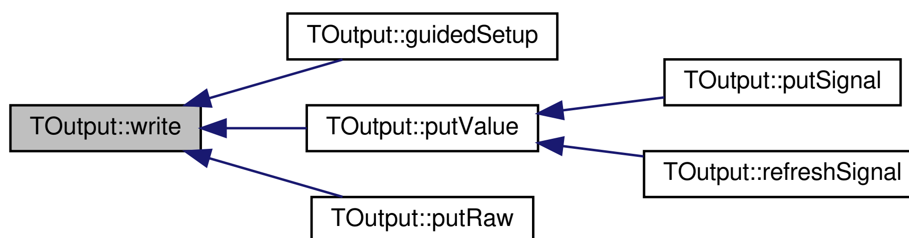
pwm	raw value to write to output

Todo use double instead of int?

Here is the call graph for this function:



Here is the caller graph for this function:



5.5.5 Member Data Documentation

5.5.5.1 port

`TPort TOutput::port [private]`

Red Pitaya's output port.

This must be a slow analog output. Fast analog outputs and digital outputs are not supported.

5.5.5.2 signal_amplitude

`double TOutput::signal_amplitude [private]`

signal amplitude

When creating a signal, e.g. with putSignal, this will hold the amplitude for later use.

5.5.5.3 signal_frequency

`double TOutput::signal_frequency [private]`

signal frequency

When creating a signal, e.g. with putSignal, this will hold the frequency for later use.

5.5.5.4 signal_started

```
high_resolution_clock::time_point TOutput::signal_started  [private]
```

signal starting point in time

When creating a signal, e.g. with putSignal, this will hold the information, when the signal was started, for later use.

5.5.5.5 signal_waveform

```
TWaveform TOutput::signal_waveform  [private]
```

signal waveform

When creating a signal, e.g. with putSignal, this will hold the waveform for later use.

5.5.5.6 slope

```
double TOutput::slope  [private]
```

physical value per raw value

This is part of the internal calibration and will automatically be set by the corresponding calibrating methods.

5.5.5.7 zero

```
double TOutput::zero  [private]
```

raw value that corresponds to physical value equal zero

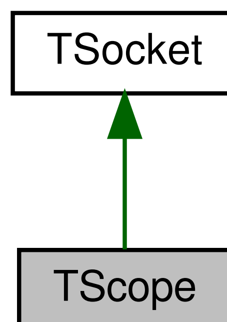
This is part of the internal calibration and will automatically be set by the corresponding calibrating methods.

5.6 TScope Struct Reference

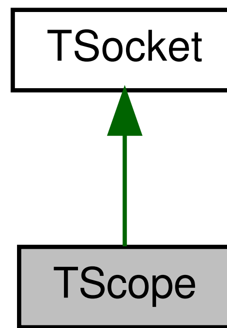
represents a connected scope (Tektronix TBS2000 series)

```
#include <scope.hpp>
```

Inheritance diagram for TScope:



Collaboration diagram for TScope:



Public Member Functions

- **TScope** (string remote_addr)

    constructor
- virtual ∼**TScope** ()

    destructor
- virtual string **execute** (string cmd, bool throw_error=true)

    executes a command
- vector< vector< double > > **measure** ()

    sets up trigger and takes measurement

Additional Inherited Members

5.6.1 Detailed Description

represents a connected scope (Tektronix TBS2000 series)

This class creates a connection to an available scope and offers methods to run custom commands or take an automated measurement. See `https://www.tek.com/oscilloscope/tbs2000-basic-oscilloscope-manual-0` for reference of executable commands

5.6.2 Constructor & Destructor Documentation

5.6.2.1 TScope()

```
TScope::TScope (
        string remote_addr )
```

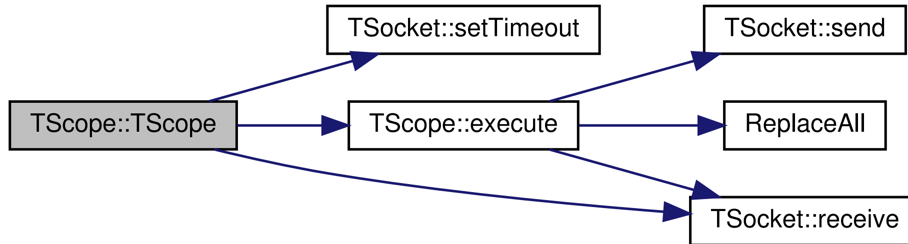
constructor

Opens connection to scope and sets timeout of **TSocket** (p. 58) and for Scope. Checks whether the remote device is actually the expected scope. Throws an exception if unsuccessful.

Parameters

remote_addr	string contaning the remote address, see **TSocket::TSocket** (p. 59) for details

Here is the call graph for this function:



5.6.2.2 ∼TScope()

```
TScope::∼TScope ( ) [virtual]
```

destructor

A virtual destructor is mandatory for polymorphic classes in order to achieve well-defined destruction behavior. It does not contain code, though.

5.6.3 Member Function Documentation

5.6.3.1 execute()

```
string TScope::execute (
            string cmd,
            bool throw_error = true ) [virtual]
```

executes a command

Sends the specified command and adds a line break. Receives scope's response and removes UI stuff (line break, cmd line "> "). If sanitized response is not empty and throw_error==true, then the function throws an exception. Otherwise, it returnes the sanitized response.

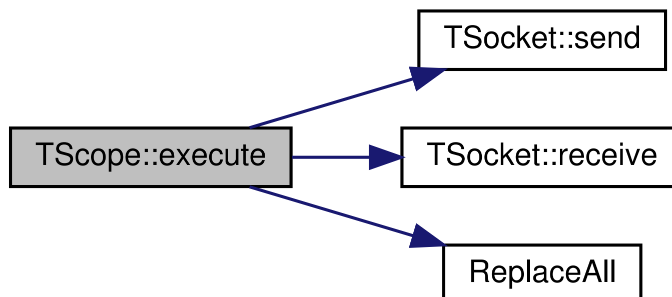
Parameters

cmd	command to execute, NOT including a line break
throw_error	whether to throw an exception on scope response

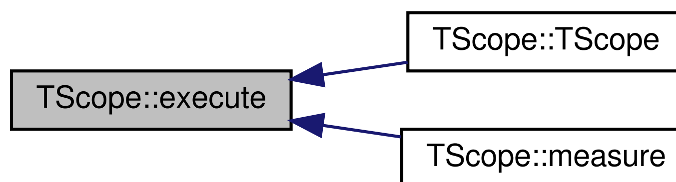
Returns

scope's response

Here is the call graph for this function:



Here is the caller graph for this function:



5.6.3.2 measure()

```
vector< vector< double > > TScope::measure ( )
```

sets up trigger and takes measurement

This method first sets the scope's acquisition mode to "single" and waits for trigger (manual trigger works, too). It then reads the time points and measured values over the entire measure span (roughly scope screen plus a bit) and returns them as {t,CH1,CH2,CH3,CH4}, where each element is a vector of roughly 2001 elements. Throws an error, if scope is not triggered.

Returns

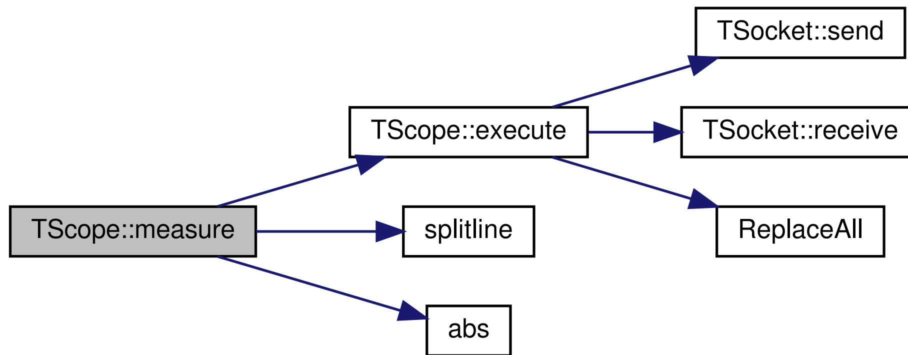
{t,CH1,CH2,CH3,CH4}, roughly 2001 values each

Todo get smallest and largest scope values automatically

Todo check preamble, whether channel is actually activaed

Todo maybe activate every channel beforehand?

Here is the call graph for this function:

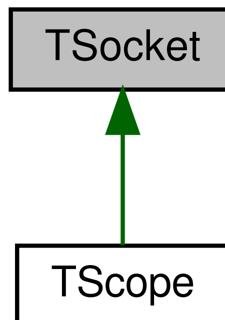


5.7 TSocket Class Reference

makes use of Linux' socket libary and provides object-oriented access

```
#include <scope.hpp>
```

Inheritance diagram for TSocket:



Public Member Functions

- **TSocket** (string remote_addr, int port)
  constructor
- virtual ∼**TSocket** ()
  destructor
- virtual void **send** (string msg)
  sends a message string
- virtual string **receive** ()
  receives a message string
- void **setTimeout** (size_t ms)
  sets timout

Private Attributes

- int **tosocket**

    handle of the socket
- struct sockaddr_in **toaddr**

    remote address data
- char **inbuf** [**BUF_SIZE**]

    input buffer
- size_t **timeout**

    timeout in milliseconds for receiving data

Static Private Attributes

- static const size_t **BUF_SIZE** = 4096

    input buffer size

5.7.1 Detailed Description

makes use of Linux' socket libary and provides object-oriented access

This class creates a socket-based TCP connection on creation and closes it on destruction. It offers only basic read and write functionality. It only support ASCII characters. See `http://heim.ifi.uio.no/michawe//teaching/cn-alt/unterlagen/csockets/index.html` for socket tutorial

Todo support UTF-8

5.7.2 Constructor & Destructor Documentation

5.7.2.1 TSocket()

```
TSocket::TSocket (
            string remote_addr,
            int port )
```

constructor

Connects to remote address via TCP immediately. If no connection can be established, an exception will be thrown. Note, that there is no hostname resolution, so only four byte IPv4 addresses can be specified.

Todo implement IPv6

implement hostname resolution

Parameters

remote_addr	remote address as IPv4 address
port	port to connect to

5.7.2.2 ∼TSocket()

```
TSocket::∼TSocket ( )  [virtual]
```

destructor

Closes the connection regardless of pending packets. Make sure to call destructor to not litter lingering connections.

5.7.3 Member Function Documentation

5.7.3.1 receive()

```
string TSocket::receive ( )  [virtual]
```

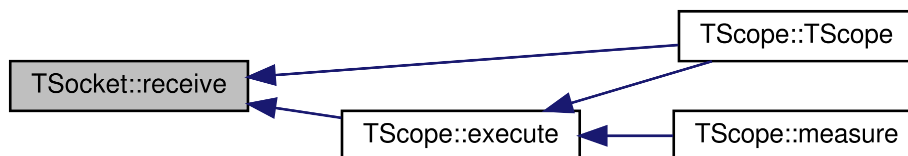
receives a message string

If there are ANY data in the OS's buffer, those will be read and returned immediately. If not, this method will retry until timout or until any data arrive. This method sanitizes messages with missing null terminators.

Returns

received message string

Here is the caller graph for this function:



5.7.3.2 send()

```
void TSocket::send (
            string msg )  [virtual]
```

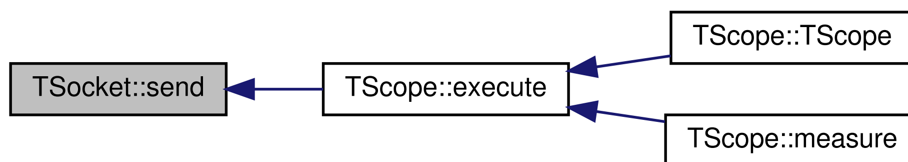
sends a message string

Throws an exception if unsuccessful.

Parameters

msg	message to be send

Here is the caller graph for this function:



5.7.3.3 setTimeout()

```
void TSocket::setTimeout (
            size_t ms )
```

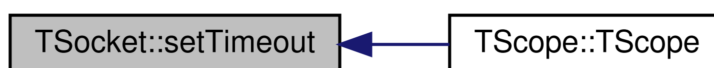
sets timout

This timeout only applies to receiving messages. Sent messages are passed to the Linux kernel and handled by that.

Parameters

ms	timout in milliseconds

Here is the caller graph for this function:



5.7.4 Member Data Documentation

5.7.4.1 BUF_SIZE

```
const size_t TSocket::BUF_SIZE = 4096 [static], [private]
```

input buffer size

4096 seems to be a good estimate. Too high values may corrupt stack, too low values result in fragmented packets.

5.7.4.2 inbuf

```
char TSocket::inbuf[ BUF_SIZE] [private]
```

input buffer

This is only used internally to access linux's socket api. see BUF_SIZE for further details.

5.7.4.3 timeout

```
size_t TSocket::timeout [private]
```

timeout in milliseconds for receiving data

See **TSocket::receive** (p. 60) for further details.

5.7.4.4 toaddr

```
struct sockaddr_in TSocket::toaddr [private]
```

remote address data

This is only used internally to access linux's socket api.

5.7.4.5 tosocket

```
int TSocket::tosocket [private]
```

handle of the socket

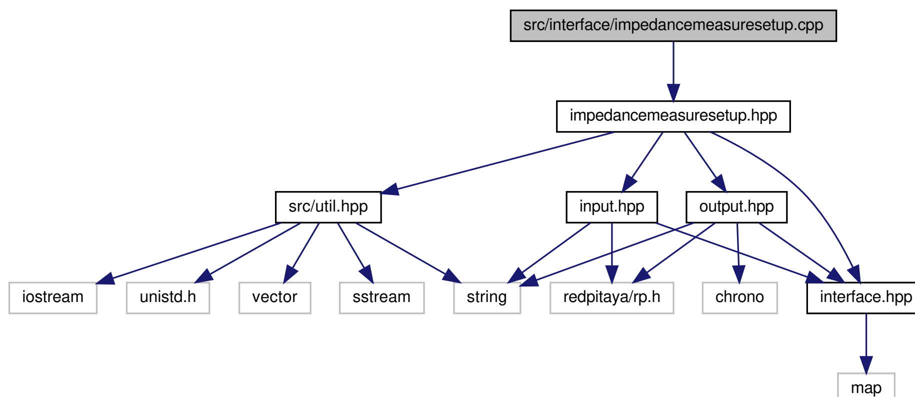
This is only used internally to access linux's socket api.

6 File Documentation

6.1 src/interface/impedancemeasuresetup.cpp File Reference

```
#include "impedancemeasuresetup.hpp"
```

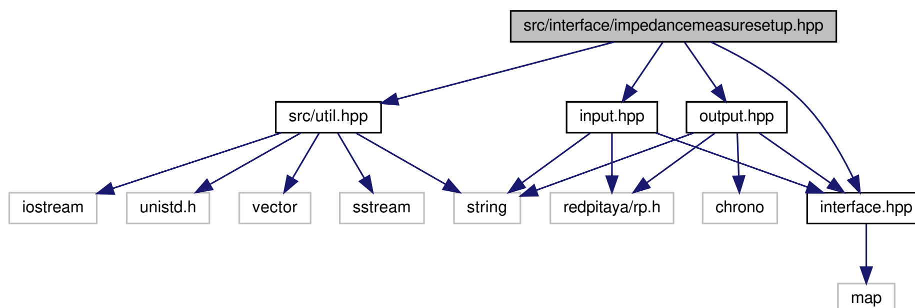
Include dependency graph for impedancemeasuresetup.cpp:



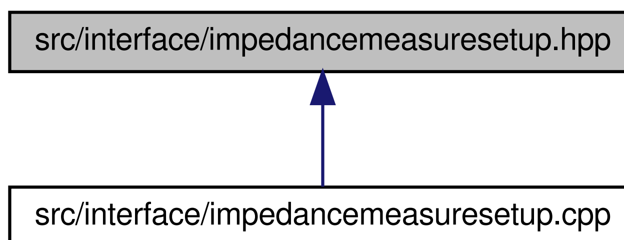
6.2 src/interface/impedancemeasuresetup.hpp File Reference

```
#include "src/util.hpp"
#include "interface.hpp"
#include "input.hpp"
#include "output.hpp"
```

Include dependency graph for impedancemeasuresetup.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class **TImpedanceMeasureSetup**

  represents an output, one voltage input and one current input

Enumerations

- enum **TImpedanceMeasureSetupOutputType** { **voltage**, **current** }

  enumerates output types.

6.2.1 Enumeration Type Documentation

6.2.1.1 TImpedanceMeasureSetupOutputType

`enum` **`TImpedanceMeasureSetupOutputType`**

enumerates output types.

This is not used in any of the tools so far and shall be reviewd before use.

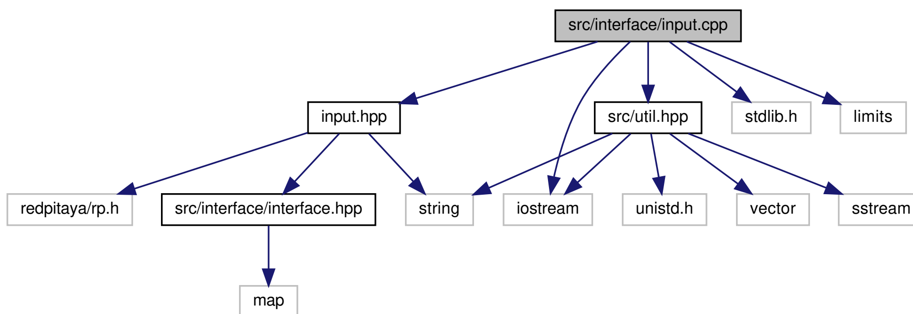
Enumerator

voltage	
current	

6.3 src/interface/input.cpp File Reference

```
#include <iostream>
#include <stdlib.h>
#include <limits>
#include "input.hpp"
#include "src/util.hpp"
```

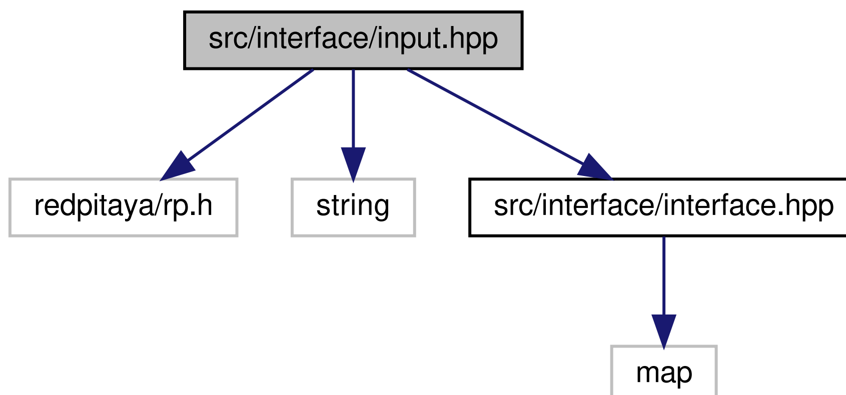
Include dependency graph for input.cpp:



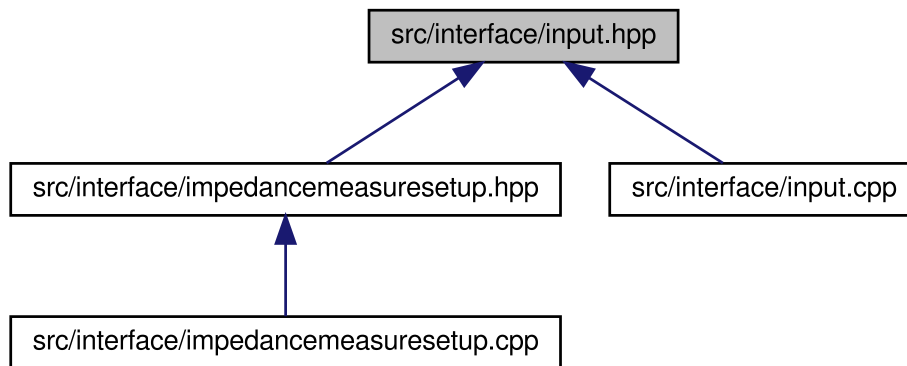
6.4 src/interface/input.hpp File Reference

```
#include "redpitaya/rp.h"
#include <string>
#include "src/interface/interface.hpp"
```

Include dependency graph for input.hpp:



This graph shows which files directly or indirectly include this file:



Classes

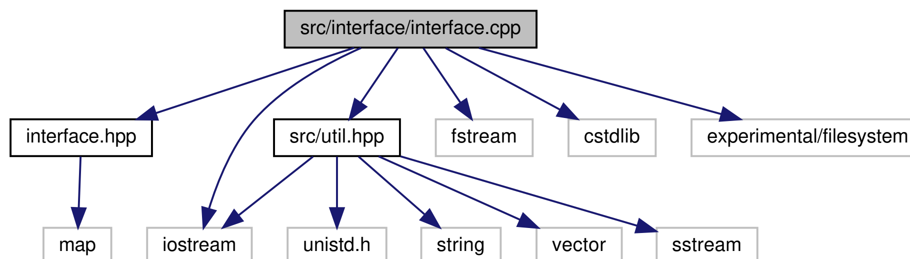
- class **TInput**

  represents an analog input including calibration.

6.5 src/interface/interface.cpp File Reference

```
#include "interface.hpp"
#include "src/util.hpp"
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <experimental/filesystem>
```

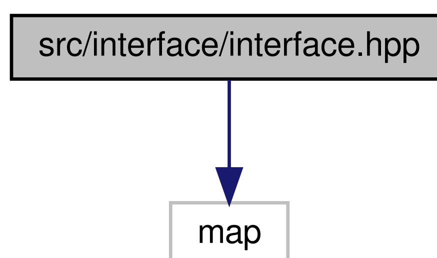
Include dependency graph for interface.cpp:



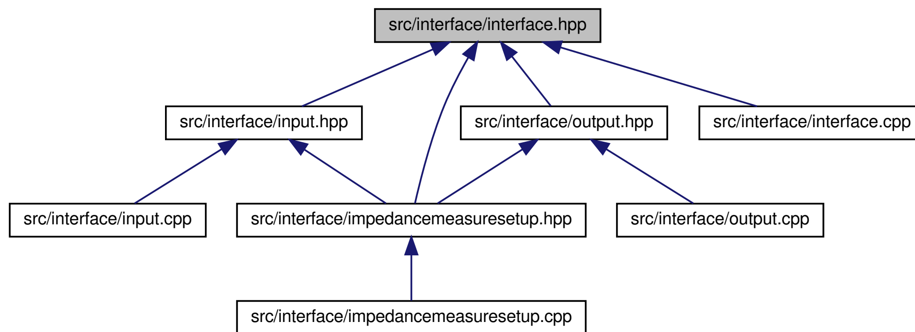
6.6 src/interface/interface.hpp File Reference

```
#include <map>
```

Include dependency graph for interface.hpp:



This graph shows which files directly or indirectly include this file:



Classes

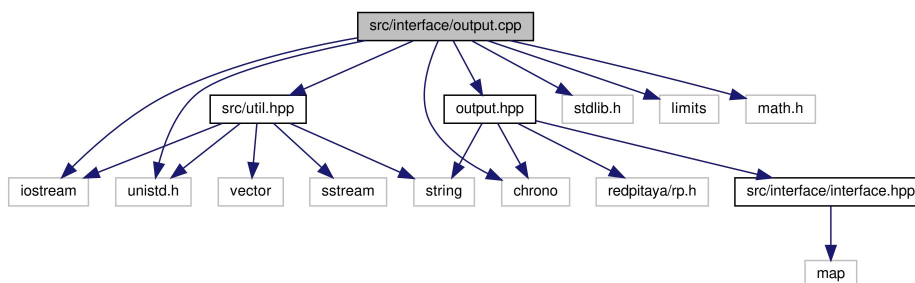
- class **TInterface**

  abstract representation of an input or output or complex hardware interface

6.7 src/interface/output.cpp File Reference

```
#include <iostream>
#include <stdlib.h>
#include <limits>
#include <unistd.h>
#include <chrono>
#include <math.h>
#include "output.hpp"
#include "src/util.hpp"
```

Include dependency graph for output.cpp:



Functions

- string **waveformToString** (**TWaveform** wf)

  converts a waveform type to a string
- **TWaveform** **stringToWaveform** (string s)

  converts a string to a waveform type
- rp_waveform_t **waveformToRPWF** (**TWaveform** wf)

  converts a waveform type to the RedPitaya's waveform type

6.7.1 Function Documentation

6.7.1.1 stringToWaveform()

```
TWaveform stringToWaveform (
            string s )
```

converts a string to a waveform type

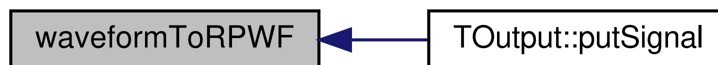
see waveformToString for details

6.7.1.2 waveformToRPWF()

```
rp_waveform_t waveformToRPWF (
            TWaveform wf )
```

converts a waveform type to the RedPitaya's waveform type

The RePitaya defines its own waveform types. See rp.h for details Here is the caller graph for this function:



6.7.1.3 waveformToString()

```
string waveformToString (
            TWaveform wf )
```

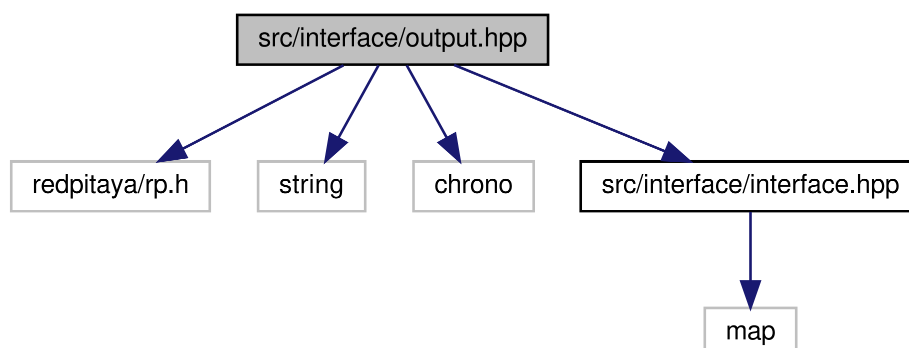
converts a waveform type to a string

This is supposed to be human-readable, e.g. wfSin is "sin"

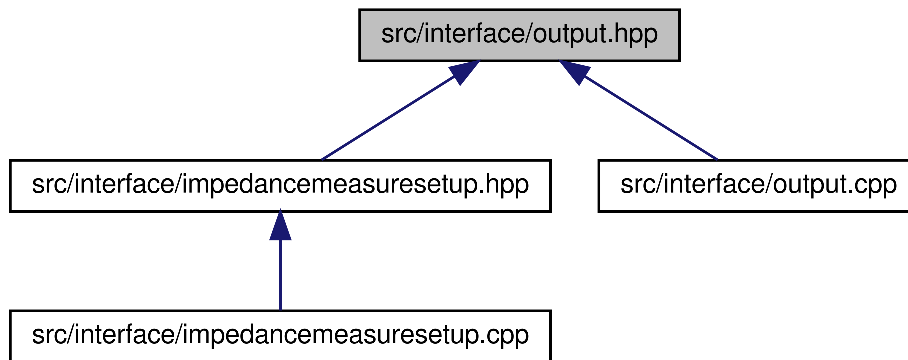
6.8 src/interface/output.hpp File Reference

```
#include "redpitaya/rp.h"
#include <string>
#include <chrono>
#include "src/interface/interface.hpp"
```

Include dependency graph for output.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class **TOutput**

  represents an analog output including calibration

Enumerations

- enum **TWaveform** { **wfDc**, **wfSin**, **wfCos**, **wfSquare** }

  enumerates different waveform types

Functions

- string **waveformToString** (**TWaveform** wf)

  converts a waveform type to a string
- **TWaveform** **stringToWaveform** (string s)

  converts a string to a waveform type
- rp_waveform_t **waveformToRPWF** (**TWaveform** wf)

  converts a waveform type to the RedPitaya's waveform type

6.8.1 Enumeration Type Documentation

6.8.1.1 TWaveform

`enum TWaveform`

enumerates different waveform types

Enumerator

wfDc	
wfSin	
wfCos	
wfSquare	

6.8.2 Function Documentation

6.8.2.1 stringToWaveform()

```
TWaveform stringToWaveform (
          string s )
```

converts a string to a waveform type

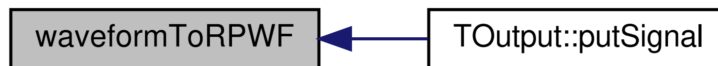
see waveformToString for details

6.8.2.2 waveformToRPWF()

```
rp_waveform_t waveformToRPWF (
          TWaveform wf )
```

converts a waveform type to the RedPitaya's waveform type

The RePitaya defines its own waveform types. See rp.h for details Here is the caller graph for this function:



6.8.2.3 waveformToString()

```
string waveformToString (
          TWaveform wf )
```

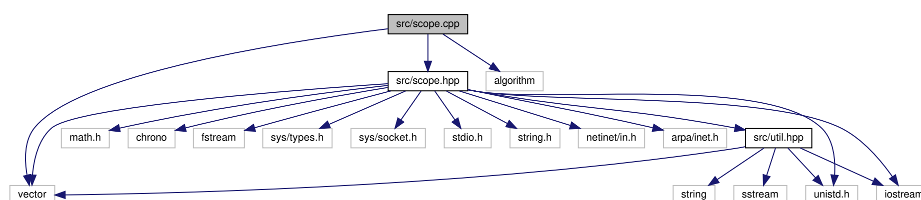
converts a waveform type to a string

This is supposed to be human-readable, e.g. wfSin is "sin"

6.9 src/scope.cpp File Reference

```
#include "src/scope.hpp"
#include <vector>
#include <algorithm>
```

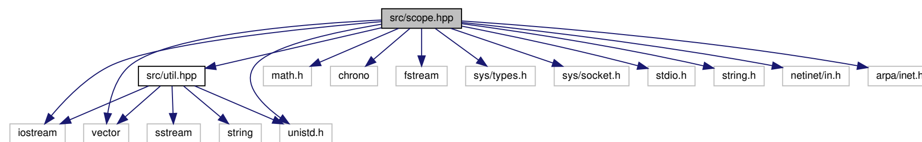
Include dependency graph for scope.cpp:



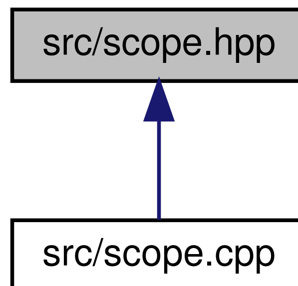
6.10 src/scope.hpp File Reference

```
#include <iostream>
#include <math.h>
#include <chrono>
#include <fstream>
#include <vector>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include "src/util.hpp"
```

Include dependency graph for scope.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class **TSocket**

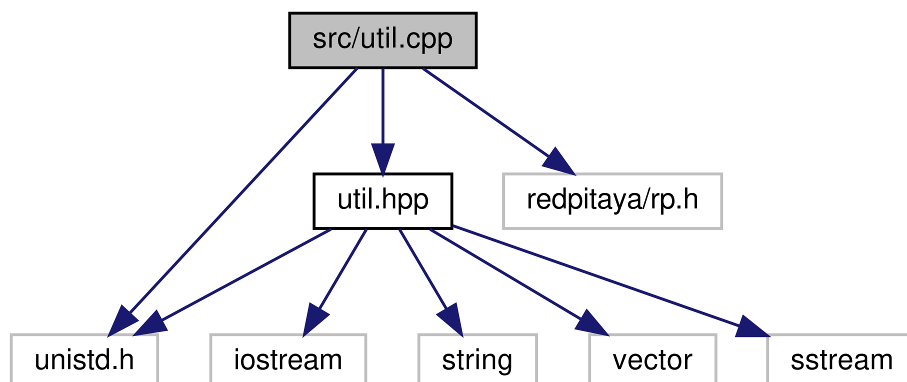
  makes use of Linux' socket libary and provides object-oriented access
- struct **TScope**

  represents a connected scope (Tektronix TBS2000 series)

6.11 src/util.cpp File Reference

```
#include "util.hpp"
#include <unistd.h>
```

```
#include "redpitaya/rp.h"
```
Include dependency graph for util.cpp:



Functions

- bool **cmdLineGetYN** (string question)
- bool **cmdLineConfirm** ()
- string **dtos** (double val)
- void **initHardwareAccess** ()
- std::string **ReplaceAll** (std::string str, const std::string &from, const std::string &to)
- vector< string > **splitline** (string line, string delimiter)
- double **abs** (double a)

Variables

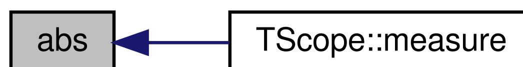
- bool **verbose** =false

6.11.1 Function Documentation

6.11.1.1 abs()

```
double abs (
            double a )
```

Here is the caller graph for this function:

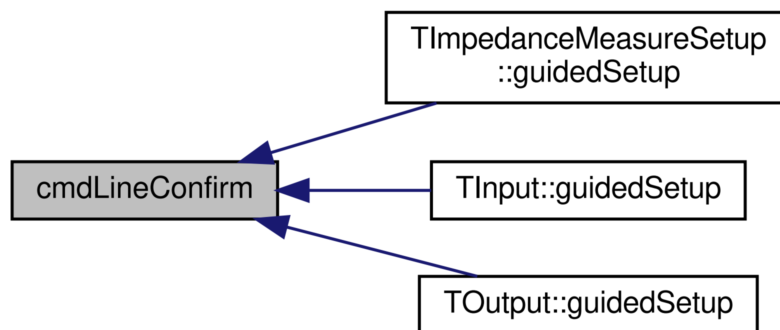


6.11.1.2 cmdLineConfirm()

```
bool cmdLineConfirm ( )
```

Here is the call graph for this function:

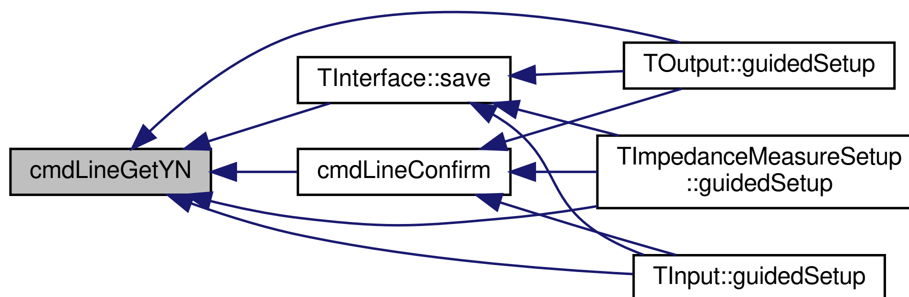
Here is the caller graph for this function:



6.11.1.3 cmdLineGetYN()

```
bool cmdLineGetYN (
            string question )
```

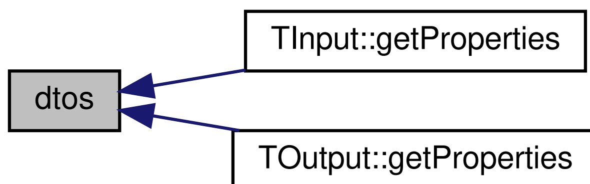
Here is the caller graph for this function:



6.11.1.4 dtos()

```
string dtos (
            double val )
```

Here is the caller graph for this function:



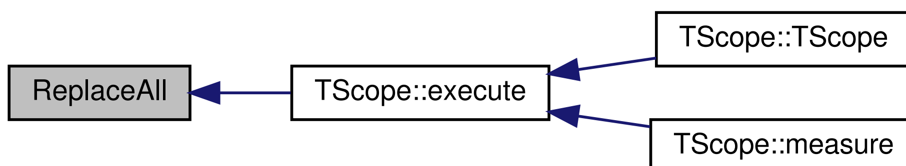
6.11.1.5 initHardwareAccess()

```
void initHardwareAccess ( )
```

6.11.1.6 ReplaceAll()

```
std::string ReplaceAll (
            std::string str,
            const std::string & from,
            const std::string & to )
```

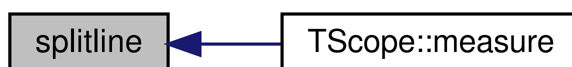
Here is the caller graph for this function:



6.11.1.7 splitline()

```
vector<string> splitline (
            string line,
            string delimiter )
```

Here is the caller graph for this function:



6.11.2 Variable Documentation

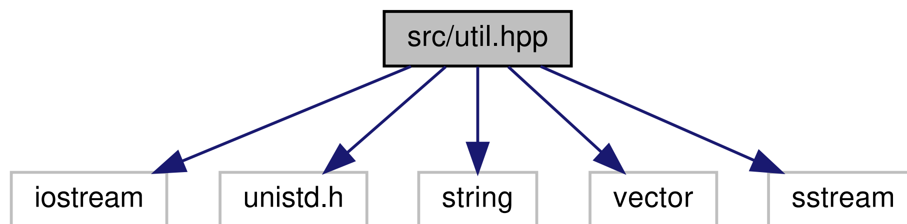
6.11.2.1 verbose

```
bool verbose =false
```

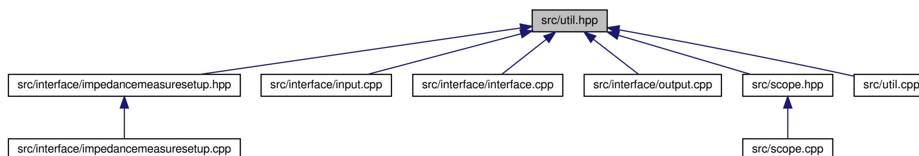
6.12 src/util.hpp File Reference

```
#include <iostream>
#include <unistd.h>
#include <string>
#include <vector>
#include <sstream>
```

Include dependency graph for util.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class **TException**

Macros

- #define **NOT_IMPLEMENTED**() throw **TException**(string("The function named ")+string(__PRETTY_FUNCTION__)+string(" is not yet (fully) implemented."))
- #define **UNREACHABLE**() throw **TException**(string("The function named ")+string(__PRETTY_FUNCTION__)+string(" reached code that is supposed to be unreachable."))
- #define **TODO**(msg) throw **TException**(string("TODO: ")+string(msg))
- #define **ERROR**(msg) throw **TException**(string("In function: ")+string(__PRETTY_FUNCTION__)+string(":\n")+string(msg))

Functions

- bool **cmdLineGetYN** (string question)
- bool **cmdLineConfirm** ()
- string **dtos** (double val)
- void **initHardwareAccess** ()
- std::string **ReplaceAll** (std::string str, const std::string &from, const std::string &to)
- vector< string > **splitline** (string line, string delimiter)
- double **abs** (double a)

Variables

- bool **verbose**

6.12.1 Macro Definition Documentation

6.12.1.1 ERROR

```
#define ERROR(
              msg ) throw  TException(string("In function:  ")+string(__PRETTY_FUNCTION__↩
)+string(" :\n")+string(msg))
```

6.12.1.2 NOT_IMPLEMENTED

```
#define NOT_IMPLEMENTED( ) throw  TException(string("The function named ")+string(__PRETTY_F↩
UNCTION__)+string(" is not yet (fully) implemented."))
```

6.12.1.3 TODO

```
#define TODO(
              msg ) throw  TException(string("TODO: ")+string(msg))
```

6.12.1.4 UNREACHABLE

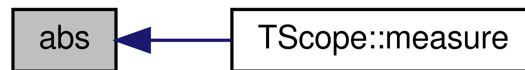
```
#define UNREACHABLE( ) throw  TException(string("The function named ")+string(__PRETTY_FUNCT↩
ION__)+string(" reached code that is supposed to be unreachable."))
```

6.12.2 Function Documentation

6.12.2.1 abs()

```
double abs (
            double a )
```

Here is the caller graph for this function:

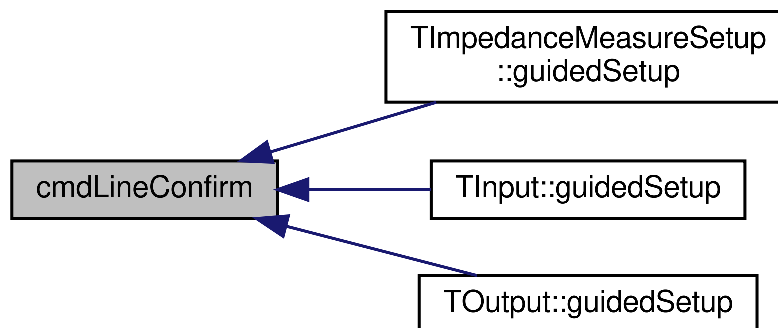


6.12.2.2 cmdLineConfirm()

```
bool cmdLineConfirm ( )
```

Here is the call graph for this function:

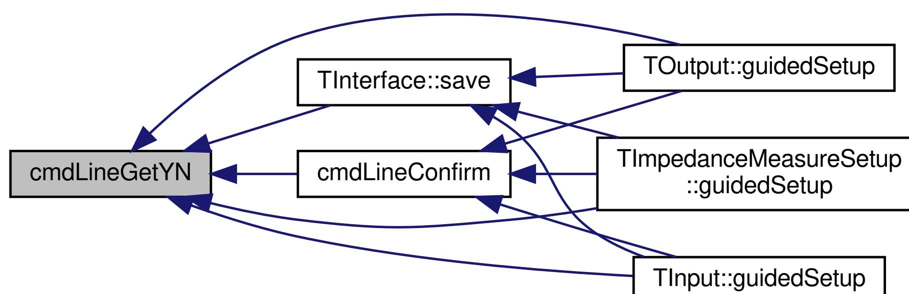
Here is the caller graph for this function:



6.12.2.3 cmdLineGetYN()

```
bool cmdLineGetYN (
            string question )
```

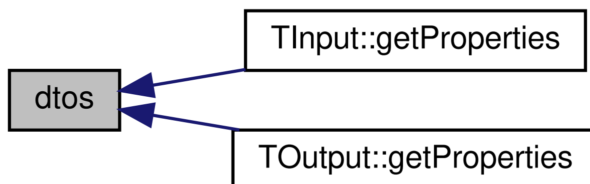
Here is the caller graph for this function:



6.12.2.4 dtos()

```
string dtos (
            double val )
```

Here is the caller graph for this function:



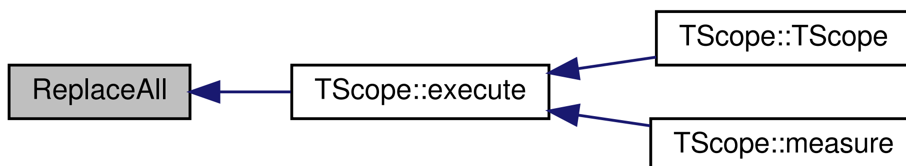
6.12.2.5 initHardwareAccess()

```
void initHardwareAccess ( )
```

6.12.2.6 ReplaceAll()

```
std::string ReplaceAll (
            std::string str,
            const std::string & from,
            const std::string & to )
```

Here is the caller graph for this function:



6.12.2.7 splitline()

```
vector<string> splitline (
            string line,
            string delimiter )
```

Here is the caller graph for this function:

6.12.3 Variable Documentation

6.12.3.1 verbose

```
bool verbose
```

Index

~TImpedanceMeasureSetup
    TImpedanceMeasureSetup, 8
~TInput
    TInput, 19
~TInterface
    TInterface, 28
~TOutput
    TOutput, 39
~TScope
    TScope, 56
~TSocket
    TSocket, 60

abs
    util.cpp, 71
    util.hpp, 75

BUF_SIZE
    TSocket, 61

CONFIG_DELIM
    TInterface, 35
CONFIG_DIR
    TInterface, 35
calibrateTwoPoints
    TInput, 19
    TOutput, 40
calibrateValue
    TInput, 20
calibrateZero
    TInput, 20
cmdLineConfirm
    util.cpp, 71
    util.hpp, 76
cmdLineGetYN
    util.cpp, 72
    util.hpp, 76
convertMeasurementToValue
    TInput, 20
convertValueToPWM
    TOutput, 40

dtos
    util.cpp, 72
    util.hpp, 76

ERROR
    util.hpp, 75
execute
    TScope, 56

getClass
    TImpedanceMeasureSetup, 8
    TInput, 21
    TInterface, 28
    TOutput, 41
getCurrentPwm
    TOutput, 41
getInputIName
    TImpedanceMeasureSetup, 9
getInputUName
    TImpedanceMeasureSetup, 9
getInputI
    TImpedanceMeasureSetup, 8
getInputU
    TImpedanceMeasureSetup, 9
getMsg
    TException, 4
getName
    TInterface, 29
getOutput
    TImpedanceMeasureSetup, 10
getOutputName
    TImpedanceMeasureSetup, 10
getOutputType
    TImpedanceMeasureSetup, 10
getPortNameOptions
    TOutput, 42
getProperties
    TImpedanceMeasureSetup, 11
    TInput, 21
    TInterface, 30
    TOutput, 42
getRangeMax
    TOutput, 43
getRangeMin
    TOutput, 43
guidedSetup
    TImpedanceMeasureSetup, 11
    TInput, 22
    TInterface, 31
    TOutput, 44

impedancemeasuresetup.hpp
    TImpedanceMeasureSetupOutputType, 63
inbuf
    TSocket, 61
initHardwareAccess
    util.cpp, 73
    util.hpp, 77
input_i
    TImpedanceMeasureSetup, 16
input_u
    TImpedanceMeasureSetup, 16
isSelfRefreshing
    TOutput, 45

load
    TInterface, 31, 32

measure
    TScope, 57
measureValue

TInput, 23
msg
TException, 5

NOT_IMPLEMENTED
util.hpp, 75
name
TInterface, 35
nameToOutputType
TImpedanceMeasureSetup, 12
nameToPort
TInput, 23
TOutput, 45

output
TImpedanceMeasureSetup, 16
output.cpp
stringToWaveform, 67
waveformToRPWF, 67
waveformToString, 67
output.hpp
stringToWaveform, 69
TWaveform, 68
waveformToRPWF, 69
waveformToString, 69
output_type
TImpedanceMeasureSetup, 16
outputTypeToName
TImpedanceMeasureSetup, 13

port
TInput, 26
TOutput, 53
portToName
TInput, 24
TOutput, 46
portToRPHS
TOutput, 47
portToRPLS
TOutput, 47
portToType
TOutput, 48
putRaw
TOutput, 49
putSignal
TOutput, 49
putValue
TOutput, 50

read
TInput, 25
receive
TSocket, 60
reconstructSignalValue
TOutput, 51
refreshSignal
TOutput, 51
ReplaceAll
util.cpp, 73
util.hpp, 77

save
TInterface, 33
send
TSocket, 60
setInputIByName
TImpedanceMeasureSetup, 13
setInputUByName
TImpedanceMeasureSetup, 14
setInputI
TImpedanceMeasureSetup, 13
setInputU
TImpedanceMeasureSetup, 14
setName
TInterface, 33
setOutput
TImpedanceMeasureSetup, 14
setOutputByName
TImpedanceMeasureSetup, 15
setOutputType
TImpedanceMeasureSetup, 15
setProperty
TImpedanceMeasureSetup, 15
TInput, 25
TInterface, 34
TOutput, 52
setTimeout
TSocket, 61
show
TException, 4
signal_amplitude
TOutput, 53
signal_frequency
TOutput, 53
signal_started
TOutput, 53
signal_waveform
TOutput, 54
slope
TInput, 26
TOutput, 54
splitline
util.cpp, 73
util.hpp, 77
src/interface/impedancemeasuresetup.cpp, 62
src/interface/impedancemeasuresetup.hpp, 63
src/interface/input.cpp, 64
src/interface/input.hpp, 64
src/interface/interface.cpp, 65
src/interface/interface.hpp, 65
src/interface/output.cpp, 66
src/interface/output.hpp, 67
src/scope.cpp, 69
src/scope.hpp, 70
src/util.cpp, 70
src/util.hpp, 74
stringToWaveform
output.cpp, 67

output.hpp, 69

TException, 4
getMsg, 4
msg, 5
show, 4
TException, 4
TImpedanceMeasureSetup, 5
∼TImpedanceMeasureSetup, 8
getClass, 8
getInputIName, 9
getInputUName, 9
getInputI, 8
getInputU, 9
getOutput, 10
getOutputName, 10
getOutputType, 10
getProperties, 11
guidedSetup, 11
input_i, 16
input_u, 16
nameToOutputType, 12
output, 16
output_type, 16
outputTypeToName, 13
setInputIByName, 13
setInputUByName, 14
setInputI, 13
setInputU, 14
setOutput, 14
setOutputByName, 15
setOutputType, 15
setProperty, 15
TImpedanceMeasureSetup, 7
TImpedanceMeasureSetupOutputType
impedancemeasuresetup.hpp, 63
TInput, 17
∼TInput, 19
calibrateTwoPoints, 19
calibrateValue, 20
calibrateZero, 20
convertMeasurementToValue, 20
getClass, 21
getProperties, 21
guidedSetup, 22
measureValue, 23
nameToPort, 23
port, 26
portToName, 24
read, 25
setProperty, 25
slope, 26
TInput, 18
zero, 26
TInterface, 26
∼TInterface, 28
CONFIG_DELIM, 35
CONFIG_DIR, 35
getClass, 28
getName, 29
getProperties, 30
guidedSetup, 31
load, 31, 32
name, 35
save, 33
setName, 33
setProperty, 34
TInterface, 28
TODO
util.hpp, 75
TOutput, 36
∼TOutput, 39
calibrateTwoPoints, 40
convertValueToPWM, 40
getClass, 41
getCurrentPwm, 41
getPortNameOptions, 42
getProperties, 42
getRangeMax, 43
getRangeMin, 43
guidedSetup, 44
isSelfRefreshing, 45
nameToPort, 45
port, 53
portToName, 46
portToRPHS, 47
portToRPLS, 47
portToType, 48
putRaw, 49
putSignal, 49
putValue, 50
reconstructSignalValue, 51
refreshSignal, 51
setProperty, 52
signal_amplitude, 53
signal_frequency, 53
signal_started, 53
signal_waveform, 54
slope, 54
TOutput, 39
TPort, 38
TType, 39
write, 52
zero, 54
TPort
TOutput, 38
TScope, 54
∼TScope, 56
execute, 56
measure, 57
TScope, 55
TSocket, 58
∼TSocket, 60
BUF_SIZE, 61
inbuf, 61
receive, 60
send, 60

setTimeout, 61
TSocket, 59
timeout, 61
toaddr, 62
tosocket, 62
TType
TOutput, 39
TWaveform
output.hpp, 68
timeout
TSocket, 61
toaddr
TSocket, 62
tosocket
TSocket, 62

UNREACHABLE
util.hpp, 75
util.cpp
abs, 71
cmdLineConfirm, 71
cmdLineGetYN, 72
dtos, 72
initHardwareAccess, 73
ReplaceAll, 73
splitline, 73
verbose, 74
util.hpp
abs, 75
cmdLineConfirm, 76
cmdLineGetYN, 76
dtos, 76
ERROR, 75
initHardwareAccess, 77
NOT_IMPLEMENTED, 75
ReplaceAll, 77
splitline, 77
TODO, 75
UNREACHABLE, 75
verbose, 77

verbose
util.cpp, 74
util.hpp, 77

waveformToRPWF
output.cpp, 67
output.hpp, 69
waveformToString
output.cpp, 67
output.hpp, 69
write
TOutput, 52

zero
TInput, 26
TOutput, 54

Handbook OrthoSim

Generated by Doxygen 1.8.13

Contents

1 Todo List 1

2 Hierarchical Index 2

2.1 Class Hierarchy 2

3 Class Index 4

3.1 Class List 4

4 File Index 6

4.1 File List 6

5 Class Documentation 8

5.1 TBitmap Class Reference 8

5.1.1 Detailed Description 9

5.1.2 Constructor & Destructor Documentation 9

5.2 TConstDouble< numerator, denominator > Class Template Reference 10

5.3 TElement Class Reference 10

5.3.1 Detailed Description 11

5.3.2 Member Function Documentation 11

5.3.3 Member Data Documentation 14

5.4 TEquationSystem< T > Class Template Reference 15

5.4.1 Member Function Documentation 16

5.5 TEquationSystemLU< T > Class Template Reference 16

5.6 TException Class Reference 17

5.6.1 Detailed Description 17

5.6.2 Constructor & Destructor Documentation 17

5.7 TFdm Class Reference 18

5.7.1 Detailed Description 20

5.7.2 Member Enumeration Documentation 20

5.7.3 Constructor & Destructor Documentation 21

5.7.4 Member Function Documentation 21

5.7.5 Member Data Documentation 21
5.8 TFdmBoundaryCondition< T > Class Template Reference 22
5.8.1 Detailed Description 23
5.8.2 Constructor & Destructor Documentation 23
5.9 TFdmElement Class Reference 24
5.9.1 Detailed Description 25
5.9.2 Member Function Documentation 25
5.10 TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur > Class Template Reference 27
5.10.1 Detailed Description 28
5.10.2 Member Function Documentation 28
5.11 TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env > Class Template Reference 28
5.11.1 Detailed Description 29
5.11.2 Member Function Documentation 29
5.12 TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter > Class Template Reference 30
5.12.1 Detailed Description 31
5.12.2 Constructor & Destructor Documentation 32
5.12.3 Member Function Documentation 32
5.13 TFdmSc< FDM_TEMPLATE_RAW, matmodeltype > Class Template Reference 33
5.13.1 Detailed Description 34
5.13.2 Constructor & Destructor Documentation 34
5.13.3 Member Function Documentation 35
5.14 TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype > Class Template Reference 36
5.14.1 Detailed Description 37
5.15 TFem Class Reference 38
5.15.1 Detailed Description 40
5.15.2 Member Enumeration Documentation 40
5.15.3 Member Function Documentation 40
5.15.4 Member Data Documentation 41
5.16 TFemBoundaryCondition< T > Class Template Reference 41

5.16.1 Detailed Description 41
5.17 TFemDemo Class Reference 42
5.17.1 Detailed Description 42
5.17.2 Member Function Documentation 42
5.18 TFemElement Class Reference 43
5.18.1 Member Function Documentation 45
5.19 TFemSC Class Reference 46
5.19.1 Detailed Description 47
5.19.2 Constructor & Destructor Documentation 47
5.20 TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype > Class Template Reference 47
5.20.1 Detailed Description 48
5.21 TMagneticComponent Class Reference 48
5.21.1 Detailed Description 50
5.21.2 Constructor & Destructor Documentation 50
5.21.3 Member Function Documentation 50
5.22 TMagneticComponentOrtho Class Reference 51
5.22.1 Detailed Description 52
5.22.2 Constructor & Destructor Documentation 52
5.22.3 Member Function Documentation 53
5.22.4 Member Data Documentation 53
5.23 TMyFStream Struct Reference 53
5.23.1 Detailed Description 54
5.23.2 Member Function Documentation 54
5.24 TPixel Class Reference 55
5.24.1 Detailed Description 56
5.24.2 Member Function Documentation 56
5.25 TShape< geometrytype, paramtype > Class Template Reference 57
5.25.1 Detailed Description 58
5.25.2 Member Function Documentation 58
5.26 TSolvable Class Reference 59

5.26.1 Detailed Description 59
5.27 TSpice Class Reference 59
5.27.1 Detailed Description 61
5.27.2 Member Function Documentation 61
5.27.3 Member Data Documentation 62
5.28 TSpiceCondition Class Reference 62
5.28.1 Detailed Description 63
5.28.2 Constructor & Destructor Documentation 63
5.28.3 Member Data Documentation 63
5.29 TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent > Class Template Reference 64
5.29.1 Detailed Description 65
5.29.2 Friends And Related Function Documentation 65
5.30 TSpiceElement Class Reference 66
5.30.1 Detailed Description 66
5.31 TSpiceMultiElement Class Reference 67
5.31.1 Detailed Description 68
5.31.2 Member Function Documentation 68
5.32 TSpiceMultiElementSCT Class Reference 70
5.32.1 Detailed Description 71
5.32.2 Member Function Documentation 71
5.33 TSpiceSingleElement Class Reference 71
5.33.1 Detailed Description 73
5.34 TSpiceSingleElementC Class Reference 73
5.34.1 Detailed Description 74
5.35 TSpiceSingleElementL Class Reference 74
5.35.1 Detailed Description 75
5.36 TSpiceSingleElementR Class Reference 75
5.36.1 Detailed Description 76
5.37 TSpiceSingleElementU Class Reference 76
5.37.1 Detailed Description 77
5.38 TSpiceSingleElementUAC Class Reference 77
5.38.1 Detailed Description 78
5.38.2 Constructor & Destructor Documentation 78
5.39 TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent > Class Template Reference 79
5.39.1 Detailed Description 80
5.39.2 Member Function Documentation 80
5.40 TVector< T > Class Template Reference 82
5.40.1 Detailed Description 82
5.40.2 Member Function Documentation 83

6 File Documentation **84**

6.1 src/demos/all.hpp File Reference 84

6.1.1 Detailed Description 85

6.2 src/demos/fdm.hpp File Reference 85

6.2.1 Detailed Description 86

6.2.2 Function Documentation 86

6.3 src/demos/fem.hpp File Reference 87

6.3.1 Detailed Description 87

6.4 src/demos/shape.hpp File Reference 88

6.5 src/main.cpp File Reference 88

6.5.1 Function Documentation 89

Index **91**

1 Todo List

Member main (p. **89) ()**
doc

Class TBitmap (p. **8)**
support newer picture file formats, eg PNG

implement load(string fn) method to load bitmaps from file

Class TElement (p. **10)**
doku

Member TElement::getStateVariablesIntegral (p. **13) (TStateVariable< U > &var)**
between when?

Member TFdm::eps_geometry (p. **21)**
move to **TShape** (p. 57)

Member TFdm::eps_H (p. **21)**
scheint zu groß zu sein

Class TFdmElement (p. **24)**
common class for FDM and FEM?

Member TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >::TFdmElementNonlinearIsotropicLossfreeLut (p. **32) ()**
remove negative H values from LUT?

check for duplicate key values?

review this convention to use milliseconds instead of seconds

Member TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >::applyModel (p. **35) () override**
implement better numeric integration for phi1

implement better numeric integration for phi2

Member TFem::buildMesh (p. **40) ()**
implement

Member TFem::eps_geometry (p. **41)**
move to **TShape** (p. 57)

Member TFem::eps_H (p. **41)**
scheint zu groß zu sein

Class TFemDemo (p. **42)**
doc

Member TFemDemo::run (p. **42) ()**

Member TFemSC::TFemSC (p. **47) ()**
implement

Member TMagneticComponent::saveDetails (p. **50) (string fn)**
implement

Member TMagneticComponentOrtho::calc_mode_ortho (p. **53)**
inherit protected

Class TSolvable (p. **59)**
doku

Class TSpice (p. **59)**
improve the solving algorithm (v.s.)

Member TSpice::addSC (p. **61) (TSpiceCondition< ELEMENT_PARAMS > sc)**
maybe make this a reference parameter?

Member TSpice::solve (p. **61) () override**
retrieve additional conditions from multi-pole elements

implement preconditioner

Member TSpiceCondition::TSpiceCondition (p. **63) (TSpiceConditionType _type, vector< TSpiceSingle↩ Element< ELEMENT_PARAMS > (p. 71) ∗> _elements, vector< int8_t > _coefficients, T _b)**
doku

Class TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent > (p. **79)**
TParent must always be derived from **TSolvable** (p. 59)

Member TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >::log (p. **80) (T time)**
maybe it would make sense to implement this procedurally instead of object-oriented

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

fstream

TMyFStream **53**

TBitmap **8**

TConstDouble< numerator, denominator > **10**

- **TElement** 10
- **TElement< ELEMENT_PARAMS >** 10
  - **TFdmElement** 24
  - **TFemElement** 43
  - **TMagneticComponent** 48
  - **TSpiceElement** 66
- **TEquationSystem< T >** 15
  - **TEquationSystemLU< T >** 16
- **TException** 17
- **TFdm< FDM_PARAMS >** 18
  - **TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >** 33
  - **TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >** 36
- **TFdmBoundaryCondition< T >** 22
- **TFdmElement< ELEMENT_PARAMS >** 24
  - **TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >** 27
  - **TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur↩_env >** 28
  - **TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >** 30
- **TFem< ELEMENT_PARAMS >** 38
  - **TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >** 47
  - **TFemSquare< ELEMENT_PARAMS, TFemElementLinearIsotropicLossfree< ELEMENT_PAR↩AMS > >** 47
    - **TTransientSolver< ELEMENT_PARAMS, TFemSquare< ELEMENT_PARAMS, TFem↩ElementLinearIsotropicLossfree< ELEMENT_PARAMS > > >** 79
      - **TFemDemo** 42
- **TFemBoundaryCondition< T >** 41
- **TMagneticComponent< ELEMENT_PARAMS >** 48
  - **TMagneticComponentOrtho** 51
- **TMagneticComponentOrtho< ELEMENT_PARAMS >** 51
  - **TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >** 33
- TParent
  - **TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >** 79
  - **TTransientSolver< ELEMENT_PARAMS, TParent >** 79

TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent > 64
TPixel 55
TShape< geometrytype, paramtype > 57
TSolvable 59
TSolvable< ELEMENT_PARAMS > 59
TFdm 18
TFem 38
TFemSC 46
TSpice 59
TSpiceCondition 62
TSpiceElement< ELEMENT_PARAMS > 66
TSpiceMultiElement 67
TSpiceMultiElementSCT 70
TSpiceSingleElement 71
TSpiceSingleElement< ELEMENT_PARAMS > 71
TSpiceSingleElementC 73
TSpiceSingleElementL 74
TSpiceSingleElementR 75
TSpiceSingleElementU 76
TSpiceSingleElementUAC 77
TSpiceSingleElementL< ELEMENT_PARAMS > 74
TSpiceSingleElementR< ELEMENT_PARAMS > 75
TSpiceSingleElementUAC< ELEMENT_PARAMS > 77
TVector< T > 82
TVector< geometrytype > 82

3 Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

TBitmap
Stores a fixed sized picture 8

TConstDouble< numerator, denominator > 10

TElement 10

TEquationSystem< T > 15

TEquationSystemLU< T > 16

TException
Holds a message and should be the only class thrown 17

TFdm
Field simulations problem represented as network of fdm elements 18

TFdmBoundaryCondition< T >
Boundary condition for field simulation problem 22

TFdmElement
Determines the material's behaviour at a single point within a FDM field simulation Field simulations based on the Finite Difference Method (TFdm (p. 18)) expect to be populated with many objects of this class 24

TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >
A linear, isotropic, lossfree magnetic material is trivial. Relative Permeability can be set as a template parameter 27

TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env >
NONlinear, isotropic, lossfree magnetic material with analytical magnetization curve 28

TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >
NONlinear, isotropic, lossfree magnetic material with LUT for magnetization curve 30

TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >
Implements a cross similar to the ones used at the Single Cross Tester 33

TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >
The TFdmSquare (p. 36) class implements a simple square 36

TFem
Field simulations problem represented as network of FEM elements 38

TFemBoundaryCondition< T >
Boundary condition for field simulation problem 41

TFemDemo 42

TFemElement 43

TFemSC
Implements a single closs (SC) of a single cross tester (SCT) 46

TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >
The TFemSquare (p. 47) class implements a simple square 47

TMagneticComponent
Used in Ilka's master thesis to represent a two-pole reluctance 48

TMagneticComponentOrtho
Used in Ilka's master thesis to represent a four-pole reluctance 51

TMyFStream
Extends fstream by convenient and error-resistant methods 53

TPixel
Stores R,G,B values of a single pixel 55

TShape< geometrytype, paramtype >
Geometric shape 57

TSolvable 59

TSpice
(possibly electrical or magnetic) network 59

TSpiceCondition
SPICE condition (Kirchhoff's laws) 62

TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >
The TSpiceDemoSolver (p. 64) class is needed for the log(T time) (p. 65) method to work 64

TSpiceElement
Serves as a common ancestor of TSpiceSingleElement (p. 71) and TSpiceMultiElement (p. 67) 66

TSpiceMultiElement
Multi-port network element 67

TSpiceMultiElementSCT
The TSpiceMultiElementSCT (p. 70) class 70

TSpiceSingleElement
Two-port network element 71

TSpiceSingleElementC
Linear capacity is trivial 73

TSpiceSingleElementL
Linear inductance is be trivial 74

TSpiceSingleElementR
Linear resistor is trivial 75

TSpiceSingleElementU
Constant voltage source is trivial 76

TSpiceSingleElementUAC
AC voltage source 77

TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >
Solves any solvable problem for consecutive time steps 79

TVector< T >
3-dimensional vector of selectable data type, eg to represent physical entities 82

4 File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

src/bitmap.hpp ??

src/csv.hpp ??

src/element.hpp ??

src/equationsystem.hpp ??

src/exception.hpp ??

src/header.hpp ??

src/ main.cpp 88

src/myfstream.hpp ??

src/shape.hpp ??

src/solvable.hpp ??

src/vector.hpp ??

src/demos/ all.hpp
This file includes all (currently working) demos' files 84

src/demos/ fdm.hpp
This file holds the FdmDemo function 85

src/demos/ fem.hpp
This file holds the FemDemo function and its assets 87

src/demos/ shape.hpp 88

src/demos/spice.hpp ??

src/fdm/boundary.hpp ??

src/fdm/component.hpp ??

src/fdm/element.hpp ??

src/fdm/fdm.hpp ??

src/fdmmodels/sc.hpp ??

src/fdmmodels/square.hpp ??

src/fem/boundary.hpp ??

src/fem/element.hpp ??

src/fem/fem.hpp ??

src/femmodels/sc.hpp ??

src/femmodels/square.hpp ??

src/matmodels/air.hpp ??

src/matmodels/linearisotropiclossfree.hpp ??

src/matmodels/nonlinearisotropiclossfree.hpp ??

src/matmodels/nonlinearisotropiclossfreelut.hpp ??

src/netmodels/c.hpp ??

src/netmodels/l.hpp ??

src/netmodels/r.hpp ??

src/netmodels/sct.hpp ??

src/netmodels/u.hpp ??

src/spice/condition.hpp ??

src/spice/element.hpp ??

src/spice/multielement.hpp ??

src/spice/singleelement.hpp ??

src/spice/spice.hpp ??

src/tools/magcurve.hpp ??

src/tools/magcurvegeo.hpp ??

src/tools/magneticcomponentreplay.hpp ??

src/tools/transientsolver.hpp ??

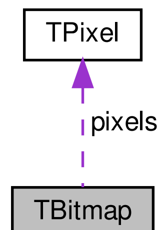
5 Class Documentation

5.1 TBitmap Class Reference

stores a fixed sized picture.

```
#include <bitmap.hpp>
```

Collaboration diagram for TBitmap:



Public Member Functions

- **TBitmap** (size_t _W, size_t _H)
  constructor
- **TBitmap** (**TBitmap** &b)
  copy constructor
- size_t **getW** ()
  returns the picture's current width
- size_t **getH** ()
  returns the picture's current height
- **TPixel** & **pix** (size_t x, size_t y)
  gets the pixel at the specified location by reference
- void **fill** (**TPixel** col)
  fills the entire bitmap picture with the specified color
- void **save** (string fn)
  saves bitmap to specified file (see `https://de.wikipedia.org/wiki/Windows_Bitmap`*)*
- **TBitmap** & **operator=** (**TBitmap** &b)
  assignment operator

Protected Attributes

- size_t **W**
  (fixed) width
- size_t **H**
  (fixed) height
- **TPixel** * **pixels**
  container for actual picture data / pixels

5.1.1 Detailed Description

stores a fixed sized picture.

obeys official Microsoft Bitmap Version 3 standard

Todo support newer picture file formats, eg PNG
implement load(string fn) method to load bitmaps from file

5.1.2 Constructor & Destructor Documentation

5.1.2.1 TBitmap()

```
TBitmap::TBitmap (
            size_t _W,
            size_t _H )
```

constructor

Parameters

_W,↩ _H	select fixed(!) width and height

5.2 TConstDouble< numerator, denominator > Class Template Reference

Public Member Functions

- **operator double** ()

Static Public Attributes

- static constexpr double **val** = double(numerator) / double(denominator)

5.3 TElement Class Reference

```
#include <element.hpp>
```

Public Member Functions

- **TElement** (size_t max_differential)
- int8_t **getMaxDifferential** ()

  returns the highest differential applied to state variables, e.g. 0 for a resistor or 1 for and inductance
- virtual void **applyModel** ()=0

  subclasses must implement phyisical laws.
- virtual void **advanceTime** (T dt)

  moves one step forward in time
- virtual void **retreatTime** ()

  moves one step back in time
- virtual void **relax** ()

  relaxes the element's state variables (clears their history and sets them to zero)
- T **getTime** ()

  gets the current time

Static Public Attributes

- static constexpr double **eps_t** = 1E-7

  smallest recognized time step.

Protected Member Functions

- template< typename U >
  void **initStateVariable** (TStateVariable< U > &var)
- template< typename U >
  void **deleteStateVariable** (TStateVariable< U > &var)
- template< typename U >
  void **setStateVariable** (TStateVariable< U > &variable, U value)
- template< typename U >
  U & **getStateVariable** (TStateVariable< U > &variable, int8_t time_offset=0)

  gets a state variables value state variable to fetch
- template< typename U >
  U **getStateVariablesDifferential** (TStateVariable< U > &var)
- template< typename U >
  U **getStateVariablesIntegral** (TStateVariable< U > &var)
- template< typename U >
  void **relax** (TStateVariable< U > &variable)
- template< typename U >
  void **advanceStateVariablesTime** (TStateVariable< U > &var)

Private Attributes

- int8_t **n**
- TStateVariable< T > **time**
- int8_t **time_idx**

  points in time for which the state variables are saved

5.3.1 Detailed Description

Todo doku

5.3.2 Member Function Documentation

5.3.2.1 advanceTime()

```
virtual void TElement::advanceTime (
            T dt ) [virtual]
```

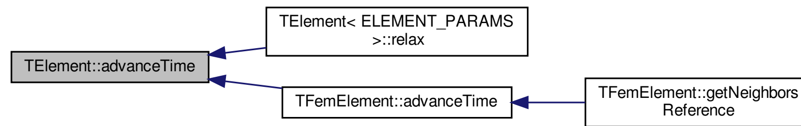
moves one step forward in time

Parameters

dt	amount of time passed in s

Reimplemented in **TFdmElement** (p. 25), and **TFemElement** (p. 44).

Here is the caller graph for this function:



5.3.2.2 applyModel()

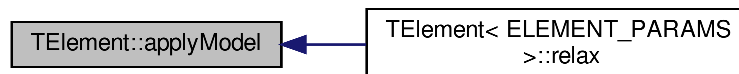
```
virtual void TElement::applyModel ( )  [pure virtual]
```

subclasses must implement phyisical laws.

E.g., an inductance must caluclate u from a given i and vice versa

Implemented in **TFdmElement** (p. 26), and **TFemElement** (p. 44).

Here is the caller graph for this function:

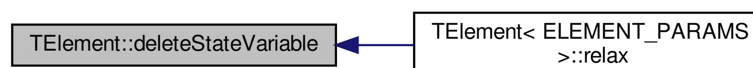


5.3.2.3 deleteStateVariable()

```
template<typename U >
void TElement::deleteStateVariable (
            TStateVariable< U > & var )  [inline], [protected]
```

< deallocates the state variable's memory.

This method must be called by derived classes' destructors to free their state variables' memory Here is the caller graph for this function:



5.3.2.4 getStateVariable()

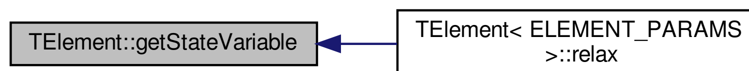
```
template<typename U >
U& TElement::getStateVariable (
            TStateVariable< U > & variable,
            int8_t time_offset = 0 ) [inline], [protected]
```

gets a state variables value state variable to fetch

Parameters

time_offset	0 for current value, positive for future, negative for past

Here is the caller graph for this function:



5.3.2.5 getStateVariablesDifferential()

```
template<typename U >
U TElement::getStateVariablesDifferential (
            TStateVariable< U > & var ) [inline], [protected]
```

< gets a state variable's temporal differential

5.3.2.6 getStateVariablesIntegral()

```
template<typename U >
U TElement::getStateVariablesIntegral (
            TStateVariable< U > & var ) [inline], [protected]
```

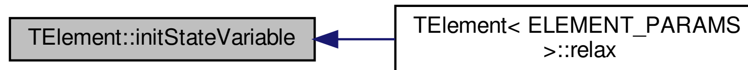
< gets a state variable's temporal integral

Todo between when?

5.3.2.7 initStateVariable()

```
template<typename U >
void TElement::initStateVariable (
            TStateVariable< U > & var ) [inline], [protected]
```

< clears the state variable's history (sets it to zero) Here is the caller graph for this function:



5.3.2.8 relax()

```
template<typename U >
void TElement::relax (
            TStateVariable< U > & variable ) [inline], [protected]
```

< relaxes the given state variable, e.g. setting all records to zero

5.3.2.9 setStateVariable()

```
template<typename U >
void TElement::setStateVariable (
            TStateVariable< U > & variable,
            U value ) [inline], [protected]
```

< sets a state variable's value

Parameters

variable	the state variable to set
value	the value to set

5.3.3 Member Data Documentation

5.3.3.1 eps_t

```
constexpr double TElement::eps_t = 1E-7 [static]
```

smallest recognized time step.

The entire softwares behavior is undefined for time steps smaller than this.

5.3.3.2 time_idx

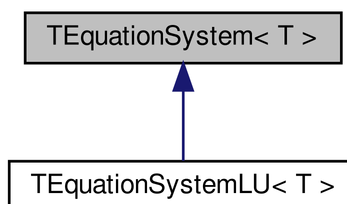
```
int8_t TElement::time_idx  [private]
```

points in time for which the state variables are saved

current time index

5.4 TEquationSystem< T > Class Template Reference

Inheritance diagram for TEquationSystem< T >:



Public Member Functions

- T & **A** (size_t row, size_t col)

    gets the coefficient matrix element at the specified index by reference
- T & **x** (size_t row)

    gets the unknown vector element at the specified index by reference
- T & **b** (size_t row)

    get the solution vector element at the specified index by reference
- virtual void **analyze** ()=0

    creates factorization of A in Ax=b in order to speed up solutions for different vectors b in Ax=b
- virtual void **solve** (bool reanalyze=true)=0

    solves the problem Ax=b for given A and b

Protected Attributes

- size_t **n**

    number of equations and variables (must be equal)
- T * **datA**

    coefficient matrix A of Ax=b
- T * **datx**

    unknown vector x of Ax=b
- T * **datb**

    solution vector b of Ax=b

5.4.1 Member Function Documentation

5.4.1.1 solve()

```
template<typename T >
virtual void  TEquationSystem< T >::solve (
            bool reanalyze = true )  [pure virtual]
```

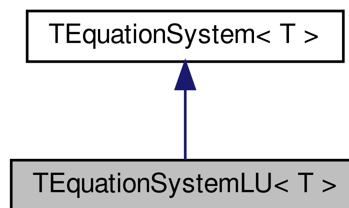
solves the problem Ax=b for given A and b

Parameters

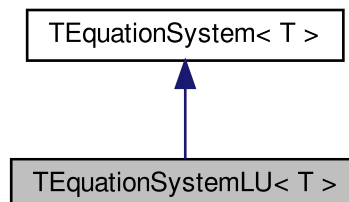
reanalyze	whether A has changed since last solution and a new factorization of A is needed; does not need to be true if **analyze()** (p. 15) has been called manually

5.5 TEquationSystemLU< T > Class Template Reference

Inheritance diagram for TEquationSystemLU< T >:



Collaboration diagram for TEquationSystemLU< T >:



Protected Attributes

- T * **L**
  L matrix of LU factorization.
- T * **U**
  U matrix of LU factorization.

Additional Inherited Members

5.6 TException Class Reference

The **TException** (p. 17) class holds a message and should be the only class thrown.

```
#include <exception.hpp>
```

Public Member Functions

- **TException** (string _msg)
  constructor
- void **show** ()
  shows the error message to the user, eg over stderr or a message box

Protected Attributes

- string **msg**
  error message

5.6.1 Detailed Description

The **TException** (p. 17) class holds a message and should be the only class thrown.

Other classes may be derived from this one to be thrown, too.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 TException()

```
TException::TException (
            string _msg )
```

constructor

Parameters

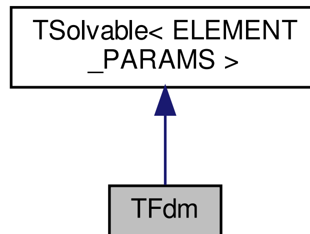
_msg	error message

5.7 TFdm Class Reference

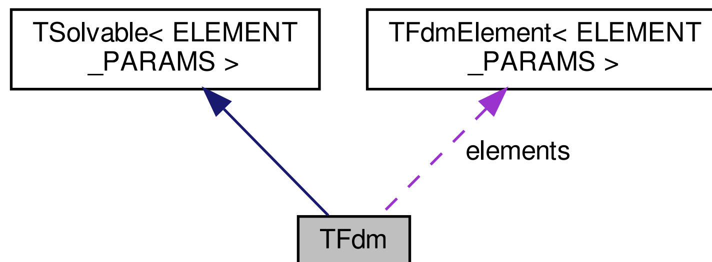
field simulations problem represented as network of fdm elements

```
#include <fdm.hpp>
```

Inheritance diagram for TFdm:



Collaboration diagram for TFdm:



Public Types

- enum **TPreconditioner** { **none**, **zero**, **lin** }

    preconditioning method for solving.

Public Member Functions

- **TFdm** (T _dx, T _dy)

    constructor

- virtual void **relax** ()

    relaxes all elements (eg H=0,B=0)

- virtual void **advanceTime** (T dt) override

  advances all elements' time
- virtual void **retreatTime** () override

  tells all elements to retreat time
- virtual void **applyModel** () override

  tells all elements to ***applyModel()*** *(p. 19)*
- virtual void **solve** () override

  solves the problem
- void **printBmp** (string fn)

  prints a picture (as bitmap) to the specified file name
- void **printTex** (string fn)

  prints a picture (as Tikz) to the specified file name
- void **saveCsv** (string fn, string delimiter=",")

  saves the result (as csv) to the specified file name

Protected Member Functions

- **TFdmBoundaryCondition**< T > & **bc** (size_t y, size_t x)

  gets the boundary condition for the specified coordinate index by reference
- template<size_t divider>
  void **solveRecursively** ()

  called by ***solve()*** *(p. 19)*
- T **getDx** ()
- T **getDy** ()

Protected Attributes

- T **phim** [nH+1][nW+1]

  temporary variable used by solveRecursively
- **TFdmElement**< ELEMENT_PARAMS > * **elements** [nH][nW]

  FDM elements.

Static Protected Attributes

- static constexpr double **eps_geometry** = 1E-6

  smallest recognized difference in geometry in m
- static constexpr double **eps_H** = 1E-0

  smallest recognized difference in field strength
- static constexpr double **eps_B** = 1E-6

  smallest recognized difference in flux density
- static constexpr size_t **n** = nH*nW

  number of elements

Private Attributes

- T **dx**

  horizontal space increment between two columns of elements
- T **dy**

  vertical space increment between two rows of elements
- bool **first_solve_done**

  internal variable used for optimization
- **TFdmBoundaryCondition**$<$ T $>$ **bcs_left** [nH+1]

  boundary conditions on the entire left column
- **TFdmBoundaryCondition**$<$ T $>$ **bcs_right** [nH+1]

  boundary conditions on the entire right column
- **TFdmBoundaryCondition**$<$ T $>$ **bcs_top** [nW-1]

  boundary conditions on the center top row (no corners)
- **TFdmBoundaryCondition**$<$ T $>$ **bcs_bottom** [nW-1]

  boundary conditions on the center bottom row (no corners)
- T **newphim** [nH+1][nW+1]

  temporary variable used by solveRecursively
- T **mu_cache** [nH][nW]

  temporary variable used by solveRecursively
- vector$<$ std::thread $*$ $>$ **threads**

  This class may use multiple threads, e.g. for simulation and for saving data to files. The destructor has to wait for them to finish.

5.7.1 Detailed Description

field simulations problem represented as network of fdm elements

This class implements the Finite Difference Method (FDM) and is therefore the most important part of this software for Ilka's master thesis. It is usually populated with elements (**TFdmElement** (p. 24)) by derived classes constructors and may take boundary conditions (bcs_...). The method **solve()** (p. 19) solves the problem. If the boundary conditions or even the elements change, solve tries to find a solution based on the last solution in order to save computing time. Results can be saved with printBmp, printTex and saveCsv. The destructor may take some time, because other threads may still be running, e.g. threads writing results to disk.

5.7.2 Member Enumeration Documentation

5.7.2.1 TPreconditioner

`enum` **`TFdm::TPreconditioner`**

preconditioning method for solving.

none: leave current element's data
zero: set unknown values to zero
lin: use linear regression between Dirichlet boundary conditions

5.7.3 Constructor & Destructor Documentation

5.7.3.1 TFdm()

```
TFdm::TFdm (
            T _dx,
            T _dy )
```

constructor

Parameters

dx	increment in x between two elements
dy	increment in y betweeen two elements

5.7.4 Member Function Documentation

5.7.4.1 bc()

```
TFdmBoundaryCondition<T>& TFdm::bc (
            size_t y,
            size_t x ) [protected]
```

gets the boundary condition for the specified coordinate index by reference

This function throws an exception, if the specified coordinate index is not on the border.

5.7.4.2 solveRecursively()

```
template<size_t divider>
void TFdm::solveRecursively ( ) [protected]
```

called by **solve()** (p. 19)

This needs to be a method on its own, because it is a recursive template.

5.7.5 Member Data Documentation

5.7.5.1 eps_geometry

```
constexpr double TFdm::eps_geometry = 1E-6 [static], [protected]
```

smallest recognized difference in geometry in m

Todo move to **TShape** (p. 57)

5.7.5.2 eps_H

```
constexpr double TFdm::eps_H = 1E-0  [static], [protected]
```

smallest recognized difference in field strength

Todo scheint zu groß zu sein

5.7.5.3 first_solve_done

```
bool TFdm::first_solve_done  [private]
```

internal variable used for optimization

If the problem has already been solved a first time, then the conditions have probably only changed for a litte. Therefore, a less aggressive approximation can be done and instead the old solution will be used to estimate the new one. This saves A LOT of computing time.

5.7.5.4 mu_cache

```
T TFdm::mu_cache[nH][nW]  [private]
```

temporary variable used by solveRecursively

This is not allocated on the stack by the using method, because it may take up enough memory to not fit and therefore raise an exception. This holds the current permeabilities of the elements

5.7.5.5 newphim

```
T TFdm::newphim[nH+1][nW+1]  [private]
```

temporary variable used by solveRecursively

This is not allocated on the stack by the using method, because it may take up enough memory to not fit and therefore raise an exception. This holds the new values for the magnetic scalar potential for each iteration step.

5.7.5.6 phim

```
T TFdm::phim[nH+1][nW+1]  [protected]
```

temporary variable used by solveRecursively

This is not allocated on the stack by the using method, because it may take up enough memory to not fit and therefore raise an exception. This holds the current values for the magnetic scalar potential after each iteration step.

5.8 TFdmBoundaryCondition< T > Class Template Reference

boundary condition for field simulation problem

```
#include <boundary.hpp>
```

Public Types

- enum **TFdmBCType** { **phim** =0, **H** =1 }

Public Member Functions

- **TFdmBoundaryCondition** ()
  default constructor
- **TFdmBoundaryCondition** (TFdmBCType _type, T _value)
  constructor

Public Attributes

- TFdmBCType **type**
  type: Dirichlet with phim; Neumann with H or B
- T **value**
  describes the boundary condition quantitatively

5.8.1 Detailed Description

template<typename T>
class TFdmBoundaryCondition< T >

boundary condition for field simulation problem

This class defines the boundary condition at a specific point in a magnetic field simulation. It allows two different types of conditions, determined by the enumerated type TFdmBCType: It can either be a Dirichlet boundary condition (type = phim) or a Neumann boundary condition (type = H). It also stores a value which is needed to describe the potential (Dirichlet) or its normal differental (Neumann) on the boundary. This class is probably only used by **TFdm** (p. 18).

5.8.2 Constructor & Destructor Documentation

5.8.2.1 TFdmBoundaryCondition()

```
template<typename T >
TFdmBoundaryCondition< T >:: TFdmBoundaryCondition (
            TFdmBCType _type,
            T _value )
```

constructor

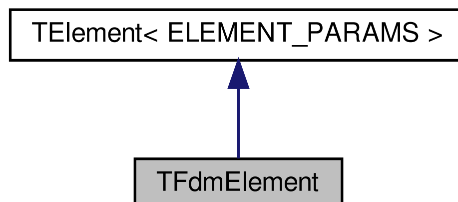
sets the specified type and vector

5.9 TFdmElement Class Reference

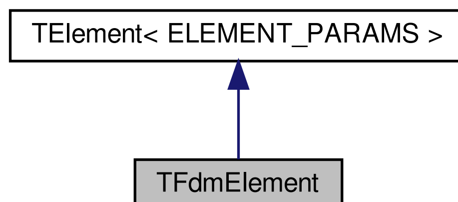
determines the material's behaviour at a single point within a FDM field simulation Field simulations based on the Finite Difference Method (**TFdm** (p. 18)) expect to be populated with many objects of this class.

```
#include <element.hpp>
```

Inheritance diagram for TFdmElement:



Collaboration diagram for TFdmElement:



Public Types

- enum **TFdmElementCalcMode** { **presetH**, **presetB**, **presetPhim**, **none** }

Public Member Functions

- **TFdmElement** (int8_t max_differential)

    standard constructor
- virtual ~**TFdmElement** ()

    virtual destructor to allow virtual functions
- void **setH** (**TVector**< T > _H)

    sets magnetic field strength (state variable)
- void **setB** (**TVector**< T > _B)

sets magnetic flux (state variable)
- **TVector**< T > **getH** ()
  gets magnetic field strength (state variable)
- **TVector**< T > **getB** ()
  gets magnetic flux (state variable)
- virtual T **getMu** ()
  get the current secant permeability
- virtual void **applyModel** ()
  calculates B fom H or vice versa
- virtual void **relax** ()
  relaxes the material (H=0, B=0)
- virtual void **advanceTime** (T dt)
  tells the material model that time has passed (dt in s) since the last calculation of H or B

Protected Member Functions

- TFdmElementCalcMode **getCalcMode** ()

Protected Attributes

- TStateVariable< **TVector**< T > > **H**
  current state of H at this element
- TStateVariable< **TVector**< T > > **B**
  current state of B at this element

Private Attributes

- TFdmElementCalcMode **calc_mode**
  calculation mode; defines preset value, the other one may then be calculated by ***applyModel()*** *(p. 26)*

Additional Inherited Members

5.9.1 Detailed Description

determines the material's behaviour at a single point within a FDM field simulation Field simulations based on the Finite Difference Method (**TFdm** (p. 18)) expect to be populated with many objects of this class.

To determine the materials behaviour, classes must inherit from this one and implement either applyModel or getMu. It it recommended to to both. If a derived class fails to do so, the running code will be trapped in an infinite recursion loop and will probably run into a stack overflow soon.

Todo common class for FDM and FEM?

5.9.2 Member Function Documentation

5.9.2.1 advanceTime()

```
virtual void TFdmElement::advanceTime (
            T dt ) [virtual]
```

tells the material model that time has passed (dt in s) since the last calculation of H or B

If this method is overwritten in derived classes, the call must propagate to this method!

Reimplemented from **TElement**< **ELEMENT_PARAMS** > (p. 11).

5.9.2.2 applyModel()

```
virtual void TFdmElement::applyModel ( ) [virtual]
```

calculates B fom H or vice versa

Either this function or **getMu()** (p. 26) has to be implemented by derived classes

Implements **TElement**< **ELEMENT_PARAMS** > (p. 12).

Reimplemented in **TFdmElementNonlinearIsotropicLossfreeLut**< **ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter** > (p. 32), and **TFdmElementNonlinearIsotropicLossfree**< **ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env** > (p. 29).

5.9.2.3 getMu()

```
virtual T TFdmElement::getMu ( ) [virtual]
```

get the current secant permeability

Either this function or **applyModel()** (p. 26) has to be implemented by derived classes

Reimplemented in **TFdmElementLinearIsotropicLossfree**< **ELEMENT_TEMPLATE_RAW, mur** > (p. 28).

5.9.2.4 relax()

```
virtual void TFdmElement::relax ( ) [virtual]
```

relaxes the material (H=0, B=0)

If this method is overwritten in derived classes, the call must propagate to this method!

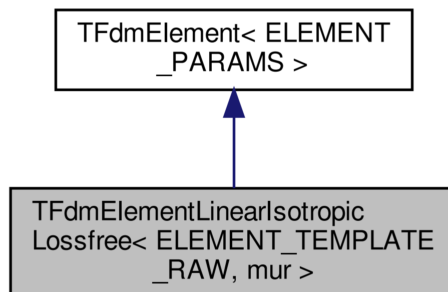
Reimplemented from **TElement**< **ELEMENT_PARAMS** > (p. 10).

5.10 TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur > Class Template Reference

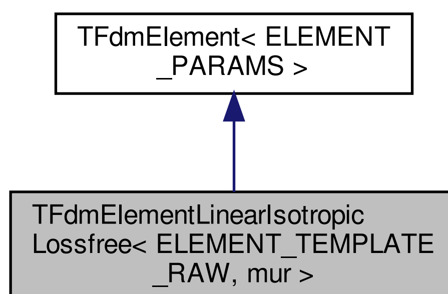
A linear, isotropic, lossfree magnetic material is trivial. Relative Permeability can be set as a template parameter.

```
#include <linearisotropiclossfree.hpp>
```

Inheritance diagram for TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >:



Collaboration diagram for TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >:



Public Member Functions

- virtual T **getMu** () override

    get the current secant permeability

Additional Inherited Members

5.10.1 Detailed Description

template< ELEMENT_TEMPLATE_RAW, size_t mur>
class TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >

A linear, isotropic, lossfree magnetic material is trivial. Relative Permeability can be set as a template parameter.

5.10.2 Member Function Documentation

5.10.2.1 getMu()

```
template<ELEMENT_TEMPLATE_RAW , size_t mur>
virtual T TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >::getMu ( ) [inline],
[override], [virtual]
```

get the current secant permeability

Either this function or **applyModel()** (p. 26) has to be implemented by derived classes

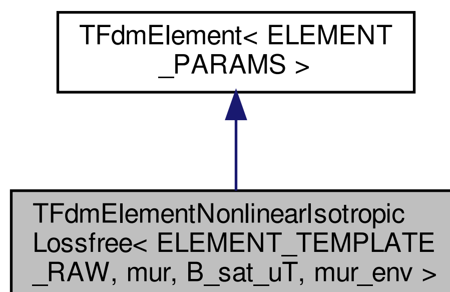
Reimplemented from **TFdmElement< ELEMENT_PARAMS >** (p. 26).

5.11 TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env > Class Template Reference

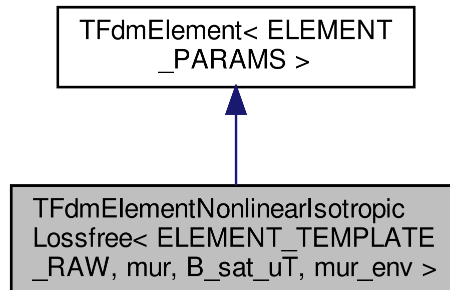
NONlinear, isotropic, lossfree magnetic material with analytical magnetization curve.

```
#include <nonlinearisotropiclossfree.hpp>
```

Inheritance diagram for TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env >:



Collaboration diagram for TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env >:



Public Member Functions

- virtual void **applyModel** () override

  calculates B fom H or vice versa

Additional Inherited Members

5.11.1 Detailed Description

template<ELEMENT_TEMPLATE_RAW, size_t mur, size_t B_sat_uT = 1500000, size_t mur_env = 1>
class TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env >

NONlinear, isotropic, lossfree magnetic material with analytical magnetization curve.

This class simply implements an exponential function as described in Ilka's master's thesis. The differential relative permeability to have near saturation should generally be 1. However, for some "hacks" it can be different values.

Parameters

mur	relative permeability at H=0
B_sat_ut	saturation flux density in MICRO tesla
mur_env	differential relative permeability to have near saturation

5.11.2 Member Function Documentation

5.11.2.1 applyModel()

```
template<ELEMENT_TEMPLATE_RAW , size_t mur, size_t B_sat_uT = 1500000, size_t mur_env = 1>
virtual void  TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT,
mur_env >::applyModel ( )  [inline], [override], [virtual]
```

calculates B fom H or vice versa

Either this function or **getMu()** (p. 26) has to be implemented by derived classes

Reimplemented from **TFdmElement**< **ELEMENT_PARAMS** > (p. 26).

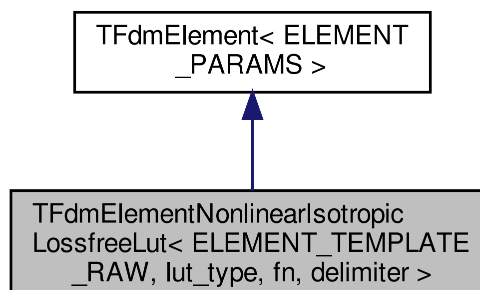
Here is the call graph for this function:

5.12 TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter > Class Template Reference

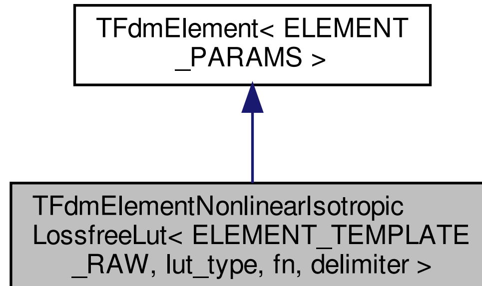
NONlinear, isotropic, lossfree magnetic material with LUT for magnetization curve.

```
#include <nonlinearisotropiclossfreelut.hpp>
```

Inheritance diagram for TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >:



Collaboration diagram for TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >:



Public Member Functions

- **TFdmElementNonlinearIsotropicLossfreeLut** ()
- virtual void **applyModel** () override

    calculates B fom H or vice versa

Static Private Attributes

- static bool **lut_created** =false
- static map< T, T > **lut**

Additional Inherited Members

5.12.1 Detailed Description

template<ELEMENT_TEMPLATE_RAW, TFdmElementNonlinearIsotropicLossfreeLutType lut_type, const char * fn, const char * delimiter>
class TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >

NONlinear, isotropic, lossfree magnetic material with LUT for magnetization curve.

This class can read a lookup table (LUT) from file and that for the magnetization curve. The LUT is saved as a static member, therefore, the file's content should not change during the creation of different objects. In every case, the objects will all have the same values from the old file version.

There are two different types of CSV files that can be read: Magnetization curves (lut_type = luttMagCurve) will be taken literally.

Measurements (lut_type = luttMeasurement) are data on the time domain that will be analyzed to find a magnetization curve, which will even work with hyseresis loops! However, the latter one is still experimental. In either case, the columns "h" and "b" will be read.

The method applyModel is slow. Use analytical options (TFdmElementNonlinearIsotropicLossfree) if possible.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 TFdmElementNonlinearIsotropicLossfreeLut()

```
template<ELEMENT_TEMPLATE_RAW , TFdmElementNonlinearIsotropicLossfreeLutType lut_type, const
char * fn, const char * delimiter>
TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >::
TFdmElementNonlinearIsotropicLossfreeLut ( ) [inline]
```

Todo remove negative H values from LUT?

Todo check for duplicate key values?

Todo review this convention to use milliseconds instead of seconds

5.12.3 Member Function Documentation

5.12.3.1 applyModel()

```
template<ELEMENT_TEMPLATE_RAW , TFdmElementNonlinearIsotropicLossfreeLutType lut_type, const
char * fn, const char * delimiter>
virtual void TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn,
delimiter >::applyModel ( ) [inline], [override], [virtual]
```

calculates B fom H or vice versa

Either this function or **getMu()** (p. 26) has to be implemented by derived classes

Reimplemented from **TFdmElement**< **ELEMENT_PARAMS** > (p. 26).

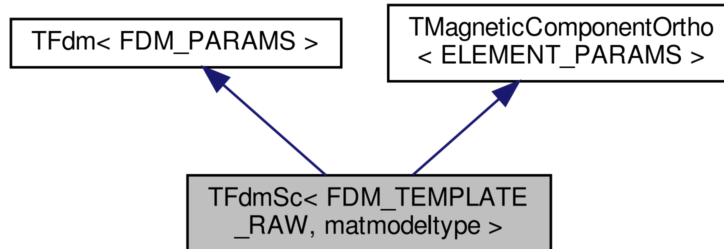
Here is the call graph for this function:

5.13 TFdmSc< FDM_TEMPLATE_RAW, matmodeltype > Class Template Reference

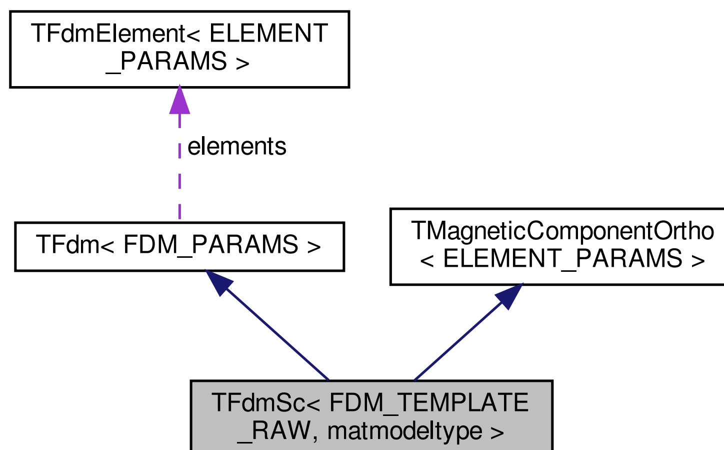
implements a cross similar to the ones used at the Single Cross Tester

```
#include <sc.hpp>
```

Inheritance diagram for TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >:



Collaboration diagram for TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >:



Public Member Functions

- **TFdmSc** (T W=0.035, T H=0.035)

    constructor
- virtual ~**TFdmSc** ()

    destructor

- virtual void **applyModel** () override

    sets field simulation boundary conditions based on reluctance's state variables, solves the problem and sets other state variables accordningly
- virtual void **relax** () override

    passes call to base classes
- virtual void **saveDetails** (string fn) override

    saves field simulation to specified file

Static Private Attributes

- static constexpr T **div** =3

    determines the part (of width and heigth) of the regarded region which is used for the cross

Additional Inherited Members

5.13.1 Detailed Description

template<FDM_TEMPLATE_RAW, typename matmodeltype>
class TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >

implements a cross similar to the ones used at the Single Cross Tester

This class creates a solvable field simulation problem and defines a single cross (invented by Ilka Schulz, see her master's thesis). The ratio of left to middle to right is the same a the one top to middle to bottom and it is 1:1:1. The arms take up four ninth of the area, the center only one ninth and another four ninth is populated with air. This means significant computing ressources go into simulation the area outside the single crosses intersection/center. However, approx. 100 by 100 elements make up for a decent field simulation.

It also inherits from TMagneticComponentOrtho, so the boundary conditions are not set manually, but by setting the primary and secondary reluctances

state variables. Also the result does not need to be analyzed manually. The other state variables are always set by applyModel accordingly.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 TFdmSc()

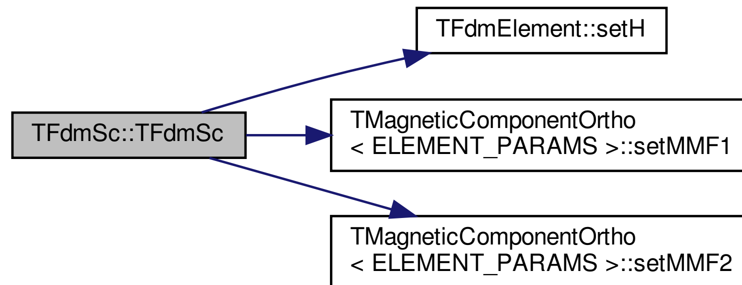
```
template<FDM_TEMPLATE_RAW , typename matmodeltype >
TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >:: TFdmSc (
            T W = 0.035,
            T H = 0.035 )
```

constructor

Parameters

W	width in m
H	height in m

Here is the call graph for this function:



5.13.2.2 ∼TFdmSc()

```
template<FDM_TEMPLATE_RAW , typename matmodeltype >
TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >::∼ TFdmSc ( ) [virtual]
```

destructor

deletes(!) all elements

5.13.3 Member Function Documentation

5.13.3.1 applyModel()

```
template<FDM_TEMPLATE_RAW , typename matmodeltype >
void TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >::applyModel ( ) [override], [virtual]
```

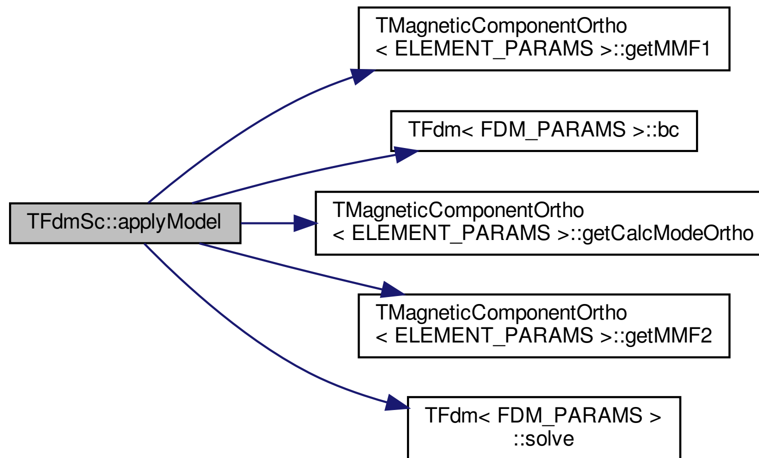
sets field simulation boundary conditions based on reluctance's state variables, solves the problem and sets other state variables accordningly

Todo implement better numeric integration for phi1

Todo implement better numeric integration for phi2

Reimplemented from **TFdm**< **FDM_PARAMS** > (p. 19).

Here is the call graph for this function:



5.13.3.2 saveDetails()

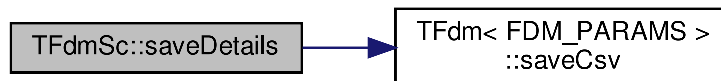
```
template<FDM_TEMPLATE_RAW , typename matmodeltype >
virtual void TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >::saveDetails (
            string fn ) [inline], [override], [virtual]
```

saves field simulation to specified file

Parameters

fn	file name to save to

Here is the call graph for this function:

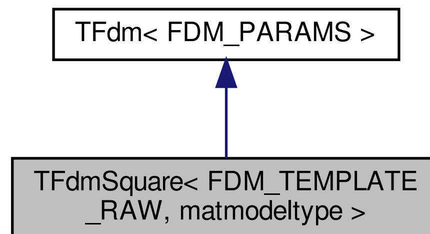


5.14 TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype > Class Template Reference

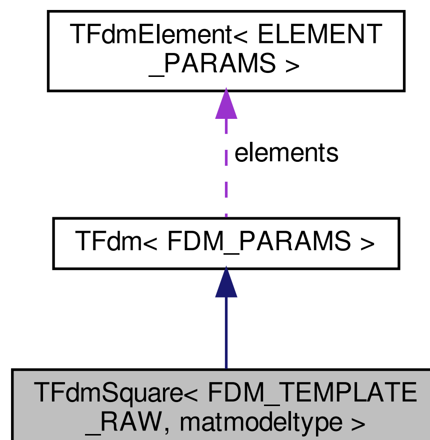
The **TFdmSquare** (p. 36) class implements a simple square.

```
#include <square.hpp>
```

Inheritance diagram for TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >:



Collaboration diagram for TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >:



Public Member Functions

- **TFdmSquare** (T W=0.01, T H=0.01)
- void **setMMF** (T mmf)

Additional Inherited Members

5.14.1 Detailed Description

template<FDM_TEMPLATE_RAW, typename matmodeltype>
class TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >

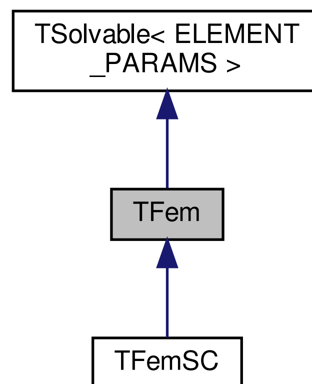
The **TFdmSquare** (p. 36) class implements a simple square.

5.15 TFem Class Reference

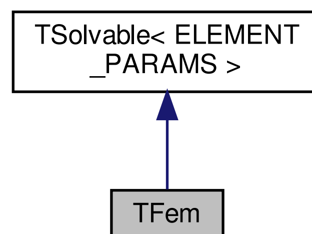
field simulations problem represented as network of FEM elements

```
#include <fem.hpp>
```

Inheritance diagram for TFem:



Collaboration diagram for TFem:



Public Types

- enum **TPreconditioner** { **none**, **zero**, **lin** }

  preconditioning method for solving.

Public Member Functions

- virtual void **relax** ()
  relaxes all elements (eg H=0,B=0)
- virtual void **advanceTime** (T dt) override
  advances all elements' time
- virtual void **retreatTime** () override
  *derived classes must inform their elements about retreated time by calling their **retreatTime()** (p. 39)*
- virtual void **applyModel** () override
  *derived classes must call **applyModel()** (p. 39) on their elements*
- virtual void **solve** () override
  solves the problem
- void **solve** (**TPreconditioner** precond, int n_threads=1)
  solves the problem
- void **printBmp** (string fn)
  prints a picture (as bitmap) to the specified file name
- void **printTex** (string fn)
  prints a picture (as Tikz) to the specified file name
- void **saveCsv** (string fn, string delimiter=",")
  saves the result (as csv) to the specified file name

Protected Member Functions

- void **clear** ()
  deletes(!) all elements, not just their pointers
- void **buildMesh** ()
  erases old and create new neighbor relations between elements
- void **setBC** (**TFemElement**< ELEMENT_PARAMS > *e, **TFemBoundaryCondition**< T > bc)
  sets boundary condition for the specified element
- void **rmBC** (**TFemElement**< ELEMENT_PARAMS > *e)
  removes boundary conditon for specified element
- void **clearBCs** ()
  removes all boundary conditions

Protected Attributes

- vector< **TFemElement**< ELEMENT_PARAMS > * > **elements**
  *FEM elements. needs to use pointer to **TFemElement** (p. 43) in order to use derivated classes.*
- map< **TFemElement**< ELEMENT_PARAMS > *, **TFemBoundaryCondition**< T > > **bcs**
  boundary conditions and the associated elements

Static Protected Attributes

- static constexpr double **eps_geometry** = 1E-6
  smallest recognized difference in geometry in m
- static constexpr double **eps_H** = 1E-0
  smallest recognized difference in field strength
- static constexpr double **eps_B** = 1E-9
  smallest recognized difference in flux density

Private Attributes

- bool **first_solve_done**

5.15.1 Detailed Description

field simulations problem represented as network of FEM elements

5.15.2 Member Enumeration Documentation

5.15.2.1 TPreconditioner

```
enum TFem::TPreconditioner
```

preconditioning method for solving.

none: leave current element's data
zero: set unknown values to zero
lin: use linear regression between Dirichlet boundary conditions

5.15.3 Member Function Documentation

5.15.3.1 buildMesh()

```
void TFem::buildMesh ( ) [protected]
```

erases old and create new neighbor relations between elements

Todo implement

5.15.3.2 solve()

```
void TFem::solve (
              TPreconditioner precond,
              int n_threads = 1 )
```

solves the problem

Parameters

precond	preconditioner to be used for iterative solving method
n_threads	amount of threads created (recommended: 1 for small nets ($<$ 20 000 elements), 4 for large FEM nets ($>$ 100 000 elements))

5.15.4 Member Data Documentation

5.15.4.1 eps_geometry

```
constexpr double TFem::eps_geometry = 1E-6  [static], [protected]
```

smallest recognized difference in geometry in m

Todo move to **TShape** (p. 57)

5.15.4.2 eps_H

```
constexpr double TFem::eps_H = 1E-0  [static], [protected]
```

smallest recognized difference in field strength

Todo scheint zu groß zu sein

5.16 TFemBoundaryCondition< T > Class Template Reference

boundary condition for field simulation problem

```
#include <boundary.hpp>
```

Public Member Functions

- **TFemBoundaryCondition** ()
  default constructor
- **TFemBoundaryCondition** (TFemBCType _type, **TVector**< T > _vec)
  constructor. sets the specified type and vector

Public Attributes

- TFemBCType **type**
  type: Dirichlet with H or B; Neumann with dH or dB
- **TVector**< T > **vec**
  describes the boudary condition quantitatively

5.16.1 Detailed Description

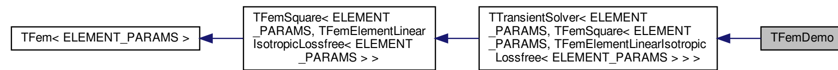
template<typename T>
class TFemBoundaryCondition< T >

boundary condition for field simulation problem

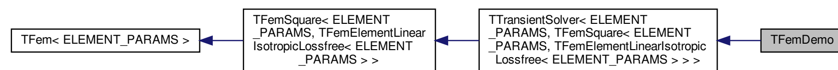
5.17 TFemDemo Class Reference

```
#include <fem.hpp>
```

Inheritance diagram for TFemDemo:



Collaboration diagram for TFemDemo:



Public Member Functions

- void **run** ()
  runs the demo

Protected Member Functions

- virtual void **log** (T time) override
  saves the current state as Bitmap (one for each time step)

Private Attributes

- fstream ∗ **f**
  output file stream to write

Additional Inherited Members

5.17.1 Detailed Description

Todo doc

5.17.2 Member Function Documentation

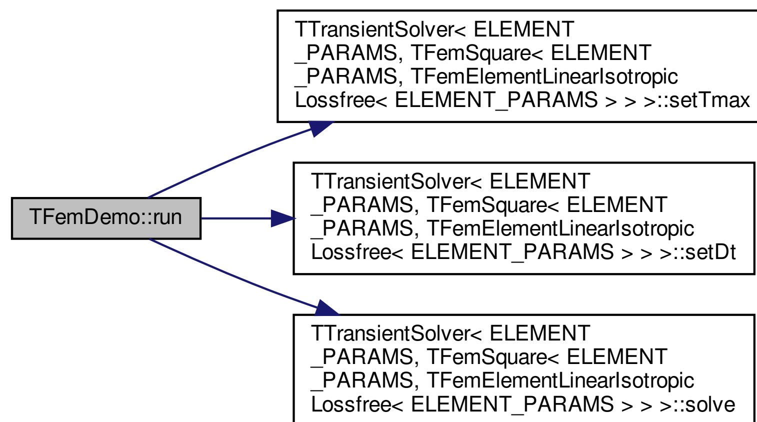
5.17.2.1 run()

```
ELEMENT_TEMPLATE void TFemDemo::run ( )
```

runs the demo

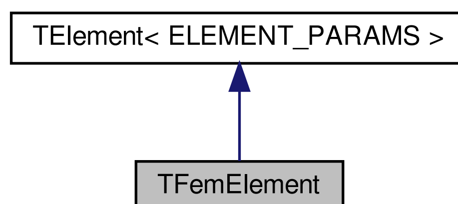
Todo

Here is the call graph for this function:

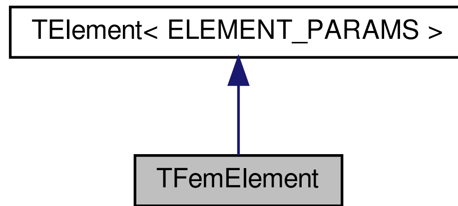


5.18 TFemElement Class Reference

Inheritance diagram for TFemElement:



Collaboration diagram for TFemElement:



Public Member Functions

- **TFemElement** (int8_t max_differential)
  standard constructor
- virtual ∼**TFemElement** ()
  virtual destructor to allow virtual functions
- void **setH** (**TVector**< T > _H)
- void **setB** (**TVector**< T > _B)
- void **setLocation** (**TVector**< T > _location)
- void **setNeighbor** (**TFemElement** ∗e)
  adds the neighbor and sets distance vector based on its and the neighbor's location
- void **clearNeighbors** ()
  clears neighbor list (does not delete the elements themselves)
- void **setNeighbors** (map< **TVector**< T >, **TFemElement** ∗> _neighbors)
  clears alls prev. neighbors with clear() and sets the new specified set of neighbors
- **TVector**< T > **getH** ()
- **TVector**< T > **getB** ()
- **TVector**< T > **getLocation** ()
- map< **TVector**< T >, **TFemElement** ∗ > **getNeighbors** ()
  gets neighbors map as copy
- map< **TVector**< T >, **TFemElement** ∗ > & **getNeighborsReference** ()
  gets neighbors bei reference
- virtual void **applyModel** ()=0
  calculates B fom H or vice versa
- virtual void **relax** ()
  relaxes the material (H=0, B=0)
- virtual void **advanceTime** (T dt)
  tells the material model that time has passed (dt in s) since the last calculation of H or B

Protected Member Functions

- TFemElementCalcMode **getCalcMode** ()
- void **setNeighbor** (**TVector**< T > d, **TFemElement** ∗e)
  adds the neighbor and hard defines the distance vector from this element to the neighbor

Protected Attributes

- TStateVariable< **TVector**< T > > **H**
  current state of H at this element
- TStateVariable< **TVector**< T > > **B**
  current state of B at this element

Static Protected Attributes

- static constexpr double **mu0** = 4∗M_PI∗1E-7
  permeability of vacuum

Private Attributes

- TFemElementCalcMode **calc_mode**
  calculation mode; defines preset value, the other one may then be calculated by ***applyModel()*** *(p. 44)*
- **TVector**< T > **location**
  location of the element in the physical space
- map< **TVector**< T >, **TFemElement**< ELEMENT_PARAMS > ∗ > **neighbors**
  elements around this one and their corresponding distance vector from this element to the neighbor

Additional Inherited Members

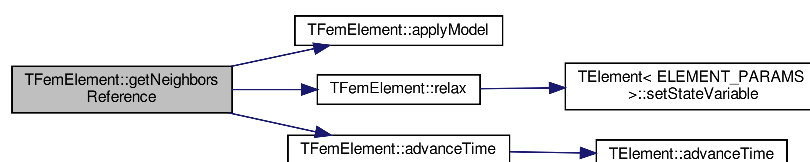
5.18.1 Member Function Documentation

5.18.1.1 getNeighborsReference()

```
map< TVector<T>, TFemElement*>& TFemElement::getNeighborsReference ( ) [inline]
```

gets neighbors bei reference

allows write access to neighbors and is much faster then **getNeighbors()** (p. 44) Here is the call graph for this function:

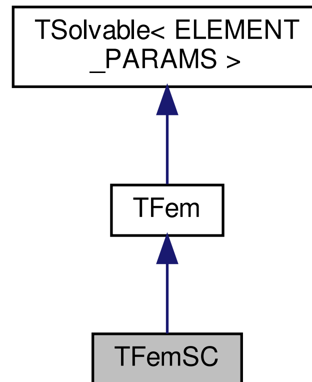


5.19 TFemSC Class Reference

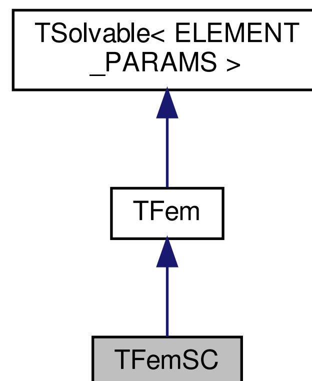
implements a single closs (SC) of a single cross tester (SCT)

`#include <sc.hpp>`

Inheritance diagram for TFemSC:



Collaboration diagram for TFemSC:



Public Member Functions

- template<typename matmodeltype >
  TFemSC ()

  constructor

Additional Inherited Members

5.19.1 Detailed Description

implements a single closs (SC) of a single cross tester (SCT)

5.19.2 Constructor & Destructor Documentation

5.19.2.1 TFemSC()

```
template<typename matmodeltype >
TFemSC::TFemSC ( )
```

constructor

Parameters

matmodeltype	desired material model type

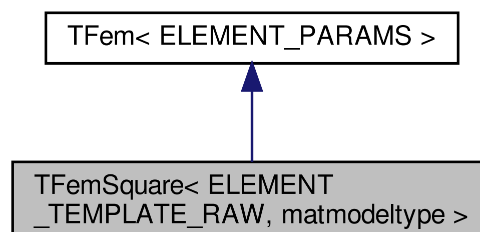
Todo implement

5.20 TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype > Class Template Reference

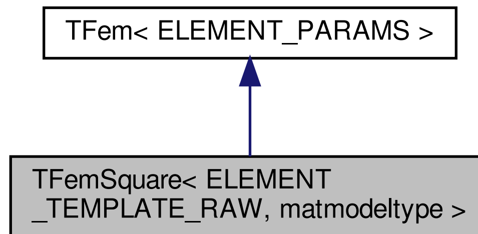
The **TFemSquare** (p. 47) class implements a simple square.

```
#include <square.hpp>
```

Inheritance diagram for TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >:



Collaboration diagram for TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >:



Public Member Functions

- virtual void **advanceTime** (T dt) override
  advances all elements' time

Additional Inherited Members

5.20.1 Detailed Description

template<ELEMENT_TEMPLATE_RAW, typename matmodeltype>
class TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >

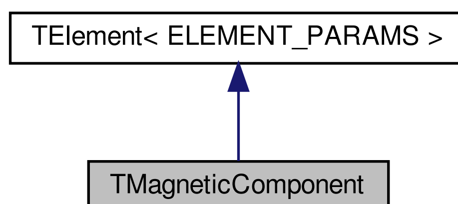
The **TFemSquare** (p. 47) class implements a simple square.

5.21 TMagneticComponent Class Reference

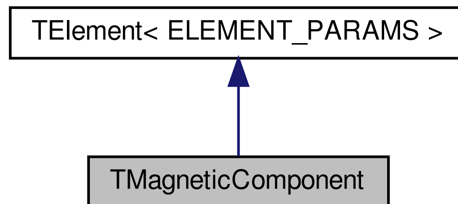
used in Ilka's master thesis to represent a two-pole reluctance

`#include <component.hpp>`

Inheritance diagram for TMagneticComponent:



Collaboration diagram for TMagneticComponent:



Public Member Functions

- **TMagneticComponent** ()
  constructor
- virtual void **setMMF** (T _mmf)
  set status variable of magnetomotive force
- virtual void **setPhi** (T _phi)
  set status variable of magnetic flux
- virtual T **getMMF** ()
  get status variable of magnetomotive force
- virtual T **getPhi** ()
  get status variable of magnetic flux
- virtual void **saveDetails** (string fn)
  may save details to filename, e.g. current status variables or details about parent class status

Protected Member Functions

- TMagneticComponentCalcMode **getCalcMode** ()
  gets current calculation mode

Protected Attributes

- TStateVariable< T > **mmf**
  magnetomotive force
- TStateVariable< T > **phi**
  magnetic flux

Private Attributes

- TMagneticComponentCalcMode **calc_mode**
  last set status variable determines which status variable must be calculated

Additional Inherited Members

5.21.1 Detailed Description

used in Ilka's master thesis to represent a two-pole reluctance

This class does NOT contain anything to calculate anything. It serves as an abstract outline for two-pole reluctances.

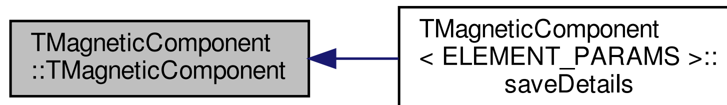
5.21.2 Constructor & Destructor Documentation

5.21.2.1 TMagneticComponent()

```
TMagneticComponent::TMagneticComponent ( )
```

constructor

Basically calls parent constructor, sets the status variables to zero and calc_mode to none. Here is the caller graph for this function:



5.21.3 Member Function Documentation

5.21.3.1 saveDetails()

```
virtual void TMagneticComponent::saveDetails (
            string fn )  [inline], [virtual]
```

may save details to filename, e.g. current status variables or details about parent class status

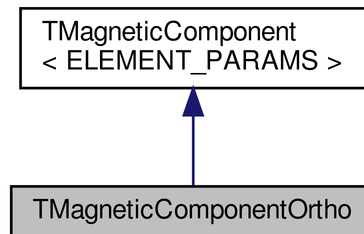
Todo implement

5.22 TMagneticComponentOrtho Class Reference

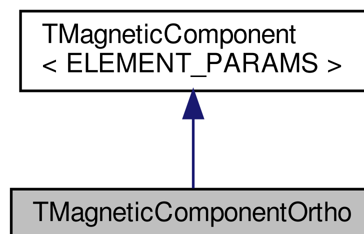
used in Ilka's master thesis to represent a four-pole reluctance

```
#include <component.hpp>
```

Inheritance diagram for TMagneticComponentOrtho:



Collaboration diagram for TMagneticComponentOrtho:



Public Member Functions

- **TMagneticComponentOrtho** ()
- virtual void **setMMF1** (T _mmf)

  sets primary magnetomotive force
- virtual void **setMMF2** (T _mmf)

  sets secondary magnetomotive force
- virtual void **setPhi1** (T _phi)

  sets primary magnetic flux
- virtual void **setPhi2** (T _phi)

  sets secondary magnetic flux
- virtual T **getMMF1** ()

gets primary magnetomotive force
- virtual T **getMMF2** ()

  gets secondary magnetomotive force
- virtual T **getPhi1** ()

  gets primary magnetic flux
- virtual T **getPhi2** ()

  gets secondary magnetic flux

Protected Member Functions

- TMagneticComponentCalcMode **getCalcModeOrtho** ()

  gets SECONDARY calculation mode

Protected Attributes

- TStateVariable$<$ T $>$ **mmf2**

  secondary magnetomotive force
- TStateVariable$<$ T $>$ **phi2**

  secondary magnetic flux

Private Attributes

- TMagneticComponentCalcMode **calc_mode_ortho**

  calculation mode only for secondary clamps

5.22.1 Detailed Description

used in Ilka's master thesis to represent a four-pole reluctance

Currently, there is no need for fancy multi-pole elements, like **TSpiceMultiElement** (p. 67). So, this class is implemented for convenience. This class does NOT contain anything to calculate anything. It serves as an abstract outline for two-pole reluctances.

5.22.2 Constructor & Destructor Documentation

5.22.2.1 TMagneticComponentOrtho()

```
TMagneticComponentOrtho::TMagneticComponentOrtho ( )  [inline]
```

$<$ constructor

Basically calls parent (primary clamps reluctance) constructor, sets the status variables to zero and calc_mode to none.

5.22.3 Member Function Documentation

5.22.3.1 getCalcModeOrtho()

`TMagneticComponentCalcMode TMagneticComponentOrtho::getCalcModeOrtho ( ) [inline], [protected]`

gets SECONDARY calculation mode

This method is not virtual, so it does not overwrite, but indeed shadow the parent's method with the same name.

5.22.4 Member Data Documentation

5.22.4.1 calc_mode_ortho

`TMagneticComponentCalcMode TMagneticComponentOrtho::calc_mode_ortho [private]`

calculation mode only for secondary clamps

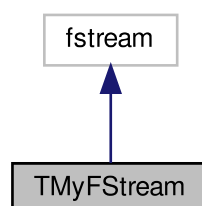
Todo inherit protected

5.23 TMyFStream Struct Reference

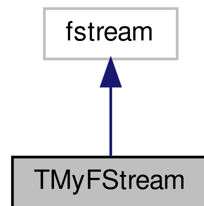
extends fstream by convenient and error-resistant methods

`#include <myfstream.hpp>`

Inheritance diagram for TMyFStream:



Collaboration diagram for TMyFStream:



Public Member Functions

- **TMyFStream** ()
  constructor
- **TMyFStream** (**TMyFStream** &)
  copy constructor
- **TMyFStream** (string fn, ios_base::openmode mode=ios_base::in|ios_base::out)
  opens the specified file in the specified mode in binary mode
- template<typename T >
  T **read** ()
  read binary representation of data; may be instantiated explicitly
- char * **read** (size_t len)
  read a char array
- string **readStr0** ()
  read a null-terminated char string
- **TMyFStream** & **read** (char *c, unsigned int len)
  read bytes
- template<typename T >
  TMyFStream & **write** (T x)
  write binary representation of data; may be instantiated explicitly
- **TMyFStream** & **writeStr** (string str)
  write a char string without any further info as length or null-terminator
- **TMyFStream** & **writeStr0** (string str)
  write a char string with null-terminator

5.23.1 Detailed Description

extends fstream by convenient and error-resistant methods

5.23.2 Member Function Documentation

5.23.2.1 read() [1/3]

```
template<typename T >
T TMyFStream::read ( )
```

read binary representation of data; may be instantiated explicitly

Parameters

T	typename of data to be read

5.23.2.2 read() [2/3]

```
char * TMyFStream::read (
            size_t len )
```

read a char array

Parameters

len	length in bytes

Returns

the character array read

5.23.2.3 read() [3/3]

```
TMyFStream & TMyFStream::read (
            char * c,
            unsigned int len )
```

read bytes

Parameters

c	buffer into which to store the result
len	number of bytes

5.24 TPixel Class Reference

stores R,G,B values of a single pixel

```
#include <bitmap.hpp>
```

Public Member Functions

- **TPixel** ()

  constructor. sets values for R,G,B to 0
- **TPixel** (uint8_t _R, uint8_t _G, uint8_t _B)

constructor. sets values for R,G,B to specified values
- void **set** (uint8_t _R, uint8_t _G, uint8_t _B)

  sets new value for R,G,B
- bool **operator==** (**TPixel** &b)

  euqality operator.
- bool **operator!=** (**TPixel** &b)

  inequality operator

Public Attributes

- uint8_t **R**

  red value: 0<=R<=255
- uint8_t **G**

  green value: 0<=G<=255
- uint8_t **B**

  blue value: 0<=B<=255

5.24.1 Detailed Description

stores R,G,B values of a single pixel

5.24.2 Member Function Documentation

5.24.2.1 operator==()

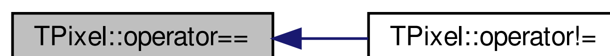
```
bool TPixel::operator== (
              TPixel & b )
```

euqality operator.

Returns

true if R,G,B are all equal to the other pixel's R,G,B

Here is the caller graph for this function:

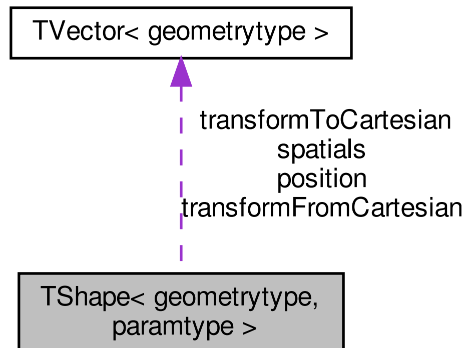


5.25 TShape< geometrytype, paramtype > Class Template Reference

represents a geometric shape

```
#include <shape.hpp>
```

Collaboration diagram for TShape< geometrytype, paramtype >:



Public Member Functions

- **TShape** ()
  constructor
- **TShape** (**TVector**< geometrytype >(∗_transformFromCartesian)(**TVector**< geometrytype >), **TVector**< geometrytype >(∗_transformToCartesian)(**TVector**< geometrytype >))
  constructor
- **TVector**< geometrytype > **paramToCoord** (**TVector**< paramtype > param)
  converts a combiantion of parameter values to a coordinate
- **TVector**< geometrytype > **coordToParam** (**TVector**< geometrytype > coord)
  converts a coordinate to a combination of parameter values
- bool **paramWithinShape** (**TVector**< paramtype > param)
  checks whether the given parameter combination points to a location within the shape
- bool **coordWithinShape** (**TVector**< geometrytype > coord)
  checks whether the given parameter combination points to a location within the shape

Public Attributes

- geometrytype **eps**
  epsilon defines the max deviation allowed for comparisons
- **TVector**< geometrytype > **position**
  location vector / position vector of shape in Cartesian form
- **TVector**< geometrytype > **spatials** [3]
  spatial vectors / directional vectors
- paramtype **param_mins** [3]

min values of each parameter

- paramtype **param_maxs** [3]

  max values of each parameter

- **TVector**< geometrytype >(∗ **transformFromCartesian**)(**TVector**< geometrytype >)

  function pointer to the transformer Cartesian –> new coordinate system

- **TVector**< geometrytype >(∗ **transformToCartesian**)(**TVector**< geometrytype >)

  function pointer to the transformer Cartesian <– new coordinate system

5.25.1 Detailed Description

template<typename geometrytype = double, typename paramtype = double>
class TShape< geometrytype, paramtype >

represents a geometric shape

Can use any coordinate system, e.g. by calling the constructor TShape<>(CYLINDER_COORDINATES) (see definitions). Be careful with periodic coordinate systems, e.g. cylinder coordinates and make sure to use only the first period; orthers, like cartesian or spiral, should make no difficulties.

5.25.2 Member Function Documentation

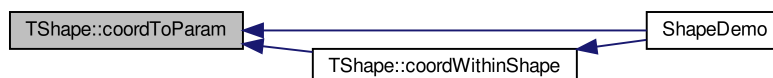
5.25.2.1 coordToParam()

```
template<typename geometrytype = double, typename paramtype = double>
SHAPE_TEMPLATE  TVector< geometrytype >  TShape< geometrytype, paramtype >::coordToParam (
            TVector< geometrytype > coord )
```

converts a coordinate to a combination of parameter values

If there are less than three non-zero spatial vectors, the additional parameters will be zero Here is the call graph for this function:

Here is the caller graph for this function:



5.26 TSolvable Class Reference

```
#include <solvable.hpp>
```

Protected Member Functions

- virtual void **solve** ()=0

  derived classes must be solvable
- virtual void **applyModel** ()=0

  *derived classes must call **applyModel()** (p. 59) on their elements*
- virtual void **advanceTime** (T dt)=0

  *derived classes must inform their elements on advancing time by calling their **advanceTime(T dt)** (p. 59)*
- virtual void **retreatTime** ()=0

  *derived classes must inform their elements about retreated time by calling their **retreatTime()** (p. 59)*

5.26.1 Detailed Description

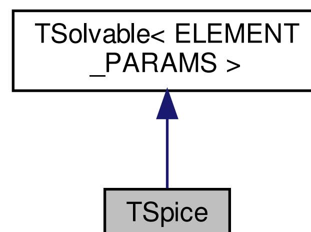
Todo doku

5.27 TSpice Class Reference

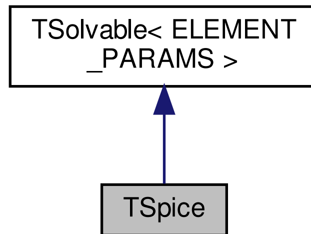
represents a (possibly electrical or magnetic) network

```
#include <spice.hpp>
```

Inheritance diagram for TSpice:



Collaboration diagram for TSpice:



Public Member Functions

- void **add** (**TSpiceElement**< ELEMENT_PARAMS > ∗e)
  adds SPICE element
- void **clear** ()
  *clears elements vector, but leaves **TElement** (p. 10) objects in memory*
- void **addSC** (**TSpiceCondition**< ELEMENT_PARAMS > sc)
  adds SPICE condition (SC)
- void **clearSCs** ()
  *removes all SPICE conditions, but leaves **TSpiceCondition** (p. 62) objects in memory*
- virtual void **relax** ()
  relaxes the network entities (e.g. I=0,U=0)
- virtual void **applyModel** () override
  reqeusts all network entities to apply their models
- virtual void **advanceTime** (T dt) override
  informs all network entities about advanced time
- virtual void **retreatTime** () override
  informs all network entities about retreated time
- virtual void **solve** () override
  solves the problem
- void **printTex** (string fn)
  prints a picture (as Tikz) to the specified file name
- void **saveCsv** (string fn, string delimiter=",")
  saves the result (as csv) to the specified file name

Protected Attributes

- vector< **TSpiceElement**< ELEMENT_PARAMS > ∗ > **elements**
- vector< **TSpiceCondition**< ELEMENT_PARAMS > > **scs**

Static Protected Attributes

- static constexpr double **eps_I** = 1E-10
  smallest recognized size in current
- static constexpr double **eps_U** = 1E-9
  smallest recognized size in voltage

Additional Inherited Members

5.27.1 Detailed Description

represents a (possibly electrical or magnetic) network

This class simply gathers elements (**TSpiceElement** (p. 66)) and conditions (**TSpiceCondition** (p. 62)). It then solves the problem and sets the element's state variables accordingly. It may be used for electrical networks or any other problems that are similarly shaped.

The solving algorithm in the method solve is not fully developed. It may find the correct solution, but needs unreasonable amounts of computing power

of microseconds for a simple network with three two-pole elements on a usual PC). This is due to the genereal description of two-poles. Implementing meta-information, like linearity or lack of hysteresis may greatly improve the algorithm.

Todo improve the solving algorithm (v.s.)

5.27.2 Member Function Documentation

5.27.2.1 addSC()

```
ELEMENT_TEMPLATE void TSpice::addSC (
              TSpiceCondition< ELEMENT_PARAMS > sc )
```

adds SPICE condition (SC)

Todo maybe make this a reference parameter?

5.27.2.2 solve()

```
ELEMENT_TEMPLATE void TSpice::solve ( )  [override], [virtual]
```

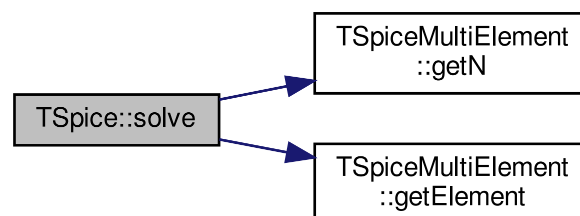
solves the problem

Todo retrieve additional conditions from multi-pole elements

Todo implement preconditioner

Implements **TSolvable**< **ELEMENT_PARAMS** > (p. 59).

Here is the call graph for this function:



5.27.3 Member Data Documentation

5.27.3.1 eps_I

```
constexpr double TSpice::eps_I = 1E-10  [static], [protected]
```

smallest recognized size in current

This has a low effect on computing power demand

5.27.3.2 eps_U

```
constexpr double TSpice::eps_U = 1E-9  [static], [protected]
```

smallest recognized size in voltage

This has a low effect on computing power demand

5.28 TSpiceCondition Class Reference

represents a SPICE condition (Kirchhoff's laws)

```
#include <condition.hpp>
```

Public Member Functions

- **TSpiceCondition** (TSpiceConditionType _type, vector< **TSpiceSingleElement**< ELEMENT_PARAMS > *> _elements, vector< int8_t > _coefficients, T _b)

  TSpiceCondition *(*p. *62).*
- size_t **getN** ()

  number of two-port elements involved
- **TSpiceSingleElement**< ELEMENT_PARAMS > *& **getElement** (size_t idx)

  gets two-port elements by reference
- int8_t & **getCoefficient** (size_t idx)

  gets coeffcicients by reference
- double & **getB** ()

  gets equations right side by reference

Public Attributes

- TSpiceConditionType **type**

  Kirchhoff's current law or voltage law.

Protected Attributes

- vector< **TSpiceSingleElement**< ELEMENT_PARAMS > * > **elements**
  two-ports involved
- vector< int8_t > **coefficients**
  sign (-1 or 1) of equation entities
- double **b**
  equation's right side

5.28.1 Detailed Description

represents a SPICE condition (Kirchhoff's laws)

Based on type, this class either implements a node's currents sum (type = current) or a loop's voltage sum (type = voltage). The vector elements holds a list of all two-ports (**TSpiceSingleElement** (p. 71)) that are involved and coefficients hold the coefficients (either 1 or -1). Note, that **TSpiceSingleElement** (p. 71) can be a pair of clamps of a multi-port element (**TSpiceMultiElement** (p. 67)). Also, the condition may be disturbed, so the equation's right side is storedin b. This class does not provide functionality to apply the condition. That is done by **TSpice** (p. 59).

5.28.2 Constructor & Destructor Documentation

5.28.2.1 TSpiceCondition()

```
ELEMENT_TEMPLATE TSpiceCondition::TSpiceCondition (
            TSpiceConditionType _type,
            vector< TSpiceSingleElement< ELEMENT_PARAMS > *> _elements,
            vector< int8_t > _coefficients,
            T _b )
```

TSpiceCondition (p. 62).

Todo doku

Parameters

_type	
_elements	
_coefficients	
_b	

5.28.3 Member Data Documentation

5.28.3.1 elements

```
vector< TSpiceSingleElement<ELEMENT_PARAMS>*> TSpiceCondition::elements [protected]
```

two-ports involved

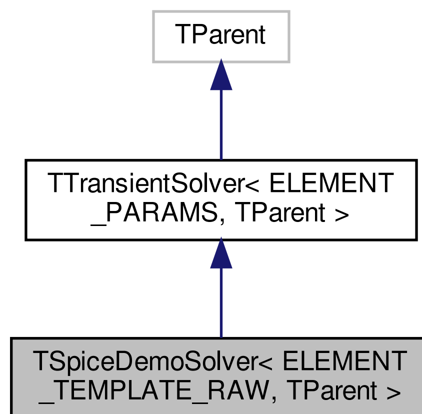
This may be a pair of clamps of a multi-port element

5.29 TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent > Class Template Reference

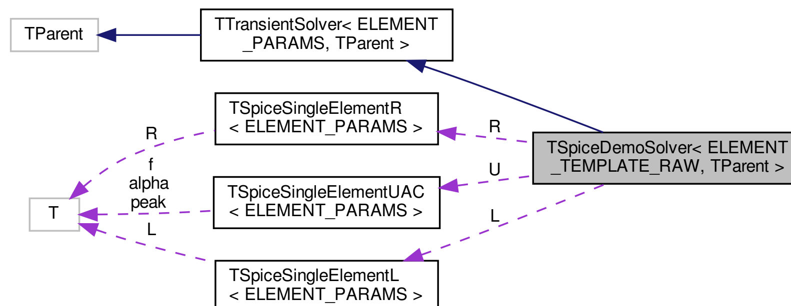
The **TSpiceDemoSolver** (p. 64) class is needed for the **log(T time)** (p. 65) method to work.

```
#include <spice.hpp>
```

Inheritance diagram for TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >:



Collaboration diagram for TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >:



Private Member Functions

- virtual void **log** (T time) override
  write current network elements' states to file stream as CSV

Private Attributes

- fstream * **f**
  filestream to write results as CSV to
- **TSpiceSingleElementUAC**< ELEMENT_PARAMS > * **U**
  voltage source
- **TSpiceSingleElementR**< ELEMENT_PARAMS > * **R**
  resistor
- **TSpiceSingleElementL**< ELEMENT_PARAMS > * **L**
  iductance

Friends

- void **SpiceDemo** (string fn)
  SpiceDemo() *(p. 65) demonstrates the usage networks and the transient solver.*

Additional Inherited Members

5.29.1 Detailed Description

template<ELEMENT_TEMPLATE_RAW, typename TParent>
class TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >

The **TSpiceDemoSolver** (p. 64) class is needed for the **log(T time)** (p. 65) method to work.

It works as an adapter.

See **SpiceDemo()** (p. 65) for further information.

5.29.2 Friends And Related Function Documentation

5.29.2.1 SpiceDemo

```
template<ELEMENT_TEMPLATE_RAW , typename TParent >
void SpiceDemo (
            string fn )  [friend]
```

SpiceDemo() (p. 65) demonstrates the usage networks and the transient solver.

It is not possible to find stationary state solution due to the nonlinear nature of genereally all network entities. However, the **TSpice** (p. 59) class is derived from **TSolvable** (p. 59) and is therefore perfectly solvable. Calling **solve()** (p. 81) on a **TSpice** (p. 59) element will calculate the network's current state based on the network entities' state variables' current values.

Parameters

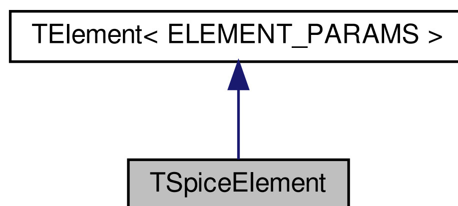
fn	filename to save the results as CSV file to

5.30 TSpiceElement Class Reference

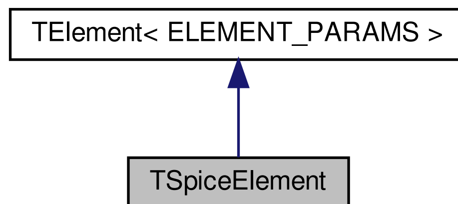
serves as a common ancestor of **TSpiceSingleElement** (p. 71) and **TSpiceMultiElement** (p. 67)

`#include <element.hpp>`

Inheritance diagram for TSpiceElement:



Collaboration diagram for TSpiceElement:



Public Member Functions

- **TSpiceElement** (int8_t max_differential)

Additional Inherited Members

5.30.1 Detailed Description

serves as a common ancestor of **TSpiceSingleElement** (p. 71) and **TSpiceMultiElement** (p. 67)

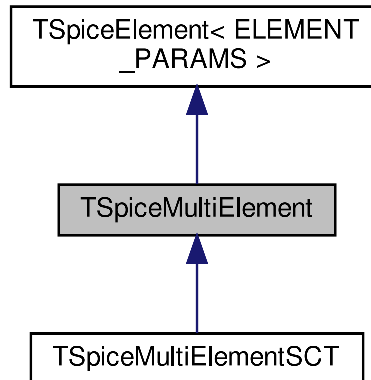
If any functionality or structure needs to be implemented, **TSpiceSingleElement** (p. 71) and **TSpiceMultiElement** (p. 67) do not need to be changed to inherit from a different class. So this is mostly for the future...

5.31 TSpiceMultiElement Class Reference

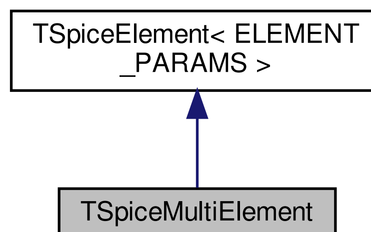
represents a multi-port network element

```
#include <multielement.hpp>
```

Inheritance diagram for TSpiceMultiElement:



Collaboration diagram for TSpiceMultiElement:



Public Member Functions

- **TSpiceMultiElement** (vector< **TSpiceSingleElement**< ELEMENT_PARAMS > *> _elements)
  constructor
- size_t **getN** ()
  number of two-poles contained in this multi-pole
- **TSpiceSingleElement**< ELEMENT_PARAMS > * **getElement** (size_t idx)
  gets idx-th two-port element

- virtual vector< **TSpiceCondition**< ELEMENT_PARAMS > * > **getSCs** ()=0

  A 2n-pole element consisting of n two-pole elements must define n SPICE conditions (e.g. Kirchhoff current/voltage law).
- virtual void **applyModel** ()

  calculates U fom I or vice versa
- virtual void **relax** ()

  relaxes the element's inner material (I=0, U=0)
- virtual void **advanceTime** (double dt)

  tells the model that time has passed (dt in s) since the last calculation of I or U

Protected Attributes

- vector< **TSpiceSingleElement**< ELEMENT_PARAMS > * > **elements**

  all two-port elements that the entire 2n-pole element consists of

5.31.1 Detailed Description

represents a multi-port network element

Though created with the clear intention to use in electrical networks, this may actually in any problem describable as a network. It holds several two-port elements and is defined by their behavior. However, derived classes may overwrite applyModel to define custom behavior.

5.31.2 Member Function Documentation

5.31.2.1 advanceTime()

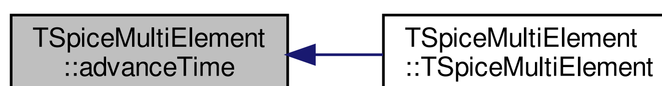
```
ELEMENT_TEMPLATE void TSpiceMultiElement::advanceTime (
            double dt )  [virtual]
```

tells the model that time has passed (dt in s) since the last calculation of I or U

The default implementation calls **relax()** (p. 69) on all two-port elemnts. Derived classes may want to implement sth. more useful.

Reimplemented in **TSpiceMultiElementSCT** (p. 71).

Here is the caller graph for this function:



5.31.2.2 applyModel()

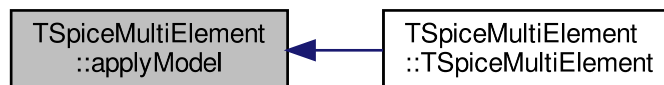
```
ELEMENT_TEMPLATE void TSpiceMultiElement::applyModel ( ) [virtual]
```

calculates U fom I or vice versa

The default implementation calls **applyModel()** (p. 68) on all two-port elemnts. Derived classes may want to implement sth. more useful.

Reimplemented in **TSpiceMultiElementSCT** (p. 71).

Here is the caller graph for this function:

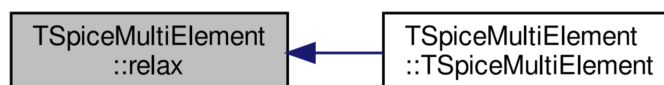


5.31.2.3 relax()

```
ELEMENT_TEMPLATE void TSpiceMultiElement::relax ( ) [virtual]
```

relaxes the element's inner material (I=0, U=0)

The default implementation calls **relax()** (p. 69) on all two-port elemnts. Derived classes may want to implement sth. more useful. Here is the caller graph for this function:

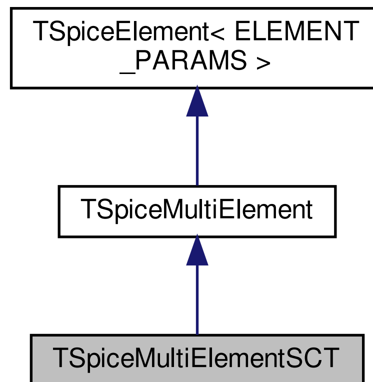


5.32 TSpiceMultiElementSCT Class Reference

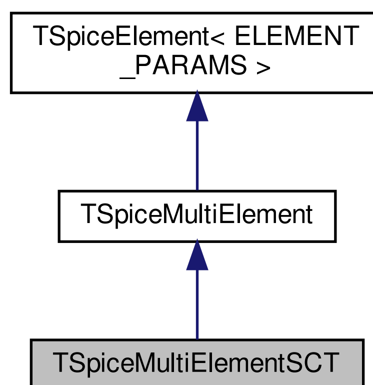
The **TSpiceMultiElementSCT** (p. 70) class.

`#include <sct.hpp>`

Inheritance diagram for TSpiceMultiElementSCT:



Collaboration diagram for TSpiceMultiElementSCT:



Public Member Functions

- virtual void **applyModel** ()

  calculates U fom I or vice versa
- virtual void **advanceTime** (double dt)

  tells the model that time has passed (dt in s) since the last calculation of I or U

Additional Inherited Members

5.32.1 Detailed Description

The **TSpiceMultiElementSCT** (p. 70) class.

5.32.2 Member Function Documentation

5.32.2.1 advanceTime()

```
virtual void TSpiceMultiElementSCT::advanceTime (
            double dt ) [inline], [virtual]
```

tells the model that time has passed (dt in s) since the last calculation of I or U

The default implementation calls **relax()** (p. 69) on all two-port elemnts. Derived classes may want to implement sth. more useful.

Reimplemented from **TSpiceMultiElement** (p. 68).

5.32.2.2 applyModel()

```
virtual void TSpiceMultiElementSCT::applyModel ( ) [inline], [virtual]
```

calculates U fom I or vice versa

The default implementation calls **applyModel()** (p. 71) on all two-port elemnts. Derived classes may want to implement sth. more useful.

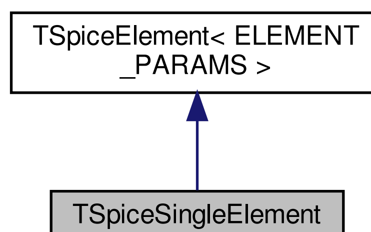
Reimplemented from **TSpiceMultiElement** (p. 68).

5.33 TSpiceSingleElement Class Reference

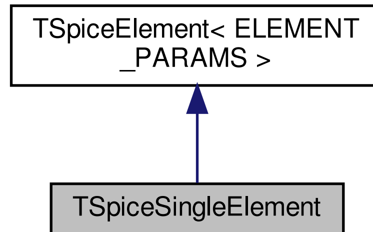
represents a two-port network element

```
#include <singleelement.hpp>
```

Inheritance diagram for TSpiceSingleElement:



Collaboration diagram for TSpiceSingleElement:



Public Member Functions

- **TSpiceSingleElement** (int8_t max_differential)
  constructor
- virtual ∼**TSpiceSingleElement** ()
  destructor
- void **setI** (T _I)
  set current (state variable)
- void **setU** (T _U)
  set voltage (state variable)
- T **getI** ()
  get current (state variable)
- T **getU** ()
  get voltage (state variable)
- virtual void **applyModel** ()=0
  calculates U fom I or vice versa
- virtual void **relax** ()
  relaxes the element's inner material (I=0, U=0)
- virtual void **advanceTime** (T dt)
  tells the model that time has passed (dt in s) since the last calculation of I or U

Protected Member Functions

- TSpiceSingleElementCalcMode **getCalcMode** ()
  whether I or U is preset, the other one will then be calculated by ***applyModel()*** (p. *72)*

Protected Attributes

- TStateVariable< T > **I**
- TStateVariable< T > **U**
  current state of I and U at this two-pole element (state variables)

Private Attributes

- TSpiceSingleElementCalcMode **calc_mode**

    whether I or U is preset, the other one will then be calculated by ***applyModel()*** (p. 72)

5.33.1 Detailed Description

represents a two-port network element

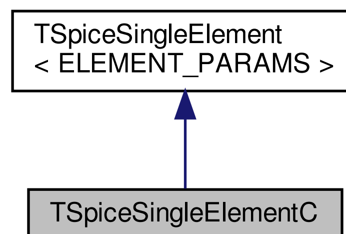
Though created with the clear intention to use in electrical networks, this may actually in any problem describable as a network. It holds a potential (voltage, U) and a flow (current, I) state variable and needs derived classes to implement applyModel in order to define its behavior.

5.34 TSpiceSingleElementC Class Reference

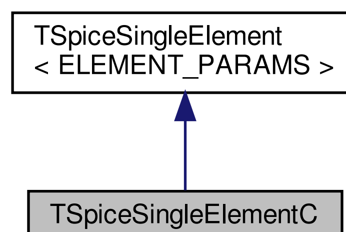
linear capacity is trivial

```
#include <c.hpp>
```

Inheritance diagram for TSpiceSingleElementC:



Collaboration diagram for TSpiceSingleElementC:



Public Member Functions

- **TSpiceSingleElementC** (double _C)
- virtual void **applyModel** ()
  calculates U fom I or vice versa

Private Attributes

- T **C**
  capacitance in F

Additional Inherited Members

5.34.1 Detailed Description

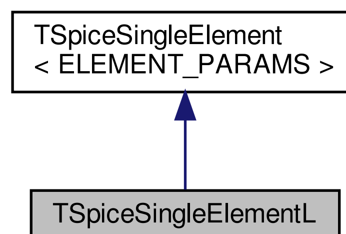
linear capacity is trivial

5.35 TSpiceSingleElementL Class Reference

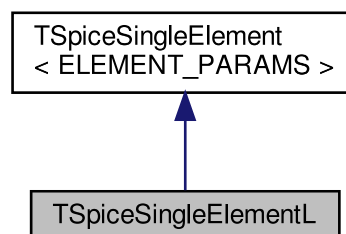
linear inductance is be trivial

```
#include <l.hpp>
```

Inheritance diagram for TSpiceSingleElementL:



Collaboration diagram for TSpiceSingleElementL:



Public Member Functions

- **TSpiceSingleElementL** (T _L)
- virtual void **applyModel** ()
  calculates U fom I or vice versa

Private Attributes

- T **L**
  inductance in H

Additional Inherited Members

5.35.1 Detailed Description

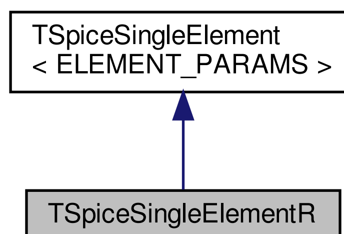
linear inductance is be trivial

5.36 TSpiceSingleElementR Class Reference

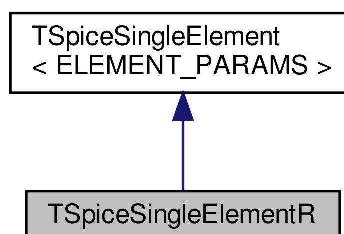
linear resistor is trivial

```
#include <r.hpp>
```

Inheritance diagram for TSpiceSingleElementR:



Collaboration diagram for TSpiceSingleElementR:



Public Member Functions

- **TSpiceSingleElementR** (T _R)
- virtual void **applyModel** () override
  calculates U fom I or vice versa

Private Attributes

- T **R**
  constant ohmic resistance in Ohm

Additional Inherited Members

5.36.1 Detailed Description

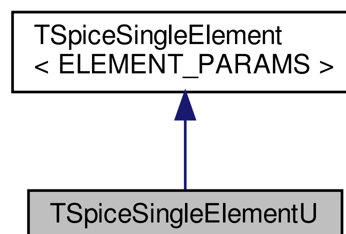
linear resistor is trivial

5.37 TSpiceSingleElementU Class Reference

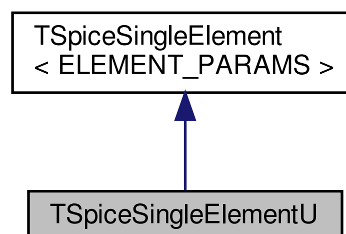
constant voltage source is trivial

`#include <u.hpp>`

Inheritance diagram for TSpiceSingleElementU:



Collaboration diagram for TSpiceSingleElementU:



Public Member Functions

- **TSpiceSingleElementU** (T _fixed)
- virtual void **applyModel** ()

  calculates U fom I or vice versa
- virtual void **relax** ()

  relaxes the element's inner material (I=0, U=0)

Private Attributes

- T **fixed**

  constant voltage

Additional Inherited Members

5.37.1 Detailed Description

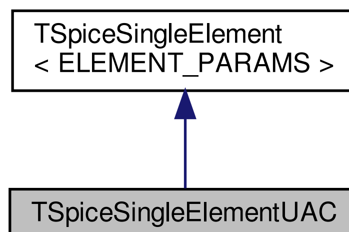
constant voltage source is trivial

5.38 TSpiceSingleElementUAC Class Reference

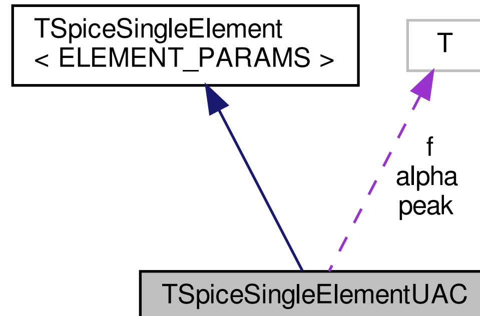
AC voltage source.

```
#include <u.hpp>
```

Inheritance diagram for TSpiceSingleElementUAC:



Collaboration diagram for TSpiceSingleElementUAC:



Public Member Functions

- **TSpiceSingleElementUAC** (T _peak, T _f, T _alpha)

  constructor
- virtual void **applyModel** ()

  calculates U fom I or vice versa
- virtual void **relax** ()

  relaxes the element's inner material (I=0, U=0)

Private Attributes

- T **peak**

  peak voltage
- T **f**

  frequency
- T **alpha**

  angular offset

Additional Inherited Members

5.38.1 Detailed Description

AC voltage source.

Generates a COS wave. Properties are not defined by template parameters but by constructor parameters in order to add flexibility. The extra performance of compiler optimized templates is not needed here.

5.38.2 Constructor & Destructor Documentation

5.38.2.1 TSpiceSingleElementUAC()

```
TSpiceSingleElementUAC::TSpiceSingleElementUAC (
            T _peak,
            T _f,
            T _alpha )  [inline]
```

constructor

Parameters

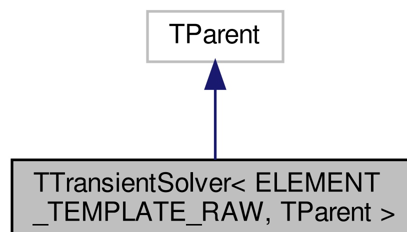
_peak	voltage amplitude
_f	frequency
_alpha	angular offset

5.39 TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent > Class Template Reference

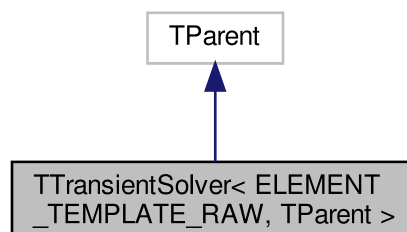
solves any solvable problem for consecutive time steps

```
#include <transientsolver.hpp>
```

Inheritance diagram for TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >:



Collaboration diagram for TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >:



Public Member Functions

- **TTransientSolver** ()
  constructor
- virtual void **log** (T time)
  logs the current state
- void **setTmax** (T _tmax)
  sets end time for simulation
- void **setDt** (T _dt)
  sets time increment for simulation
- virtual void **solve** () override
  solves the transient problem

Protected Attributes

- T **tmax**
  ending time, e.g. 0.1 s
- T **dt**
  time step, e.g. 1E-4 s

5.39.1 Detailed Description

template<ELEMENT_TEMPLATE_RAW, typename TParent>
class TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >

solves any solvable problem for consecutive time steps

A solvable problem (derived from **TSolvable** (p. 59)) may be specified to inherit from. This may be a network or field simulation problem or any other. The solve method is overwritten and will step through a specified time span. It will call the parent's solve method for each step and call log afterwards to give derived classes the possibility to save data at each time step.

Todo TParent must always be derived from **TSolvable** (p. 59)

5.39.2 Member Function Documentation

5.39.2.1 log()

```
template<ELEMENT_TEMPLATE_RAW , typename TParent>
virtual void  TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >::log (
            T time )  [virtual]
```

logs the current state

This method does nothing by default, but derived classes may implement it

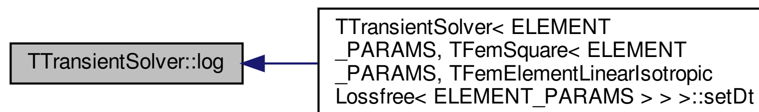
Parameters

time	current state's time

Todo maybe it would make sense to implement this procedurally instead of object-oriented

Reimplemented in **TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >** (p. 65), and **TFemDemo** (p. 42).

Here is the caller graph for this function:

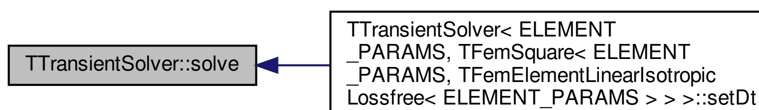


5.39.2.2 solve()

```
template<ELEMENT_TEMPLATE_RAW , typename TParent>
virtual void TTransientSolver< ELEMENT_TEMPLATE_RAW, TParent >::solve ( ) [override], [virtual]
```

solves the transient problem

Method is to be called after setting the start vector. The start vector is set by directly setting the elements' state variables. The method applyModel() does not need to be called on the elements. Here is the caller graph for this function:

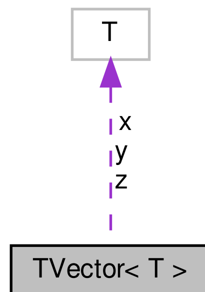


5.40 TVector< T > Class Template Reference

3-dimensional vector of selectable data type, eg to represent physical entities

```
#include <vector.hpp>
```

Collaboration diagram for TVector< T >:



Public Member Functions

- **TVector** (T _x, T _y, T _z)
  constructor. inits components with specified values
- **TVector** (T _x, T _y)
  constructor for 2D vector. inits x,y components with specified values, z component with 0
- **TVector** (T preset)
  constuctor. inits all components with the same specified value
- **TVector** ()
  default constructor. inits all components with 0
- T & **component** (const int i)
- double **norm** (const int i) const

Public Attributes

- T **x**
  x component
- T **y**
  y component
- T **z**
  z component

5.40.1 Detailed Description

template<typename T = double>
class TVector< T >

3-dimensional vector of selectable data type, eg to represent physical entities

Parameters

T	data type of vector components

5.40.2 Member Function Documentation

5.40.2.1 component()

```
template<typename T = double>
T& TVector< T >::component (
            const int i ) [inline]
```

< gets i-th component by reference

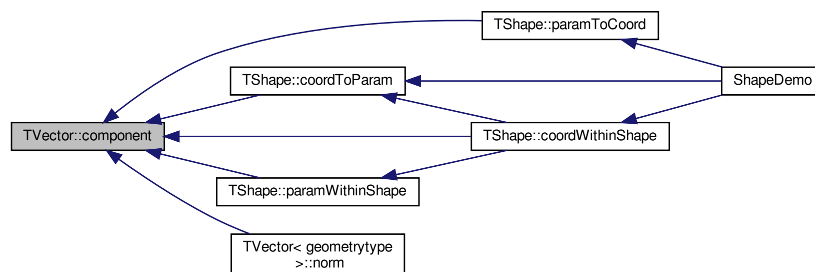
Parameters

i	component index (starting from 0)

Returns

reference to component

Here is the caller graph for this function:



5.40.2.2 norm()

```
template<typename T = double>
double TVector< T >::norm (
            const int i ) const [inline]
```

< calculates norm

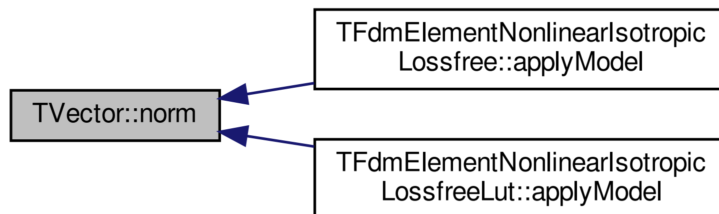
Parameters

i	norm, eg i=2 for euclid, i=-1 for infinity

Returns

calculated norm value (always of type double)

Here is the caller graph for this function:



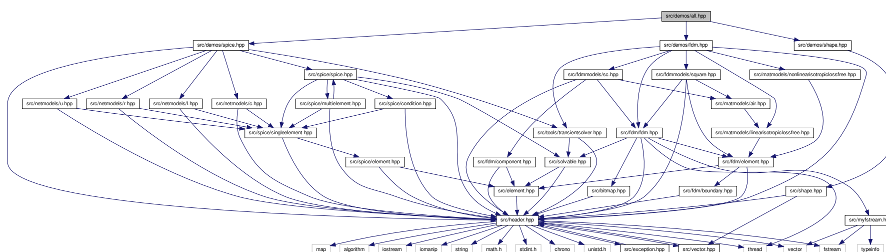
6 File Documentation

6.1 src/demos/all.hpp File Reference

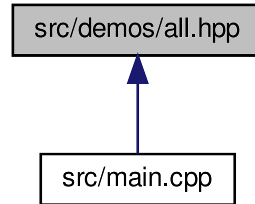
This file includes all (currently working) demos' files.

```
#include "src/demos/spice.hpp"
#include "src/demos/fdm.hpp"
#include "src/demos/shape.hpp"
```

Include dependency graph for all.hpp:



This graph shows which files directly or indirectly include this file:



6.1.1 Detailed Description

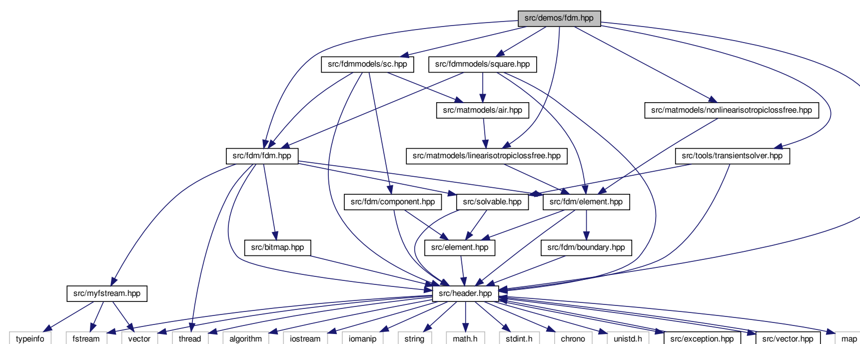
This file includes all (currently working) demos' files.

6.2 src/demos/fdm.hpp File Reference

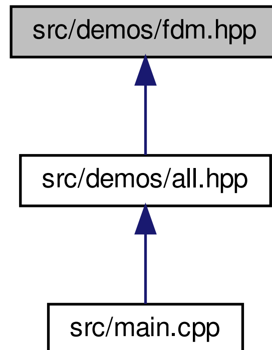
This file holds the FdmDemo function.

```
#include "src/header.hpp"
#include "src/tools/transientsolver.hpp"
#include "src/fdm/fdm.hpp"
#include "src/fdmmodels/square.hpp"
#include "src/fdmmodels/sc.hpp"
#include "src/matmodels/linearisotropiclossfree.hpp"
#include "src/matmodels/nonlinearisotropiclossfree.hpp"
```

Include dependency graph for fdm.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- template<typename T = double, TRungeKuttaMethod method = TRungeKuttaMethod::PseudoEuler, size_t nH = 30, size_t nW = 30, typename matmodeltype = TFdmElementLinearIsotropicLossfree<ELEMENT_PARAMS,50>>
  void **FdmDemo** (string fn="/home/ilka/test/fdm.csv")

  FdmDemo() *(p. 86) demonstrates the usage of the* ***TFdm*** *(p. 18) class.*

6.2.1 Detailed Description

This file holds the FdmDemo function.

6.2.2 Function Documentation

6.2.2.1 FdmDemo()

```
template<typename T = double, TRungeKuttaMethod method = TRungeKuttaMethod::PseudoEuler, size←
_t nH = 30, size_t nW = 30, typename matmodeltype = TFdmElementLinearIsotropicLossfree<ELEME←
NT_PARAMS,50>>
void FdmDemo (
            string fn = "/home/ilka/test/fdm.csv" )
```

FdmDemo() (p. 86) demonstrates the usage of the **TFdm** (p. 18) class.

It creates a specific FDM problem class (Single Cross), solves it, saves its outcome and deletes it afterwards.

Parameters

nH	number of FDM elements in vertical direction
nW	number of FDM elements in horizontal direction
matmodeltype	type derived from **TFdmElement** (p. 24) specifying material model
fn	file name to save result to (as CSV)

Here is the call graph for this function:

FdmDemo

TFdm< FDM_PARAMS >::solve

TFdm< FDM_PARAMS >::saveCsv

6.3 src/demos/fem.hpp File Reference

This file holds the FemDemo function and its assets.

```
#include "src/header.hpp"
#include "src/transientsolver.hpp"
#include "src/fem/fem.hpp"
#include "src/femmodels/square.hpp"
#include "src/matmodels/linearisotropiclossfree.h"
```

Include dependency graph for fem.hpp:

Classes

- class **TFemDemo**

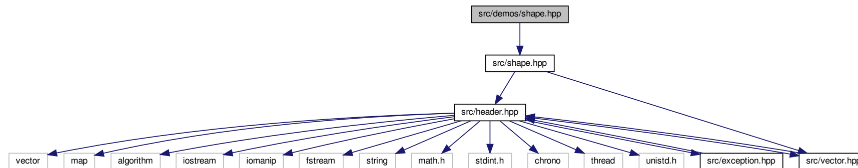
6.3.1 Detailed Description

This file holds the FemDemo function and its assets.

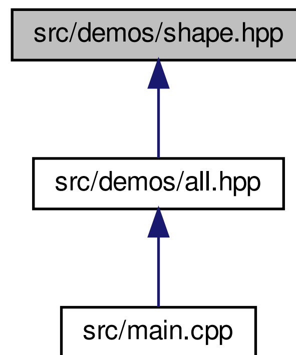
6.4 src/demos/shape.hpp File Reference

```
#include "src/shape.hpp"
```

Include dependency graph for shape.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void **ShapeDemo** ()

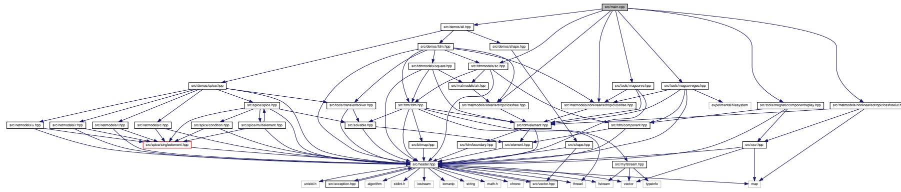
    This demo shows the use of parametric shapes.

6.5 src/main.cpp File Reference

```
#include "src/demos/all.hpp"
#include "src/matmodels/linearisotropiclossfree.hpp"
#include "src/matmodels/nonlinearisotropiclossfree.hpp"
#include "src/matmodels/nonlinearisotropiclossfreelut.hpp"
#include "src/fdmmodels/sc.hpp"
#include "src/tools/magcurve.hpp"
#include "src/tools/magcurvegeo.hpp"
```

```
#include "src/tools/magneticcomponentreplay.hpp"
```

Include dependency graph for main.cpp:



Functions

- int **main** ()

Variables

- bool **VERBOSE** = false
- constexpr char **measurement_NO10par_fn** [ ] = "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SST-par.csv"
- constexpr char **measurement_NO10orth_fn** [ ] = "/mnt/laptop/Documents/Masterarbeit/Messungen/NO10-SST-orth.csv"
- constexpr char **measurement_NO20_fn** [ ] = "/mnt/laptop/Documents/Masterarbeit/Messungen/NO20_↩ fine.csv"
- constexpr char **measurement_delimiter** [ ] = ";"

6.5.1 Function Documentation

6.5.1.1 main()

```
int main ( )
```

Todo doc

Index

~TFdmSc
TFdmSc, 35

addSC
TSpice, 61
advanceTime
TElement, 11
TFdmElement, 25
TSpiceMultiElement, 68
TSpiceMultiElementSCT, 71
applyModel
TElement, 12
TFdmElement, 26
TFdmElementNonlinearIsotropicLossfree, 29
TFdmElementNonlinearIsotropicLossfreeLut, 32
TFdmSc, 35
TSpiceMultiElement, 68
TSpiceMultiElementSCT, 71

bc
TFdm, 21
buildMesh
TFem, 40

calc_mode_ortho
TMagneticComponentOrtho, 53
component
TVector, 83
coordToParam
TShape, 58

deleteStateVariable
TElement, 12
demos/fdm.hpp
FdmDemo, 86

elements
TSpiceCondition, 63
eps_geometry
TFdm, 21
TFem, 41
eps_H
TFdm, 21
TFem, 41
eps_I
TSpice, 62
eps_t
TElement, 14
eps_U
TSpice, 62

FdmDemo
demos/fdm.hpp, 86
first_solve_done
TFdm, 22

getCalcModeOrtho
TMagneticComponentOrtho, 53
getMu
TFdmElement, 26
TFdmElementLinearIsotropicLossfree, 28
getNeighborsReference
TFemElement, 45
getStateVariable
TElement, 12
getStateVariablesDifferential
TElement, 13
getStateVariablesIntegral
TElement, 13

initStateVariable
TElement, 13

log
TTransientSolver, 80

main
main.cpp, 89
main.cpp
main, 89
mu_cache
TFdm, 22

newphim
TFdm, 22
norm
TVector, 83

operator==
TPixel, 56

phim
TFdm, 22

read
TMyFStream, 54, 55
relax
TElement, 14
TFdmElement, 26
TSpiceMultiElement, 69
run
TFemDemo, 42

saveDetails
TFdmSc, 36
TMagneticComponent, 50
setStateVariable
TElement, 14
solve
TEquationSystem, 16
TFem, 40
TSpice, 61
TTransientSolver, 81
solveRecursively

- TFdm, 21

SpiceDemo
- TSpiceDemoSolver, 65

src/demos/all.hpp, 84
src/demos/fdm.hpp, 85
src/demos/fem.hpp, 87
src/demos/shape.hpp, 88
src/main.cpp, 88

TBitmap, 8
- TBitmap, 9

TConstDouble< numerator, denominator >, 10
TElement, 10
- advanceTime, 11
- applyModel, 12
- deleteStateVariable, 12
- eps_t, 14
- getStateVariable, 12
- getStateVariablesDifferential, 13
- getStateVariablesIntegral, 13
- initStateVariable, 13
- relax, 14
- setStateVariable, 14
- time_idx, 14

TEquationSystem
- solve, 16

TEquationSystem< T >, 15
TEquationSystemLU< T >, 16
TException, 17
- TException, 17

TFdm, 18
- bc, 21
- eps_geometry, 21
- eps_H, 21
- first_solve_done, 22
- mu_cache, 22
- newphim, 22
- phim, 22
- solveRecursively, 21
- TFdm, 21
- TPreconditioner, 20

TFdmBoundaryCondition
- TFdmBoundaryCondition, 23

TFdmBoundaryCondition< T >, 22
TFdmElement, 24
- advanceTime, 25
- applyModel, 26
- getMu, 26
- relax, 26

TFdmElementLinearIsotropicLossfree
- getMu, 28

TFdmElementLinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur >, 27
TFdmElementNonlinearIsotropicLossfree
- applyModel, 29

TFdmElementNonlinearIsotropicLossfree< ELEMENT_TEMPLATE_RAW, mur, B_sat_uT, mur_env >, 28
TFdmElementNonlinearIsotropicLossfreeLut
- applyModel, 32
- TFdmElementNonlinearIsotropicLossfreeLut, 32

TFdmElementNonlinearIsotropicLossfreeLut< ELEMENT_TEMPLATE_RAW, lut_type, fn, delimiter >, 30
TFdmSc
- ∼TFdmSc, 35
- applyModel, 35
- saveDetails, 36
- TFdmSc, 34

TFdmSc< FDM_TEMPLATE_RAW, matmodeltype >, 33
TFdmSquare< FDM_TEMPLATE_RAW, matmodeltype >, 36
TFem, 38
- buildMesh, 40
- eps_geometry, 41
- eps_H, 41
- solve, 40
- TPreconditioner, 40

TFemBoundaryCondition< T >, 41
TFemDemo, 42
- run, 42

TFemElement, 43
- getNeighborsReference, 45

TFemSC, 46
- TFemSC, 47

TFemSquare< ELEMENT_TEMPLATE_RAW, matmodeltype >, 47
TMagneticComponent, 48
- saveDetails, 50
- TMagneticComponent, 50

TMagneticComponentOrtho, 51
- calc_mode_ortho, 53
- getCalcModeOrtho, 53
- TMagneticComponentOrtho, 52

TMyFStream, 53
- read, 54, 55

TPixel, 55
- operator==, 56

TPreconditioner
- TFdm, 20
- TFem, 40

TShape
- coordToParam, 58

TShape< geometrytype, paramtype >, 57
TSolvable, 59
TSpice, 59
- addSC, 61
- eps_I, 62
- eps_U, 62
- solve, 61

TSpiceCondition, 62
- elements, 63
- TSpiceCondition, 63

TSpiceDemoSolver
- SpiceDemo, 65

TSpiceDemoSolver< ELEMENT_TEMPLATE_RAW, TParent >, 64
TSpiceElement, 66
TSpiceMultiElement, 67
  advanceTime, 68
  applyModel, 68
  relax, 69
TSpiceMultiElementSCT, 70
  advanceTime, 71
  applyModel, 71
TSpiceSingleElement, 71
TSpiceSingleElementUAC, 77
  TSpiceSingleElementUAC, 78
TSpiceSingleElementC, 73
TSpiceSingleElementL, 74
TSpiceSingleElementR, 75
TSpiceSingleElementU, 76
TTransientSolver
  log, 80
  solve, 81
TTransientSolver< ELEMENT_TEMPLATE_RAW, T↩ Parent >, 79
TVector
  component, 83
  norm, 83
TVector< T >, 82
time_idx
  TElement, 14