

Bachelorarbeit

Tobias Wulf

Winkelmessung durch magnetische Sensor-Arrays und
Toleranzkompensation mittels Gauß-Prozess

Tobias Wulf

Winkelmessung durch magnetische Sensor-Arrays und Toleranzkompensation mittels Gauß-Prozess

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Elektro- und Informationstechnik*
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuer Prüfer: Prof. Dr. Karl-Ragmar Riemschneider
Zweitgutachter: Prof. Dr. Klaus Jünemann

Eingereicht am: TT. Monat Jahr

Tobias Wulf

Thema der Arbeit

Winkelmessung durch magnetische Sensor-Arrays und Toleranzkompensation mittels Gauß-Prozess

Stichworte

Sensor-Array Simulation, Dipol, Magnetfeld, Kugelmagnetapproximation, TMR, TDK TAS2141, AMR, NXP KMZ60, Toleranzkompensation, Gauß-Prozess, Kovarianzmatrix, Regression, Winkelvorhersage

Kurzzusammenfassung

...

Tobias Wulf

Title of Thesis

Angular Measurement by Magnetic Sensor Arrays and Tolerance Compensation by Gaussian Process

Keywords

Sensor Array Simulation, Dipole, Magnetic Field, Spherical Magnet Approximation, TMR, TDK TAS2141, AMR, NXP KMZ60, Tolerance Compensation, Gaussian Process, Covariance Matrix, Regression, Angular Prediction

Abstract

...

Inhaltsverzeichnis

1 Motivation 0.0.1 17.02.2021	1
1.1 Stand der Vorarbeiten	2
1.2 Zielstellung	7
2 Grundlagen 0.0.3 13.04.2021	8
2.1 Kreisdarstellung des klassischen Anwendungsfalls	8
2.2 Euklidischer Abstand in Normschreibweise	11
2.3 Magnetische Sensoren und Drehwinkelerfassung	13
2.4 Kennfeldmethode zur Charakterisierung von Sensoren	17
2.5 Prinzip des Sensor-Arrays	21
2.6 Sensor-Array-Simulation über Dipol-Feldgleichung	25
2.7 Gauß-Prozesse für Regressionsverfahren	29
3 Software-Entwicklung für Optimierungsexperimente 0.0.2 19.02.2021	32
3.1 Aufgabe und Funktionen der Software	32
3.2 Aufbau und Vorgehen	33
3.2.1 Sensor-Array-Simulation	34
3.2.2 Gauß-Prozess-Regression	35
3.3 Simulationsprozesse	37
3.3.1 Sensor-Array-Simulation	37
3.3.2 Gauß-Prozess-Regression	38
4 Erprobungs- und Optimierungsexperimente 0.0.1 13.01.2021	41
4.1 Festlegung des Startpunktes	41
4.2 Festlegung des Verfahrweges ohne Verkippung	42
4.3 Simulationsdurchführung	42
5 Auswertung 0.0.1 13.01.2021	43
5.1 Gegenüberstellung der GPR-Modelle	43

6 Zusammenfassung und Bewertung 0.0.1 13.01.2021	44
Algorithmenverzeichnis	46
Glossar	47
Abkürzungen	49
Literatur	50
Anhang	52
A TDK TAS2141-AAAB Kennfelddatensatz 0.0.1 29.03.2021	52
B Sensor-Array-Simulation Implementierung 0.0.1 07.04.2021	55
C Gauß-Prozess-Regression Implementierung 0.0.1 13.04.2021	59
C.1 Modellinitialisierung	60
C.2 Modelloptimierung	70
C.3 Modellvorhersagen	72
C.4 Modellgeneralisierung	75
D Genutzte Software 0.0.3 08.01.2021	77
E Software-Dokumentation 0.0.5 14.04.2021	78
E.1 GaussianProcessDipoleSimulation	79
E.2 Workflows	81
E.2.1 Project Preparation	82
E.2.2 Project Structure	90
E.2.3 Git Feature Branch Workflow	94
E.2.4 Documentation Workflow	95
E.2.5 Simulation Workflow	98
E.3 Executable Scripts	99
E.3.6 publishProjectFilesToHTML	100
E.3.7 generateConfigMat	103
E.3.8 generateSimulationDatasets	111
E.3.9 deleteSimulationDatasets	113
E.3.10 deleteSimulationPlots	114

E.3.11	exportPublishedToPdf	115
E.3.12	demoGPRModule	120
E.3.13	investigateKernelParameters	124
E.3.14	compareGPRKernels	130
E.4	Source Code	134
E.4.15	sensorArraySimulation	135
E.4.15.1	rotate3DVector	137
E.4.15.2	generateDipoleRotationMoments	139
E.4.15.3	generateSensorArraySquareGrid	143
E.4.15.4	computeDipoleH0Norm	146
E.4.15.5	computeDipoleHField	148
E.4.15.6	simulateDipoleSquareSensorArray	152
E.4.16	gaussianProcessRegression	157
E.4.16.7	initGPR	159
E.4.16.8	initGPROptions	161
E.4.16.9	initTrainDS	163
E.4.16.10	initKernel	165
E.4.16.11	initKernelParameters	166
E.4.16.12	tuneKernel	168
E.4.16.13	computeTuneCriteria	170
E.4.16.14	predFrame	171
E.4.16.15	predDS	173
E.4.16.16	lossDS	175
E.4.16.17	optimGPR	177
E.4.16.18	computeOptimCriteria	179
E.4.16.19	kernelQFCAPX	181
E.4.16.19.1	QFCAPX	182
E.4.16.19.2	meanPolyQFCAPX	184
E.4.16.19.3	initQFCAPX	186
E.4.16.20	kernelQFC	188
E.4.16.20.4	QFC	189
E.4.16.20.5	meanPolyQFC	191
E.4.16.20.6	initQFC	193
E.4.16.21	basicMathFunctions	195
E.4.16.21.7	sinoids2angles	196
E.4.16.21.8	angles2sinoids	199

E.4.16.21.9	decomposeChol	201
E.4.16.21.10	frobeniusNorm	203
E.4.16.21.11	computeInverseMatrixProduct	205
E.4.16.21.12	computeTransposeInverseProduct	207
E.4.16.21.13	addNoise2Covariance	209
E.4.16.21.14	computeAlphaWeights	211
E.4.16.21.15	computeStdLogLoss	212
E.4.16.21.16	computeLogLikelihood	214
E.4.16.21.17	estimateBeta	216
E.4.17	util	218
E.4.17.22	removeFilesFromDir	219
E.4.17.23	publishFilesFromDir	221
E.4.17.24	plotFunctions	223
E.4.17.24.18	plotTDKCharDataset	225
E.4.17.24.19	plotTDKCharField	230
E.4.17.24.20	plotTDKTransferCurves	234
E.4.17.24.21	plotKMZ60CharDataset	238
E.4.17.24.22	plotKMZ60CharField	243
E.4.17.24.23	plotKMZ60TransferCurves	247
E.4.17.24.24	plotDipoleMagnet	251
E.4.17.24.25	plotSimulationDataset	255
E.4.17.24.26	plotSingleSimulationAngle	264
E.4.17.24.27	plotSimulationSubset	271
E.4.17.24.28	plotSimulationCosSinStats	278
E.4.17.24.29	plotSimulationDatasetCircle	285
E.5	Datasets	292
E.5.18	TDK TAS2141 Characterization	293
E.5.19	NXP KMZ60 Characterization	297
E.5.20	Config Mat	301
E.5.21	Training and Test Datasets	302
E.6	Unit Tests	305
E.6.22	runTests	307
E.6.23	removeFilesFromDirTest	308
E.6.24	rotate3DVectorTest	309
E.6.25	generateDipoleRotationMomentsTest	311
E.6.26	generateSensorArraySquareGridTest	312

Inhaltsverzeichnis

E.6.27 computeDipoleH0NormTest	313
E.6.28 computeDipoleHFieldTest	314
E.6.29 tiltRotationTest	317
Selbstständigkeitserklärung	319

1 Motivation 0.0.1 17.02.2021

Magnetische Sensoren erlauben die berührungslose Erfassung von Drehzahlen und Winkelinformationen. In modernen Automobilen werden sie unter anderem in der Motorelektronik und im Bremsystem eingesetzt. Neuentwicklungen in der Halbleitertechnik, auf Basis des TMR-Effekts, ermöglichen den Aufbau komplexerer Sensorstrukturen [15]. Die Arbeitsgruppe Sensorik an der HAW Hamburg erforscht moderne Ansätze der Signalverarbeitung für neu gewonnene Sensorstrukturen, verwirklicht als magnetische Sensor-Arrays. Durch den Aufbau von Sensoren als Arrays, bieten sich Möglichkeiten zur Nutzung von Algorithmen und Regressionsverfahren an, die eine Kompensation und Detektion von mechanischen Toleranzen zulassen [19].

Das Verarbeiten einer Vielzahl an Messwerten, bedingt durch Sensor-Array-Strukturen, ist hierbei eine der Herausforderungen die es zu bewältigen gilt. Mit Hilfe moderner Algorithmen, die Ansätze des maschinellen Lernens beinhalten, ergeben sich weitere Problemstellungen in Bezug auf Modellabbildung- und Optimierung. Das übergeordnete Ziel bei der Lösung und Bewältigung der einzelnen Etappen ist die Verbesserung der Messgenauigkeit, indem individuelle Abweichungen des Sensors einem geeigneten Modell antrainiert und Modellparameter optimiert werden.

Moderne Regressionsverfahren liefern dabei statistische Ansätze um geeignete Qualitätskriterien zu bilden und somit trainierte Modelle und ihre Messwertgenauigkeit bewerten zu können, sodass eine Erprobung und Bewertung der erstellten Modelle, mit Toleranz-Abweichungen in den Eingangsdaten, während einer Arbeitsphase untersucht werden können. Diese Arbeit konzentriert dabei auf die simulative Abbildung eines Tunnel-Magnetoresistance (TMR)-Sensormodells für die Drehwinkelerfassung.

1.1 Stand der Vorarbeiten

Zur Erörterung der Ziele und Inhalte dieser Arbeit, findet einleitend eine kurze Zusammenfassung der Vorarbeiten statt. Für den Inhalt relevante Aspekte der Vorarbeiten werden im Kapitel 2 näher beleuchtet und erklärt.

Aktuell steht kein magnetisches TMR-Sensor-Array als integrierte Lösung zur Verfügung. Im Zuge des Forschungsprojekts Signalverarbeitung für Integrated-Sensor-Array (ISAR) sind in der Arbeitsgruppe Sensorik Machbarkeitsstudien erbracht worden [14][17]. Zielstellung war dabei die Untersuchung der generellen Funktionalität und technischen Umsetzung eines magnetischen Sensor-Arrays im Maßstabsmodell.

Platinen-Sensor-Array

Für den Aufbau des Platinen-Sensor-Arrays sind einzelne Winkelsensoren in Sensorbändern angeordnet, Abbildung 1.1. Die Messwerterfassung erfolgt über ein Multiplexing-Verfahren. Eine Steuerung des Multiplexings und die weitere Messwertverarbeitung erfolgt mit Hilfe eines Mikrocontrollers.

Diese Herangehensweise lässt eine Untersuchung der technischen Machbarkeit auf der Basis von aktuell verfügbaren Technologien und Winkelsensoren zu. So ist das Platinen-Sensor-Array in verschiedenen Versionen, mit Anisotrope-Magnetoresistance (AMR)-Sensoren der Firma NXP Semiconductors (KMZ60) [8] und TMR-Sensoren der Firma TDK (TAS2141-AAAB) [11] verwirklicht worden. Das Maßstabsmodell des magnetischen Sensor-Arrays kann zu Vergleichs- und weiteren Erprobungsarbeiten genutzt werden. Diese können beispielsweise in Simulationen und Hardware-Optimierungsarbeiten einfließen.

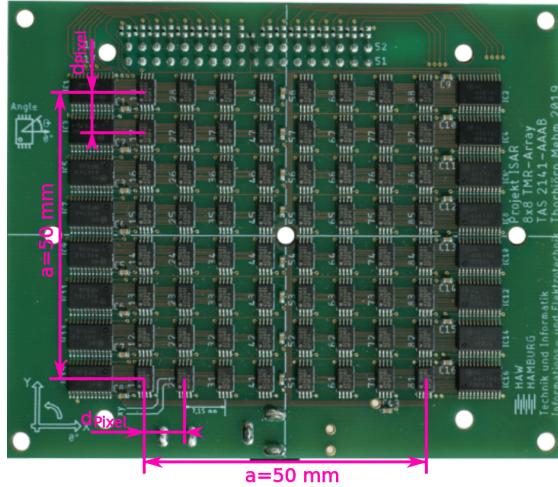


Abbildung 1.1: Platinen-Sensor-Array im Maßstab aufgebaut als 8×8 Sensor-Array, dass als Aufsteckmodul für eine Mikrocontroller getriebene Signalverarbeitung bereitsteht. Die einzelnen Sensoren sind in Sensorbänken angeordnet. Die Anordnung erfolgt in eine linke und rechte Sensorbank pro Reihe auf der Platine. Eine Sensorbank besteht jeweils aus einem Multiplexer-IC und vier daneben liegenden Sensor-ICs. Abbildung entnommen und bearbeitet aus [14].

Simulationsmodell des Sensor-Arrays

Einen weiteren Ansatz, der durch die Arbeitsgruppe Sensorik verfolgt wird, ist die Entwicklung eines Simulationsmodells auf Grundlage von Charakterisierungsdatensätzen. Hierfür wird ein einzelnes Sensor-IC, z.B. der TMR-Sensor TAS2141-AAAB der Firma TDK, nach einer bestimmten Kennfeldmethode [15] charakterisiert. Der so gewonnene Datensatz kann dann, durch geeignete Interpolationsverfahren, in einer Simulation zur Generierung eines magnetischen Sensor-Arrays genutzt werden. In Abbildung 1.2 ist das Kernprinzip des Simulationsansatzes vereinfacht dargestellt. Es wird ein Simulationsmodell aufgebaut, dass Charakterisierungsdatensätze verarbeiten kann und entsprechende Charakteristiken eines einzelnen Sensor-ICs zu einem Sensor-Array interpoliert. Abhängig von weiteren gewählten Eigenschaften des Sensor-Arrays, wie geometrische Anordnung und Größe, produziert das interpolierte Modell Simulationsdatensätze, die das Verhalten des einzelner Sensor-ICs ortsabhängig im Sensor-Array abbilden.

Der Simulationsansatz besitzt ebenfalls den Vorteil Modelle aufzubauen, die sich auf heute zur Verfügung stehenden Technologien beziehen. Weitere Vorteile sind die Mani-

pulationsfähigkeit der Sensor-Array-Geometrie und -Größe. So bieten sich Möglichkeiten magnetische Sensor-Arrays in verschiedenen Maßstäben und geometrischen Formen zu simulieren. Des weiteren können verschiedene Anwendungsszenarien simuliert werden. Eine Problemstellung die sich dabei ergibt, ist die physikalisch sinnvolle Stimulanz des Simulationsmodell. Für das Platinen-Sensor-Array ist im trivialen Anwendungsfall die Stimulanz ein simpler Permanentmagnet. In der Simulation muss eine entsprechende Stimulierung des Sensor-Arrays über magnetische Feldgleichungen gelöst werden [12][15], wobei weitere Problemstellungen zur richtigen Dimensionierung oder Approximation des zu simulierenden Magnetfeldes auftreten.



Abbildung 1.2: Ansatzdarstellung zur Generierung eines Simulationsmodells des magnetischen Sensor-Arrays. Sensor spezifische Charakteristiken (Kennfelder) werden in einem Charakterisierungsdatensatz gespeichert und im Anschluss das Verhalten des Einzelexemplars zu einem Sensor-Array interpoliert. Die Simulation des interpolierten Sensor-Arrays erzeugt eine höhere Abstraktionsebene, deren Ergebnisse wiederum in Simulationsdatensätze gespeichert sind und zur weiteren Analyse und Evaluierung genutzt werden können. Die Abstraktion der Kennfelder soll hier das Prinzip des Simulationsansatzes veranschaulichen. Im Simulationsmodell werden keine Arrays von Kennfeldern aufgebaut, sondern Charakteristiken des einzelnen Kennfeldes entnommen und interpoliert. Die grau unterlegten Abschnitte kennzeichnen Verfahrensschritte, in denen Datensätze zur Verfügung stehen oder erzeugt werden.

Das Sensor-Array-Modell, ob als Platinen-Modell oder Simulationsmodell, repräsentiert im Kontext nur die erste Hälfte eines modernen, vollwertigen Sensor-ICs. Seine Aufgabe besteht darin eine physikalische Anregung (Magnetfeld) in elektrische, analoge Signale umzuwandeln. Dieser Teil eines Sensor-ICs wird zumeist als Sensorkopf bezeichnet, da eine sinnbildliche darunter liegende Einheit die weitere Signalverarbeitung und -Auswertung übernimmt. Es handelt sich dabei um eine anwendungsspezifische integrierte Schaltung, engl. Application-Specific-Integrated-Circuit (ASIC). Beide Teile zusammen, der Sensorkopf und das Signalverarbeitungs-ASIC, bilden ein vollständiges Sensor-IC mit der Fähigkeit zur modernen Signalverarbeitung. Unterstützend zeigt Abbildung 1.3 die allgemeine Aufbaubeschreibung eines Sensor-IC und Unterteilung in Sensorkopf und ASIC, respektive Signalerzeugung und Signalverarbeitung.

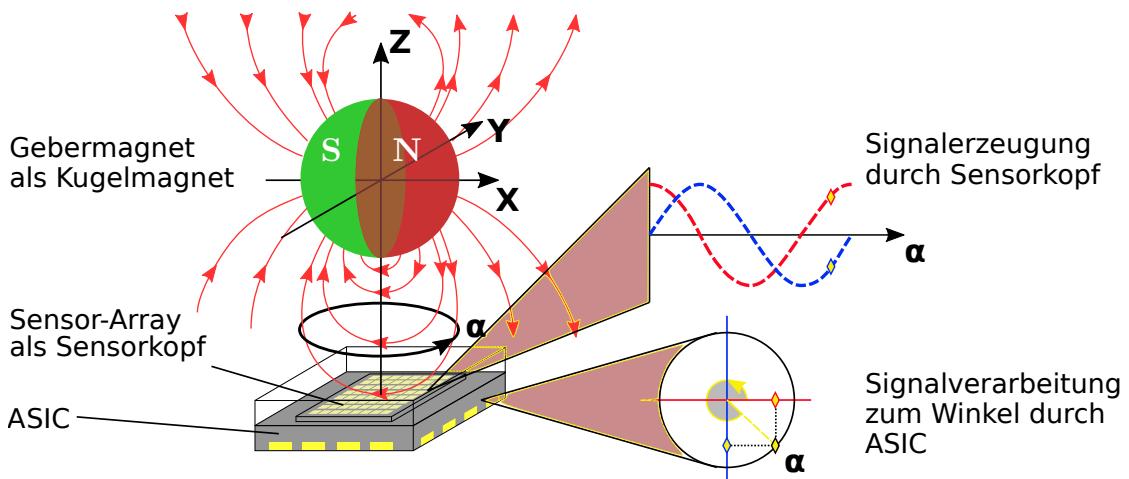


Abbildung 1.3: Veranschaulichung eines vollständigen Sensor-ICs für die Drehwinkel erfassung. Stark vereinfachte Darstellung eines Sensor-IC bestehend aus einem Sensorkopf und ASIC. Zu sehen sind die übergeordneten Aufgaben von Sensorkopf und ASIC. Der Sensorkopf erfasst die physikalische Stimulation (hier Kugelmagnetfeld) und setzt diese in analoge Signale um. Eine anschließende Signalverarbeitung findet im ASIC statt, der die elektrischen Signale zur entsprechenden Winkelausgabe abstrahiert. Dargestellt ist die Signalerzeugung eines einzelnen Punktes auf dem magnetischen Sensor-Arrays. Grafik entnommen und bearbeitet aus [19].

ASIC - Konzeptionierung der Kernfunktionalität

Derzeitig befinden sich die Forschungsprojektarbeiten für einen tauglichen ASIC in der Konzeptionsphase. Die Kernfunktionalität eines ASIC-Designs wird durch ein mathematisches Modell oder Verfahren abgebildet, dass in der Lage ist vom Sensorkopf erzeugte Messwerte adäquat und ausreichend schnell zu verarbeiten. Dabei muss ein solches Modell oder Verfahren grundlegende Eigenschaften des physikalischen Gesamtsystems in sich vereinigen und diese repräsentativ in den Gesamtkontext der Applikation setzen können. Im Kontext dieser Arbeit ist die Sensorapplikation, durch die Drehwinkelerfassung einer kreisförmigen Sensoranregung dargestellt, wie es in Abbildung 1.3 angedeutet ist.

Erfolgte Vorarbeiten der Arbeitsgruppe Sensorik für ein ASIC-Design, umfassen die Entwicklung eines mathematischen Modells und erste theoretische Simulationen [15][18][19]. Die Simulation bindet dabei Datensätze ein, die durch das Sensor-Array-Simulationsmodell erzeugt werden. Das mathematische Modell der ASIC-Kernfunktionalität ist auf Grundlage von Gauß-Prozessen für Regressionsverfahren entwickelt [3] worden. Die bisherigen Simulationsarbeiten beschränken sich auf mathematische Simulationen, die auf eine Gültigkeitsprüfung des mathematischen ASIC-Modells abzielen und Ansätze zur Modellqualifizierung und Qualitätskriterien für die Signalverarbeitung mit beinhalten.

1.2 Zielstellung

- Bezug zu Vorarbeiten
- Verfeinerung des Simulationsmodell des magnetischen Sensor-Arrays
- Skalierung des approximierten Kugelmagnetanregungsfeldes
- Optimierung des mathematischen Model für die ASIC-Kernfunktionalität
- Aufschlüsselung der Modellparameter
- Überführung von Skript basierten Entwürfen in Funktionsmodule
- Modularer Modellaufbau, der Modularerweiterungen zulässt

2 Grundlagen 0.0.3 13.04.2021

Das Fundament für die Drehwinkelerfassung mittels magnetischen Sensor-Array und lernender Signalverarbeitung [15][20][19] bildet das Regressionsverfahren für Gauß-Prozesse [3] und die damit verbundene Abstandsmessung von Winkelpositionen auf einer Kreisbahn. Für eine anschauliche Erklärung der Grundlagen, sollen die Zusammenhänge anhand einfacher Kreisdarstellung des Messprinzips eines einzelnen Winkelsensors gezeigt werden. Sodass dieses später in der Verwendung eines Sensor-Arrays adaptierbar ist und mittels geeigneter Rechen- und Normierungsverfahren auf die Problemstellung eines höherdimensionalen Systems projiziert werden kann.

2.1 Kreisdarstellung des klassischen Anwendungsfalls

Im klassischen Anwendungsfall, zu sehen in Abbildung 2.1, ist ein Gebermagnet räumlich zentriert über einen magnetischen Sensor platziert. Bei Drehung des Gebermagneten rotiert sein Magnetfeld entsprechend mit. Die Rotation findet um die Z -Achse des Gebermagneten statt. Die Nord-Süd-Ausrichtung des Magneten liegt in der X - bzw. Y -Achse des Koordinatensystems [8][11].

Der Winkelsensor misst, die zueinander und zur Rotationsachse orthogonal stehenden, X - und Y -Feldstärkenkomponenten des Gebermagneten H_x und H_y . Diese setzt der Winkelsensor in elektrische Spannungssignale um. Die Winkelstellung α des Magnet wird somit nicht direkt gemessen. Sie kann aber, mittels der gemessenen H_x und H_y Feldstärkenkomponenten, durch einfache Vektorrechnung berechnet werden.

Bei idealer und gleichbleibender Position des Gebermagneten in Relation zum Winkelsensor, liefern die aufgenommen H_x - / H_y -Messwerte eine Cosinus-Funktion $V_{cos}(H_x, H_y)$, sowie eine um 90° zur Cosinus-Funktion phasenverschobene Sinus-Funktion $V_{sin}(H_x, H_y)$. Genaue physikalische Größenzusammenhänge und technische Umsetzung sind dabei vorerst in Darstellung vernachlässigt.

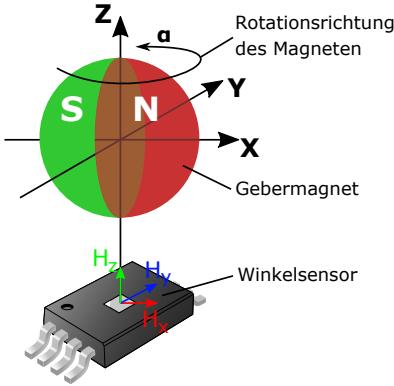


Abbildung 2.1: Klassischer Anwendungsfall für die Drehwinkelerfassung. Zeigt einen, um seine Z -Achse rotierenden, Gebermagneten und einen Winkelsensor in zentrierter und orthogonaler Ausrichtung zur Z -Achse des Magneten. Idealerweise befinden sich Magnet und Sensor, ohne Verkipplungen in X - oder Y -Richtung, parallel zueinander. Grafik entnommen und bearbeitet aus [20].

Durch die Phasenverschiebung der Sinus-Funktion stehen die Messwerte $V_{cos}(H_x, H_y)$ und $V_{sin}(H_x, H_y)$ vektoriell orthogonal zueinander. Bedingt durch die Orthogonalität der Messwerte $V_{cos}(H_x, H_y) \perp V_{sin}(H_x, H_y)$ und gleichförmige Kreisbewegung des Magneten um seine Z -Achse, beschreibt die Winkelmessung in polarer Darstellung eine konstante Kreisbahn. Diese besitzt einen konstanten Bahnradius r und die Winkelstellung α des Gebermagneten [15].

Für eine beliebige Winkelmessung \mathbf{A} , die eine entsprechende Winkelstellung α des Gebermagneten abbildet $\mathbf{A} \mapsto \alpha$, ergibt sich somit folgender vektorieller Zusammenhang in Gleichung 2.1 [19].

$$\underbrace{\begin{pmatrix} H_x(\alpha) \\ H_y(\alpha) \end{pmatrix}}_{\text{Gebermagnetfeld}} \Rightarrow \underbrace{\begin{pmatrix} V_{cos}(H_x, H_y) \\ V_{sin}(H_x, H_y) \end{pmatrix}}_{\text{Winkelsensormesswerte}} = \underbrace{\begin{pmatrix} r \cdot \cos(\alpha) \\ r \cdot \sin(\alpha) \end{pmatrix}}_{\text{Kreisdarstellung}} = \underbrace{\begin{pmatrix} a_x \\ a_y \end{pmatrix}}_{\text{Winkelmessung}} = \mathbf{A}(\alpha) \quad (2.1)$$

Die so erhobenen Winkelmessung \mathbf{A} nach Gleichung 2.1, bildet ein eindimensionales Vektorfeld mit $\{a_x, b_x\} \in \mathbb{R}$ ab. Wobei sich der Bahnradius r für die Kreisdarstellung, aus dem Betrag der Messung $|\mathbf{A}|$, nach Gleichung 2.2 gewinnen lässt.

$$r = |\mathbf{A}| = \sqrt{(V_{cos}(H_x, H_y))^2 + (V_{sin}(H_x, H_y))^2} = \sqrt{a_x^2 + a_y^2} \quad (2.2)$$

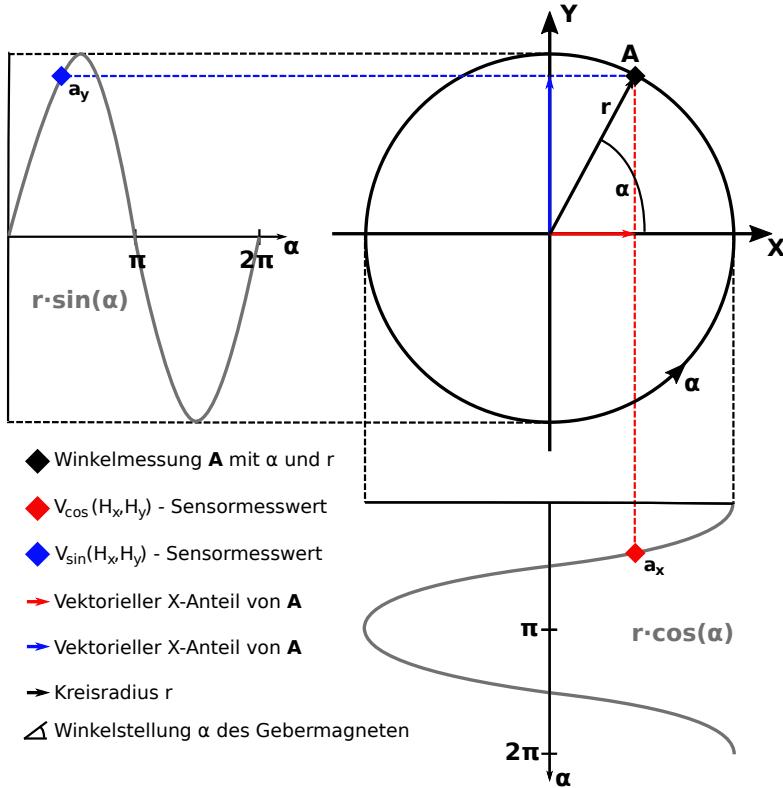


Abbildung 2.2: Kreisdarstellung der Winkelmessung. Als Abbildung der Winkelmessung $\mathbf{A} \mapsto \alpha$ aus Gleichung 2.1. Die Zusammensetzung der Messung erfolgt durch die vom Winkelsensor gemessenen, vektoriellen Anteile für die polare Darstellung der Gebermagnetwinkelstellung.

Der entsprechende Winkel des Gebermagneten lässt sich, mittels Überführung in Polarkoordinaten, nach Gleichung 2.3 zurückrechnen. Die Abbildung 2.2 veranschaulicht den Zusammenhang zwischen Messwerten und Abbildung der Gebermagnetwinkelstellung. Die Sinoiden Messergebnisse sind der arctan2 Funktion zuzuführen. Diese bildet einen Winkel von null bis π ab und besitzt eine Sprungstelle bei π . Der Y -Anteil kann dabei als Entscheider genutzt werden, um eine Abbildung des Winkels auf eine volle Kreisumdrehung (2π) umzusetzen.

$$\alpha = \begin{cases} \text{arctan2}(a_y, a_x) & \text{f. } a_y > 0 \\ \pi & \text{f. } a_y = 0 \\ \text{arctan2}(a_y, a_x) + 2\pi & \text{f. } a_y < 0 \end{cases} \quad (2.3)$$

2.2 Euklidischer Abstand in Normschreibweise

Um adäquate Bezüge bzw. Abstände zwischen einzelnen Messwerten herzustellen ist ein Wechsel der Betrachtungsweise notwendig. Es erleichtert die Handhabung der Problemstellung Vektorbeträge als normierte Längen und Distanzen zu sehen. Betrachtet man die vektoriellen Zusammenhänge, der klassischen Anwendung aus Abschnitt 2.1, in Normschreibweise. Ergibt sich der Radius r für eine Winkelstellung $\mathbf{A} \mapsto \alpha_1$ nach Gleichung 2.4. Die einzelnen Vektorelemente sind entsprechend der Vektor-2-Norm [9] zum Radius r normiert.

$$r = |\mathbf{A}| = \sqrt{a_x^2 + a_y^2} = \sqrt{\sum_{i=1}^n |A_i|^2} = \|\mathbf{A}\|_2 \quad (2.4)$$

Es ist weithin von einer idealen Ausrichtung von Sensor und Gebermagnet wie in Abbildung 2.1 auszugehen. Somit bleibt der Kreisbahn Radius r für eine zweite Winkelstellung mit $\mathbf{B} \mapsto \alpha_2$ konstant.

$$r = \|\mathbf{A}\|_2 = \|\mathbf{B}\|_2 = \text{konst.} \quad (2.5)$$

Der direkte Abstand zwischen den beiden Winkelstellung $\mathbf{A} \mapsto \alpha_1$ und $\mathbf{B} \mapsto \alpha_2$ lässt sich geometrisch über den Satz des Pythagoras ermitteln. Dafür werden Abstandquadrate aus den Einzeldifferenzen der vektoriellen X -/ Y -Anteile gebildet. Das Resultierende Abstandsquadrat bildet mit seiner Kantenlänge dann den Winkelabstand zwischen beiden Winkelstellungen. Abbildung 2.3 veranschaulicht das Vorgehen.

$$\begin{aligned} d_E(\mathbf{A}, \mathbf{B}) &= \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \\ &= \sqrt{\sum_{i=1}^n (A_i - B_i)^2} = \|\mathbf{A} - \mathbf{B}\|_2 \end{aligned} \quad (2.6)$$

Die Überführung in Normschreibweise des Abstandes ergibt nach Gleichung 2.6 eine Vektor-2-Differenznorm und ist allgemein als euklidischer Abstand bekannt. Analog dazu bildet sich das Quadrat nach Gleichung 2.7.

$$d_E^2 \langle \mathbf{A}, \mathbf{B} \rangle = (a_x - b_x)^2 + (a_y - b_y)^2 = \|\mathbf{A} - \mathbf{B}\|_2^2 \quad (2.7)$$

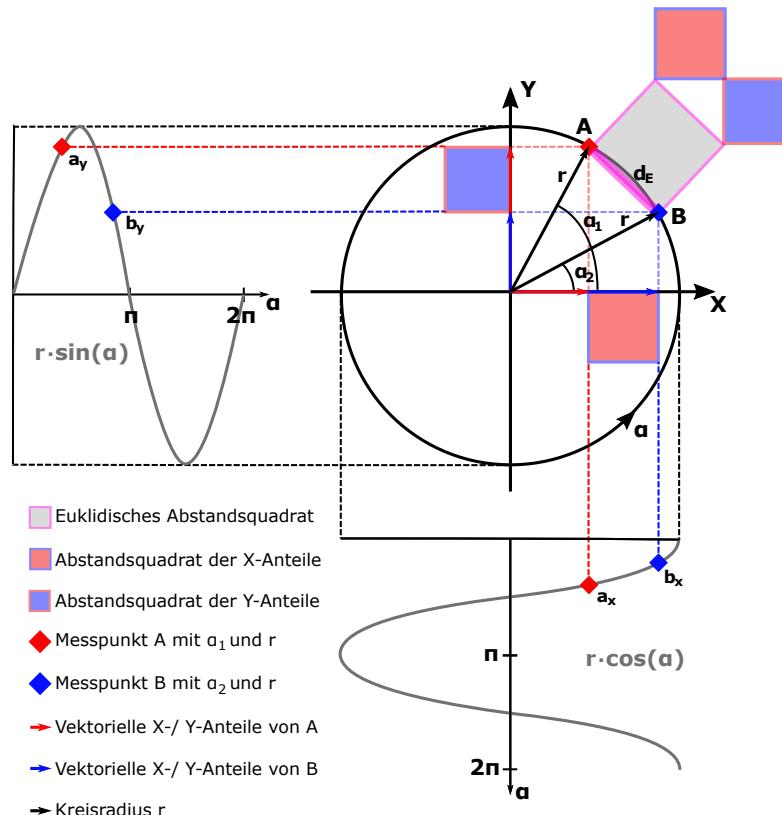


Abbildung 2.3: Allg. Kreisdarstellung des euklidischen Winkelabstands. Die Kreisdarstellung zeigt den euklidischen Winkelabstand zweier Winkelmesspunkte **A** und **B** mit gleichen Kreisradius r . Der euklidische Abstand, bzw. das Abstandsquadrat, zwischen den Winkelposition **A** und **B** ist zerlegt in Abstandsquadratanteile. Die Abstandquadratanteile ergeben sich aus der vektoriellen Zusammensetzung in X -/ Y -Anteile für die einzelnen Messpunkte **A** und **B**.

Für Vektor-2-Normen muss die Dreiecksungleichung aus Gleichung 2.2 [4][9] gelten. Über die Ungleichung lassen sich Einzelnormen approximiert im Vergleich zu Differenznormen zwischen zwei Punkten **A** und **B** darstellen. Dieser Ansatz kann genutzt werden wenn die Bahnradius r nicht mehr konstant ist und somit $\|\mathbf{A}\|_2 \neq \|\mathbf{B}\|_2$ ist. Der Ansatz begünstigt die Projektion von orthogonalen Systemen in höheren Normraum [4] und stellt somit die Grundlage für eine Adaptierung auf ein Sensor-Array dar.

$$\begin{aligned} |\|\mathbf{A}\|_2 - \|\mathbf{B}\|_2| &\leq \|\mathbf{A} \pm \mathbf{B}\|_2 \leq |\|\mathbf{A}\|_2 + \|\mathbf{B}\|_2| \\ &\Updownarrow \\ (\|\mathbf{A}\|_2 - \|\mathbf{B}\|_2)^2 &\leq \|\mathbf{A} \pm \mathbf{B}\|_2^2 \leq (\|\mathbf{A}\|_2 + \|\mathbf{B}\|_2)^2 \end{aligned} \quad (2.8)$$

2.3 Magnetische Sensoren und Drehwinkelerfassung

Magnetische Sensoren besitzen eine lange Tradition in der Automobilindustrie. Sie eignen sich besonders durch die berührungslose Erfassung von mechanischen Bewegungen und die kontaktlose Strommessung für den Einsatz in der Fahrzeugtechnik. Es existieren verschiedene Sensoren, die durch unterschiedliche magnetoresistive Effekte realisiert sind. Dabei bildet sich das Grundprinzip, durch anlegen eines äußeren Magnetfeldes und eine resultierende Änderung des elektrischen Widerstandes eines Materials [21].



Abbildung 2.4: Schichtmodelle dreier magnetoresistiver Effekte. a) AMR-Effekt, schwache Widerstandsänderung. b) GMR-Effekt stärkere Widerstandsänderung. c) TMR-Effekt stärkste Widerstandsänderung. Grafik entnommen aus [10].

AMR-Effekt

In der Mitte des 19. Jahrhunderts entdeckte der britische Physiker William Thomson den anisotropen magnetoresistiven Effekt (AMR). Der AMR-Effekt basiert auf einer von Strom- und Magnetisierungsrichtung abhängigen Streuung von Elektronen in einer einzelnen aktiven Schicht, Teil a) der Abbildung 2.4. Diese Schicht besteht in der Praxis oftmals aus einer Nickel-Eisen-Legierung. Die typische Variation der relativen Widerstandsänderung $\Delta R/R$ liegt im Bereich von 2% bis 3% [21]. Für eine eindeutig Winkelmessung werden zwei Wheatstone'sche Brücken aus dem Schichtmaterial aufgebaut. Die Stromdurchflussrichtung ist horizontal. Bedingt durch den AMR-Effekt ist eine Periodizität von 180° abgedeckt [10][21]. Ein mittels AMR-Effekt entwickelter Sensor für die Drehwinkelerfassung, besitzt daher zwei um 45° verdrehte Wheatstone-Brücken. Durch die schwache Widerstandsänderung des Materials ist eine nachgeschaltet Verstärkerschaltung notwendig [8].

GMR-Effekt

Der riesige magnetoresistive Effekt, engl. Giant-Magnetoresistance (GMR), ist 1988 von Grünberg und Fert entdeckt worden. Beide erhielten dafür 2007 den Nobelpreis für Physik, da unter Ausnutzung des GMR-Effekts sich die Speicherkapazität von Computerfestplatten stark erhöhen lies [10]. Das Minimalprinzip für einen solchen Sensor bildet sich aus zwei magnetischen Dünnschichten, die durch eine nicht magnetische Schicht (z.B. Kupfer) voneinander getrennt sind. Dabei folgt die Magnetisierung der aktiven Schicht (z.B. Nickel-Eisen) einem von außen angelegten Magnetfeld, während die Magnetisierung der zweiten Schicht (z.B. Kobalt-Eisen) durch eine darunter liegende antiferromagnetische Schicht (z.B. Platin-Mangan) fixiert ist. Die Stromdurchflussrichtung bleibt wie beim AMR horizontal [10][21]. Die relativen Widerstandsänderung $\Delta R/R$ ist abhängig von der relativen Ausrichtung der Magnetisierungen in beiden magnetischen Schichten und liegt für einfache Schichtsystem, wie in Teil b) der Abbildung 2.4 gezeigt, bei etwa 10%. Die Herstellung eines GMR-Sensors ist deutlich aufwendiger, als es beim AMR-Effekt der Fall ist. So können aber in Multilagen mit vielfacher Wiederholung der magnetischen Schichten bis zu 80% $\Delta R/R$ erreicht werden [21]. Mit der GMR Technologie aufgebaute Drehwinkelsensoren haben eine Periodizität von 360° und besitzen zwei um 90° verdrehte Wheatstone-Brücken und ebenfalls nachgeschaltete Verstärkereinheiten [13].

TMR-Effekt

Im Jahr 1975 ist der tunnel-magnetoresistive Effekt (TMR) durch M. Jullière entdeckt worden. Im einfachsten Fall, wie Teil c) Abbildung 2.4 zeigt, tritt der Effekt bei Schichtsystemen auf, die durch eine isolierende Schicht (z.B. Magnesiumoxid) getrennt sind [10]. Die Stromdurchflussrichtung ist im Gegensatz zum AMR und GMR vertikal zu den Schichten. Die relativen Widerstandsänderung $\Delta R/R$ erfolgt in Abhängigkeit zur relativen Ausrichtung der Magnetisierungen beider magnetischen Schichten, die an der Isolationsschicht angrenzen.

Wie beim GMR folgt die aktive Schicht einem äußeren Magnetfeld, ebenso ist die zweite Schicht durch eine antiferromagnetische Schicht fixiert [21]. Der zugrunde liegende Effekt ist aber physikalisch ein gänzlich anderer. Hier “tunnelt” der Stromfluss durch die Isolationsschicht. Das ist ein quantenmechanischer Effekt und kann mit Ansätzen der “normalen” Physik nicht mehr erklärt werden. Zurückzuführen ist der Effekt auf die Spin-Polarisation der einzelnen Elektroden eines magnetischen Tunnel-Kontaktes [21]. In praktischen Ausführungen bei Raumtemperatur liegen heute relative Widerstandsänderungen $\Delta R/R$ im Bereich von 30% bis zu 200% und sind somit deutlich höher als beim GMR [21]. Unter Laborbedingungen konnten bei sehr tiefen Temperaturen mittlerweile Widerstandsänderungen bis zu 1000% erreicht werden [10].

Praxistaugliche Ausführungen von TMR-Sensoren stehen erst seit einigen Jahren zur Verfügung. Die Herstellung eines Sensors erfordert einen enormen apparativen Aufwand und Produktionsanlagen mit entsprechenden Fertigkeiten mussten erst entwickelt werden. Der Aufwand ist mit Vorteilen gegenüber dem AMR- und GMR-Sensor entlohnt worden. Somit besitzt ein TMR-Sensorelement einen viel höheren Widerstand bei gleicher Abmessung. AMR/ GMR Technologien müssen im Vergleich flächenhungrige Strukturen realisieren. Aufgrund der vergleichsweise kleinen Flächen und einer äußerst geringeren Stromaufnahme ist ein engmaschiger Aufbau von Array-Strukturen möglich [10]. TMR-Flächen sind typischer Weise $< 2 \mu\text{m}$ im Radius. Die hohe Widerstandsänderung generiert entsprechend hohe Signalamplituden in der Magnetfelderfassung, daher kann eine nachgeschaltete Verstärkung entfallen. Es ist eine Periodizität von 360° abgedeckt. Ein TMR-Sensor für die Drehwinkelerfassung besteht aus zwei um 90° verdrehte Wheatstone-Brücken [11].

Wheatstone-Brücke

Ein einzelnes TMR-Element bzw. Widerstand kann bereits als eigenständiger magnetischer Sensor betrachtet werden. Allerdings können Temperatureinflüsse den Widerstand mitunter variieren lassen. Um den Temperatureinfluss entgegenzuwirken, sind daher Wheatstone'sche Brückenschaltungen genutzt. Die einzelnen Widerstände der Brücke sind dabei so angeordnet, dass sie einen gemeinsamen Temperaturkoeffizienten besitzen und entsprechend miteinander gleichmäßig schwanken [21]. Über die Differenzmessung an den Brückenmittelabgriffen wird somit der Temperatureinfluss weitestgehend unterdrückt [11][21]. Abbildung 2.5 zeigt den schematischen Brückenaufbau für einen TMR-Sensor. Bei gleichförmiger Rotation eines Anregungsmagnetfeldes wird durch die 90° Verdrehung beider Brücken zueinander erreicht, dass diese benötigte Cosinus- und Sinus-Funktion, mit entsprechender Phasenverschiebung um 90° , für den Anwendungsfall in ?? ausgeben [11].

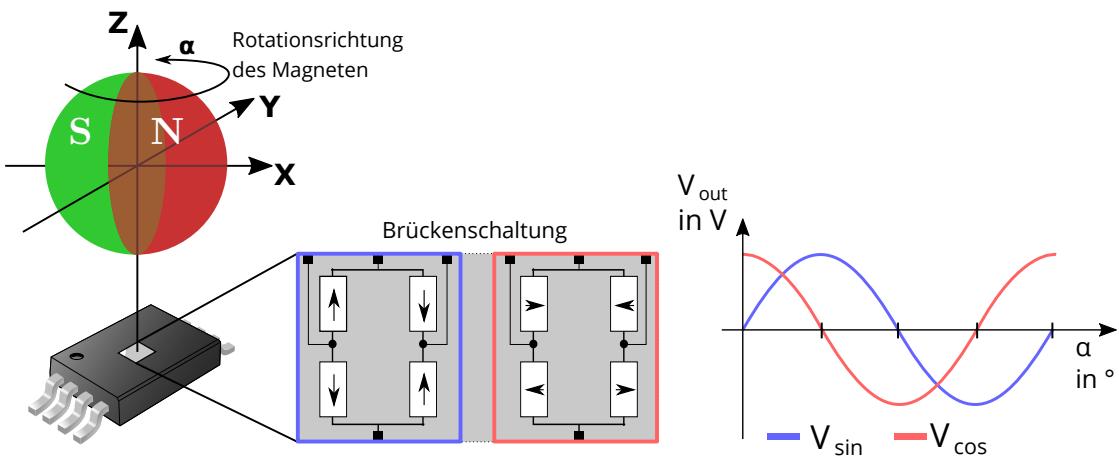


Abbildung 2.5: TMR Drehwinkelapplikation. Schematisch gezeigt für eine volle Rotation des Gebermagneten um 360° . Zu sehen sind die um 90° verdrehten Wheatstone-Brücken des Sensors. Die Brücken bilden, bei rotierenden Gebermagnetfeldern, eine Sinus- und Cosinus-Funktion nach. Die Pfeile in den einzelnen Widerständen weisen auf ihre magnetische Ausrichtung hin. Grafik entnommen und bearbeitet aus [20].

2.4 Kennfeldmethode zur Charakterisierung von Sensoren

Die physikalisch-mathematische Beschreibung von magnetischen Sensormodellen, für eine simulative Nutzung, ist nicht trivial. Jede in Abschnitt 2.3 zusammengefasste Sensortechnologie birgt bestimmte verhaltensbezogene Eigenschaften in sich. So müssen technologiebasierte Abhängigkeiten in Linearität und Sättigungsverhalten in komplexen mathematischen Gleichungen beschrieben sein. Wobei bestimmte Parameter der Modelle experimentell bestimmt werden müssen [15]. Das stellt einen erheblichen Arbeitsaufwand dar. Der Arbeitsgruppe Sensorik ist es gelungen diesen Aufwand durch Messmethodik zu umgehen. So ist die Kennfeldmethode zur Charakterisierung von magnetoresistiven Sensoren entwickelt worden. Das Ziel der Messmethode ist es repräsentative Datensätze zu generieren, die physikalische Charakteristiken eines Sensors in sich vereinigen und sein Verhalten zur weiteren Nutzung zugänglich machen.

Im Labor der Arbeitsgruppe Sensorik sind automatisierte Messstände für Sensoren verschiedenster Couleur eingerichtet. Je nach Sensortechnologie und Applikation können diese mit unterschiedlichen Messmitteln bestückt werden. So ist es möglich die richtige Stimulanz für die jeweilige Sensorapplikation zu erzeugen. Für das Ausmessen von Winkelsensoren ist ein Kreuzspulen-Messstand zur Anwendung gekommen [15][19]. Der Messstand eignet sich besonders gut dazu rotierende Magnetfelder zu erzeugen. Was einer idealen Stimulanz für Winkelsensoren entspricht. Das Magnetfeld wird dabei durch ein speziell abgestimmtes Kreuzspulen-System erzeugt. Die dafür eingespeisten Spulenströme sind dabei direkt proportional zum erzeugten Magnetfeld. Das Anregungsfeld kann somit über Spulenfaktoren zurückgerechnet werden [19].

Abbildung 2.6 zeigt das verwendete Anregungsfeld zur Charakterisierung. Stimuliert wird in X - und Y -Richtung, der räumlichen Sensorebene. So verwendet die H_x -Feldgenerierung ein dreicksmodulierter Cosinus-Strom. Die H_y -Feldgenerierung nutzt ein, mit gleicher Frequenz, dreicksmodulierten Sinus-Strom. Es entstehen dabei steigende und fallende Modulationsflanken bzw. Messverläufe. Es ergeben sich in polarer Darstellung Trajektorien mit wachsender und schrumpfende Amplituden, die durch Rotation einen nach Außen (steigend) bzw. Innen (fallend) gerichteten Verlauf besitzen [15]. Es werden die Spulenströme und Spannungsausgaben des Sensors aufgezeichnet.

In einem weiteren Evaluierungsschritt sind Stimuli und Sensorsignale programmatisch zu indizieren, um eine gegenseitig Referenzierung zu ermöglichen. Sodass resultierend zweidimensionale Kennfeldpaare, bestehend aus je ein Kennfeld für entsprechende Winkelsensor-Wheatstone-Brücke, als Charakterisierungsergebnis zur Verfügung stehen [15].

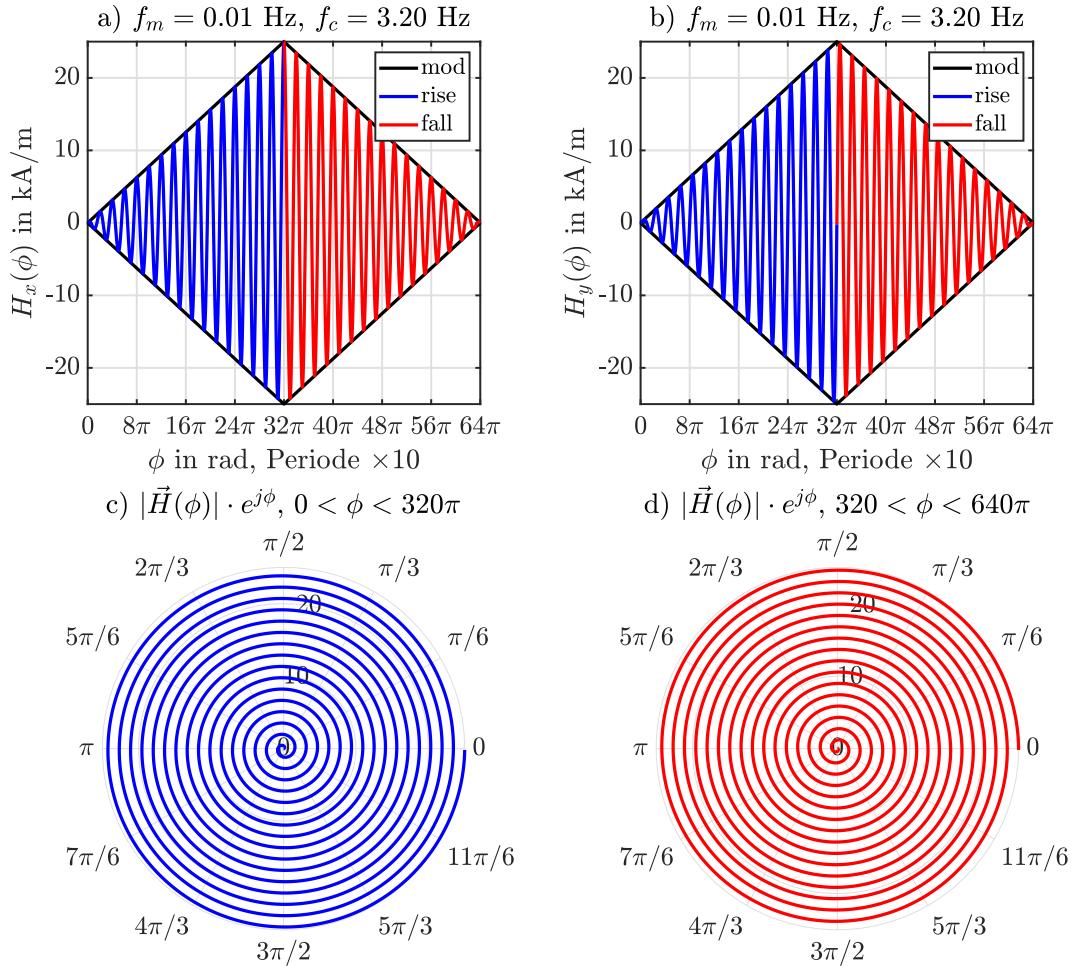


Abbildung 2.6: Magnetfeldstimulus zur Erzeugung von Sensorkennfeldern. Es sind die Bestandteile des magnetischen Sensorstimuli dargestellt, die zum Ausmessen des Sensorkennfeldes in H_x - und H_y -Richtung verwendet worden sind. Es ist das Prinzip des Verfahrens dargestellt. In a) und b) ist die Dreiecksmodulation des magnetischen Anregungsfeldes abgebildet. Für a) die H_x -Feldanregung mit Cosinus-Trägerwelle und für b) die H_y -Feldanregung mit Sinus-Trägerwelle. Es sind für beide Anregungsrichtungen niedrige Frequenzen gewählt um ein quasi-statisches Anregungsmagnetfeld zu erzeugen. Es ergeben sich für die Betragssamplitude des Stimulus, in polarer Darstellung c) und d), konzentrische Trajektorien. Diese verlaufen von Innen nach Außen für die steigende Flanke der Amplitudenmodulation c) und von Außen nach Innen für die fallende Flanke d). Die Dreieckmodulationsfrequenz liegt bei $f_m = 0,1 \text{ Hz}$ und einer Trägerwellenfrequenz $f_c = 3,2 \text{ Hz}$. Grafik nachempfunden aus [15].

Im Anhang A ist der Kennfelddatensatz eines TMR-Sensors [11] gezeigt. Der Datensatz dient, als Arbeitsgrundlage für die Sensor-Array-Simulation und ist von der Arbeitsgruppe Sensorik zur Verfügung gestellt worden. Zur Simulation sind die Kennfelder aus Abbildung 2.7 zu verwenden, a) für die Erzeugung der Cosinus-Funktion und b) für die Sinus-Funktion. Beide Kennfelder zeichnen sich besonders durch ihren linearen Arbeitsbereich zwischen $\pm 8,5 \text{ kA m}^{-1}$ aus. Der Arbeitsbereich ist für beide Kennfelder nahezu identisch, zu sehen in c) und als Kreis in a) und b) markiert.

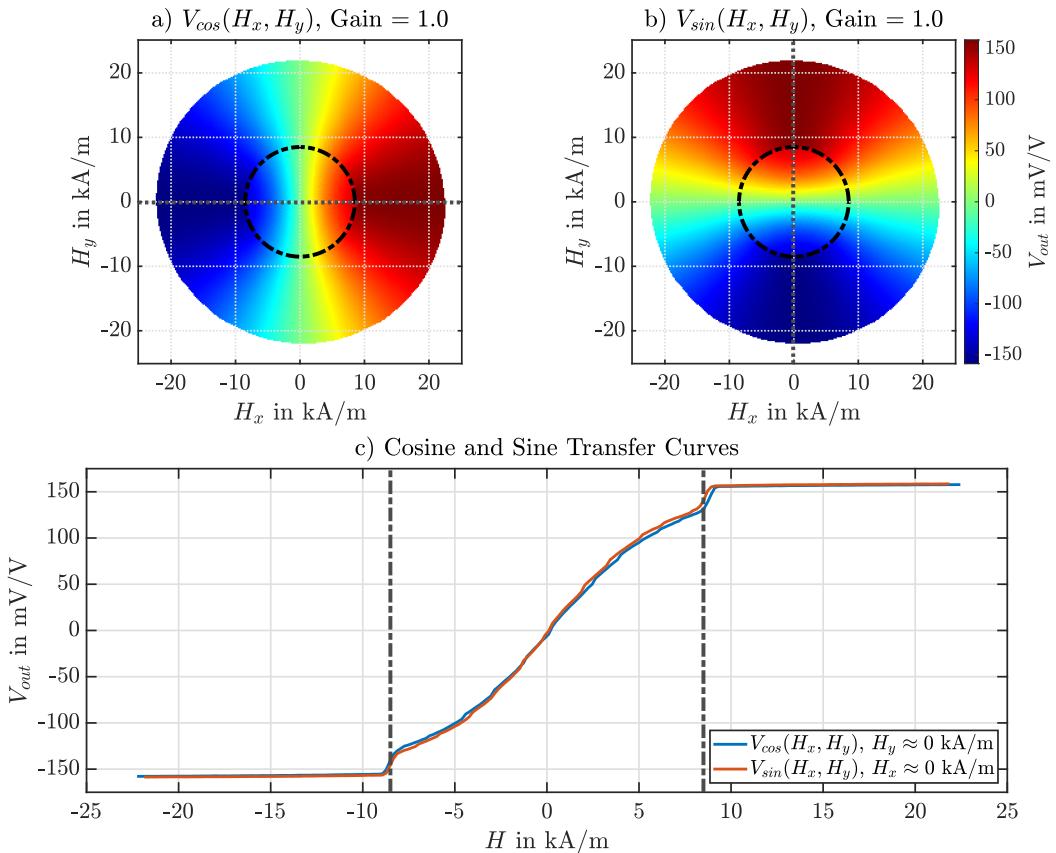


Abbildung 2.7: TDK TAS2141-AAAB Übertragungskennlinie. Es sind wieder die Kennfelder aus der steigenden Amplitudenmodulation in a) und b). In c) sind die Übertragungskennlinien für den Sensor gezeigt mit Kennzeichnung für den Betrieb auf dem linearen Plateau des Kennfeldes bei $8,5 \text{ kA m}^{-1}$. Ebenfalls zu sehen in a) und b) durch die sich ergebene Kreisbahn mit einem Radius des aufgelegten Intervalls aus c). Grafik nachempfunden aus [15].

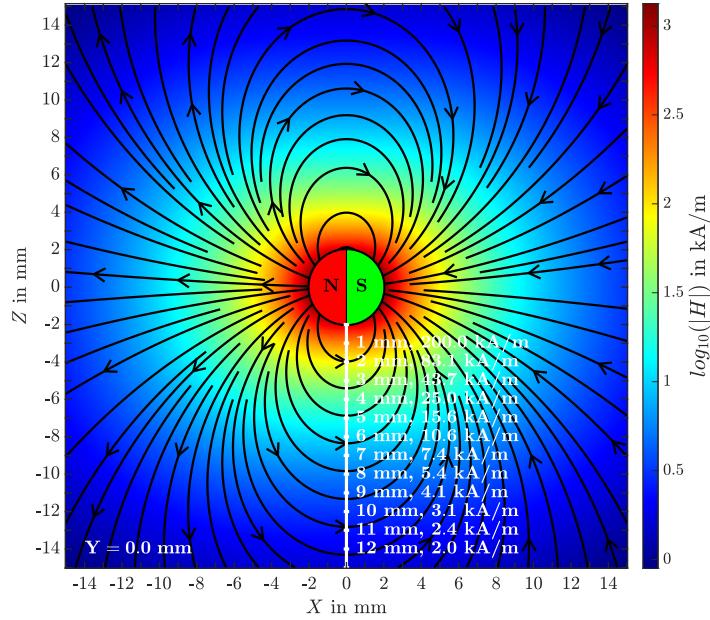


Abbildung 2.8: Approximierter Kugelmagnet. Die Approximation des Kugelmagneten erfolgt über Dipol-Feldgleichung in Näherung des Kugelmagnetfernfeldes. Das Magnetfeld ist auf 200 kA m^{-1} bei einem Abstand von 1 mm zur Kugelmagnetoberfläche normiert. Der Radius des Kugelmagneten beträgt 2 mm.

Auf den TMR-Sensor-Kennfeldern basierende Simulationen, sollten daher so parametriert sein, dass sie innerhalb des markierten Bereiches stattfinden. Zur Generierung von Spannungsausgaben in einer Sensorsimulation, sind für korrespondiere Anregungsfeldstärken entsprechende Referenzwerte aus den Kennfeldern zu entnehmen. Die Referenzwerte sind normiert und nach Gleichung A.1 in Spannungsausgaben der Wheatstone-Brückenschaltungen umzurechnen. Ein geeignetes Verfahren für die Entnahme von Referenzwerten bietet hier, die in Matlab implementierte 2D-Interpolation. Diese kann so parametriert werden, dass sie nach dem Nearest-Neighbor-Verfahren für beliebige Feldstärkeneingaben den nächstgelegenen Referenzwert ausgibt. Wichtig sind hierbei eine gute magnetische Stimulanz, die den Arbeitsbereich des Kennfeldes trifft.

Als Beispiel für eine passende Anregung soll hier, der in Abbildung 2.8 approximierte, Kugelmagnet dienen. Das Magnetfeld ist so normiert, dass eine Betragsfeldstärke von 200 kA m^{-1} bei 1 mm Abstand zur Magnetenoberfläche anliegt. Die angelegte Skala zeigt, dass ein Sensor mit seinen Kennfeldern aus Abbildung 2.7, einen Abstand in Z -Richtung größer als 6 mm einhalten sollte, um den Arbeitsbereich des Kennfeldes zu treffen. Weitere Beschreibungen und Erläuterungen zur Simulation und Dimensionierung der magnetischen Feldanregung finden sich in beiden folgenden Unterkapiteln Abschnitt 2.5 und Abschnitt 2.6.

2.5 Prinzip des Sensor-Arrays

Ein Sensor-Array stellt in seiner Funktionsweise ein Array aus einzelnen Winkelsensoren dar. Jeder einzelne Winkelsensor des Arrays bildet somit einen Sensor-Pixel. Die einzelnen Sensor-Pixel besitzen im Simulationsbetrieb das selbe Verhalten, das basierend auf den TMR-Senor [11], aus Kennfeldern (Anhang A) entnommen wird. Resultierende Messwerte der einzelnen Sensor-Pixel sind positionsabhängig von ihrer Lage im Sensor-Array[15]. Die Sensor-Pixel-Anordnung erfolgt quadratisch mit gleichen Abständen zu einander. Das Gesamtsystem ist somit eine Adaption des klassischen Anwendungsfall aus Abbildung 2.1 für multiple Winkelsensoren [14][15]. So ergibt sich der Anwendungsfall für das Sensor-Array nach Abbildung 2.9.

Das Gesamtsystem aus Gebermagnet und Sensor-Array behält seinen Koordinatenursprung in der Gebermagnetmitte. Das Sensor-Array ist zentriert und lotrecht zur Z -Achse des Magneten auszurichten [15]. Sodass ein konstantes Flächenniveau in Z eingehalten wird. Bedingt durch die aufgespannte Array-Fläche, sind die einzelnen Sensor-Pixel nicht mehr ideal unter dem Magneten ausgerichtet [18]. Es ist somit davon auszugehen, dass die Kreisbahnen der einzelnen Pixel verzerrt sind. Wobei sich die Bahnbeschreibungen, der Pixel mit dem geringsten X -/ Y -Versatz, einem idealen Kreisverlauf annähern müssen. Physikalische Kleinstabstände zwischen den Sensorbrücken eines Sensor-Pixels sind vernachlässigt. Im Simulationsbetrieb ist die Annahme getroffen, dass die Brückenabstände innerhalb eines Sensor-Pixel vernachlässigbar klein sind.

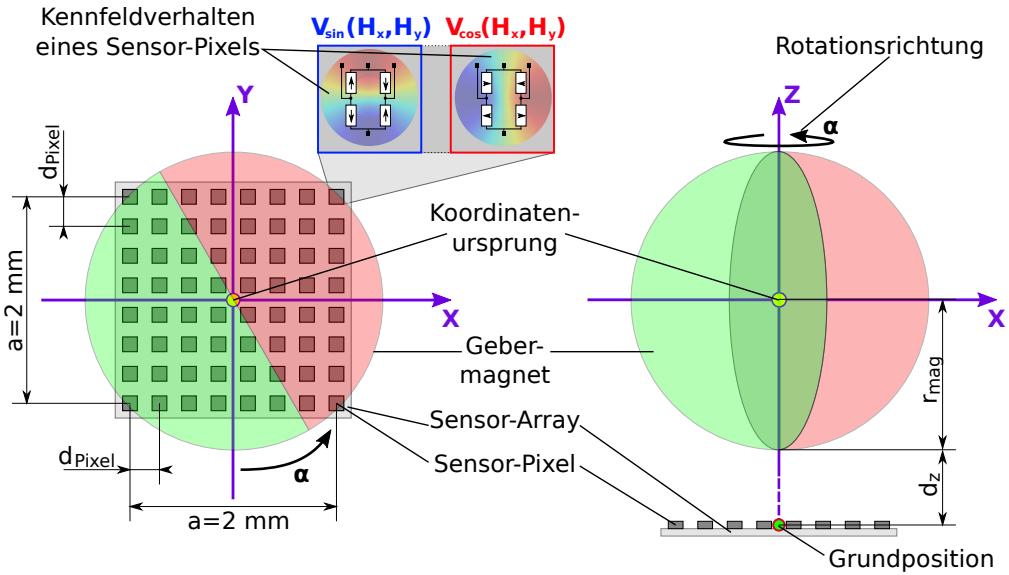


Abbildung 2.9: Geometrischer Aufbau und Ausrichtung des Sensor-Arrays. Quadratische Anordnung von Sensor-Pixeln zu einem 8×8 Sensor-Array. Alle Pixel sind gleich verteilt auf der Array-Fläche. Sensor-Pixel-Verhalten ist aus Kennfeldern entnommen und ortsabhängig von Pixel-Position im Koordinatensystem. Array-Kantenlängen sind mittig von Eck-Pixel zu Eck-Pixel bestimmt. Ebenfalls der Z -Abstand zur Magnetoberfläche. Abstände innerhalb der Pixel sind vernachlässigt. Das Array ist zentriert in der Z -Achse ausgerichtet. Ideal lotrecht zur Nord-Süd-Ausrichtung des Magneten. Koordinatenursprung des Gesamtsystems liegt in der Gebermagnetmitte. Gebermagnet ist ein Kugelmagnet, der um seine Z -Achse rotiert. Grafik nachempfunden und bearbeitet aus [18].

Gemäß der geometrischen Vorgaben, konstantem Flächenniveau in Z und der Wahl des Koordinatenursprungs, fächer sich ein Koordinaten-Meshgrid für $N_{Pixel} \times N_{Pixel}$ für $i, j \in \{1 \dots N_{Pixel}\}$ Sensor-Pixel auf. Dabei ist von einer, im Mittelpunkt des Sensor-Array festgelegten, Grundposition \vec{p} des Sensor-Arrays nach Gleichung 2.9 vorzugehen.

$$\begin{aligned}
 \vec{p} &= \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} & A_{Array} &= a_{Array}^2 & x_{i,j} &= p_x - \frac{a_{Array}}{2} + j \cdot d_{Pixel} \\
 d_{Pixel} &= \frac{a_{Array}}{N_{Pixel} - 1} & y_{i,j} &= p_y + \frac{a_{Array}}{2} - i \cdot d_{Pixel} & & (2.9) \\
 z_{i,j} &= p_z - r_{mag} = konst.
 \end{aligned}$$

Die Z -Koordinate entspricht dem Z -Abstand zum Magneten mit $d_z = p_z$. Das Sensor-Array ist über diesen Vektor in Koordinatensystem zu verschieben. Der Koordinatenursprung bleibt im Magneten. Das Meshgrid fächert sich für die $i - te$ Reihe und die $j - te$ Spalte des Sensor-Arrays auf. Somit gibt jedes Sensor-Pixel, entsprechend seiner Zuordnung im Array, Spannungsausgaben wie ein einzelnes Sensor-IC aus. Es stehen dadurch nicht mehr Skalare bzw. ein Vektor als Winkelmesswert zur Verfügung, sondern Array-Daten für korrespondierende Cosinus- und Sinus-Vektorfelder [14][19]. Abbildung 2.10 zeigt den dimensionalen Zuwachs.



Abbildung 2.10: Resultierende Sensor-Array-Daten. Zwei Matrizen, je eine für alle Cosinus-Brückenausgaben und Sinus-Brückenausgaben. Die $i - ten$ und $j - ten$ Matrixelemente bilden einen Vektor entsprechend des klassischen Anwendungsbeispiels. Jeweils ein Matrix-Paar für die $n - te$ Winkelstellung α .

Bedingt durch den Zuwachs an Daten und ihre Anordnung, benötigt es eine modifizierte Abstandsfunktion [18][19] im Vergleich zur Gleichung 2.7. Als erstes braucht es skalare Repräsentanten der Matrizen. Diese sind durch eine zutreffende Matrix-Norm zu bilden. Matrizen können als lange Vektoren betrachtet werden, daher bietet sich die Rechenvorschrift für $j - te$ Spalten nach Gleichung 2.10 an. Diese Norm ist als Frobenius-Norm bezeichnet.

$$\|\mathbf{A}_x\|_F = \sqrt{\sum_{j=1}^n \|A_{xj}\|_2^2} = \sqrt{\mathbf{A}_x \mathbf{A}_x^T} \quad (2.10)$$

Angewandt auf beide Vektormatrizen für Cosinus- und Sinus-Anteile, ergibt sich eine normierte Kreisbahn nach Gleichung 2.11. Der Radius ist durch Versatz der einzelnen Sensor-Pixel nicht konstant und muss je nach Position des Sensor-Arrays eine weniger oder stärkere Ellipsenform beschreiben [15]. Das ergibt sich durch Überlagerung der Sinoide nach Frobenius-Norm. Der Versatz jedes einzelnen Sensor-Pixel wirkt dabei wie eine Dämpfung auf die Ausgangsspannungen der Sensor-Pixel. So zeigt sich ein Versatz in X-Richtung in einer Dämpfung der Cosinus-Funktion und entsprechender Versatz in Y-Richtung mit einer Dämpfung der Sinus-Funktion [15].

$$\|r\|_F = \|\mathbf{A}\|_F = \sqrt{\|\mathbf{A}_x\|_F^2 + \|\mathbf{A}_y\|_F^2} \quad (2.11)$$

Durch simples einsetzen der normierten Messwertmatrizen in die euklidische Abstandsquadratfunktion Gleichung 2.7, folgt ein approximiertes Abstandsergebnis nach Gleichung 2.12. Über die Dreiecksungleichung erhält man die genau Lösung und Projektion in den höheren Normraum [4][9] und somit die modifizierte Abstandsfunktion nach Frobenius-Norm [19][18] in Gleichung 2.13. Es sind gemachte Beschreibungen aus Abschnitt 2.2, für zwei Winkelstellungen \mathbf{A} und \mathbf{B} , entsprechend der Array-Datenformate angepasst.

$$d_E^2 \langle \mathbf{A}, \mathbf{B} \rangle = (\|\mathbf{A}_x\|_F - \|\mathbf{B}_x\|_F)^2 + (\|\mathbf{A}_y\|_F - \|\mathbf{B}_y\|_F)^2 \quad (2.12)$$

$$\leq$$

$$d_F^2 \langle \mathbf{A}, \mathbf{B} \rangle = \|\mathbf{A}_x - \mathbf{B}_x\|_F^2 + \|\mathbf{A}_y - \mathbf{B}_y\|_F^2 = \|\mathbf{A} - \mathbf{B}\|_F^2 \quad (2.13)$$

2.6 Sensor-Array-Simulation über Dipol-Feldgleichung

Die Sensor-Array-Simulation nutzt einen Kugelmagneten als Stimulanz [15]. Es ist die Anwendungsbeschreibung aus Abbildung 2.9 gewählt. Der Vorteil darin liegt, dass ein Kugelmagnetfeld mittels der Feldgleichung für einen magnetischen Dipol approximiert werden kann [12]. Das innere Magnetfeld des Kugelmagneten ist dabei zu vernachlässigen. Der Radius des Kugelmagneten r_{mag} ist als Offset für den räumlichen Abstand zum Magneten zu verwenden. Weitere physikalische Effekte wie magnetische Remanenz werden vernachlässigt.

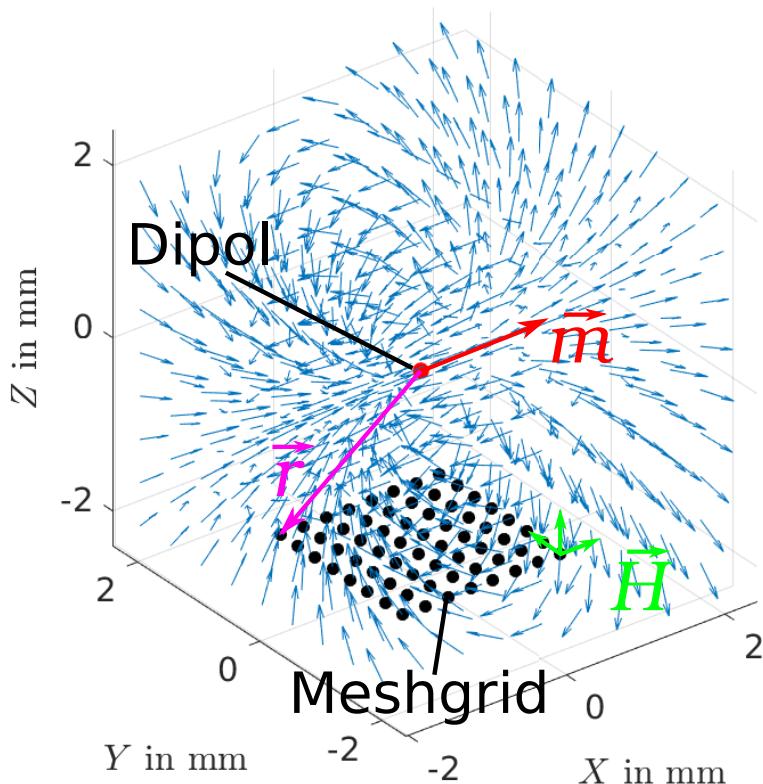


Abbildung 2.11: Simulation der Dipol-Feldgleichung. Als veranschaulichendes Beispiel ist ein Magnetfeld über die Dipol-Feldgleichung simuliert. Der Dipol bildet den Koordinatenursprung bei $\vec{r} = (0, 0, 0)^T$. In Relation zum Dipol ist ein Meshgrid, der einzelnen Sensor-Pixel-Positionen \vec{r} , unterhalb des Dipoles gelegt. Abhängig vom magnetischen Moment \vec{m} des Dipoles wird an jeder Meshgrid-Position \vec{r} die Dipol-Feldgleichung gelöst und punktuell die magnetische Feldstärke \vec{H} berechnet. Das magnetische Moment \vec{m} bestimmt die Nord-Süd-Ausrichtung und Winkelstellungen des Dipoles.

Es wird ein Meshgrid für die einzelnen Sensor-Pixel nach Gleichung 2.9 in ein dreidimensionale Koordinatensystem gelegt. Wie in Abbildung 2.11 zu sehen, liegt der Dipol im Koordinatenursprung bei $\vec{r} = (0, 0, 0)^T$. Die einzelnen Pixel-Position \vec{r} und das magnetische Dipol-Moment \vec{m} , sind durch Gleichung 2.14 beschrieben. Das Dipol-Moment \vec{m} bestimmt die räumliche Ausrichtung des Magnetfeldes.

$$\vec{r} = \hat{r} \cdot |\vec{r}| = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \vec{m} = \begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix} \quad (2.14)$$

Die Feldstärken \vec{H} , an den jeweiligen Pixel-Positionen \vec{r} , sind für das aktuelle Moment \vec{m} nach Gleichung 2.15 zu berechnen. Gleichung 2.15 ist durch einsetzen von Gleichung 2.14 zu Gleichung 2.16 vereinfacht. Somit ist die Feldstärke \vec{H} , mit dazugehörigen Moment \vec{m} , nur von der Richtung des Einheitsvektors \hat{r} abhängig und durch $\frac{1}{4\pi|\vec{r}|^3}$ skaliert. Die entsprechende Betragfeldstärke $|\vec{H}|$ setzt sich aus den resultierenden Feldstärkenkomponenten in Gleichung 2.17 zusammen.

$$\vec{H}(\vec{r}, \vec{m}) = \frac{1}{4\pi} \cdot \left(\frac{3\vec{r} \cdot (\vec{m}^T \cdot \vec{r})}{|\vec{r}|^5} - \frac{\vec{m}}{|\vec{r}|^3} \right) \quad (2.15)$$

$$= \frac{1}{4\pi|\vec{r}|^3} \cdot \left(3\hat{r} \cdot (\vec{m}^T \cdot \hat{r}) - \vec{m} \right) = \begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} \quad (2.16)$$

$$|\vec{H}| = \sqrt{H_x^2 + H_y^2 + H_z^2} \quad (2.17)$$

Um eine Rotation und etwaige Verkippungen des Dipol-Magnetfeldes im Raum zu erwirken, müssen nach Gleichung 2.18 entsprechende axiale Rotationen in X , Y und Z , durch aufschalten von Drehmatrizen hergestellt werden. Durch drehen bzw. verkippen des Dipol-Momentes \vec{m} nach Gleichung 2.18, ergibt sich das neue Dipol-Moment \vec{m}' und somit weiter resultierende Feldstärken \vec{H}' bei gleichbleibenden Pixel-Positionen \vec{r} [15].

$$\underbrace{\begin{pmatrix} m'_x \\ m'_y \\ m'_z \end{pmatrix}}_{\vec{m}'} = \underbrace{\begin{pmatrix} \cos \alpha_z & -\sin \alpha_z & 0 \\ \sin \alpha_z & \cos \alpha_z & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{R_z(\alpha_z)} \underbrace{\begin{pmatrix} \cos \alpha_y & 0 & \sin \alpha_y \\ 0 & 1 & 0 \\ -\sin \alpha_y & 0 & \cos \alpha_y \end{pmatrix}}_{R_y(\alpha_y)} \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha_x & -\sin \alpha_x \\ 0 & \sin \alpha_x & \cos \alpha_x \end{pmatrix}}_{R_x(\alpha_x)} \underbrace{\begin{pmatrix} m_x \\ m_y \\ m_z \end{pmatrix}}_{\vec{m}} \quad (2.18)$$

Für den Standardanwendungsfall aus Abschnitt 2.5, also Rotation in Magnet- Z -Achse ohne Verkippung, sind die Drehmatrizen R_x und R_y auszuschalten. Das kann durch zu Null setzen der Verkippungswinkel α_x und α_y erreicht werden. Die Drehmatrizen sind dadurch zur Einheitsmatrix I gleichgeschaltet. Es ergeben sich somit $i - te$ Dipol-Rotationsmomente \vec{m}_i , für $i - te$ Winkelstellungen des Magneten α_i mit $\alpha_i \in \{0^\circ, \dots, 360^\circ\}$, nach Gleichung 2.19. Dabei ist $\vec{m}_0 = -(m_0, 0, 0)^T$ das Startmoment und legt die Nord-Süd-Ausrichtung des Gebermagneten zu Beginn in seine X -Achse. Für Rotationen mit konstanten Verkippungen sind die Verkippungswinkel $\alpha_x \neq 0$ bzw. $\alpha_y \neq 0$ zu setzen und allgemein nach Gleichung 2.18 zu berechnen.

$$\vec{m}_i(\alpha_i) = R_z(\alpha_i) \cdot I \cdot I \cdot \vec{m}_0 \quad \text{f. } \vec{m}_0 = - \begin{pmatrix} m_0 \\ 0 \\ 0 \end{pmatrix} \quad (2.19)$$

Als Anfangswert für das Startmoment empfiehlt sich $m_0 > 1000 \text{ A m}^2$ zu wählen, dass unterdrückt numerische Fehler beim Berechnen der Feldstärke \vec{H} . In einem weiteren Normierungsschritt zum Aufprägen einer Betragsfeldstärke H_{mag} , bei definierten Abstand $r_{mag} + d_z$ zur Magnetenoberfläche, löscht sich der hohe Anfangswert für m_0 rechnerisch aus. Sodass über das Dipol-Moment nur die Ausrichtung des Gebermagneten gesteuert ist und kein nominaler Einfluss auf errechnete Feldstärken \vec{H} besteht.

Damit in der Sensor-Array-Simulation magnetische Anregungen erzeugt werden können, die den empfohlenen Kennfeldarbeitsbereich aus Anhang A treffen, ist es notwendig das approximierte Kugelmagnetfeld in einem weiteren zu manipulieren. Die Manipulation des Magnetfeldes erfolgt, durch das Aufprägen einer Betragfeldstärke H_{mag} , für die Ruhelage des Magneten mit dazugehöriger Feldstärke \vec{H}_0 . Dabei ist ein definierter Abstand zur Kugelmagnetoberfläche festzulegen, bei dem sich die aufzuprägende Betragfeldstärke H_{mag} einstellt.

$$\vec{r}_0(\alpha_1, \alpha_y, \alpha_x) = R_z(\alpha_1) \cdot R_y(\alpha_y) \cdot R_x(\alpha_x) \cdot (0, 0, -(r_{mag} + d_z))^T \quad (2.20)$$

$$\vec{m}_0(\alpha_1, \alpha_y, \alpha_x) = R_z(\alpha_1) \cdot R_y(\alpha_y) \cdot R_x(\alpha_x) \cdot (-m_0, 0, 0)^T \quad (2.21)$$

Die Ruhelage bezieht sich auf den Startwinkel α_1 der Simulation und ist gemäß gewünschter Verkippungen axial getreu einzustellen. Die Normierungsposition ist mit Gleichung 2.20 vorgegeben und definiert den Abstand entlang der Magnet-Z-Achse und zur Magnetoberfläche. Der Kugelmagnet ist mit entsprechenden Ruhemoment, der Normierungsposition folgend, nach Gleichung 2.21 auszurichten. Anschließend ist die Betragfeldstärke $|\vec{H}_0(\vec{r}_0, \vec{m}_0)|$ auszurechnen. Über den Quotient, aus gewünschter Prägung H_{mag} und Betrag $|\vec{H}_0(\vec{r}_0, \vec{m}_0)|$ in Ruhelage, mündet die Berechnung für ein normiertes Kugelmagnetfeld $\vec{H}_{Norm}(\vec{r}, \vec{m}_i)$, für beliebige Positionen \vec{r} im Koordinatenraum und $i - te$ Rotationsmomente \vec{m}_i in Gleichung 2.22.

$$\vec{H}_{Norm}(\vec{r}, \vec{m}_i) = \vec{H}(\vec{r}, \vec{m}_i) \cdot \frac{H_{mag}}{|\vec{H}_0(\vec{r}_0, \vec{m}_0)|} \quad (2.22)$$

Die Anwendungskonfigurierung für eine optimale Simulation und treffen der Arbeitsbereiche ist Anhang B zu entnehmen. Simulierte Feldstärken sind gemäß der Meshgrid-Anordnung in Matrizen zu speichern, sodass sich in Abschnitt 2.5 beschriebene, Array-Datenformate ergeben. Diese können im Simulationsverlauf fortführend, direkt im Array-Format auf die Kennfelder, zur Entnahme von korrespondierenden Spannungsausgaben angewandt und gespeichert werden. In der Sensor-Array-Simulation ist die Verkippung in X-Achse deaktiviert mit $\alpha_x = 0$. Im weiteren Kontext bezieht sich der Begriff Verkippung (engl. “tilt”), ausschließlich auf Verkippungen in der Y-Achse des Gebermagneten.

2.7 Gauß-Prozesse für Regressionsverfahren

Das in der Arbeit zur Anwendung kommende Regressionsverfahren für Gauß'sche Prozesse, orientiert sich maßgebend, an der 2006 von Rasmussen und Williams veröffentlichter Leitliteratur [3]. Vorarbeiten der Arbeitsgruppe Sensorik [18][19] basieren dabei auf Winkelvorhersagen, die über den Mittelwert freien Ansatz gewonnen werden [3]. Als funktionaler Entwurf des Regressionsmodells [19], decken die Vorarbeiten Modellinitialisierung Abschnitt C.1 und Vorhersage mit Winkelkonfidenzintervall Abschnitt C.3 ab. Dabei bezieht sich die Modellinitialisierung auf die Kernel-Implementierung, mittels Kovarianzfunktion nach Gleichung C.4 und für die Abstandsfunktion d_F^2 nach Gleichung 2.13. Das aufgestellte Regressionsmodell ist in der Lage Simulationsergebnisse aus Anhang B zu verarbeiten [18][19].

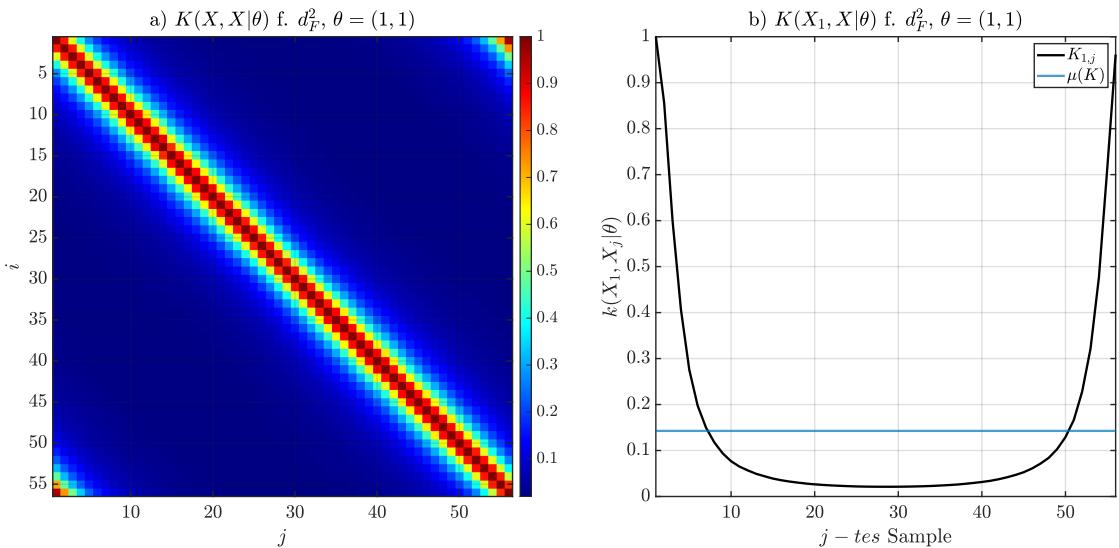


Abbildung 2.12: Kernel-Implementierung der Vorarbeiten, mittels Kovarianzfunktion $k(X_i, X_j)$ nach Gleichung C.4 f. d_F^2 nach Gleichung 2.13 und einen Trainingsdatensatz X mit $N_{Ref} = 56$ gleich verteilten Simulationswinkeln $X_i \mapsto \alpha_i$. Die Skalierung durch Kernel-Parameter ist mit $\theta = (1, 1)$ ausgeschaltet. In a) ist Algorithmus 3 f. $K(X, X|\theta)$ genutzt und alle Trainingsdaten X somit gegenseitig referenziert. Abbildung b) zeigt, Kovarianzen $K_{1,j}$ des Teildatensatz X_1 , gegenüber sich selbst und jedem weiteren Satz X_j . Der Matrix-Mittelwert $\mu(K)$ in b), ist als Indikator über die gegenseitige Beeinflussung Datensätze im Regressionsverfahren zu interpretieren. Der genutzte Trainingsdatensatz X basiert auf, gemachter TMR-Sensor [11] Charakterisierung in Anhang A. Grafik nachempfunden aus [7]

Ab hier folgt ein Wechsel der Notation um Bezüge zur Fachliteratur [3] zeigen zu können. Die veränderte Schreibweise ist im Abschnitt C.1 unter Trainings- bzw. Testdatensätze zusammengefasst und beschrieben.

Die Parametrierung des Regressionsverfahren ist bisher empirisch ermittelt worden und stellt einen Angelpunkt in dieser Arbeit dar. Für eine selbstständige Optimierung, von Modellparametern und verbundener Modellgeneralisierung, müssen entsprechende Kriterien gebildet werden. Diese sind, in genutzter Literatur, als Minimierungsprobleme beschrieben [3][5][7].

Im Gegensatz zur Leitliteratur [3], die Lösungsansätze für Regressionen eindimensionaler Funktionen beschreibt, ist für die Winkelvorhersage eine kombinierte Regression einer zweidimensionalen Funktion herzustellen. Da Ableitungen von Winkelstellungen, über orthogonal zueinander stehenden, Cosinus- und Sinus-Funktionen erfolgen. Dabei ist die Adaption der Array-Daten-Formate aus Abschnitt 2.5, bereits in den Vorarbeiten gelöst worden [19] und über die Kovarianzfunktion Gleichung C.4 für d_F^2 Gleichung 2.13 implementiert. Abbildung 2.12 zeigt die Implementierung aus den Vorarbeiten, anhand der Kovarianzfunktion und resultierender Kovarianzmatrix, für ein Beispiel eines $N_{Ref} = 56$ Observierungen großen Trainingsdatensatz. Für diese Implementierungsform, wäre das ein viel zu großer Datensatz in der realen Anwendung. In Bezug auf das Sensor-Array Abschnitt 2.5, müssten $2 \times 56 = 112$ Matrizen abgelegt werden, sodass diese dem Regressionsprozess als Referenzen zur Verfügung stehen. Was hier zwar ein drastisches Beispiel ist, dass allerdings sehr schön das Verhalten der Kovarianz in Verbindung mit TMR basierten Daten zeigt.

Die Kovarianzfunktion mit resultierender Kovarianzmatrix, muss in der Lage sein systemische Eigenschaften wiedergeben zu können. Auf den TMR-Sensor [11] gemünzt, muss die Matrix einfach periodisch sein. Das ist durch Kurvenverlauf in Abbildung 2.12 b) und durch ansteigenden Ecken links unten und rechts oben in a) zu erkennen. Es gibt nur eine vollwertige Diagonale. Bei Systemen höherer Periodizität, müsste die Kovarianzfunktion, entsprechend mehrere dieser Diagonalen, durch Superposition oder Trigonometrie-Funktionen erzeugen [3].

Die homogenen Bereiche der Matrix zeigen, dass die Trainingsdaten X mit gleichbleibender magnetischer Stimulanz erzeugt wurden. Gäbe es Fehllagen in der Erzeugung, Sprunghaften Versatz des Sensor-Arrays, oder Verkippung des Magneten, wären die Flächen unterbrochen. Ebenfalls indiziert die durchgehende Diagonale, dass die Rotation konstant mit gleichbleibenden Abständen vollzogen worden ist. Würden Sprünge in der Rotation auftauchen, müssten diese durch Schnitte in der Diagonale und eventuell durch Absenkungen der Ecken ersichtlich werden. Durch Bruch in der Periodizität würden Ecken der Matrix ganz verschwinden.

Der annähernd keilförmige Kurvenverlauf in Abbildung 2.12 b), bei ausgeschalteter Skalierung $\theta = (1, 1)$, weist auf ein System mit einfacher Komplexität hin [3]. Das heißt, es benötigt entweder eine erhöhte Anzahl von Trainingssamples, oder eine Parameteroptimierung und aufbohren der Modellkomplexität, um von den Trainingsdaten abweichende Daten, mit geringen Regressionsvarianzen prozessieren zu können [3]. Andersherum gesagt, gibt man alle möglichen in Winkelpositionen über die Trainingsdaten vor, muss der Regressionsfehler automatisch gegen null gehen. Die Modellanpassung auf eingespeiste Trainingsdaten, sowie gleichzeitige und sinnvolle Verringerung des Datenumfangs, bilden dabei die zu haltende Balance in Bezug auf akzeptable Winkelfehler [3].

3 Software-Entwicklung für Optimierungsexperimente 0.0.2

19.02.2021

3.1 Aufgabe und Funktionen der Software

- Identifizierung der Grundfunktionen
- Datengenerierung
- Datenanalyse
- Sonderfunktion
- Darstellungs- und Plot-Funktionen

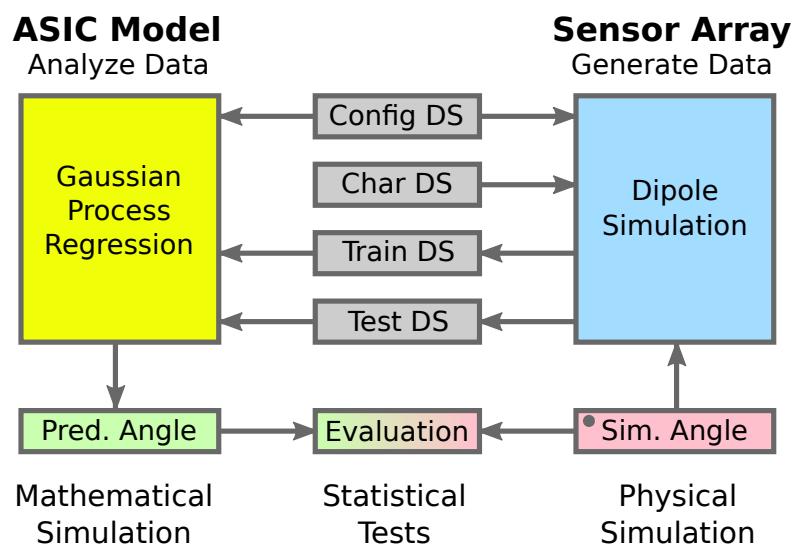


Abbildung 3.1: Simulationsaufbau im Überblick

Die Software-Entwicklung erfolgt unter dem Gesichtspunkt zur Durchführung von Versuchsreihen zu Parameterfindung und teilweise auf Zwischenergebnissen basieren. Gut strukturierte Archivierung von Ergebnisse. Graphische Unterstützung von Auswertung.

3.2 Aufbau und Vorgehen

- Skriptbasierte Entwurfsarbeit
- Überführen in modularen Aufbau von Kernfunktion
- Parametrierte Steuerung der Software über Zentrale Konfigurierung
- Ausführbare Skripte (Einbindung von Modulen und nutzen der Konfigurierung)
- Speicherung von Ergebnissen in Datensätzen
- Versionierung der Arbeitsschritte

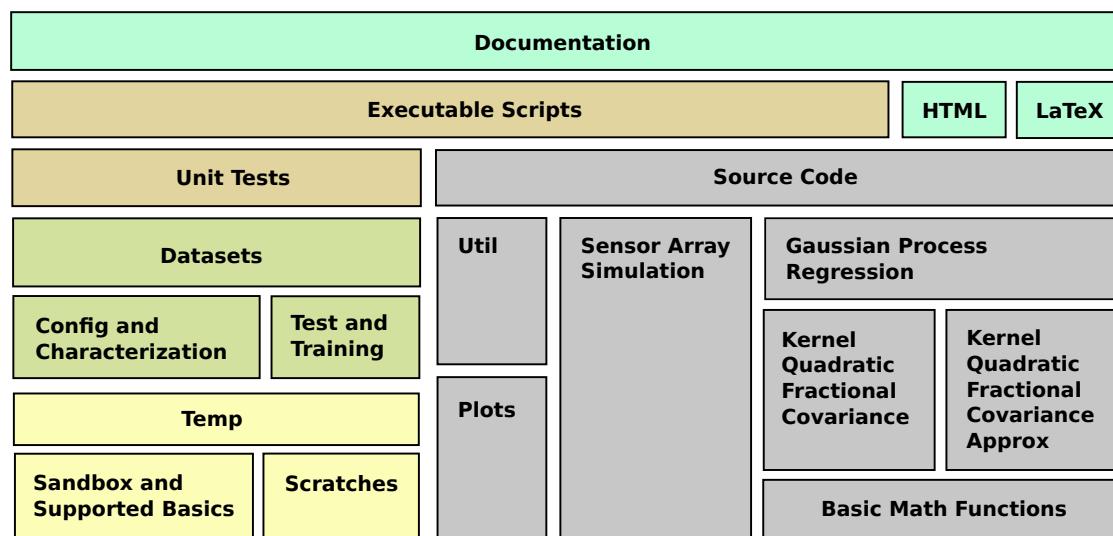


Abbildung 3.2: Blockschema Simulations-Software

3.2.1 Sensor-Array-Simulation

- Zuordnung Datengenerierung
- Nutzung von vorarbeiten
- Darstellung des Modul-Funktionsablaufdiagramm
- Darstellung des Algorithmus für die Simulation mehrere Positionen
- Nutzung des Moduls für eingestellte Konfigurierung

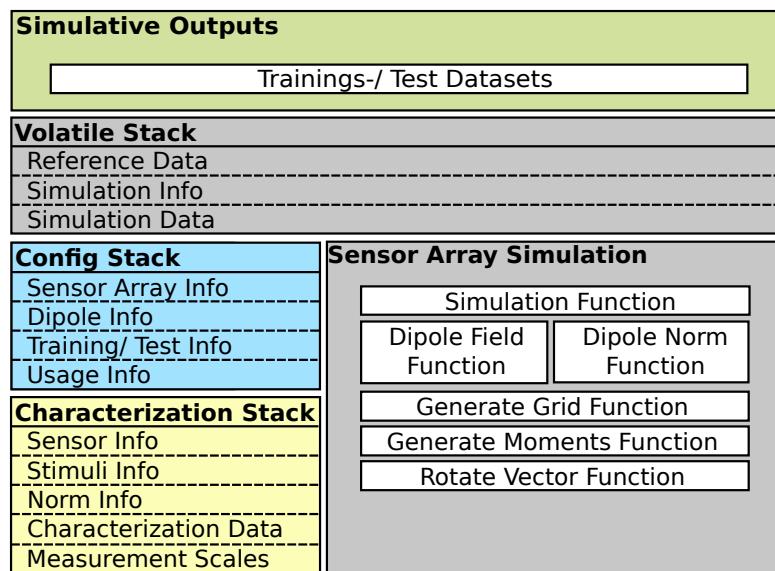


Abbildung 3.3: Blockschema Sensor-Array-Simulation

3.2.2 Gauß-Prozess-Regression

- Zuordnung Datenanalyse
- Nutzung von Vorarbeiten
- Darstellung des Modul-Funktionsablaufdiagramm
- Aufbau der Modell-Engine und Schnittstellen für neue Kovarianzfunktionen
- Darstellung der einzelnen Optimierungsverfahren und Aufzeigen der Unterschiede im vorgehen
- Nutzung des Moduls für eingestellte Konfigurierung

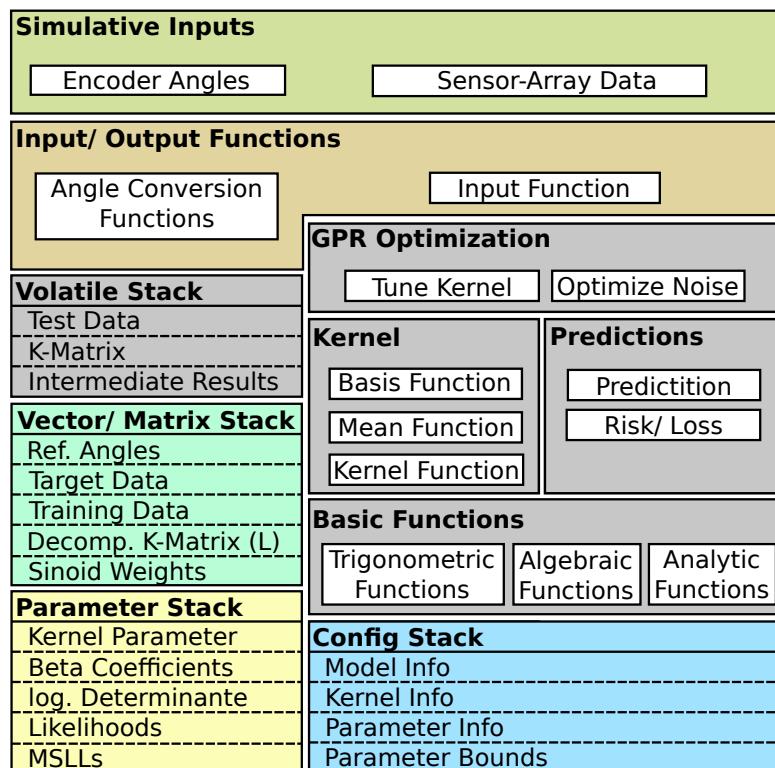


Abbildung 3.4: Blockschema Trainingsphase Regression

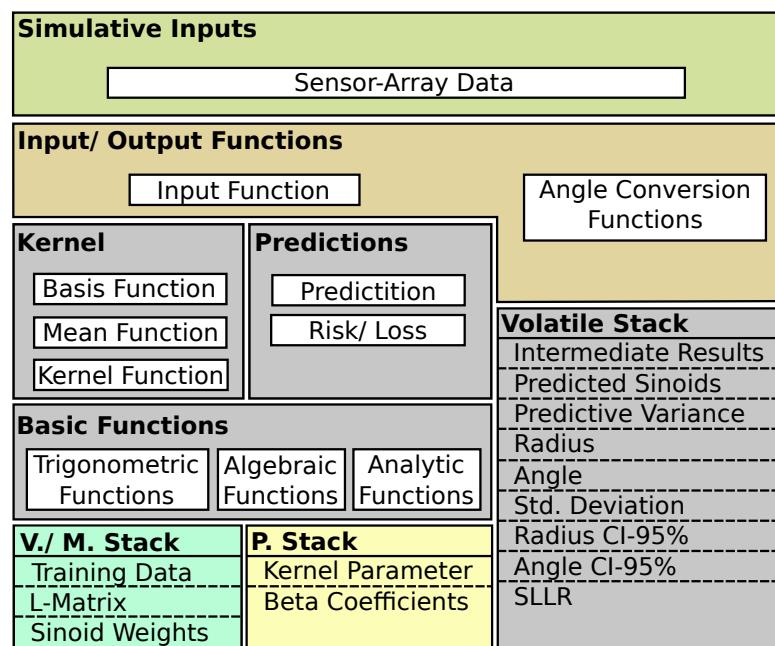


Abbildung 3.5: Blockschema Arbeitsphase Regression

3.3 Simulationsprozesse

3.3.1 Sensor-Array-Simulation

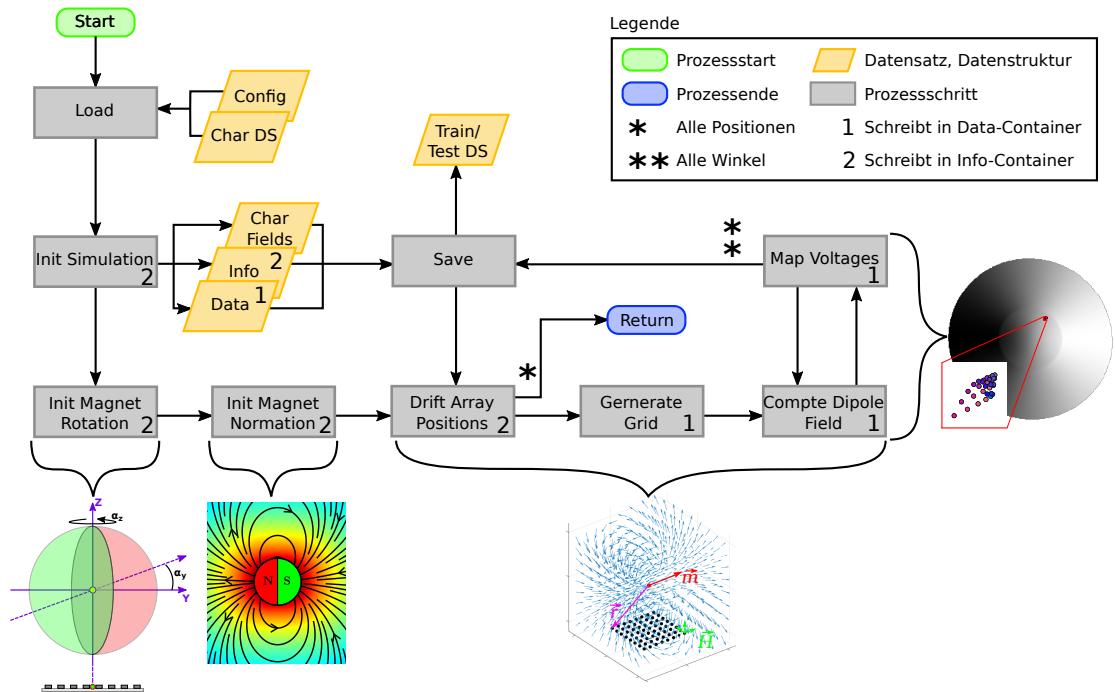


Abbildung 3.6: Sensor-Array-Simulation Prozessansicht

3.3.2 Gauß-Prozess-Regression

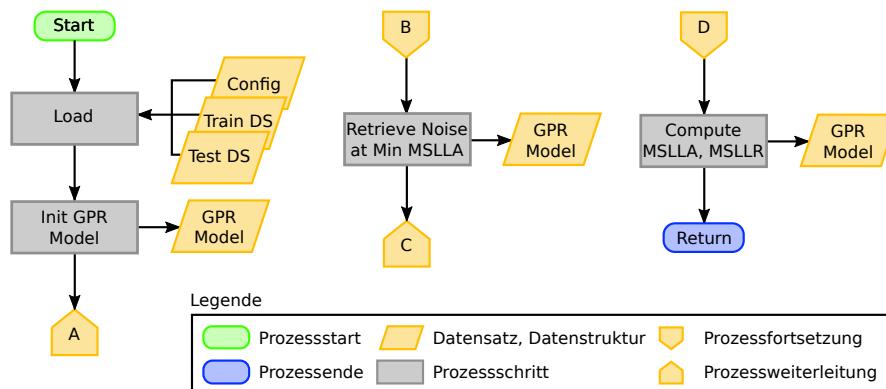


Abbildung 3.7: Regressionsoptimierung/-Generalisierung Prozessansicht

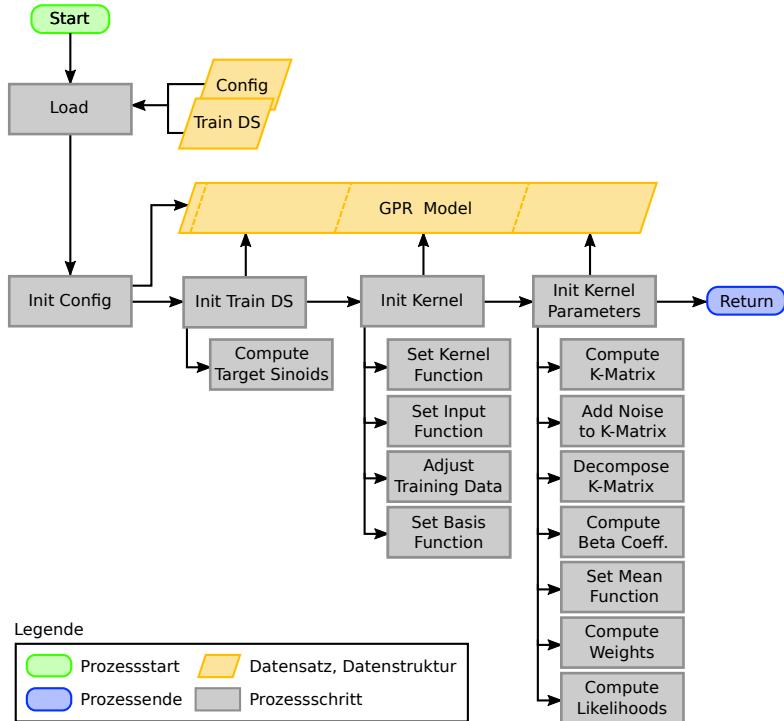


Abbildung 3.8: Regressionsinitialisierung Prozessansicht

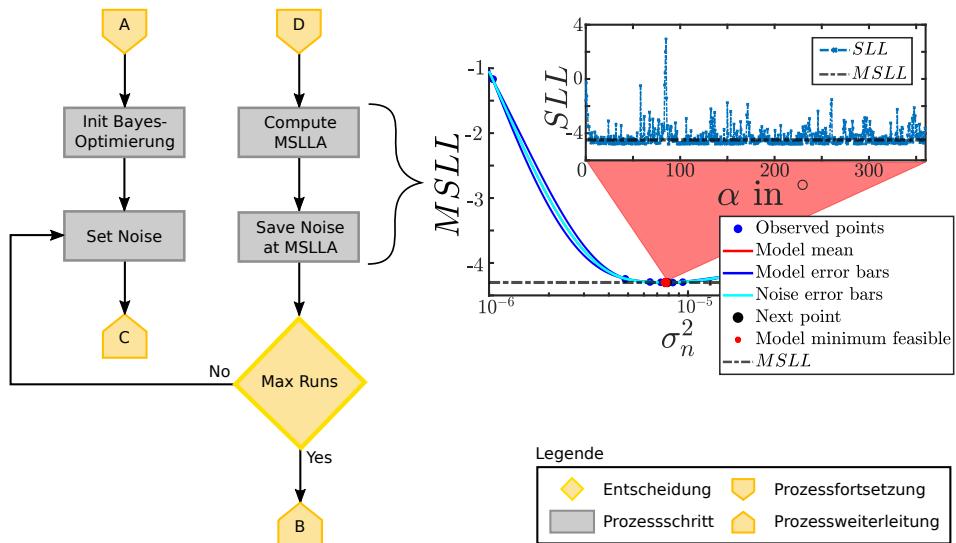


Abbildung 3.9: Rauschniveaoptimierung Prozessansicht

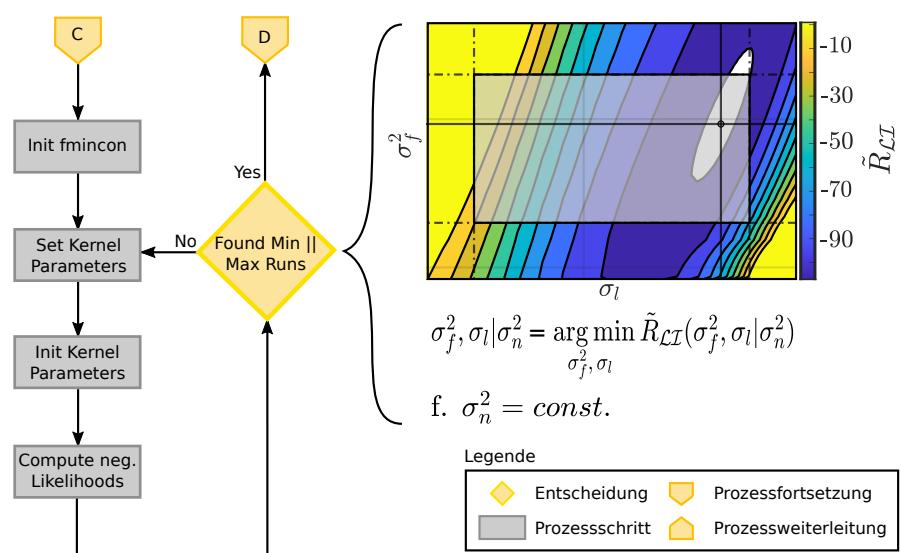


Abbildung 3.10: Regressionsparameteroptimierung Prozessansicht

4 Erprobungs- und Optimierungsexperimente 0.0.1

13.01.2021

- Klassifizierung (Diagnose)
- Stabilitätskriterium
- Fehlererkennung Max. Mittelwert, Qualitätsmaß
- Allg. Vorgehen "Batch-Job"
- Konfigurierung der Simulationssoftware

4.1 Festlegung des Startpunktes

- Startpunkt, 1. Position gleich Anlernpunkt für Trainingsphase
- Auswahl des Senortyps
- Konfigurierung des Magneten
- Auswahl des GPR-Modells nach Optimierung
- Konfigurierung des GPR-Modells mit ermittelten Parametern

4.2 Festlegung des Verfahrweges ohne Verkippung

- Vorbetrachtung des Magnetsfeldes
- Aufteilung in Sektoren
- Abfahren in Z-Richtung ohne Versatz
- Festlegen des X-Y-Versatzes, Symmetrie-Sektor

4.3 Simulationsdurchführung

- Festhalten der Ergebnisse
- Position, Winkelfehler (Max, Mittel), Qualitätsmaß (Max, Mittel)
- Drift-Darstellung

5 Auswertung 0.0.1 13.01.2021

5.1 Gegenüberstellung der GPR-Modelle

- Aufwand der Trainingsphase
- Nötige Parameter und zu Speichernde Werte
- Arbeitsphase, Genauigkeit, Fehlererkennung, Stabilität

6 Zusammenfassung und Bewertung 0.0.1

13.01.2021

- Kurzdarstellung der Ergebnisse der Arbeit
- Offene Punkte und Probleme
- Ansätze zur Weiterführung für zukünftige Arbeiten
- Bewertung der Ergebnisse in Bezug auf die Anwendung

Abbildungsverzeichnis

1.1	Platinen-Sensor-Array im Maßstab	3
1.2	Ansatzdarstellung zur Generierung eines Simulationsmodell des magnetischen Sensor-Arrays	4
1.3	Veranschaulichung eines vollständigen Sensor-ICs für die Drehwinkelerfassung	5
2.1	Klassischer Anwendungsfall für die Drehwinkelerfassung	9
2.2	Kreisdarstellung der Winkelmessung	10
2.3	Allg. Kreisdarstellung des euklidischen Winkelabstands	12
2.4	Schichtmodelle dreier magnetoresistive Effekte	13
2.5	TMR Drehwinkelapplikation	16
2.6	Magnetfeldstimulus zur Erzeugung von Sensorkennfeldern	18
2.7	TDK TAS2141-AAAB Übertragungskennlinie	19
2.8	Approximierter Kugelmagnet	20

2.9	Geometrischer Aufbau und Ausrichtung des Sensor-Arrays	22
2.10	Resultierende Sensor-Array-Daten	23
2.11	Simulation der Dipol-Feldgleichung	25
2.12	Kernel-Implementierung der Vorarbeiten	29
3.1	Simulationsaufbau im Überblick	32
3.2	Blockschema Simulations-Software	33
3.3	Blockschema Sensor-Array-Simulation	34
3.4	Blockschema Trainingsphase Regression	35
3.5	Blockschema Arbeitsphase Regression	36
3.6	Sensor-Array-Simulation Prozessansicht	37
3.7	Regressionsoptimierung/ -Generalisierung Prozessansicht	38
3.8	Regressionsinitialisierung Prozessansicht	39
3.9	Rauschniveaoptimierung Prozessansicht	39
3.10	Regressionsparameteroptimierung Prozessansicht	40
A.1	TDK TAS2141-AAAB Brückenkennfelder	53
A.2	TDK TAS2141-AAAB Kennfeldquerschnitte	54
B.1	Kennfeld-Mapping	57
B.2	Sensor-Array-Datensatz Teilansicht	58

Tabellenverzeichnis

A.1	Eckdaten TDK TAS2141-AAAB Kennfelder	52
B.1	Sensor-Array-Simulationsparameter	55
C.1	Gauß-Prozess-Regression-Simulationsparameter	59
D.1	Genutzte Software	77

Algorithmenverzeichnis

1	Sensor-Array-Simulation	56
2	Modellinitialisierung mit konst. Trainingsdaten und Parametern	60
3	Berechnung der Kovarianzmatrix $K(X, X \theta)$	63
4	Berechnung der β Polynomkoeffizienten aus Gleichung C.10	66
5	Modelloptimierung über Fmincon-Funktion f. $\sigma_n^2 = \text{konst.}$	70
6	Modellvorhersage f. Sinoide eines Testwinkel mit $X_* \mapsto \alpha_*$	72
7	Modellgeneralisierung über BayesOpt-Funktion f. alle $X_* \mapsto \alpha_*$	76

Glossar

AMR-Effekt Anisotroper-Magnetoresistiver-Effekt.

Arbeitsgruppe Sensorik Die Arbeitsgruppe Sensorik steht unter Leitung von Prof. Dr.-Ing. Karl-Ragmar Riemschneider und ist unter dem Department Informations- und Elektrotechnik Teil der Fakultät Technik un Informatik an der HAW Hamburg.

GMR-Effekt Riesiger-Magnetoresistiver-Effekt.

HAW Hamburg Die HAW Hamburg ist die Hochschule für Angewandte Wissenschaften in Hamburg und war die ehemalige Fachhochschule am Berliner Tor.

Kennfeld Zweidimensionales Charakterisierungsabbild einer Wheatstone'schen Sensorbrücke eines magnetoresistiven Winkelsensors. Erstellt durch die Kennfeldmethode zur Charakterisierung von magnetischen Winkelsensoren.

Kennfeldmethode Charakterisierungsverfahren zur Ausmessung magnetoresistiver Winkelsensoren, bestehend aus zwei zueinander verdrehten Wheatstone-Brücken. Die resultierenden Charakterisierungsergebnisse können als Kennfelddatensätze zur Simulation von magnetischen Sensoren genutzt werden.

Kennfeldpaar Charakterisierungsergebnis der Kennfeldmethode für die Charakterisierung magnetoresistiver Winkelsensoren. Bestehend aus zwei Kennfeldern, jeweils als Repräsentanten der Wheatstone-Brücken eines Winkelsensors.

Kreuzspulen-Messstand Automatisierter Messtand zur Charakterisierung von Winkelsensoren. Der Messtand nutzt ein Kreuzspulen-System, in dessen Mitte der Winkelsensor platziert ist. Das Spulensystem erzeugt ein moduliertes, langsam rotierendes Anregungsmagnetfeld. Parallel zeichnet der Messtand, die zur Charaktisierung nötigen, Spannungsausgaben des Sensors und Anregungsströme der Spulen auf. Beides

erfolgt programmatisch. Die aufgezeichneten Messdaten können im Anschluss zu Kennfeldern evaluiert werden.

Kreuzspulen-System Spulensystem in Kreuzanordnung in dessen Mitte ein zu messendes Senor-IC platziert wird. Die Spulen sind Maßanfertigungen mit ganz bestimmten Messeigenschaften. Spulenfaktoren sind speziell ausgerechnet und nachgemessen. Kernelement des Kreuzspulen-Messstandes. Eingespeiste Spulenströme erzeugen entsprechende magnetische Felder in X - und Y -Richtung. Die Spulenströme erzeugen, entsprechend der Spulenfaktoren, H_x - und H_y -Feldstärken. Die Feldstärken sind direkt proportional zu den Einspeiseströmen. Die Ströme werden über niederohmige Shunt-Widerstände mit gemessen. Die Feldstärken können so über die Spulenfaktoren zurückgerechnet und zur Auswertung genutzt werden..

Sensorkopf Signal erzeugender Teil eines Sensor-ICs, dem eine Einheit zur weiteren Signalverarbeitung nachgeschaltet ist. Für die Drehwinkelerfassung besteht besteht die Signalerzeugung zumeist aus zwei verdrehten Wheatstone-Brücken, deren einzel Widerstände mittels magnetoresistiven Materialien aufgebaut sind.

TMR-Effekt Tunnel-Magnetoresistiver-Effekt.

Wheatstone'sche Brückenschaltungen Messbrückenschaltung bestehend aus zwei Spannungsteilern, die parallel zu einer gemeinsamen Quelle geschaltet sind. Es wird eine Differenzspannung über die Mittelabgriffe der Spannungsteiler gemessen. Allgemein bekanntes Messprinzip. 1833 von Samuel Hunter Christie erfunden und nach dem britischen Physiker Sir Charles Wheatstone benannt. Abgekürzt auch Wheatstone-Brücken oder in der Sensorik auch Sensorbrücken genannt. Ein Sensorkopf für die Winkelmessung setzt sich in der Regel aus zwei solch gearteter Brückenschaltungen zusammen.

Abkürzungen

AMR Anisotrope-Magnetoresistance.

ASIC Application-Specific-Integrated-Circuit.

CPU Prozessorkern.

GMR Giant-Magnetoresistance.

HDD Festplattenlaufwerk.

IC Integrated-Circuit.

ISAR Integrated-Sensor-Array.

OS Betriebssystem.

RAM Arbeitsspeicher.

SW Software.

TMR Tunnel-Magnetoresistance.

Literatur

- [1] N. I. Fisher. *Statistical Analysis of Circular Data*. Cambridge University Press, 1993. ISBN: 9780511564345.
- [2] K. V. Mardia und P. E. Jupp. *Directional Statistics*. Bd. Wiley Series in Probability and Statistics. John Wiley und Sons, Inc., 1999. ISBN: 9780471953333.
- [3] C. E. Rasmussen und C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. ISBN: 026218253X. URL: www.gaussianprocess.org/gpml (besucht am 30.10.2020).
- [4] M. Plum. „Orthogonalprojektionen, Orthonormalsysteme und -basen“. In: *Vorlesung Differentialgleichungen und Hilberträume*. Vorlesung (2011). KIT, 2012.
- [5] P. Guerrero und J. Ruiz del Solar. „Circular Regression Based on Gaussian Processes“. In: *2014 22nd International Conference on Pattern Recognition*. 2014. DOI: [10.1109/ICPR.2014.631](https://doi.org/10.1109/ICPR.2014.631).
- [6] R. Johnson. *MATLAB Style Guidelines 2.0*. Version 2. MATLAB Central File Exchange, 2014. URL: <https://de.mathworks.com/matlabcentral/fileexchange/46056-matlab-style-guidelines-2-0> (besucht am 21.09.2020). Online.
- [7] M. Lang, O. Dunkley und S. Hirche. „Gaussian process kernels for rotations and 6D rigid body motions“. In: *2014 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2014. DOI: [10.1109/ICRA.2014.6907617](https://doi.org/10.1109/ICRA.2014.6907617).
- [8] NXP Semiconductors. *KMZ60 Angle sensor with integrated amplifier*. Datenblatt, 2014.
- [9] R. A. van de Geijn. *Notes on Vector and Matrix Norms*. The University of Texas Austin, 2014.
- [10] H. Lemme. *Messung durch den Tunnel*. Hrsg. von Elektroniknet. 2016. URL: <https://www.elektroniknet.de/messen-testen/sensorik/messung-durch-den-tunnel.133265.html> (besucht am 25.01.2021). Online.

- [11] TDK. *TMR Angle Sensor TAS2141-AAAB*. Datenblatt, 2016.
- [12] H. Pape. „Simulation und Auswertung von Permanentmagneten für manetoresistive Sensor-Arrays“. Bachelorarbeit HAW Hamburg, 2017.
- [13] infineon. *TLE5x09A16(D) Analog AMR/GMR Angle Sensors*. Datenblatt, 2018.
- [14] T. Mehm. „Schaltungsentwurf und Mikrocontrollersteuerung für ein Tunnel-Magnetoresistives Sensor-Array“. Bachelorarbeit HAW Hamburg, 2019.
- [15] T. Schüthe, A. Albounyan und K. Riemschneider. „Two-Dimensional Characterization and Simplified Simulation Procedure for Tunnel Magnetoresistive Angle Sensors“. In: *Sensors Applications Symposium (SAS)*. (13. März 2019). IEEE, 2019. DOI: [10.1109/SAS.2019.8706125](https://doi.org/10.1109/SAS.2019.8706125). URL: <https://ieeexplore.ieee.org/document/8706125> (besucht am 05.10.2020). Online.
- [16] Bitbucket. *Feature Branch Workflow in Git*. Hrsg. von ATlassian. 2020. URL: <https://www.atlassian.com/de/git/tutorials/comparing-workflows/feature-branch-workflow> (besucht am 10.09.2020). Online.
- [17] J. Ernsting. „Funktionsdemonstrator für magnetische Sensor-Arrays auf Basis des Mikrocomputers Raspberry PI“. Bachelorarbeit HAW Hamburg, 2020.
- [18] T. Schüthe, K. Jünemann und K. Riemschneider. *Tolerance Compensation based on Gaussian Processes for Angle Measurements with Magnetic Sensor Arrays*. HAW Hamburg, 2020.
- [19] T. Schüthe u. a. „Positionserfassung mittels Sensor-Array aus Tunnel-Magnetoresistiven Vortex-Dots und lernender Signalverarbeitung“. In: *Tille T. (eds) Automobil-Sensorik 3*. Springer Vieweg, Berlin, Heidelberg, 2020. ISBN: 978-3-662-61259-0. URL: https://doi.org/10.1007/978-3-662-61260-6_14.
- [20] T. Schüthe u. a. „Positionserfassung mittels Sensor-Array aus Tunnel-Magnetoresistiven Vortex-Dots und lernender Signalverarbeitung“. In: *8. Fachtagung Sensoren im Automobil*. 2020.
- [21] T. Tille. *Automobil-Sensorik-3*. Springer Vieweg, 2020. ISBN: 978-3-662-61259-0.

A TDK TAS2141-AAAB

Kennfelddatensatz 0.0.1 29.03.2021

Der Anhang beinhaltet die Kennfelddarstellung eines TAS2141-AAAB TMR-Winkelsensor der Firma TDK [11]. Die Charakterisierung des Sensor-ICs ist mittels Kennfeldmethode [15] vorgenommen worden. Das Ausmessen des TMR-Sensors hat im Labor der Arbeitsgruppe Sensorik stattgefunden. Als Charakterisierungsergebnis liegt entsprechender Datensatz vor und dient in dieser Arbeit als Simulationsgrundlage für die Sensor-Array-Simulation. Der Datensatz ist von der Arbeitsgruppe Sensorik zur Verfügung gestellt worden. Eine detaillierte Zusammensetzung des Datensatzes ist im Anhang E aufgeführt.

Eigenschaft	Wert	Einheit
H_x -Skala	$-25 \dots 25$	kA m^{-1}
H_y -Skala	$-25 \dots 25$	kA m^{-1}
H_x -Schrittweite	0,1961	kA m^{-1}
H_y -Schrittweite	0,1961	kA m^{-1}
Auflösung	256×256	Pixel
Wertebereich $V(H_x, H_y)$	Normiert	mV V^{-1}
Normfaktor	$1 \cdot 10^3$	mV
Gain	1	-
Brückenverdrehung	90	°
Periodizität	360	°

Tabelle A.1: Eckdaten TDK TAS2141-AAAB Kennfelder

Die Charakterisierung mittels Kennfeldmethode generiert zwei Kennfeldpaare zu sehen in Abbildung A.1. Das erste Kennfeldpaar a) und c) referenziert sich aus dem steigenden Messverlauf, der amplitudenmodulierten H_x -/ H_y -Stimuli. Das zweite b) und d) setzt sich aus dem fallenden Stimuli zusammen. Ein Kennfeld repräsentiert dabei eine Wheatstone-Brücke des Winkelsensors [15]. Bedingt durch die Verdrehung beider Brücken [11], ist ein

entsprechendes Kennfeldpaar zueinander um 90° verdreht. Die Kennfelder besitzen, je ein Minimum und Maximum, dass bei Abfahren eines Kreises auf einem Kennfeld zur 360° Periodizität führt. Die Kennfelder entsprechen somit dem Kernverhalten des Winkelsensors [11]. Tabelle A.1 fasst die Grundeigenschaften der Kennfelder zusammen. Beim zusammensetzen der Kennfelder, sind die gemessenen Ausgangsspannungen normiert und von Offsets bereinigt worden. Das erleichtert den Simulationseinsatz mit variablen Betriebsspannungen. So können für beliebige Feldstärken, Ausgangsspannungen nach Gleichung A.1 aus den Kennfeldern entnommen werden.

$$V_{out}(H_x, H_y) = Gain \cdot \frac{V_{cos,sin}(H_x, H_y)}{Normfaktor} \cdot V_{CC} + V_{Offset} \quad (\text{A.1})$$

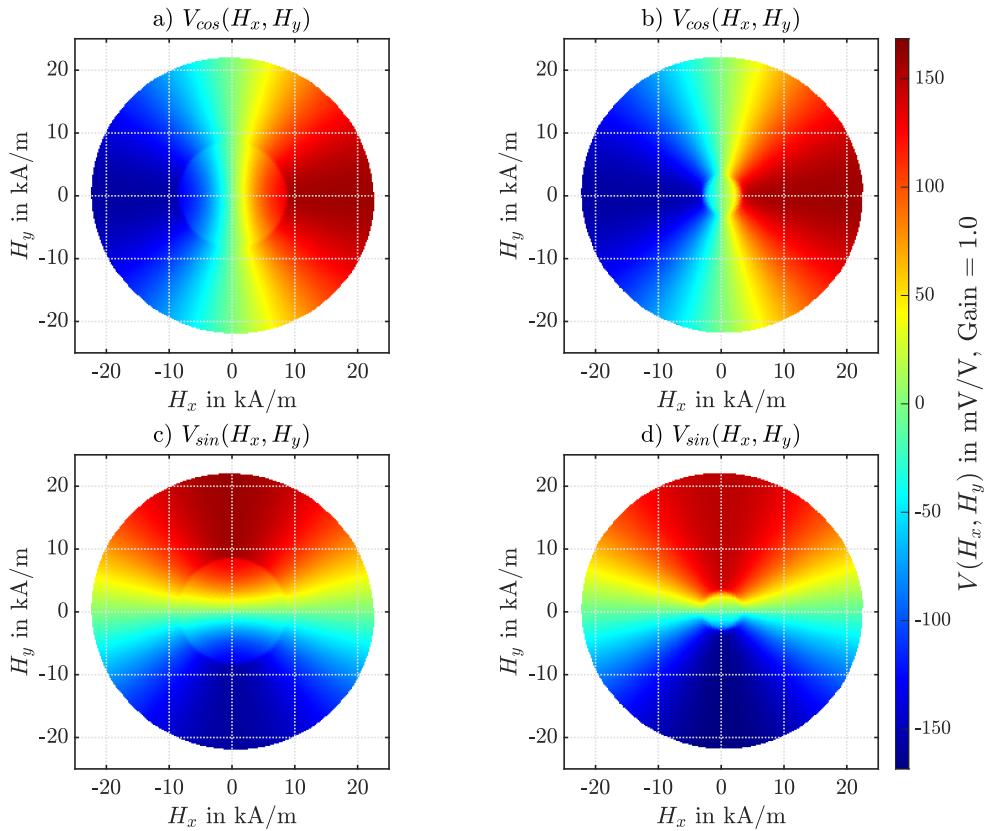


Abbildung A.1: TDK TAS2141-AAAB Brückenkennfelder. Kennfelder der Cosinus-Brücke a) und b). Kennfelder der Sinus-Brücke c) und d). a) und c) gewonnen aus steigenden Amplitudenmodulation. b) und d) gewonnen fallenden Modulation. Die Kennfelder sind normiert in mV V^{-1} . Grafik nachempfunden aus [15].

Im Vergleich der Kennfeldpaare biete sich das erste aus aus Abbildung A.1 a) und c) für eine Simulation an. Die Kennfelder besitzen größere Plateauflächen [15]. In Abbildung A.2 a) und c) ist das Kennfeldpaar nochmals gesondert dargestellt. Für das Kennfeld, der Cosinus-Wheatstone-Brücke in a), sind Querschnitte für variable H_x - und verschiedenen konstante H_y -Feldstärken in b) aufgetragen. Das gleiche vice versa in d) für Sinus-Wheatstone-Brücke aus c). Die Plateau-Grenzen liegen in H_x - und H_y -Richtung ca. bei $\pm 8,5 \text{ kA m}^{-1}$ und sind als Limits in Abbildung A.2 b) und d) gekennzeichnet. Es zeigt sich ein annähernd linearer Bereich für die Übertragungskennlinien bei $H_{x,y} = 0 \text{ kA m}^{-1}$. Dieser Arbeitsbereich ist für die Simulation einzustellen [15].

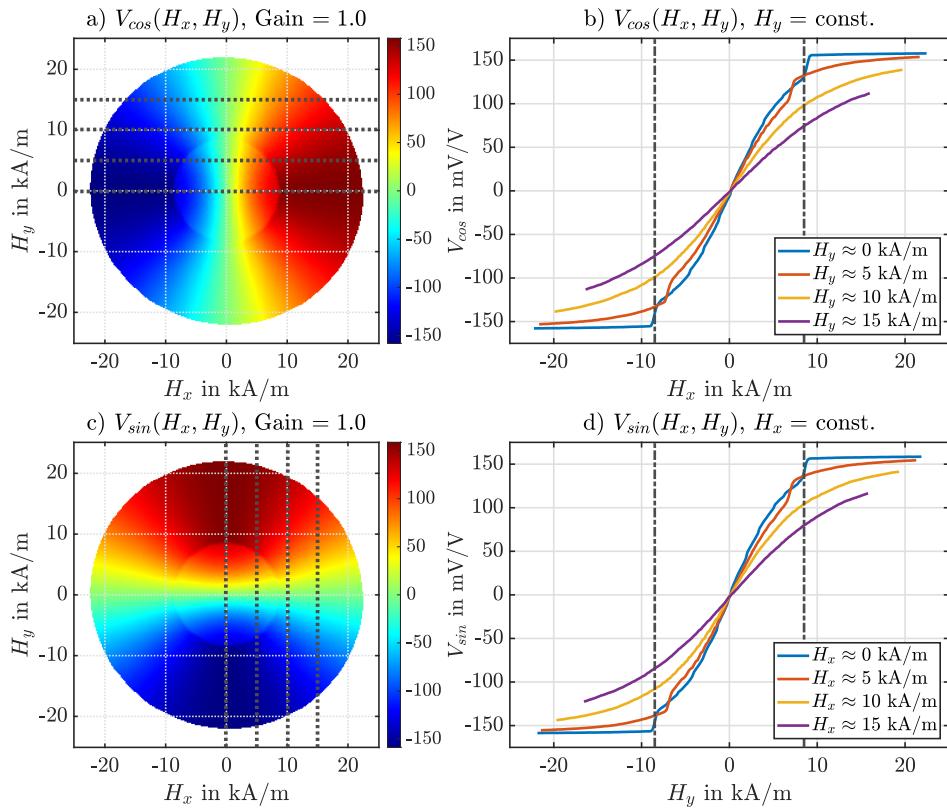


Abbildung A.2: TDK TAS2141-AAAB Kennfeldquerschnitte. Cosinus-Brücken-Kennfeld in a) und Sinus-Brücke in c). In b) und d) sind folgend der Verdrehung Querschnitte aus a) und c) aufgetragen. In b) V_{cos} f. variable H_x - und verschieden konst. H_y -Feldstärken. In d) vice versa für V_{sin} mit verschiedenen konst. H_x - bei variablen H_y -Feldstärken. Breite lineare Plateaus liefern einen annähernd linearer Arbeitsbereich in c) und d) zw. $\pm 8,5 \text{ kA m}^{-1}$ f. Übertragungskennlinien $H_{x,y} = 0 \text{ kA m}^{-1}$. Grafik aus [15].

B Sensor-Array-Simulation

Implementierung 0.0.1 07.04.2021

Der Anhang veranschaulicht die Handhabung und Standardparametrierung für die Sensor-Array-Simulation. Es sind optimale Simulationsparameter in Tabelle B.1 aufgeführt. Diese sind im Konfigurationsskript einzustellen und vorab der Simulation auszuführen. Das Skript erstellt eine MAT-Datei. Diese enthält, die in Gruppen zusammengefasste Simulationsparametrierung. Bei Simulationsausführung sind betreffende Parametergruppen aus der Konfigurationsdatei zu laden.

Parametergruppe	Parameter	Wert	Einheit	Kurzbeschreibung
SensorArrayOptions	geometry	'square'	-	Array-Geometrie-Indikator
	dimension	8	-	Sensor-Array-Pixel $N_{Pixel} \times N_{Pixel}$
	edge	2	mm	Sensor-Array-Kantenlänge
	V_{cc}	5	V	Sensor-Array-Betriebsspannung
	V_{off}	2,5	V	Sensor-Brücken-Offset-Spannung
	V_{norm}	$1 \cdot 10^3$	mV	Kennfeldnormierung
DipoleOptions	sphereRadius	2	mm	Kugelmagnetradius
	H_{0mag}	200	kA m ⁻¹	Betragsfeldstärke Magnetfeldnormierung
	z_0	1	mm	Z-Abstand Magnetfeldnormierung
	m_{0mag}	$1 \cdot 10^6$	A m ²	Magnitude d. mag. Moments
Training-/ TestOptions	useCase	'Training' / 'Test'	'char'	Datensatzindikator f. Anwendungszweck
	xPos	[0,]	mm	Sensor-Array X-Positionsvektor
	yPos	[0,]	mm	Sensor-Array Y-Positionsvektor
	zPos	[7,]	mm	Sensor-Array Z-Positionsvektor
	tilt	0	°	Magnetverkippung in Y-Achse
	angleRes	0,5	°	Winkelauflösung f. Magnetrotation
	phaseIndex	0	-	Phasenverschiebung-Index f. Startwinkel
	nAngles	20 / 720	-	Anzahl gleich verteilter Simulationswinkel
	BaseReference	'TDK'	char	Kennfelddatensatzindikator
	BridgeReference	'Rise'	char	Kennfeldindikator

Tabelle B.1: Sensor-Array-Simulationsparameter. Default-Parameter für die Simulation mit ideal ausgerichteten Gesamtsystem, bestehend aus mag. Dipol und Sensor-Array.

Die Sensor-Array-Simulation folgt der Aufführung in Algorithmus 1. Zur Generierung von Trainings-/ Testdatensätzen dient die Konfigurationsdatei als Eingabe. Entsprechend der Parametrierung in Tabelle B.1, sind notwendige Kennfelddatensätze initialisiert und Funktionsmodule eingebunden. Für die eingestellte Anzahl von Simulationswinkeln und angegebenen Winkelauflösung, wird ein Rotationsvektor aufgestellt, indem alle Simulationswinkel gleich verteilt sind. Die Simulation fährt alle X, Y, Z Sensor-Array-Positionen im Koordinatenraum ab. Für jede angefahrene Position wird ein Meshgrid und entsprechender Datensatz mit voller Rotation erzeugt. Für verschiedene Magnetverkippung ist die Konfigurationsdatei anzupassen und die Simulation zu wiederholen.

Algorithmus 1 : Sensor-Array-Simulation

Input : Konfigurationsdatensatz

Output : MAT-Dateipfad

Result : Sensor-Array-Datensätze (Training/ Test)

1. Laden der Konfigurierung \leftarrow Tabelle B.1;

2. Laden der Kennfelder \leftarrow Anhang A;

3. Initialisierung Simulationssparameter \leftarrow Tabelle B.1;

4. Initialisierung Rotation \leftarrow Gleichung 2.20, Gleichung 2.21;

5. Initialisierung Kennfelder \leftarrow Gleichung A.1;

6. Initialisierung Dipol-Rotationsmomente \leftarrow Gleichung 2.18;

7. Initialisierung Sensor-Array-Positionen \leftarrow Tabelle B.1;

8. Initialisierung Dipol-Feldnormierung \leftarrow Betragkonstante in Gleichung 2.22;

10. Speicherallokation f. Ergebnisse;

11. Anlegen Metadaten (Info) und Simulationsdaten (Data) Structs;

12. for z in Z -Positionsvektor **do**

for x in X -Positionsvektor **do**

for y in Y -Positionsvektor **do**

 Update Info-Struct (Positionsdaten);

 Initialisierung Sensor-Array-Meshgrid \leftarrow Gleichung 2.9;

for \vec{m}_i in Momentenmatrix **do**

 Normierte Dipol-Feldberechnung auf Meshgrid \leftarrow Gleichung 2.22;

 Kennfeld-Mapping interp2(Nearest-Neighbor) \leftarrow Abbildung B.1;

end

 Update Data-Struct (Ergebnisse);

 Speichern Info- und Data-Struct in Ziel-MAT-Datei;

 Ausgabe MAT-Dateipfad;

end

end

end

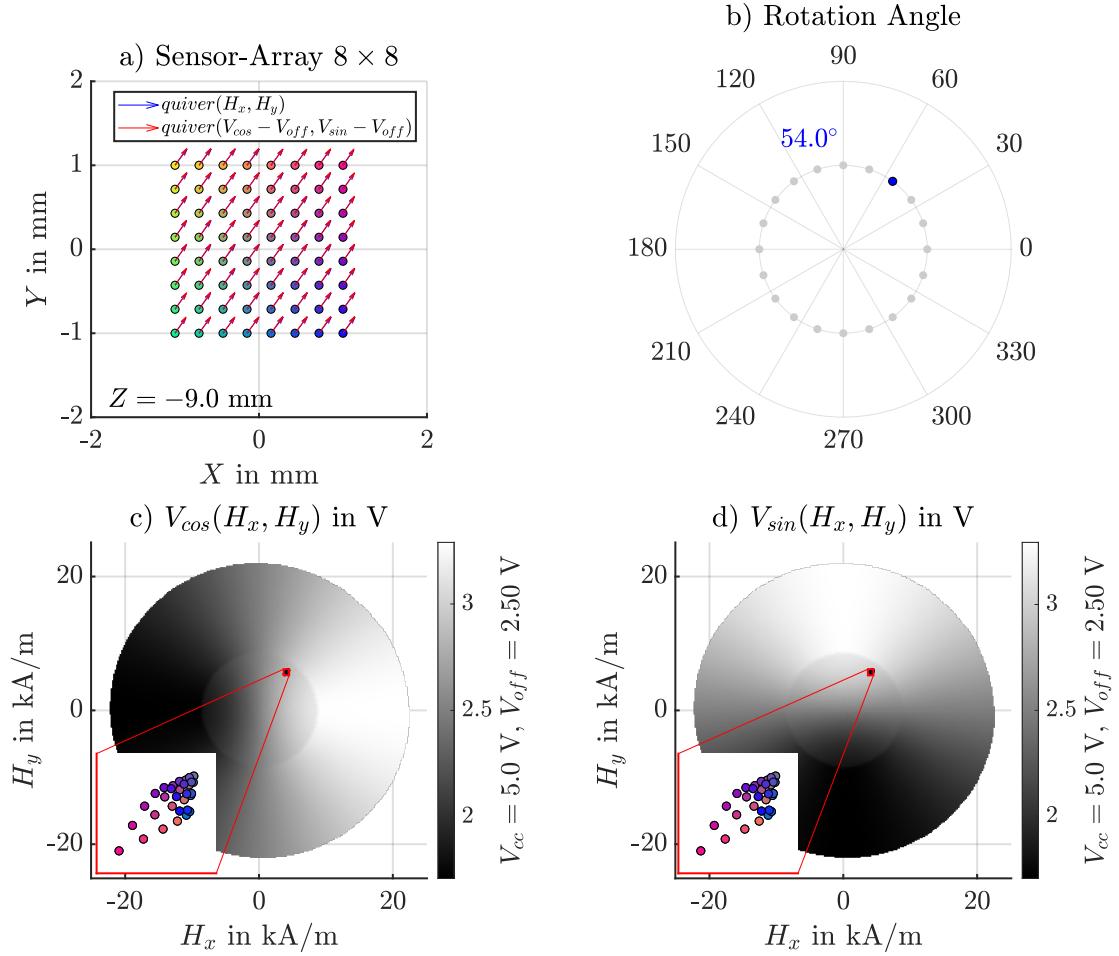


Abbildung B.1: Kennfeld-Mapping. Entnahme von Referenzspannungen aus Sensor-Kennfelder, gezeigt für einen beliebigen Simulationswinkel. In a) Meshgrid des Sensor-Arrays mit Grundposition $(0, 0, -9)^T$ mm relativ zum Koordinatenursprung (mag. Dipol). Simulation ohne Dipol-Verkippung. b) Simulation für 20 gleich verteilte Winkel, gezeigt ist der vierte Winkel bei 54° . Simulationsparameter sind wie in Tabelle B.1 eingestellt. Magnet und Array-Position sind ideal konfiguriert. Ersichtlich anhand des eng gegliederten Feldstärken-Mappings auf c) Cosinus-Kennfeld und d) Sinus-Kennfeld. Die simulierten Feldstärken für alle Sensor-Pixel-Koordinaten, liegen innerhalb des linearen Kennfeldarbeitsbereich. Die Kennfelder sind entsprechend Betriebsspannungsparametrierung in V umgerechnet.

Abbildung B.1 zeigt das Mapping, auf TMR-Sensor-Kennfeldern (Anhang A), für prozessierte H_x - und H_y -Feldstärken. Die Simulation ist mit der Standardparametrierung aus Tabelle B.1 ausgeführt worden. Hier am Beispiel für 20 Simulationswinkel. Gezeigt ist der vierte Winkel bei 54° . Die errechneten Feldstärken sind auf die Kennfelder projiziert und Spannungswerte mittels Matlab-2D-Interpolation für Nearest-Neighbor entnommen. Abbildung B.2 zeigt, das nochmals für 720 Winkel und vier Sensor-Pixel. Es sind die Eck-Pixel angezeigt. Da sich der mag. Dipol ohne Verkipfung, zentriert über dem Sensor-Array befindet, überlagern sich die Signalverläufe für diagonal gegenüberliegende Sensor-Pixel. Die leicht ellipsenförmige Verlauf der projizierten Feldstärken, der äußeren Pixel, ergibt sich durch den Versatz zur Magnetfeldmitte. Ein Pixel direkt lotrecht zur Magnet-Z-Achse platziert, erzeugt einen optimale Kreisbahn. Eine detaillierte Beschreibung der Skripte und zugehöriger Funktionsmodule ist in Anhang E einzusehen.

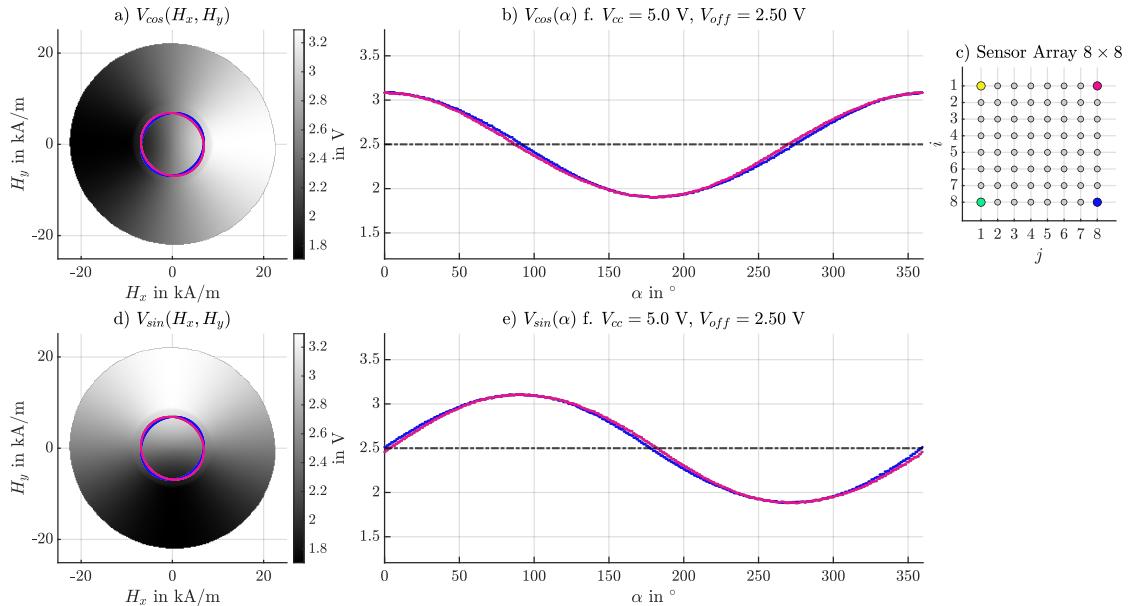


Abbildung B.2: Sensor-Array-Datensatz Teilansicht. Teilansicht der Simulationsergebnisse, entsprechend der Parametrierung nach Tabelle B.1. Es ist der gesamte Simulationsdurchlauf für die Eck-Sensor-Pixel c) gezeigt. a) und d) zeigen das Kennfeld-Mapping, für korrespondierende Feldstärken, bei einer vollständigen Dipol-Drehung. b) und e) stellt die Rotation aufgetragen über alle Simulationswinkel dar. Die sich diagonal gegenüberliegenden Sensor-Pixel, überlagern sich in den Darstellungen a), b), d) und e). Das entspricht der Kreuzsymmetrieeigenschaft Dipol-Magnetfeldes. Das Sensor-Array ist zentriert zum Dipol ausgerichtet. Dipol ist nicht verkippt. Grafik nachempfunden aus [15]

C Gauß-Prozess-Regression

Implementierung 0.0.1 13.04.2021

Im Anhang befindet sich die Beschreibung, der Regression mittels Gauß'scher Prozesse. Die implementierten Mechanismen sind auf die Sensor-Array-Simulation Anhang B angepasst. Es wird sich an der Anforderungsbeschreibung für ein TMR-Sensor-Array Abschnitt 2.5 orientiert. Eine Standardparametrierung für die Simulationsdurchführung ist in Tabelle C.1 einzusehen. Die Notation ist der kompakten Schreibweise für Gauß'sche Prozesse [3] angepasst. Implementiert ist ein Kernel aus den Vorarbeiten [18][19]. Darauf aufbauend ist ein zweiter mit angepasster Eingangswertverarbeitung entworfen worden. Beide Kernel besitzen die Fähigkeit zur Mittelwert freien und Polynom gestützten Regression bzw. Vorhersage [3]. Das Regressionsmodell kann ganze Sensor-Array-Datensätze verarbeiten. Es sind verschiedene Qualitätskriterien implementiert, die Aussagen zur Modellgenauigkeit, Vorhersagesicherheit und Generalisierung treffen. Die Trainingsphase für das Regressionsmodell wird durch Algorithmus 7 durchgeführt. Eine anschließende Arbeitsphase ist durch Algorithmus 6 umgesetzt. Einen Überblick über die Gesamtsoftware, in der dieser Teil ein Modul einnimmt, ist im Anhang E einzusehen. Der Anhang besitzt einen Glossar ähnlichen Charakter und soll als Schnellnachschlagewerk unterstützen.

Parametergruppe	Parameter	Wert	Einheit	Kurzbeschreibung
GPROptions	kernel	'QFCAPX'	char	Kernel-Funktion-Indikator (C.4), ' $QFC \leftarrow d_F^2$ '
	θ	(1, 1)	-	Kernel-Parametervektor θ (C.5)
	σ_f^2 -Bounds	(0.1, 100)	-	Parameter-Bounds θ_1 f. Algorithmus 5
	σ_l -Bounds	(0.1, 100)	-	Parameter-Bounds θ_2 f. Algorithmus 5
	σ_n^2	$1 \cdot 10^{-6}$	-	Rauschniveau, Rauschauaufschaltung (C.6)
	σ_n^2 -Bounds	($1 \cdot 10^{-8}, 1 \cdot 10^{-4}$)	-	Parameter-Bounds σ_n^2 f. Algorithmus 7
	OptimRuns	30	-	Durchlaufanzahl f. Algorithmus 7
	SLL	'SLLA'	char	Verlust-Indikator f. Winkel (A)/ R (Radius) Algorithmus 7
	mean	'zero'	char	Indikator Mittelwertpolynom Ein ('poly')/ Aus ('zero')
	polyDegree	1	-	Grad des Mittelwertpolynoms wenn mean = 'poly'

Tabelle C.1: Gauß-Prozess-Regression-Simulationsparameter. Default-Parameter für die Prozessierung von Simulationsergebnissen aus der Sensor-Array-Simulation Anhang B.

C.1 Modellinitialisierung

Die Modellinitialisierung zur Gauß-Prozess-Regression erfolgt nach Algorithmus 2. Dabei sind Modellparametrierung über einen Konfigurationsdatensatz Tabelle C.1 zu laden. Ebenfalls sind alle gewählten Trainingsdaten, mit dazugehörigen Simulationswinkel $X \mapsto \alpha_{Ref}$, in die Initialisierung einzuspeisen. Das Modell wird Schritt für Schritt in einem Struct aufgebaut. Dabei sind bestimmte Funktionalitäten, entsprechend der gewählten Konfigurierung, in Funktions-Handles zugewiesen. So sind die Schritte 2, 4, 5 und 7 als Funktions-Handles umgesetzt. Das verringert den Speicheraufwand und benötigte Rechenergebnisse können dynamisch bei Bedarf erzeugt werden. Nach der Initialisierung müssen, Modellkonfiguration aus Schritt 1, die Regressionsziele aus Schritt 3, die L -Matrix aus Schritt 8 und die Regressionsgewichte aus Schritt 12 als gespeicherte Werte, für die Vorhersage nach Algorithmus 6 vorliegen. Die Modellkonfiguration aus Schritt 1 und die berechneten Modellplausibilitäten aus Schritt 13 sind für die Modelloptimierung nach Algorithmus 5 entscheidend. Das Modell wird in der Optimierung z.T. reinitialisiert. Die einzelnen Initialisierungsschritte sind nachfolgend zusammengefasst aufgeführt. Es ist ein mathematisch Bezug zur Implementierung in Anhang E und Notation nach Fachliteratur [3] vorgenommen worden.

Algorithmus 2 : Modellinitialisierung mit konst. Trainingsdaten und Parametern

Input : Konfigurationsdatensatz, Trainingsdatensatz $X \mapsto \alpha_{Ref}$

Result : Regressionsmodell mit Fähigkeit zur Datensatzverarbeitung aus Anhang B

1. Initialisierung Modellkonfiguration \leftarrow Tabelle C.1;
 2. Initialisierung X, α und X -Formatierung \leftarrow Gleichung C.1, Gleichung C.2;
 3. Initialisierung Regressionsziele \leftarrow Gleichung C.3;
 4. Initialisierung Kernel-Funktion \leftarrow Gleichung C.4;
 5. Initialisierung Basis-Funktion \leftarrow Gleichung C.8;
 6. Berechnung $K(X, X|\theta) \leftarrow$ Algorithmus 3;
 7. Rauschaufschaltung $K_y \leftarrow$ Gleichung C.6;
 8. Cholesky-Zerlegung von K_y zu L u. Berechnung $\log |K_y| \leftarrow$ Gleichung C.7;
 9. Initialisierung Mittelwertpolynome \leftarrow Gleichung C.9;
 10. Berechnung Polynomkoeffizienten \leftarrow jeweils Algorithmus 4 f. Gleichung C.10;
 11. Initialisierung Mittelwertfunktion \leftarrow Gleichung C.11;
 12. Berechnung Regressionsgewichte \leftarrow Gleichung C.12;
 13. Berechnung Modellplausibilität \leftarrow Gleichung C.13;
-

Trainingsdatensatz definiert nach der kompakten Notation aus [3]. Ein Trainingsdatensatz X beinhaltet alle Referenzwinkelstellungen α_i mit $X_i \mapsto \alpha_i$, nach der Beschreibung in Abschnitt 2.5 mit $X_{cos,i} = A_x$ und $X_{sin,i} = A_y$. Für die Implementierung mit Gleichung 2.7 müssen alle Trainingsdatenmatrizen normiert sein, sodass sich Vektoren als Trainingsdaten abbilden, siehe Gleichung C.1.

$$X = [X_i, \dots X_{N_{Ref}}] \quad \text{f. } i = 1, 2, 3, \dots, N_{Ref} \quad (\text{C.1})$$

$$X_i = \begin{cases} [X_{cos,i}, X_{sin,i}] & \text{f. } d_F^2 \text{ (2.13)} \\ [\|X_{cos,i}\|_F, \|X_{sin,i}\|_F] & \text{f. } d_E^2 \text{ (2.7)} \end{cases}$$

$$X_i \mapsto \alpha_i$$

Testdatensatz definiert in kompakter Schreibweise nach [3]. Ein Testdatensatz X_* repräsentiert einen Testwinkel mit $X_* \mapsto \alpha_*$. Auch hier gilt, wie für Trainingsdatensätze $X_i \mapsto \alpha_i$, die Umschreibung nach Abschnitt 2.5 mit $X_{cos*} = A_x$ und $X_{sin*} = A_y$. Jeweils für die gewählte Implementierung ist auch hier eine Eingangsverarbeitung der Datensätze notwendig Gleichung C.2, sodass die Implementierung nach Gleichung 2.7 mit Skalaren statt Matrizen arbeitet.

$$X_* = \begin{cases} [X_{cos*}, X_{sin*}] & \text{f. } d_F^2 \text{ (2.13)} \\ [\|X_{cos*}\|_F, \|X_{sin*}\|_F] & \text{f. } d_E^2 \text{ (2.7)} \end{cases} \quad (\text{C.2})$$

$$X_* \mapsto \alpha_*$$

Regressionsziele sind als Spaltenvektoren nach [3] wie in Gleichung C.3 definiert. Die Besonderheit hier sind zwei Zielvektoren statt einer, wie in der Fachliteratur [3] angegeben. Abstrahiertes Regressionsziel ist der Einheitskreis, daher ergeben sich einfache Sinoide aus den Referenzwinkeln in Gleichung C.3.

$$y_{cos} = (\cos \alpha_i, \dots, \cos \alpha_{N_{Ref}})^T \quad \text{f. } i = 1, 2, 3, \dots, N_{Ref} \quad (\text{C.3})$$

$$y_{sin} = (\sin \alpha_i, \dots, \sin \alpha_{N_{Ref}})^T$$

Kernel-Funktion als Kernelement des Regressionsverfahren für Gauß'sche Prozesse [3]. Es sind zwei Versionen nach gleichen Vorbild der fraktalen Kovarianz [18][19] implementiert. Die Implementierung mittels Gleichung 2.13 resultiert aus den Vorarbeiten der Arbeitsgruppe Sensorik und stellt die genaue Lösung dar und arbeitet direkt mit Matrizen als Trainingsdaten. Die Implementierung nach mit Gleichung 2.7 ist innerhalb dieser Arbeit entstanden und bedingt ein vorab Prozessieren der Trainingsdaten zu Vektoren. Ebenfalls müssen weitere Testdaten eingangs zu skalaren verarbeitet werden. Dazu wird die Frobenius-Norm aus Gleichung 2.10 verwendet. Die Implementierung nach Gleichung 2.7 folgt dem Beispiel, aus einer Veröffentlichung der TU-München [7], für die Anwendung der Gauß-Prozess-Regression auf Themenfeld der Robotik.

$$k(X_i, X_j) = \begin{cases} \frac{a}{b+d_F^2 \langle X_i, X_j \rangle} & \text{f. } d_F^2 \text{ (2.13)} \\ \frac{a}{b+d_E^2 \langle X_i, X_j \rangle} & \text{f. } d_E^2 \text{ (2.7)} \end{cases} \quad (\text{C.4})$$

mit $i, j = 1, 2, 3, \dots, N_{Ref}$

Kernel-Parameter für die Kovarianz- oder Kernel-Funktion bilden sich die Funktionsparameter, wie in Gleichung C.5 beschrieben. Bei der Parameteridentifizierung ist das Auslöschkriterium für gültige Kovarianzfunktionen elementar [3]. Dabei ist der Fall abzudecken, dass die Abstandsfunktion der Kovarianz zu null wird, wenn Datensätze mit sich selbst prozessiert werden. In diesem Fall muss die Kovarianzfunktion σ_f^2 ergeben.

$$a = \sigma_f^2 \cdot 2\sigma_l^2 \quad b = 2\sigma_l^2 \quad \theta = [\sigma_f^2, \sigma_l] \quad (\text{C.5})$$

Kovarianzmatrix als Autokorrelationsergebnis, ist für alle bereitgestellten Trainingsdaten untereinander mit Algorithmus 3 zu berechnen [3]. Das Ergebnis, in quadratischer Matrixform, definiert das Verhalten des Gesamtsystems über gemessene Einzelabstände der Trainingsdaten zueinander. Es bedeutet, dass für ein Sensor-Array auf Basis des TMR-Sensors [11] in Drehwinkelapplikation Abbildung 2.9, die 360° Periodizität ersichtlich sein muss. Was eine Maxima-Diagonale von links nach rechts und annähernde Max-Werte in den beiden übrigen Ecken der Matrix impliziert.

Algorithmus 3 : Berechnung der Kovarianzmatrix $K(X, X|\theta)$

Input : Kernel-Funktion $k(X_i, X_j)$, Trainingsdaten X , Kernel-Parameter θ

Result : K -Matrix $N_{Ref} \times N_{Ref}$

```

1. Initialisierung Parameter  $(a, b) \leftarrow \theta$  Gleichung C.5;
2. for  $i = 1, 2, 3, \dots, N_{Ref}$  do
    | for  $j = 1, 2, 3, \dots, N_{Ref}$  do
        | |  $K_{i,j} = k(X_i, X_j) \leftarrow$  Gleichung C.4;
    | end
end

```

Rauschaufschaltung durch konstantes Rauschniveau σ_n^2 und minimaler Anhebung der Kovarianzmatrix-Diagonalen, für verrauschte bzw. fehlerbehaftete Regression [3]. Entsprechend der Fachliteratur ist die Kovarianzmatrix inklusive additives Rauschen als K_y bezeichnet [3].

$$K_y = K(X, X|\theta) + \sigma_n^2 I \quad (\text{C.6})$$

Cholesky-Zerlegung K_y als Ansatz zum Lösen der Regressionsmechanismen. Dem Regressionsverfahren zugrundeliegenden Lösungen der Wahrscheinlichkeitsdichte-Integrale sind über Matrix- und Vektormultiplikation mit Inversen Matrizen in linearen Gleichungssystemen gelöst [3]. Für das Lösen der Gleichungssysteme mit inversen Matrizen, ist die Zerlegung der nicht inversen Matrix in eine untere Dreiecksmatrix möglich. Die Cholesky-Zerlegung schafft entsprechenden Repräsentant L für die Matrix K_y . Die logarithmierte Determinante ist mit Gleichung C.7 zu berechnen. Matrizen müssen symmetrisch und positiv definit sein, um die Cholesky-Zerlegung anwenden zu können [3].

$$LL^T = K_y \quad (C.7)$$

$$\log|K_y| = 2 \sum_{i=1}^{N_{Ref}} \log L_{i,i}$$

Es sollen damit Gleichungssysteme wie $Ax = b \Leftrightarrow x = A^{-1}b$ über zerlegten Repräsentanten gelöst sein $Ly = b \Leftrightarrow L^Tx = y$. Als Notation für die notwendige Lösung der linearen Gleichungssystem ist der Backslash-Operator genutzt $x = L^T \backslash (L \backslash b)$ [3]. Der Backslash-Operator steht ebenfalls in Matlab zur Verfügung.

Basis-Funktion als Aufbaufunktion für Polynome aus Trainings- und Testdaten. Das Regressionsverfahren mittels Gauß'scher Prozesse kann in zwei Ausführungen betrieben werden. Die erste ist eine Regression ohne weitere Mittelwerte als Regressionshilfe. In zweiter Ausführung können Mittelwerte der Daten z.B. über Polynomfindung gebildet sein. Dafür bedingt es eine Basis-Funktion nach Gleichung C.8, die entsprechend der Kernel-Funktion und resultierende Datenformate Polynome bildet [3]. In der Implementierung sind Polynome ersten Grades verwendet. Was einer Offset- und Amplitudenkorrektur der Trainings- und Testdaten entspricht. Die in Gleichung C.8 gezeigten Funktionen müssen, jeweils für beide Cosinus und Sinus Datentypen gebildet werden und beziehen sich hier in der Darstellung für einen einzigen Simulationswinkel $X_* \mapsto \alpha_*$.

$$h_{\cos}(X_{\cos*}) = \begin{cases} 0 & \text{f. } m_{\cos}(X_{\cos*}) = 0 \\ (1, \|X_{\cos*}\|_F, \|X_{\cos*}\|_F^2, \dots)^T & \text{f. } d_F^2 \text{ (2.13), } m_{\cos}(X_{\cos*}) \neq 0 \\ (1, X_{\cos*}, X_{\cos*}^2, \dots)^T & \text{f. } d_E^2 \text{ (2.7), } m_{\cos}(X_{\cos*}) \neq 0 \end{cases} \quad (\text{C.8})$$

$$h_{\sin}(X_{\sin*}) = \begin{cases} 0 & \text{f. } m_{\sin}(X_{\sin*}) = 0 \\ (1, \|X_{\sin*}\|_F, \|X_{\sin*}\|_F^2, \dots)^T & \text{f. } d_F^2 \text{ (2.13), } m_{\sin}(X_{\sin*}) \neq 0 \\ (1, X_{\sin*}, X_{\sin*}^2, \dots)^T & \text{f. } d_E^2 \text{ (2.7), } m_{\sin}(X_{\sin*}) \neq 0 \end{cases}$$

Mittelwertpolynome bauen sich über die Basis-Funktionen aus Gleichung C.8 für Trainingsdaten auf. Es resultieren Matrizen [3], deren erste Reihe gleich eins ist und jede weitere Reihe mit entsprechenden Exponenten für die Polynomgenerierung versehen ist. Die Polynombildung ist jeweils für beide Cosinus- und Sinus-Datensätze durchzuführen, wenn die Mittelwertbildung aktiv ist.

$$H_{\cos}(X_{\cos}) = \begin{cases} 0 & \text{f. } m_{\cos}(X_{\cos}) = 0 \\ [h_{\cos}(X_{\cos,i}), \dots, h_{\cos}(X_{\cos,N_{Ref}})] & \text{f. } m_{\cos}(X_{\cos}) \neq 0 \end{cases} \quad (\text{C.9})$$

$$H_{\sin}(X_{\sin}) = \begin{cases} 0 & \text{f. } m_{\sin}(X_{\sin}) = 0 \\ [h_{\sin}(X_{\sin,i}), \dots, h_{\sin}(X_{\sin,N_{Ref}})] & \text{f. } m_{\sin}(X_{\sin}) \neq 0 \end{cases}$$

jeweils für alle $X_{\cos} = [X_{\cos,i}, \dots, X_{\cos,N_{Ref}}]$ und

alle $X_{\sin} = [X_{\sin,i}, \dots, X_{\sin,N_{Ref}}]$

mit $i = 1, 2, 3, \dots, N_{Ref}$

Polynomkoeffizienten zur Mittelwertbildung über gebildet Polynome nach Gleichung C.8 und Gleichung C.9, sind benötigte Polynomkoeffizienten nach Gleichung C.10 zu berechnen [3]. Zur Berechnung der Koeffizienten sind mehrere inverse Matrix-Produkte verschachtelt zu lösen.

$$\beta_{cos} = \begin{cases} 0 & \text{f. } m_{cos}(X_{cos}) = 0 \\ (H_{cos}K_y^{-1}H_{cos}^T)^{-1}H_{cos}K_y^{-1}y_{cos} & \text{f. } m_{cos}(X_{cos}) \neq 0 \end{cases} \quad (\text{C.10})$$

$$\beta_{sin} = \begin{cases} 0 & \text{f. } m_{sin}(X_{sin}) = 0 \\ (H_{sin}K_y^{-1}H_{sin}^T)^{-1}H_{sin}K_y^{-1}y_{sin} & \text{f. } m_{sin}(X_{sin}) \neq 0 \end{cases}$$

jeweils für alle $X_{cos} = [X_{cos,i}, \dots, X_{cos,N_{Ref}}]$ und

alle $X_{sin} = [X_{sin,i}, \dots, X_{sin,N_{Ref}}]$

mit $i = 1, 2, 3, \dots, N_{Ref}$

Zur Bewältigung des Problems ist Algorithmus 4 implementiert worden und jeweils für beide Cosinus- und Sinus-Polynome aus Gleichung C.9 durchzuführen. Die Polynom- und Koeffizientenbestimmung entfällt, wenn die Mittelwertbildung ausgeschaltet ist.

Algorithmus 4 : Berechnung der β Polynomkoeffizienten aus Gleichung C.10

Input : Polynommatrix H , Untere Dreiecksmatrix $L(K_y)$, Regressionsziel y

Result : β -Koeffizienten

1. $a_0 \leftarrow$ Lösen von $K_y^{-1}y$;
 $a_0 = L^T \setminus (L \setminus y)$;
 2. $A_1 \leftarrow$ Lösen von $HK_y^{-1}H^T$;
for j -te Spalte in H^T **do**
| $V_j = L \setminus H_j^T$;
| **end**
| $A_1 = V^T V$;
 3. $L_1 \leftarrow$ cholesky(A_1);
 4. $A_2 \leftarrow$ Lösen von $A_1^{-1}H$;
for j -te Spalte in H **do**
| $V_j = L_1^T \setminus (L_1 \setminus H_j)$;
| **end**
| $A_2 = V$;
 5. $\beta = A_2 \cdot a_0$;
-

Mittelwertfunktionen für die Regression setzen sich aus gebildeten Polynomen und den bestimmten Polynomkoeffizienten nach Gleichung C.11 zusammen [3]. Für alle Trainingsdaten mittels Polynommatrizen und für einzelne Testdaten über die Basis-Funktion. Bei eingeschalteter Mittelwertbildung, bildet sich das Regressionsergebnis über die Summe aus Mittelwertberechnung und Stützwertsumme [3]. Die Mittelwertrechnung ist für beide Cosinus- und Sinus-Datensätze umzusetzen.

$$m_{cos}(X_{cos(*)}) = \begin{cases} 0 & \text{f. mittelwertfreie Regression} \\ H_{cos}(X_{cos}) \cdot \beta_{cos} & \text{f. Trainingsdaten } X_{cos} \\ h_{cos}(X_{cos*}) \cdot \beta_{cos} & \text{f. Testdaten } X_{cos*} \end{cases} \quad (\text{C.11})$$

$$m_{sin}(X_{sin(*)}) = \begin{cases} 0 & \text{f. mittelwertfreie Regression} \\ H_{sin}(X_{sin}) \cdot \beta_{sin} & \text{f. Trainingsdaten } X_{sin} \\ h_{cos}(X_{sin*}) \cdot \beta_{sin} & \text{f. Testdaten } X_{sin*} \end{cases}$$

jeweils für alle $X_{cos} = [X_{cos,i}, \dots, X_{cos,N_{Ref}}]$ und
 alle $X_{sin} = [X_{sin,i}, \dots, X_{sin,N_{Ref}}]$
 mit $i = 1, 2, 3, \dots, N_{Ref}$

Regressionsgewichte oder Stützwerte für die Vorhersage beider Sinoide sind jeweils, in Abhängigkeit der dazugehörigen Regressionsziele und Mittelwerte, über inverse Matrix-Produkt aus K_y^{-1} und das Residual aus Ziel und Mittelwert zu bilden. Gleichung C.12 beschreibt die Lösung des resultierenden Gleichungssystem über die untere Dreiecksmatrix L der Kovarianzmatrix K_y [3]. Die Gewichtsbildung veranschaulicht am besten den Gesamtlauf des Verfahrens. Es sind zwei unterschiedliche Regressionen, jeweils für Cosinus- und Sinus-Funktionen durchzuführen. Dabei stützen sich beide Regressionen auf eine gemeinsame Kovarianzbewertung, der zugrundeliegenden Trainingsdatensätze [19]. Die Kovarianzmatrix stellt somit die vektorielle und orthogonale Kopplung der Daten her und impliziert ihre gegenseitige Abhängigkeit.

$$\begin{aligned}\alpha_{cos} &= K_y^{-1} \cdot (y_{cos} - m_{cos}(X_{cos})) \\ &= L^T \backslash (L \backslash (y_{cos} - m_{cos}(X_{cos})))\end{aligned}\tag{C.12}$$

$$\begin{aligned}\alpha_{sin} &= K_y^{-1} \cdot (y_{sin} - m_{sin}(X_{sin})) \\ &= L^T \backslash (L \backslash (y_{sin} - m_{sin}(X_{cos})))\end{aligned}$$

jeweils für alle $X_{cos} = [X_{cos,i}, \dots, X_{cos,N_{Ref}}]$ und
alle $X_{sin} = [X_{sin,i}, \dots, X_{sin,N_{Ref}}]$
mit $i = 1, 2, 3, \dots, N_{Ref}$

Modellplausibilitäten oder Regressionsevidenzen, sind entsprechend der Regressionsausrichtung, über Residuale und Regressionsgewichte in Gleichung C.13 zu bilden [3]. Jeweils wieder für beide Cosinus- und Sinus-Datensätze. Die so aufgestellten Plausibilitäten bewerten den Regressions-Fit in Bezug auf die Trainingsdaten. In der Fachliteratur [3] sind diese auch als Logarithmic-Marginal-Likelihoods bezeichnet. Sie bieten einen Indikator für den Daten-Fit, der < 0 wird für eine schlechte Anpassung, ≈ 0 ist bei mäßiger Anpassung und > 0 ist für eine gute Modellanpassung. Dabei sind Werte > 30 als sehr gute Anpassung Modellanpassung für jeweilige Sinoide zu interpretieren. Bei Plausibilitäten größer > 60 und zu eng gewählter Parameter-Bounds, kann sich ein zu Starker Fit auf die Trainingsdaten einstellen. Das wird als Overfitting bezeichnet. Ein so über parametrisiertes Modell, verliert dabei seine Fähigkeit zur Generalisierung und liefert nur für die Trainingsdaten selber valide Ergebnisse. Testdaten, die von den Trainingsdaten abweichen, können somit nicht mehr korrekt prozessiert werden. Die Interpretation der Likelihoods ist aus der Fachliteratur [3] entnommen und für empfohlene Werte empirisch bestimmt worden. Die einzelnen Plausibilitäten müssen ungefähr gleich groß sein, andernfalls besteht ein Ungleichgewicht in der Vorhersage. Resultierende Ergebnisse sind dann im Winkel und Radius verfälscht. Hergestellt wird das Gleichgewicht durch die Kovarianzkopplung, mittels gemeinsamer Kovarianzmatrix.

$$\log p(y_{cos}|X_{cos}) = -0,5 \left((y_{cos} - m_{cos}(X_{cos}))^T \alpha_{cos} + \log |K_y| + N_{Ref} \log 2\pi \right) \quad (\text{C.13})$$

$$\log p(y_{sin}|X_{sin}) = -0,5 \left((y_{sin} - m_{sin}(X_{sin}))^T \alpha_{sin} + \log |K_y| + N_{Ref} \log 2\pi \right)$$

jeweils für alle $X_{cos} = [X_{cos,i}, \dots, X_{cos,N_{Ref}}]$ und
alle $X_{sin} = [X_{sin,i}, \dots, X_{sin,N_{Ref}}]$
mit $i = 1, 2, 3, \dots, N_{Ref}$

C.2 Modelloptimierung

Die Optimierung bezieht sich auf ein fertig initialisiertes Modell nach Algorithmus 2. Dieses muss dafür einen vollständigen Parametersatz Tabelle C.1 inklusive Bounds beinhalten. Ebenfalls müssen alle Trainingsdaten entsprechend der gewählten Implementierung im Modell enthalten sein. Die Optimierung in Algorithmus 5 ist mittels Fmincon-Funktion (Matlab) implementiert und nutzt einen Sequential-Quadratic-Programming-Algorithmus um das Minimum-Kriterium aus Gleichung C.14 zu steuern. Das Modell wird im Prozess so lange reinitialisiert, bis keine graduelle Änderung des Kriteriums mehr festgestellt werden können. Kritisch im Optimierungsverfahren sind dabei, die zu setzenden Parameter-Bounds für θ . Sind die Grenzen des Suchfeldes zu eng gesetzt, kann das Minimum nicht erreicht werden. Der Algorithmus wird dann die Bounds selbst als Ergebnis liefern. Sind die Bounds zu weit abgesteckt ist es theoretisch möglich, dass das Minimum nicht vor Abbruch gefunden werden kann. In der Regel findet sich das Minimum, mit der getroffenen Implementierung, nach 6 – 24 Durchläufe. Das ist empirisch durch ausgegebene Grafiken beobachtet worden.

Algorithmus 5 : Modelloptimierung über Fmincon-Funktion f. $\sigma_n^2 = \text{konst.}$

Input : Modell inkl. $X, \theta, \sigma_n^2 + \text{Bounds} \leftarrow \text{Algorithmus 2}$

Result : Optimiertes Modell mit neuen Kernel-Parameter $\theta|\sigma_n^2$, f. $\sigma_n^2 = \text{konst.}$

1. Initialisierung Fmincon-Funktion;
 2. Initialisierung Parameter-Bounds \leftarrow Modell-Bounds Tabelle C.1;
 3. Initialisierung Fmincon-Startwert \leftarrow Model-Kernel-Parameter Tabelle C.1;
 4. Initialisierung Min-Kriterium $\tilde{R}_{\mathcal{L}} \leftarrow$ Gleichung C.14;
 5. **while** $\neg(\tilde{R}_{\mathcal{L}} = \text{konst. f. 7 Iterationen}) \wedge (\min \neq \tilde{R}_{\mathcal{L}})$ **do**
 - | Zuweisung innerhalb Parameter-Bounds $\theta \leftarrow$ Fmincon-Funktion;
 - | Modell-Teilreinitialisierung \leftarrow Algorithmus 2, Schritte 6. bis 13.;
 - | Berechnung $\tilde{R}_{\mathcal{L}} \leftarrow$ Gleichung C.14;
 6. **end**
 7. Speichern $\theta \leftarrow$ Fmincon-Funktion;
 8. Modell-Teilreinitialisierung \leftarrow Algorithmus 2, Schritte 6. bis 13.;
-

Min-Kriterium als zusammengesetztes Kriterium aus den einzelnen Modellplausibilitäten für die Cosinus- und Sinus-Vorhersage. Der Bildungsansatz ist aus einem Regressionsproblem der Computer-Vision [5] adaptiert und nach dem Leitwerk zur Verfahrensentwicklung [3] angepasst worden. Die Findung optimaler Kovarianz- bzw. Kernel-Parameter, kann nach Gleichung Gleichung C.14 vorgenommen werden. Dafür sind Modellplausibilitäten, für die einzelnen Sinoiden, als Funktion von Kernel-Parametern zu betrachten. Das Kriterium ergibt sich, als negative Summe der einzelnen Plausibilitäten Gleichung C.13. Das aufgestellte Minimierungsproblem ist bei einem konstanten Rauschniveau σ_n^2 und verschiedenen Kernel-Parameter θ zu untersuchen.

$$\theta|\sigma_n^2 = \arg \min_{\theta} \tilde{R}_{\mathcal{L}}(\theta|\sigma_n^2) \quad \text{f. } \sigma_n^2 = \text{konst.}$$

(C.14)

$$\tilde{R}_{\mathcal{L}}(\theta|\sigma_n^2) = -(\log p(y_{cos}|X_{cos}, \theta, \sigma_n^2) + \log p(y_{sin}|X_{sin}, \theta, \sigma_n^2))$$

C.3 Modellvorhersagen

Die Implementierung für Winkelvorhersagen, auf Grundlage von Datensätzen der Sensor-Array-Simulation Anhang B, läuft nach Algorithmus 6 ab. Der Algorithmus zeigt, die Vorhersage für eine Winkelstellung und ist für mehrere Winkel zu wiederholen. Die geschriebene Software in Anhang E, kann einen Winkelsatz oder komplett Datensätze, bestehend aus mehreren Winkelsätzen prozessieren. Für letzteres stehen Ergebnisse als Vektoren zur weiteren Analyse oder Optimierung bereit.

Algorithmus 6 : Modellvorhersage f. Sinoide eines Testwinkel mit $X_* \mapsto \alpha_*$

Input : Modell inkl. X, θ, σ_n^2 , Testdatensatz (inkl. Testwinkel α_*) $X_* \mapsto \alpha_*$

Result : Sinoide $\bar{f}_{cos*}, \bar{f}_{sin*}$, Radius \bar{r}_* , Winkel $\bar{\alpha}$, Sinoide Varianz $\mathbb{V}[\bar{f}_*]$, Sinoide Std.-Abweichung s_* , Konfidenzintervalle $CIA_{95\%}, CIR_{95\%}$, std. log. Verluste (*SLLA*), *SLLR*

1. Berechnung Kovarianzvektor $\mathbf{k}_* \leftarrow$ Gleichung C.15;
 2. Berechnung Varianzvorhersage $\mathbb{V}[\bar{f}_*] \leftarrow$ Gleichung C.18;
 3. Berechnung Mittelwertvorhersage $\bar{f}_{cos*}, \bar{f}_{sin*} \leftarrow$ Gleichung C.16;
 4. Berechnung Radius \bar{r}_* , Winkel $\bar{\alpha} \leftarrow$ Gleichung C.17;
 4. Berechnung Std.-Abweichung $s_* \leftarrow$ Gleichung C.19;
 6. Berechnung Konfidenzintervalle $CIA_{95\%}, CIR_{95\%} \leftarrow$ Gleichung C.20;
 7. Berechnung std. log. Verluste (*SLLA*), *SLLR* \leftarrow Gleichung C.21;
-

Kovarianzvektor als Regressionsmaß für einen einzigen Testdatensatz mit zugehörigen Simulationswinkel $X_* \mapsto \alpha_*$. Ist der Vektor \mathbf{k}_* nach Algorithmus 3 zu bilden. Er stellt den Vergleichsbezug von Testdaten X_* zu allen Trainingsdaten X her und löst die neue Winkelstellung in Relation zu den Trainingsdaten auf [3]. Der Kovarianzvektor \mathbf{k}_* ist mit aktuellen Modellparametern zu berechnen Gleichung C.15.

$$\mathbf{k}_* = K(X, X_* | \theta) \quad \text{mit Algorithmus 3} \quad (\text{C.15})$$

f. Traingsdaten X und einen Testwinkel $X_* \mapsto \alpha_*$

Mittelwertvorhersage als Mittelwertergebnis für eine Standardnormalverteilung, sind über die Summe aus Mittelwertschätzung Gleichung C.11 und Produkt aus Kovarianzvektor Gleichung C.15 mit Regressionsgewichte Gleichung C.12 zu bilden [3]. Jeweils für beide Funktionen Cosinus und Sinus. Die Systematische Kopplung erfolgt über den gemeinsamen Kovarianzvektor.

$$\begin{aligned}\bar{f}_{cos*} &= m_{cos}(X_{cos*}) + \mathbf{k}_*^T \cdot \alpha_{cos} \\ \bar{f}_{sin*} &= m_{sin}(X_{sin*}) + \mathbf{k}_*^T \cdot \alpha_{sin}\end{aligned}\tag{C.16}$$

Regressierter Winkel und Radius aus der Cosinus- und Sinus-Vorhersage, ergeben sich aus der Anwendungsbeschreibung im Abschnitt 2.1 durch Gleichung C.17.

$$\begin{aligned}\bar{r}_* &= \sqrt{\bar{f}_{cos*}^2 + \bar{f}_{sin*}^2} \quad \text{wie Gleichung 2.2} \\ \bar{\alpha}_* &= \text{atan2}(\bar{f}_{sin*}, \bar{f}_{cos*}) \quad \text{wie Gleichung 2.3}\end{aligned}\tag{C.17}$$

Varianzvorhersage als Korrelation eines Testdatensatz X_* mit sich selbst. Dafür ist der Datensatz X_* Algorithmus 3 zuzuführen und zur Varianz der Vorhersage $\mathbb{V}[\bar{f}_*]$ im Vergleich zur Kovarianzmatrix und Kovarianzvektor mit Gleichung C.18 aufzulösen. Die Varianz $\mathbb{V}[\bar{f}_*]$ gilt jeweils für beide Sinoide Regressionsprozesse [3]. Dieser Fall deckt sich mit dem Auslöschungskriterium für gültige Kovarianzfunktion und kann in diesem Fall durch einen numerischen Fehler $\Delta\epsilon$ leicht vom theoretischen Rechenergebnis abweichen.

$$\begin{aligned}\mathbb{V}[\bar{f}_*] &= K(X_*, X_* | \theta) - \mathbf{k}_*^T K_y^{-1} \mathbf{k}_* \\ &= K(X_*, X_* | \theta) - v^T v \\ &= (\sigma_f^2 + \Delta\epsilon) - v^T v\end{aligned}\tag{C.18}$$

mit $v = L \setminus \mathbf{k}_*$ und $\Delta\epsilon$ numerischer Fehler

Standardabweichung als zugehörige Abweichung der Mittelwertvorhersage für eine Standardnormalverteilung, ergibt sich aus der Varianzvorhersage und dem verwendeten Rauschniveau nach Gleichung C.19 [3]. Die Standardabweichung gilt, jeweils als Abweichung für beide Sinoide als eigenständige und statistische Prozesse. Fehler der einzelnen Prozesse, müssen sich in einer kombinierten Auswertung, durch Addition ihrer Einzelvarianzen s_*^2 , für eine gemeinsame Abschätzung fortpflanzen.

$$s_* = \sqrt{\mathbb{V}[\bar{f}_*] + \sigma_n^2} \quad (\text{C.19})$$

Qualitätskriterien können über die ermittelte Standardabweichung s_* der Einzelregressionen gebildet werden. Dafür muss diese gemäß der Fehlerfortpflanzung für die einzelnen statistischen Prozesse mit dem Faktor $\sqrt{2}$ versehen werden, da sich die Varianz für Cosinus und Sinus zusammensetzt und verdoppelt mit $s_*\sqrt{2} = \sqrt{2(\mathbb{V}[\bar{f}_*] + \sigma_n^2)}$. Konfidenzintervalle für ermittelten Winkel $\bar{\alpha}_*$ und Radius \bar{r}_* , können daher direkt mit $95\% \leftarrow z_{CDF}$ -Faktor für normalverteilte Wahrscheinlichkeiten und kumulativer Dichtefunktion berechnet werden. Der Radikant für die Stichprobenanzahl entfällt, da immer nur ein Testwert prozessiert wird. Es ergeben sich das Konfidenzintervall für Winkel $CIA_{95\%}$ und für Radius $CIR_{95\%}$ nach der Gleichung C.20. Für das Winkelintervall ist die statistische Aussage mit $\arcsin(z_{CDF} \cdot s_*\sqrt{2})$ ins Winkelmaß überführt.

$$\begin{aligned} CIA_{95\%} &= \bar{\alpha}_* \pm \arcsin(z_{CDF} \cdot s_*\sqrt{2}) \quad \text{f. } z_{CDF} = 1,96 \leftarrow 95\% \\ CIR_{95\%} &= \bar{r}_* \pm z_{CDF} \cdot s_*\sqrt{2} \end{aligned} \quad (\text{C.20})$$

Zwei weitere Qualitätskriterien, zur Interpretation der Modellgeneralisierung, können über standardisierte logarithmische Verluste (engl. std. log. loss) berechnet werden [3]. Die Berechnung erfolgt, als Vergleich zwischen Soll- und errechneten Istwerten, unter Berücksichtigung der Fehlerfortpflanzung und Winkelmaß nach Gleichung C.21. Es ergibt sich der Verlust $SLLA$ für Winkel und $SLLR$ für Radius.

Verluste $SLLA$ stehen nur zur Verfügung, wenn Simulations- oder Encoder-Winkel in der Vorhersage mit einbezogen sind. Verluste $SLLR$ für Radius stehen immer zur Verfügung, da mit Einheitskreis als Regressionsziel, der Sollradius gleich eins ist. Ein schlecht generalisiertes Modell liefert positive Verlustwerte > 0 . Eine mäßige Generalisierung liefert Verluste ≈ 0 . Eine gute bis sehr gute Generalisierung liefert strikt Verlustwerte < 0 [3].

$$SLLA = 0,5 \cdot \left(\log(2\pi \arcsin^2(s_*\sqrt{2})) + \frac{(\alpha_* - \bar{\alpha}_*)^2}{\arcsin^2(s_*\sqrt{2})} \right) \quad (\text{C.21})$$

$$SLLR = 0,5 \cdot \left(\log(2\pi(s_*\sqrt{2})^2) + \frac{(1 - \bar{r}_*)^2}{(s_*\sqrt{2})^2} \right)$$

C.4 Modellgeneralisierung

Algorithmus 7 zur Modellgeneralisierung vereinigt die Algorithmen 2 und 5 in sich und nutzt diese in Verbindung mit einem Bayes-Optimierungsverfahren zur Ermittlung des passenden Rauschniveau σ_n^2 . Das Bayes-Optimierungsverfahren ist in Matlab über die BayesOpt-Funktion implementiert und wird mit dem Probier-Algorithmus Improve-Per-2^{nd+} betrieben. Entscheidend bei der Ausführung ist die Durchlaufzahl der Bayes-Optimierung, da sich das Verfahren durch Ausprobieren von σ_n^2 und Vergleich des resultierenden Min-Kriterium Gleichung C.22, Schritt für Schritt der optimalen Lösung annähert.

$$\sigma_n^2 | X_* = \arg \min_{\sigma_n^2} MSLL(\sigma_n^2 | X_*) \quad (\text{C.22})$$

$$MSLL = \begin{cases} \frac{SLLA(X_*)}{N_*} & \text{f. Verluste üb. Winkel} \\ \frac{SLLR(X_*)}{N_*} & \text{f. Verluste üb. Radius} \end{cases}$$

f. alle N_* Testdaten X_* und

Testwinkel $X_* \mapsto \alpha_*$ nach Gleichung C.21

Je nach dem wie die Grenzen der einzelnen Modellparameter gewählt sind, kann das unterschiedlich schnell passieren. Wird der Algorithmus zu früh abgebrochen, ist die optimale Lösung wahrscheinlich nicht gefunden. Daher empfiehlt sich für Anfangsuntersuchungen mit weiten Parameter-Bounds eine Durchlaufzahl ≥ 50 zu wählen.

Algorithmus 7 : Modellgeneralisierung über BayesOpt-Funktion f. alle $X_* \mapsto \alpha_*$

Input : Kofigurationsdatensatz, Trainingsdatensatz X, Testdatensatz X*

Result : Generalisiertes Modell mit optimierten Rauschniveau σ_n^2

1. Initialisierung Modell \leftarrow Algorithmus 2;
 2. Initialisierung Rauschniveau-Bounds \leftarrow Tabelle C.1;
 3. Initialisierung Min-Kriterium $MSLL \leftarrow$ Gleichung C.22;
 4. Initialisierung BayesOpt-Funktion mit Durchlaufzahl \leftarrow Tabelle C.1;
 5. **while** Durlaufzahl nicht erreicht **do**
 - Zuweisung innerhalb Rauschniveau-Bounds $\sigma_n^2 \leftarrow$ BayesOpt-Funktion;
 - Modelloptimierung von 1. mit neuen $\sigma_n^2 \leftarrow$ Algorithmus 5;
 - Berechnung f. alle Testwinkel $MSLL \leftarrow$ Gleichung C.22;
 - Speichern und indizieren von σ_n^2 f. jedes Ergebnis $MSLL$;
 - end**
 6. Entnahme von σ_n^2 bei $\min MSLL$;
 7. Speichern von σ_n^2 in 1.;
 8. Finale Modelloptimierung von 1. \leftarrow Algorithmus 5;
 9. Berechnung und Mittelung f. alle Testwinkel $SLLA, SLLR \leftarrow$ Gleichung C.21;
-

Min-Kriterium als Mittelwertbildung aller standardisierten logarithmischen Winkelverluste aus Gleichung C.21. Es sind für jeden verfügbaren Testdatensatz und zugehörigen Simulationswinkel $X_* \mapsto \alpha_*$ die Verluste nach Gleichung C.22 auszurechnen und zu $MSLL$ zu mitteln [3]. Im Anschluss ist gebildeter Mittelwert einem Minimierungsverfahren zur Ermittlung des passenden Rauschniveaus σ_n^2 zuzuführen. Das Minimierungsproblem induziert dabei für jedes ausprobierte σ_n^2 ein Regressionsmodell. Die berechneten Modelle sind über ihre mittleren Verlust $MSLL$ miteinander zu vergleichen [3]. Das Modell für $\min MSLL$, besitzt die stärkste Generalisierung und somit das optimiert Rauschniveau σ_n^2 und sich nach Algorithmus 5 ergebenen optimierten Kernel-Parameter $\theta|\sigma_n^2$. Auch hier gilt wie für Gleichung C.21, dass sich eine gute Generalisierung für $MSLL < 0$ einstellt. Empirisch beobachtet Werte, liegen dabei im Intervall von $-2 < MSLL < -5$.

D Genutzte Software 0.0.3 08.01.2021

Für die Nachvollziehbarkeit der getätigten Entwicklungsarbeiten und die Erstellung der Bachelor-Thesis, ist das dafür jeweilige Betriebssystem (OS) und die verwendete Software (SW) tabellarisch aufgeführt. Es finden sich genutzte Versionen der SW und Angaben zur Minimalanforderung für deren Nutzung. Die Anforderungen sind für Prozessorkern (CPU), Arbeitsspeicher (RAM), Festplattenlaufwerk (HDD) näher aufgeschlüsselt. Die Programmierarbeiten mit MATLAB sind jeweils mit Windows und Linux geschrieben bzw. getestet worden.

Software	Verwendungszweck (Typ)	Min.-Anforderung	Version	Erscheinungstag
Ubunut Budgie	Linux-Betriebssystem (Laptop OS)	2 GHz Dual-Core-CPU 4 GB RAM 25 GB freier HDD-Speicher	18.04 LTS	26.04.2018
Windows 10 Enterprise	Windows-Betriebssystem (Laptop OS)	1 GHz Core-CPU 1 GB RAM 32 GB freier HDD-Speicher	1909	12.11.2020
MATLAB	Simulationssoftware (Multi-Paradigmen Programmier- Sprache, IDE)	Intel/ AMD x86-64 CPU 4 GB RAM 3.5 GB freier HDD-Speicher	2020b	17.09.2020
Git	Versionierung (Kommandozeilenprogramm)	-	2.29	29.10.2020
Inkscape	Vektorgrafikzeichenprogramm (Grafikaufbereitung)	1 GHz CPU 256 MB RAM 302 MB freier HDD-Speicher	0.92.3	11.03.2018
Texstudio	Textbearbeitung f. LaTeX Dokumente (Editor)	- - 24.7 MB freier HDD Speicher	2.12.6	25.07.2020
wkhtmltopdf	HTML- zu Pdf-Konvertierung	- - -	0.12.6	11.06.2020
JabRef	Literaturverwaltungsprogramm f.BibLaTeX (Editor)	- - -	5.1	30.08.2020

Tabelle D.1: Genutzte Software zur Erstellung der Thesis und Dokumentation der Ergebnisse, Entwicklungsumgebung für die geschriebene Simulationssoftware zur Generierung und Auswertung der Sensor-Array-Simulation.

E Software-Dokumentation 0.0.5

14.04.2021

Die Software-Dokumentation ist automatisiert mit MATLAB-Skripten erstellt worden. Es ist dafür ein zweistufiger Prozess implementiert, der im ersten Schritt eine in MATLAB integrierte HTML-Dokumentation erstellt und im Anschluss diese zu eigenständigen PDF-Dateien exportiert. Als letzter Schritt sind diese zu einem LaTeX-Manual zusammengefasst im Anhang eingebunden. Mit diesem Verfahren ist es möglich, eine Dokumentation direkt aus geschriebenen M-Dateien zu generieren. Allerdings ist es dafür nötig, eine spezielle Formatierung und einen gewissen Programmierstil einzuhalten [6]. Die Dokumentation enthält neben dem erstellten Quellcode eine Reihe von Arbeitsanweisungen, wie mit der Software umzugehen ist. Zusätzlich sind Beschreibungen für die Erstellung und Pflege des Software-Projektes mit beigelegt. Die geschriebene Software ist mithilfe des Software-Versionierungsprogramms Git erstellt worden, was eine genaue Nachvollziehbarkeit in Bezug auf die einzelnen Arbeitsschritte ermöglicht. Zur Versionierung ist der Git-Feature-Branch-Workflow [16] angewandt worden. Aus stilistischen Gründen ist die gesamte Software-Dokumentation in Englisch verfasst.

GaussianProcessDipoleSimulation

The project of sensor array simulations and Gaussian Processes for angle predictions on simulation datasets started in

May 06. 2019

with IEEE paper by Thorben Schüthe which is a base investigation of "Two-Dimensional Characterization and Simplified Simulation Procedure for Tunnel Magnetoresistive Angle Sensors". This produces characterization datasets of different current available angular sensors on the market.

June 11. 2019

Thorben Schüthe came up with a high experimental scripting for abstracting sensor characterization fields to an array of sensor fields which was stimulated by magnetic dipole field equations to approximate a spherical magnet.

November 06. 2019

Prof. Dr. Klaus Jünemann supports the team around Prof. Dr.-Ing. Karl-Ragmar Riemschneider and Thorben Schüthe with an application of Gaussian Process learning to investigate on angle predictions for sensor array simulation results. The attempt of the solution was working for a tight set of parameters and was highly experimental with rare documentation and few sets of functions and scripts. The math of this very solution based on the standard book for Gaussian Process by Williams and Rasmussen. The algorithm is related to the guideline for linear regression model which worked fine for a setup of standard use cases but needed further investigation for a wider set of parameters and functions to identify general and relevant parameter settings to provide an applicable angular prediction.

September 21. 2020

Tobias Wulf establishes a Matlab project structure and programming guidance and flows to document the source code integrated in the Matlab project architecture. That includes templating for scripts and functions and general descriptions of project structure and guidance for testing and documenting project results or new source code including automation for publishing html in Matlab integrated fashion.

October 22. 2020

Tobias Wulf adds TDK TAS2141 TMR characterization to the project. Thorben Schüthe provided a raw dataset which was manually modified by Tobias Wulf to a dataset which is plotable and reconstructable in stimulus and characterization field investigations.

October 31. 2020

Tobias Wulf establishes a general configuration flow to control part of software via config file which is partly loaded as needed into workspace.

November 29. 2020

Tobias Wulf finished the implementation of sensor array simulation which uses TDK TAS2141 as base of simulation. The software includes now simulation for stimulus magnet (dipole sphere) and automated way fast generate training and test datasets by set configuration. Various plots and animation for datasets and a best practice workflow for simulation. Also included are unit test and Matlab integrated documentation in html files. A full description of generated datasets is included too.

December 05. 2020

Tobias Wulf integrated a second characterization dataset for NXP KMZ60 into the sensor array simulation software. The dataset was manually modified in the same way as the TDK TAS2141 dataset. The KMZ60 raw data was provided from Thorben Schüthe. The simulation software was adjusted to run with both datasets now. Additional plots for transfer curves are included for both and same plots for characterization view of KMZ60 as for TAS2141 too.

April 01. 2021

Tobias Wulf integrated GPR algorithms made by Klaus Jünemann as gaussianProcessRegression modul. Additionally a second kernel was implemented based on the first one by Jünemann. The implementation was transferred from a functional and script based draft version of GPR mechanism into fully initialized model based version which loads needed functionality and parameters into a struct. So prediction and optimization algorithms are working on a structured model frame. Missing model optimization is added to fit model on training data and generalize it to test data. Interface to Sensor Array simulations are done by work on datasets.

Created on September 21. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Workflows

Developing software needs conventions to produce common results and good working software. There are certain points which matches good written software:

1. The reuse factor of the written source code.
2. Good source code structure or hierarchy to expand.
3. Testing with automated frameworks e.g. Unit test.
4. Source code versioning.
5. Source code readability and detailed commenting and documentation

The last point can be split into two points but Matlab provides a publish process where in source code comments can be used for documentation. What is probably not detailed enough and needs further documents in completion. Ongoing on that to provide support in guidance for current or upcoming project work it is recommended to declare common workflows for those points.

Coding conventions are used from MATLAB Style Guidelines 2.0 by Richard Johnson.

See Also

- [MATLAB Style Guidelines 2.0](#)

Project Preparation

How to setup a Matlab project with Git support and simple backup plan.

Project Structure

Directory structure, associated tasks and how to add new elements.

Git Feature Branch Workflow

How to work in the project with Git support in feature driven way.

Documentation Workflow

How to document the project work in progress and introduce new project elements to publishing process.

Simulation Workflow

Best practice simulation workflow for sensor array simulations to generate training and test datasets.

Created on September 21, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Project Preparation

The first steps to setup a scalable software project are none trivial and need a good strcuture for later project expands. Either to setup further new projects a well known scalable project structure helps to combine different software parts to bigger environment packages. Therefore a project preparation flow needs to be documented. It unifies the outcome of software projects and partly guarantees certain quality aspects.

The following steps can be used as guidance to establish a propper Matlab project structure in general. Each step is documented with screenshots to give a comprehensible explanation.

See Also

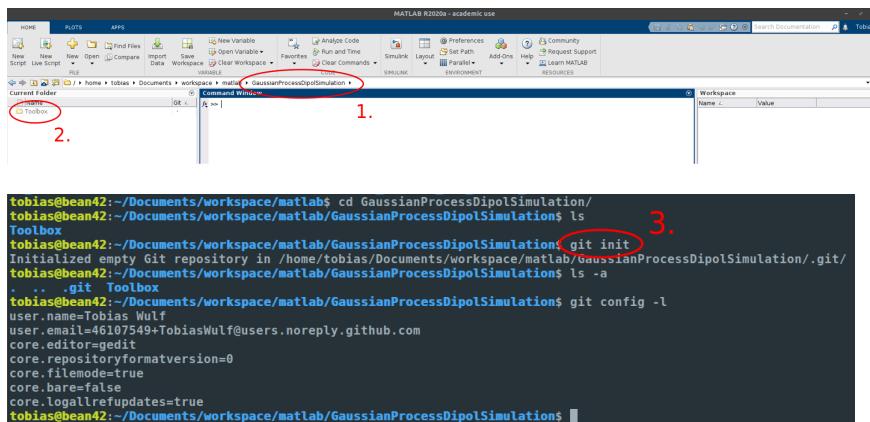
- [Create a New Project From a Folder](#)
- [Add a Project to Source Control](#)
- [Setup Git Source Control](#)
- [Use Source Control with Projects](#)
- [Git Attributes](#)
- [Git Ignores](#)
- [Add Files to the Project](#)
- [Commit Modified Files to Source Control](#)
- [Clone Git Repository](#)

Create Main Project Directory

The main project directory contains only two subfolders. The first one is the Toolbox folder where the project, m-files and other project files like documentation are placed. The folder is also called sandbox folder in Matlab project creation flows which is just another description for a project folder where the coding takes place. The second folder is a hidden Git repository folder which keeps the versionation in final. It is respectively seen a remote repository that establish basics to setup backup plans via Git clone or can be laterly replaced by remote repository on a server or a GitHub repository to work in common on the project.

First step:

1. Create an empty project folder, open Matlab navigate to folder path.
2. Right click in the Current Folder pane and create New> Folder "Toolbox".
3. Open a Git terminal and in the project directory and initialize an empty Git repository.



Create Matlab Project with Git Support

In second it is needed to create the Matlab project files in a certain way to get full Git support and support for the Matlab help browser

environment. In this use case the before created local Git repository is used as remote origin. So several settings are automatically made during the creation process by Matlab and as mentioned before the "local remote" repository can be replaced later by a remote origin located on a server or GitHub. The Toolbox folder must be empty to process the following steps.

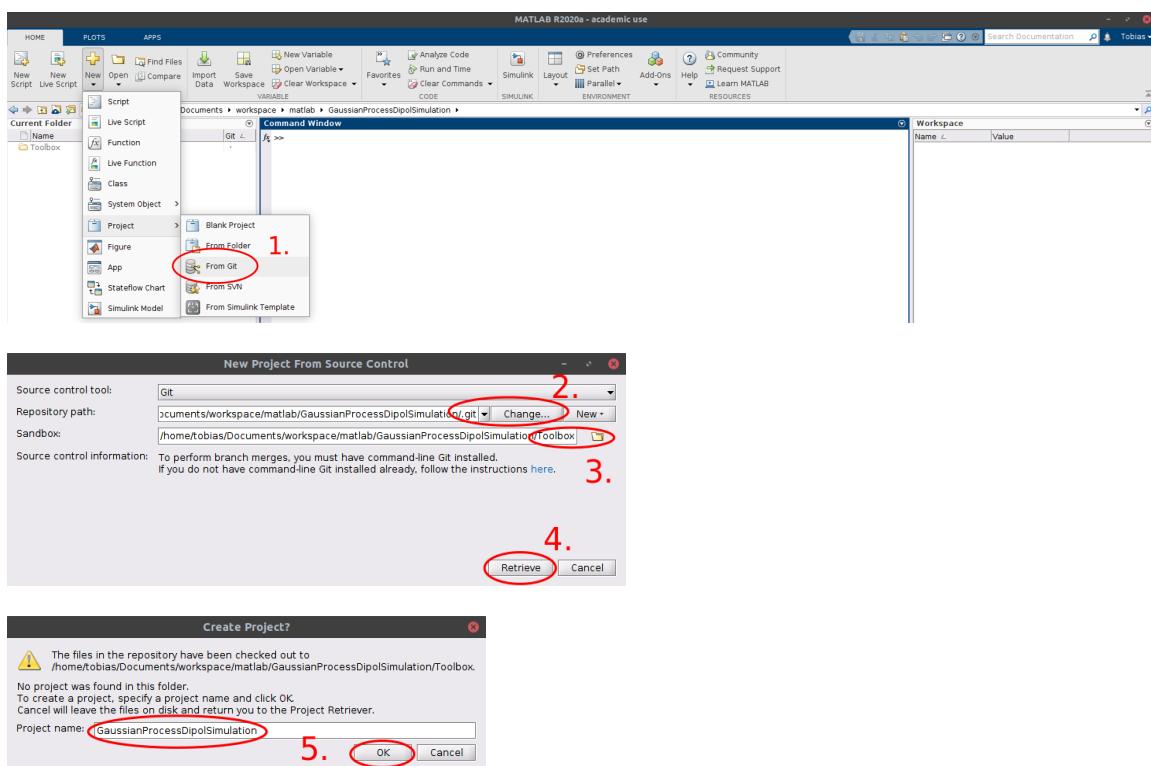
It is recommend to do no further Git actions on the created Git repository via Git terminal!

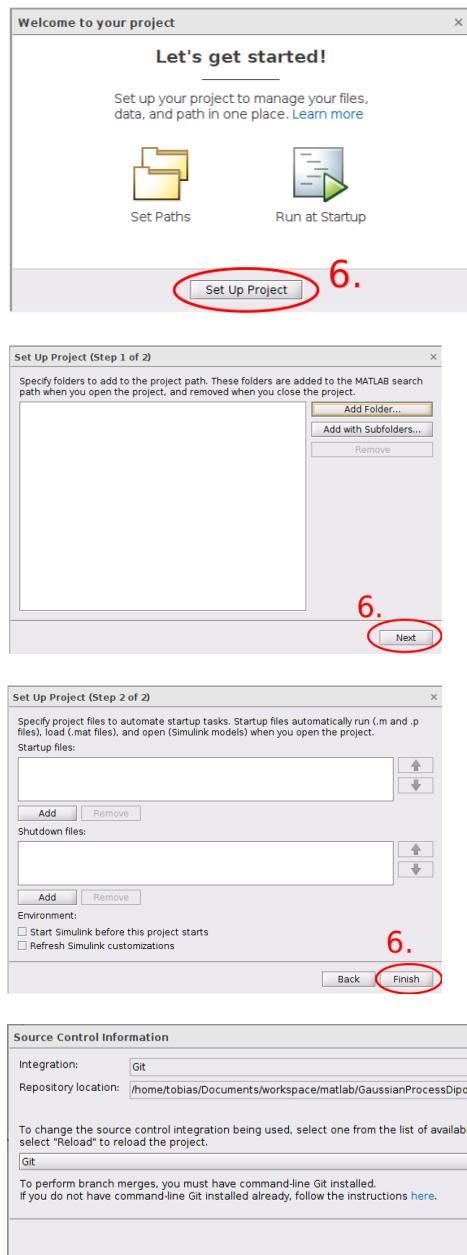
These steps only proceed the project setup, further Matlab framework functionality is added later.

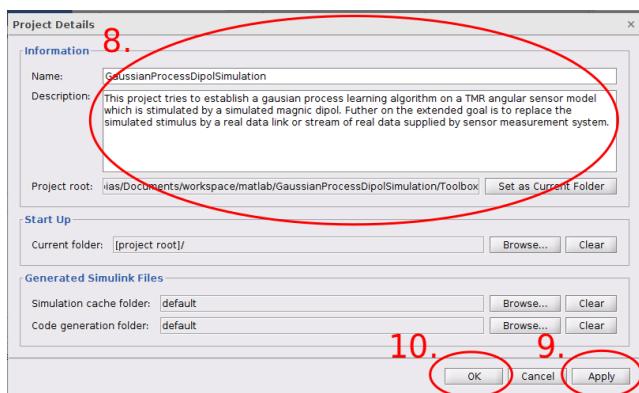
Second step:

1. In the created main project directory create a New > Project > From Git.
2. Change the repository path to the hidden Git repository path in the main project directory.
3. Change the sandbox path to the Toolbox path in the main project directory.
4. Click Retrieve.
5. Enter the project name given by the main project directory name and click OK.
6. Click on Set Up Project and skip the two following steps via Next and Finish.
7. Switch to Toolbox directory by double click on the folder in the Current Folder pane, open the created Matlab project file with a double click and check source control information under PROJECT tab by clicking Git Details.
8. Add a short project summary by click on Details under the ENVIRONMENT section of the PROJECT tab.
9. Click Apply.
10. Click OK.

The project itself is under source control now.







Register Binaries to Git and Prepare Git Ignore Cases

The root of Git is to work as text file versioner. Source code files are just text files. So Git versionates, tags and merges them in various ways in a work flow process. That means Git edits files. This point can be critical if Git does edit a binary file and corrupts it, so that is not executable any more. Therefore binary files must be registered to Git. Another good reason is to register binary or other none text files because Git performs no automatic merges on file if they are not known text files. To keep the versionating Git makes a taged copy of that file every time the file changed. That can be a very junk of memory and lets repository expands to wide.

To prevent Git from mishandling binaries it is able to register them in a certain file and mark the file types how to handle them in progress. The file is called `.gitattributes` must be placed in the Git working directory which is the sandbox folder for Matlab projects. The `.gitattributes` file itself is hidden.

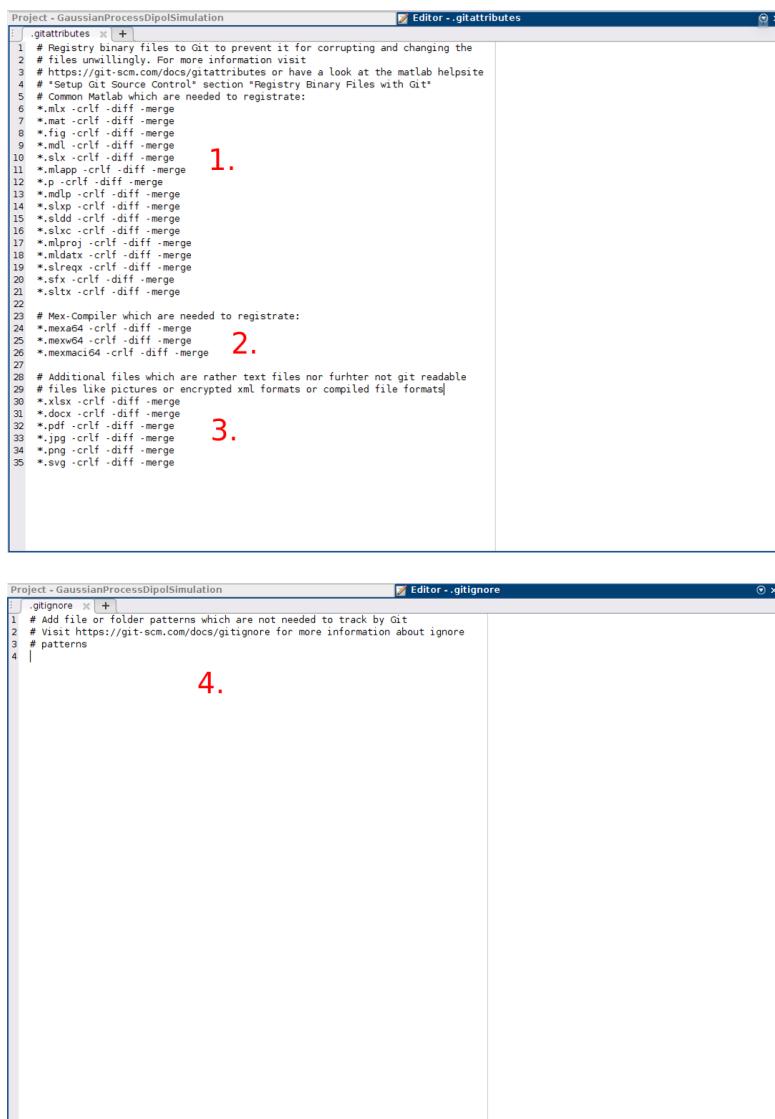
Three options are needed to mark a file type as binary. The `-crlf` option disables end of line conversion and the `-diff` option in combination with the `-merge` option to mark the file as binary.

In addition to that it is possible to declare several ignore cases to Git. So certain directories or file types are not touched or are left out from source control. This is done in `.gitignore` file. The must be placed in the sandbox folder too.

From the sandbox directory enter in the Matlab command prompt `edit .gitattributes` and `edit .gitignore` and save both files. The files are not shown in Current Folder pane (hidden files). Edit both files in the Matlab editor and save the files.

Third step:

1. Add common Matlab file types to `.gitattributes`.
2. Add Matlab compiler file types to `.gitattributes`.
3. Add other file types which can appear during the work to `.gitattributes`.
4. Add ignore cases to `.gitignore` if needed.



```

Project - GaussianProcessDipolSimulation
Editor - .gitattributes

1 # Registry binary files to Git to prevent it for corrupting and changing the
2 # files unwillingly. For more information visit
3 # https://git-scm.com/docs/gitattributes or have a look at the matlab helpsite
4 # in the "Source Control" section "Registry Binary Files with Git"
5 # Common Matlab files are needed to register:
6 *.mlx -crlf -diff -merge
7 *.mat -crlf -diff -merge
8 *.fig -crlf -diff -merge
9 *.mdl -crlf -diff -merge
10 *.slx -crlf -diff -merge
11 *.mapp -crlf -diff -merge
12 *.mldat -crlf -diff -merge
13 *.mdlp -crlf -diff -merge
14 *.slxp -crlf -diff -merge
15 *.sldd -crlf -diff -merge
16 *.sldx -crlf -diff -merge
17 *.mlproj -crlf -diff -merge
18 *.mldata -crlf -diff -merge
19 *.sldemo -crlf -diff -merge
20 *.sldfr -crlf -diff -merge
21 *.sldt -crlf -diff -merge
22

23 # Mex-Compiler which are needed to register:
24 *.mexa64 -crlf -diff -merge
25 *.mexw64 -crlf -diff -merge
26 *.mexmaci64 -crlf -diff -merge
27

28 # Additional files which are rather text files nor further not git readable
29 # files like pictures or encrypted xml formats or compiled file formats
30 *.xlsx -crlf -diff -merge
31 *.docx -crlf -diff -merge
32 *.pdf -crlf -diff -merge
33 *.jpg -crlf -diff -merge
34 *.png -crlf -diff -merge
35 *.svg -crlf -diff -merge

```



```

Project - GaussianProcessDipolSimulation
Editor - .gitignore

1 # Add file or folder patterns which are not needed to track by Git
2 # Visit https://git-scm.com/docs/gitignore for more information about ignore
3 # patterns
4

```

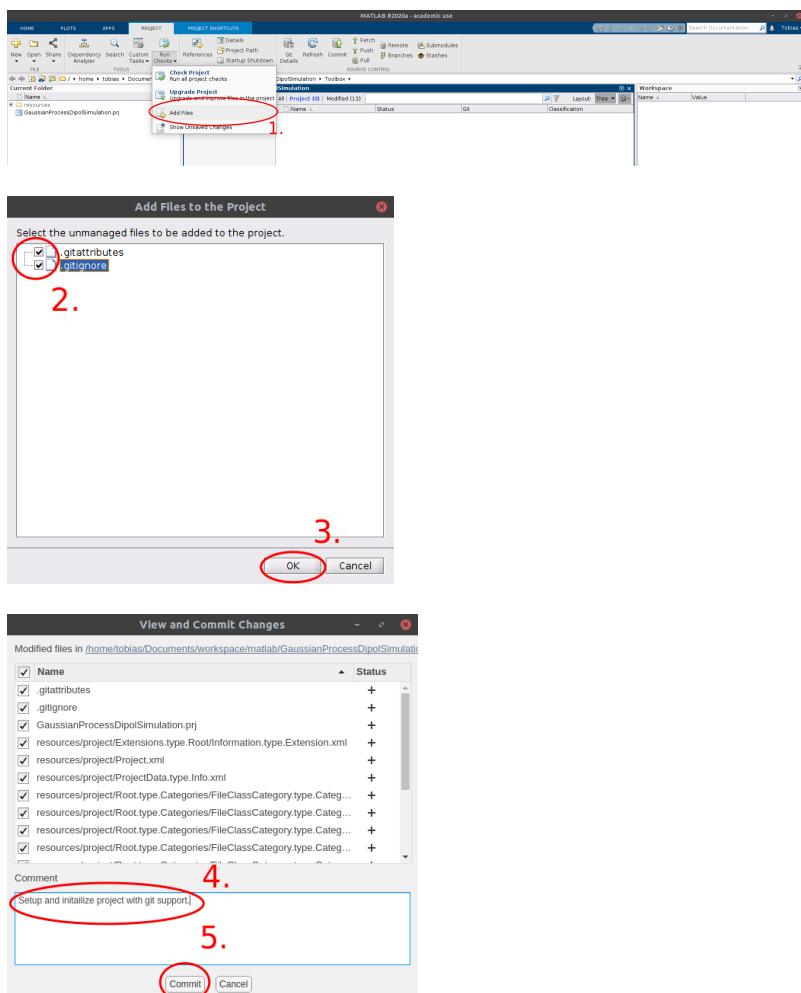
Checkout Project State and Do an Initial Commit

The main part is done. It just needs a few further step to save the work and add the created files to the project.

Fourth step:

1. Add created files to the project. In the PROJECT tab under TOOLS section click Run Checks > Add Files.
2. Check the files to add to the project.
3. Click OK.
4. Right click in the white space of Current Folder pane and click Source Control> View and Commit Changes... and add comment to the commit.
5. Click Commit.

The project is now initialized.



Push to Remote and Backup

The project is ready to work with. Finally it needs a backup mechanism to save the done work after closing the Matlab session. Git and how the project is built up to provide an easy way to make backups.

1. Push the committed changes to remote repository.
2. Insert a backup medium e.g. USB stick and open a git terminal there.
3. Clone the project remote repository from project directory.
4. Change the directory to cloned project.
5. Check if everything was cloned.
6. Check if the remote url fits to origin.
7. Pull from remote to check if everything is up to date.



```

1:tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation
tobias@bean42:/media/tobias/DEP17364$ git clone ~/Documents/workspace/matlab/GaussianProcessDipolSimulation/
Cloning into 'GaussianProcessDipolSimulation'...
done.
Checking out files: 100% (120/120), done. 4.
tobias@bean42:/media/tobias/DEP17364$ cd GaussianProcessDipolSimulation/ 5.
tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ ls
docs GaussianProcessDipolSimulation.prj info.xml resources scripts
tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ git config -l
user.name=Tobias Wulf
user.email=46107549+TobiasWulf@users.noreply.github.com
core.editor=gedit
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
remote.origin.url=/home/tobias/Documents/workspace/matlab/GaussianProcessDipolSimulation/
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ git pull origin master
From /home/tobias/Documents/workspace/matlab/GaussianProcessDipolSimulation
 * branch      master       -> FETCH_HEAD
Already up to date.
tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ 6.

```

If further changes are committed to the project push again to the remote from Matlab environment and update the backup from time to time by inserting your medium and make a fresh pull. Change the directory to the folder and just pull again. See below as an example how does it look like.

```

tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ git pull origin master
remote: Counting objects: 37, done.
remote: Compressing objects: 100% (24/24), done.
remote: Total 37 (delta 23), reused 25 (delta 11)
Unpacking objects: 100% (37/37), done.
From /home/tobias/Documents/workspace/matlab/GaussianProcessDipolSimulation
 * branch      master       -> FETCH_HEAD
   B2c80a7..4d177e8  master    -- origin/master
Updating 02c80a7..4d177e8
Fast-forward
  docs/Project_Preparation.m | 65 ++++++=====
  docs/Project_Preparation_Scripts.html | 2 ++
  docs/html/Introduction.html | 2 ++
  docs/html/Project_Preparation.html | 93 ++++++=====
  docs/html/Work_Flows.html | 2 ++
  docs/html/Helpsearch_v3/0.cfe | Bin 260 -> 260 bytes
  docs/html/Helpsearch_v3/0.rtf | Bin 26025 -> 27914 bytes
  docs/html/Helpsearch_v3/0.si | Bin 267 -> 267 bytes
  docs/html/Images/Project_Preparation/13_check_add_files.png | Bin 99341 -> 121756 bytes
  docs/html/Images/Project_Preparation/14_add_git_files.png | Bin 10541 -> 15143 bytes
  docs/html/Images/Project_Preparation/15_commit_initialized_project.png | Bin 62587 -> 62932 bytes
  docs/html/Images/Project_Preparation/16_publish_to_web.html | 10 ++++++=====
  scripts/publishProjectfilesToHTML.m | 6 -----
resources/project/Root.type.Files/docs.type.File/html.type.File/Flows.html.type.File.xml | 10 ++++++=====
14 files changed, 146 insertions(+), 48 deletions(-)
create mode 100644 resources/project/Root.type.Files/docs.type.File/html.type.File/Flows.html.type.File.xml
tobias@bean42:/media/tobias/DEP17364/GaussianProcessDipolSimulation$ 8.

```

Port Remote Repository to GitHub

The remote repository is ported to GitHub laterly. Therfore some minimal changes are made manually to the local repository.

1. According to new rules on GitHub the master branch is renamed to main.
2. Due to that a new upstream is set to origin/main from origin/master
3. To fetch all casualties a merge was needed from origin/main on local main. The origin/master reference was included.
4. Change remote repository to GitHub URL <https://github.com/TobiasWulf/GaussianProcessDipolSimulation.git>
5. At the moment the GitHub repository is private and not visible in the web. After finishing the general work the repository will be set to publish in consultation with HAW TMR research project and team.
6. After publish on GitHub, clone or fork to work with.
7. The source code is hosted under MIT license.
8. Use GitHub flows to clone or fork and push changes to backup done work.
9. Toolbox folder is not needed anymore because remote is elsewhere now
10. Re clone from remote to get new structurew without Toolbox folder

Created on September 30. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Project Structure

A good project directory structure is the key to build scalable and expandable software projects. Therefore each project folder has to fulfill an associated task. Additionally, a good structure facilitates project navigation and the retrieval and reuse of project content. Further on Matlab provides strategies to add content to existing project structures and label it for script based execution of project task to manage project files. To add new content have a look at the links below.

See Also

- [Specify Project Path](#)
- [Add Files to the Project](#)
- [Add Labels to Files](#)

Directory Overview

```
GaussianProcessDipoleSimulation
├── [4.0K]  data
│   ├── [4.0K]  test
│   └── [4.0K]  training
├── [4.0K]  docs
│   ├── [ 12K]  html
│   │   ├── [4.0K]  helpsearch-v3
│   │   └── [4.0K]  images
│   └── [4.0K]  latex
│       ├── [4.0K]  BA_Thesis_Tobias_Wulf
│       └── [4.0K]  Manual
├── [4.0K]  resources
├── [4.0K]  scripts
├── [4.0K]  src
│   ├── [4.0K]  gaussianProcessRegression
│   │   ├── [4.0K]  basicMathFunctions
│   │   ├── [4.0K]  kernelQFC
│   │   └── [4.0K]  kernelQFCAPX
│   ├── [4.0K]  sensorArraySimulation
│   └── [4.0K]  util
│       └── [4.0K]  plotFunctions
└── [4.0K]  temp
    ├── [4.0K]  Functions
    └── [4.0K]  GaussProc
└── [4.0K]  tests
```

24 directories

Generated with linux shell command from on directory above the main project directory.

```
tree -dhn GaussianProcessDipoleSimulation ...
-o GaussianProcessDipoleSimulation/docs/html/Directory_Tree.txt -I ...
"project|Project_*|thesis|literature"
```

Directory Tasks

Directory	Task
-----------	------

Directory	Task
./	Main project directory which contains the Matlab project sandbox files and the hidden repository files. Matlab project sandbox directory. Project root directory which contains the Matlab project file, the info.xml, .gitignore, .gitattributes files and all other project related subdirectories. Startup directory.
.git	Hidden repository for local standalone work. Saves daily working results. Provide a Git clonable instance of sandbox the directory. Replacable. Not Matlab driven, simulates remote repository.
./resources	Autogenerated directory from Matlab project. Contains the local project versioning and project xml-files.
./data	Contains all project related datasets e.g. mat-files.
./data/training	Contains mat-files from sensor array simulation for training cases of the gaussian process.
./data/test	Contains mat-files from sensor array simulation for test cases of the gaussian process.
./docs	Documentation directory which contains m-files only for documentation use and the directory where all project remarked files are published into HTML output files.
./docs/html	Publish directory where published m-files are collected and bind to a Matlab help browser readable documentation. It contains html-files and subdirectory for images and figures which are used in the documentaion. The help browser search database is placed here too. Much more important the directory contains the helptoc.xml which pointed by the info.xml from root project directory.
./docs/html/helpsearch-v3	Contains autogenerated help search database entries. The directory is rewritten during the publish documentation process.
./docs/html/images	Contains all needed image files like png-files which are used in the documentation.
./docs/latex	Documentation directory which LaTeX documentation of the project including subfolders for Thesis of each project participant.
./docs/latex/BA_Thesis_Tobias_Wulf	Bachelor Thesis directory of Tobias Wulf.
./docs/latex/Manual	Export directory for documentation written in Matlab as pdf export.
./scripts	The scripts directory contains all executable script m-files to solve certain tasks in the project, to generate datasets or execute parts of the toolbox source code.
./src	Source code directory which contains reusable source code clustered in submodule directories. The code can be function oriented or class oriented or a mix of both. Contains no bare script files.

Directory	Task
./src/sensorArraySimulation	Sensor Array Simulation function and class. Contains functions, mathematical functions and classes to simulate an N x N sensor array on base of the TDK TAS2141 characterization dataset.
./src/gaussianProcessRegression	Gaussian Process Regression module which contains basic math functions and submodules to implement GPR models with different kernels using same regression and optimization process.
./src/gaussianProcessRegression/basicMathFunctions	Basic math functions to perform GPR angular predictions.
./src/gaussianProcessRegression/kernelQFC	Exact Quadratic Frobenius Covariance kernel functions which bases on matrice training data.
./src/gaussianProcessRegression/kernelQFCAPX	approximated Quadratic Frobenius Covariance kernel functions which bases on vector training data and uses norm scalar presentation of input matrix data. Using triangle inequation to norm matrix data before compute the covariances.
./src/util	Util function and class space. Function and class source code to solve upcoming help tasks e.g. to manage project content, to support plot framework or reporting or publishing processes.
./src/util/plotFunctions	Contains plot functions for reuse.
./tests	For test driven development each function or class needs a own test space or file. The directory contains these tests.
./temp	Temporally working directory to save intermediate results or the last software state from session before or scratch files which flies around.

Add New Elements

Add new folder to project:

1. Create a new folder and add to Project Path after Matlab flow.
2. Run Checks > Add Files.
3. Run tree command from shell to update directory for the documentation (optional).
4. Update directory task table of this document.

Add new file to project:

1. Create new File and edit the file after Documentation Workflow. and Conventions.
2. Run Checks > Add Files.
3. Label the new file from project pane.
4. Commit file into active branch.
5. Registrate to the documentation if needed (publish, toc and listings docs).

Created on October 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Git Feature Branch Workflow

The project work with Git requires a consistent workflow to apply changes to the Matlab project in a way that no broken source code affects the current state of the project. Therefore Git has the ability to work on new features, issues or bugs in the certain workflow which matches those requirements. This workflow is called Feature Branch Workflow. The workflow describes that for every change in the source code a new branch must be opened in the Git tree. The following changes are committed to the new branch and so that changing commits are not listed in the master branch of the Git tree and have no effect on the made work until the branch is merged back into the master branch. That makes it possible to work on several new features at a time and guarantees a functional working version of the project.

For a deeper understanding in example have a look at the description of Atlassian tutorial page of the Feature Branch Workflow. The listed Matlab help pages describe to use the embedded Matlab Git tooling to apply changes with branching merging.

See Also

- [Feature Branch Workflow](#)
- [Branch and Merge with Git](#)
- [Pull, Push and Fetch Files with Git](#)
- [Update Git File Status and Revision](#)

Examples

1. The master branch is created. Project starts with a first commit.
2. The second commit adds to the master branch files like `.gitattributes`.
3. But there was an issue with that attributes declaration so a new branch is opened to solve that issue.
4. On the same time a new feature must be established e.g. a new script or function. So a second branch is opened.
5. Also a third for a small bug fix.
6. Now the work at those three different task can be done in parallel without affecting each other.
7. Switch between the different branches by checkout the branch and commit the ongoing work into each branch for itself.
8. If the work is done in a branch, the branch must be merged on the master branch. Git makes automated merge commits where the changes from the branches are integrated in master branch files.
9. At this point it is possible that merging conflicts are raised. Those conflicts in the files must be solved manually.
10. Just open a new branch for the next change, switch to it and commit the work until its done and the branch is ready to merge back into master

It is best practice to push all created local branches to a remote repository too! It completes the backup on the one hand and on the other it makes the ongoing work accessible to third.

Created on October 07. 2020 by Tobias Wulf. Copyright Tobias 2020.

Published with MATLAB® R2020b

Documentation Workflow

The documentation workflow describes how to document new m-file scripts or functions and where they must be registered into the publishing process of the documentation. So the published m-file is available in the Matlab help browser of this project.

1. Create a new m-file in the project structure
2. Use the script or function template for initial edit and fill the template with new content.
3. Make introducing documentation entries. If it is a new module, so introduce the module with its own doc where all scripts, functions and classes are listed. If this document already exist, make a new entry.
4. Make help entry in the helptoc.xml via tocitem tag. List all sections of the doc comment as sub tocitems.
5. Introduce the new file to the publish script and make an entry under a fitting section or make a new one if it is a new module or folder.
6. Introduce the new file to export published files script and do toc entries into script file generate pdf-manual.
7. Commit the done work.

See Also

- [Project Structure](#).
- [Display Custom Documentation](#)
- [publishProjectFilesToHTML](#)
- [exportPublishedToPdf](#)

Script Template

```
%% scriptName
% Detailed description of the script task and summary description of
% underlaying script sections.
%
%
%% Requirements
% * Other m-files required: None
% * Subfunctions: None
% * MAT-files required: None
%
%
%% See Also
% * Reference1
% * Reference2
% * Reference3
%
%
% Created on Month DD, YYYY by Creator. Copyright Creator YYYY.
%
% <html>
% <!--
% Hidden Clutter.
% Edited on Month DD, YYYY by Editor: Single line description.
% -->
% </html>
%
%
%% First Script Section
% Detailed section description of step by step executed script code.
disp("Prompt current step or meaningful information of variables.")
Enter section source code
```

```
%> Second Script Section  
% Detailed section description of step by step executed script code.  
disp("Prompt current step or meaningful information of variables."  
Enter section source code
```

Function Template

```
%> functionName  
% Single line summary.  
%  
%% Syntax  
%   outputArg = functionName(positionalArg)  
%   outputArg = functionName(positionalArg, optionalArg)  
%  
%  
%% Description  
% *outputArg = functionName(positionalArg)* detailed use case description.  
%  
% *outputArg = functionName(positionalArg, optionalArg)* detailed use case  
% description.  
%  
%  
%% Examples  
%   Enter example matlab code for each use case.  
%  
%  
%% Input Arguments  
% *positionalArg* argument description.  
%  
% *optionalArg* argument description.  
%  
%  
%% Output Arguments  
% *outputArg* argument description.  
%  
%  
%% Requirements  
% * Other m-files required: None  
% * Subfunctions: None  
% * MAT-files required: None  
%  
%  
%% See Also  
% * Reference1  
% * Reference2  
% * Reference3  
%  
%  
% Created on Month DD. YYYY by Creator. Copyright Creator YYYY.  
%  
% <html>  
% <!--  
% Hidden Clutter.  
% Edited on Month DD. YYYY by Editor: Single line description.  
% -->  
% </html>  
%  
function [outputArg] = functionName(positionalArg, optionalArg)  
    arguments  
        % validate positionalArg: dim class {validator}  
        positionalArg (1,:) double {mustBeNumeric}  
        % validate optionalArg: dim class {validator} = defaultValue
```

```
optionalArg (1,:) double {mustBeNumeric, mustBeEqualSize(optionalArg, optionalArg)} = 4
end
outputArg = positionalArg + optionalArg;
end

% Custom validation function
function mustBeEqualSize(a,b)
    % Test for equal size
    if ~isequal(size(a),size(b))
        eid = 'Size:notEqual';
        msg = 'Size of first input must equal size of second input.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Created on October 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Simulation Workflow

That workflow describes a best practice way to simulate a sensor array with dipole (spherical magnet).

1. Clean up old simulation datasets and plots of by executing `deleteSimulationDatasets` and `deleteSimulationPlots`.
2. Edit `generateConfigMat` to needed specifications for simulation and generate or regenerate `config.mat` by executing the script.
3. Execute `generateSimulationDatasets` to generate configure training and test datasets.
4. Execute the needed plots to describe the simulation as wished.
5. Execute other parts of the software to work with current setup of simulation datasets.
6. Rename plots or move them to a subfolder to save them.
7. Move or rename Datasets if it is needed to keep them after done work.
8. Restart workflow for a next configuration to investigate on.

See Also

- [generateConfigMat](#)
- [deleteSimulationDatasets](#)
- [generateSimulationDatasets](#)
- [deleteSimulationPlots](#)

Created on December 03, 2020 by Tobias. Copyright Tobias 2020.

Published with MATLAB® R2020b

Executable Scripts

Executable scripts of the project to solve various actions or project tasks. The main approach of project scripts is an automated way to collect and execute certain actions in an example to run project documentation at once or generate project configuration file which are used by other scripts or loaded by functions to control and execute task in a unified project structure.

compareGPRKernels

C.compares GPR kernel functions with each and another.

investigateKernelParameters

Analyzes covariance kernel parameters with contour plots.

demoGPRModule

Demonstrates the use of the gaussianProcessRegression module.

exportPublishedToPdf

E.export published HTML files to a pdf manual.

deleteSimulationPlots

D.delete simulation training and test dataset plots from figures and images path with training and test filename pattern.

deleteSimulationDatasets

D.delete generated simulation datasets from data path.

generateSimulationDatasets

G.generate simulation datasets from sensor array simulation configuration.

publishProjectFilesHTML

P.publish Matlab help browser integrated HTML documentation.

generateConfigMat

G.generate configuration for generic use or part use in different program layers.

Created on September 21. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

[publishProjectFilesToHTML](#)

The script is used to publish all toolbox included files to HTML documentation folder docs/html. The script runs a section with certain options for each project part and uses the built-in function to generate the documentation files. For a complete documentation support each generated html document needs to get listed in the project helptoc file with toc entry.

Requirements

- Other m-files required: src/util/removeFilesFromDir.m
- Subfunctions: None
- MAT-files required: data/config.mat

See Also

- [generateConfigMat](#)
- [publishFilesFromDir](#)
- [builddocsearchdb](#)
- [removeFilesFromDir](#)
- [Documentation Workflow](#)

Created on September 21. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Start Publishing Script, Clean Up and Load Config

At first clean up junk from workspace and clear prompt for new output. Set project root path to create absolute file path with fullfile function. Load absolute path variables and publishing options from config.mat

```
disp('Workspace cleaned up ...');
clearvars;
clc;
disp('Load configuration ...');
try
    load('config.mat', 'PathVariables', 'PublishOptions');
catch ME
    rethrow(ME);
end
```

Remove Equation PNG Files

Remove equation png file from HTML output folder before create or recreate publishing files. To prevent the directory expanse of old or edited equation files.

```
yesno = input('Renew eqautions in docs [y/n]: ', 's');
if strcmp(yesno, 'y')
    removeFilesFromDir(PublishOptions.outputDir, '*_eq*.png');
end
```

Project Documentation Files

In this section of the publish script every bare documentation script should be handled and executed to publish. These are m-files without any executeable code so they exist just to transport the documentation content into html output. Get all m-files from docs path. Not recursively but verbose. No expected directory tree search for m-files.

```
disp('Publish project documentation files ...');
publishFilesFromDir(PathVariables.docsPath, PublishOptions, false, true);
```

Executable Script Files

The section collects all ready to execute scripts from project scripts folder and publish them to html documentation folder. Every script must be noticed in Executable_Scripts.m file with one line description. That is very important to not execute the scripts during publishing. If a script contains critical or loop gaining code. In example the publishProjectFilesToHTML.m script such loop gaining code. If eval code during publishing is enabled the script starts publishing itself over and over again because it contains the loop entry via the publish function. So routine is minimal adjusted by evalCode parameter in PublishOptions struct. No expected directory to search for m-files so no recursively but verbose.

```
disp('Publish executable scripts ...');
PublishOptions.evalCode = false;
publishFilesFromDir(PathVariables.scriptsPath, PublishOptions, false, true);
```

Source Code Functions and Classes

That part of the publish script collects function and class m-files from the util section of the source code located in src/. Introduce every new m-file to the source code related documentation m-file and add a description. In general functions and class files are not executed on publishing execution so set evalCode option to false in PublishOptions struct. In addition to that the source code itself should not be in the published document, so the showCode option is switched to false. Publish recursively from underlying directory tree, verbose.

```
disp('Publish source code functions and classes ...');
PublishOptions.evalCode = false;
publishFilesFromDir(PathVariables.srcPath, PublishOptions, true, true);
```

Unit Test Scripts

Publish unit tests scripts for each made test script and overall test runner.

```
disp('Publish unit tests scripts ...');
PublishOptions.evalCode = false;
publishFilesFromDir(PathVariables.unittestPath, PublishOptions, false, true);
```

Build Documentation Database for Matlab Help Browser

To support Matlabs help browser it is needed build searchable help browser entries including a searchable database backend. Matlabs built-in function builddocsearchdb does the trick. The function just needs the output directory of built HTML documentation and it creates a subfolder which includes the database. About the info.xml from the project root and the helptoc.xml file the html documentation folder all listet documentation is accessible. At first remove old database before build the new reference database. Remove autogenerated directory helpsearch-v3. At first get folder content and remove first two relative directory entries from struct. Then delete files and check if files do not exist any more. At least build up new search database entries to Matlab help.

```
disp('Remove old search entries ...');
clearvars;
close all;
clc;
disp('Reload configuration after unit test execution ...');
try
    load('config.mat', 'PathVariables', 'PublishOptions');
catch ME
    rethrow(ME);
end
```

```
if removeFilesFromDir(PathVariables.helpsearchPath)
    buildDocSearchDb(PublishOptions.outputDir);
else
    disp('Could not remove old search entries ...');
end
```

Open Generated Documentation.

Open generated HTML documentation from documentation root HTML file which should be a project introduction or project roadmap page. Comment out if this script is added to project shutdown tasks.

```
open(fullfile(PublishOptions.outputDir, ...
    'gaussianProcessRegression.html'));
disp('Done ...');
```

Published with MATLAB® R2020b

generateConfigMat

Generate configuration mat-file which contains reusable configuration to control the software or certain function parameters.
Centralized collection of configuration. If it is certain configuration needed place it here.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [save](#)
- [load](#)
- [matfile](#)

Created on October 29. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Clean Up

Clear variables from workspace to build up a fresh new configuration workspace.

```
disp('Clean up workspace ...');
close all;
clearvars;
clc;
```

Default Plot Settings

Set default settings for plots and graphics like text interpreter and font size and so on. If script runs on start up, defaults are working for all plots. disp('Set plot defaults ...');

```
set(groot, 'DefaultTextInterpreter', 'latex');
set(groot, 'DefaultLegendInterpreter', 'latex');
set(groot, 'DefaultAxesTickLabelInterpreter', 'latex');
set(groot, 'DefaultAxesFontSize', 20);
set(groot, 'DefaultLineLineWidth', 2.5);
set(groot, 'DefaultAxesLineWidth', 1.5);
set(groot, 'DefaultAxesFontSize', 20);
% set(groot, 'DefaultAxesFontWeight', 'bold');
set(groot, 'DefaultTextFontSize', 20);
% set(groot, 'DefaultaxesFontName', 'Times new Roman')
% set(groot, 'DefaultlegendFontName', 'Times new Roman');
set(groot, 'DefaultAxesXGrid','on');
set(groot, 'DefaultAxesYGrid','on');
set(groot, 'DefaultFigureNumberTitle' , 'off');
set(groot, 'DefaultFigureWindowState', 'normal');
set(groot, 'DefaultFigureMenuBar', 'figure');
set(groot, 'DefaultFigureToolBar', 'figure');
% set(groot, 'DefaultFigureUnits', 'centimeters');
set(groot, 'DefaultFigurePosition', [100, 100, 800, 700]);
set(groot, 'DefaultFigureWindowState', 'normal')
set(groot, 'DefaultFigurePaperType', 'a4');
set(groot, 'DefaultFigurePaperUnits', 'centimeters');
set(groot, 'DefaultFigurePaperOrientation', 'landscape');
```

```
set(groot, 'DefaultFigurePaperPositionMode', 'auto');
set(groot, 'DefaultFigureDoubleBuffer', 'on');
set(groot, 'DefaultFigureRendererMode', 'manual');
set(groot, 'DefaultFigureRenderer', 'painters');
set(groot, 'DefaultTiledlayoutPadding', 'normal');
set(groot, 'DefaultTiledlayoutTileSpacing', 'compact');
set(groot, 'DefaultPolarAxesTickLabelInterpreter', 'latex');
set(groot, 'DefaultPolarAxesFontSize', 20);
% set(groot,);
```

GeneralOptions

General options like formats for strings or date or anything else what has no special relation to a theme complex. Fix parameters.

```
disp('Set general options ...');
GeneralOptions = struct;
GeneralOptions.dateFormat = 'yyyy-mm-dd_HH-MM-SS-FFF';
```

Path Variables

Key path variables and directories, often used in functions or scripts. Collet the path in a struct for easier save the struct fields as variables to config.mat via -struct flag. Fix parameters.

```
disp('Create current project instance to gather information ...');

% create current project instance to retrieve root information
projectInstance = matlab.project.currentProject;

disp('Set path variables ...');
PathVariables =struct;

% project root path, needs to be recreated generic to work on
% different machines
PathVariables.rootPath = projectInstance.RootFolder;

% path to data folder, which contains datasets and config.mat
PathVariables.dataPath = fullfile(PathVariables.rootPath, 'data');

% path to TDK TAS2141 TMR angular sensor characterization dataset
PathVariables.tdkDatasetPath = fullfile(PathVariables.dataPath, ...
    'TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat');

% path to TDK TAS2141 TMR angular sensor characterization dataset
PathVariables.kmz60DatasetPath = fullfile(PathVariables.dataPath, ...
    'NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat');

% path to config file dataset
PathVariables.configPath = fullfile(PathVariables.dataPath, ...
    'config.mat');

% path to training dataset folder
PathVariables.trainingDataPath = fullfile(PathVariables.dataPath, ...
    'training');

% path to test dataset folder
PathVariables.testDataPath = fullfile(PathVariables.dataPath, ...
    'test');

% path to documentation and m-files only for documentation
```

```

PathVariables.docsPath = fullfile(PathVariables.rootPath, ...
    'docs');

% path to publish html documentation output directory, helptoc.xml location
PathVariables.publishHtmlPath = fullfile(PathVariables.docsPath, 'html');

% path to save plots as images svg, eps, png, etc.
PathVariables.saveImagePath = fullfile(PathVariables.publishHtmlPath, ...
    'images');

% path to latex docs folder
PathVariables.latexDocsPath = fullfile(PathVariables.docsPath, ...
    'latex');

% path to latex Thesis Tobias Wulf (take care if comment in)
% PathVariables.thesisTobiasWulf = fullfile(PathVariables.latexDocsPath, ...
%     'BA_Thesis_Tobias_Wulf');

% path to docs export folder for Manual
PathVariables.exportPublishPath = fullfile(PathVariables.latexDocsPath, ...
    'Manual');

% path to style sheet for html documentation, Matlab provided style sheet
PathVariables.publishStyleSheetPath = fullfile(PathVariables.publishHtmlPath, ...
    'docsHtmlStyleSheet.xsl');

% path to documentation search database entries for Matlab help browser support
PathVariables.helpsearchPath = fullfile(PathVariables.publishHtmlPath, ...
    'helpsearch-v3');

% path to executable m-file scripts of the project
PathVariables.scriptsPath = fullfile(PathVariables.rootPath, 'scripts');

% path to source code files, function and class files
PathVariables.srcPath = fullfile(PathVariables.rootPath, 'src');

% path to unittest files, scripts and script suite
PathVariables.unittestPath = fullfile(PathVariables.rootPath, 'tests');

```

Publish Options

These are general options for documents to publish. They are passed to the matlab publish function via a struct where each option gets its own field. The option struct can be copied and adjusted for differing publish conditions in example for scripts, functions, and bare document m-files. Initialize the option struct with output format field name and field value and add further fields (options) with point value. Fix parameters.

```

disp('Set publish options struct for publish function ...');
PublishOptions = struct('format', 'html');
PublishOptions.outputDir = PathVariables.publishHtmlPath;
PublishOptions.stylesheet = PathVariables.publishStyleSheetPath;
PublishOptions.createThumbnail = false;
PublishOptions.figureSnapMethod = 'entireFigureWindow';
PublishOptions.imageFormat = 'png';
PublishOptions.maxHeight = 600;
PublishOptions.maxWidth = 600;
PublishOptions.useNewFigure = false;
PublishOptions.evalCode = false;
PublishOptions.catchError = true;
PublishOptions.codeToEvaluate = [];
PublishOptions.maxOutputLines = Inf;
PublishOptions.showCode = true;

```

Sensor Array Options

The options control the built-up of the sensor array in geometry and technical behavior. This means number of sensors in the array and its size in mm. The supply and offset voltage of each sensor which is needed for using the characterization which is normed in mV/V. These parameters should be fix during generation a bulk of training or test data sets. The simulation function does not covers vectors yet.

```
disp('Set sensor array option for geometry and behavior ...');
SensorArrayOptions = struct;

% Geometry of the sensor array current sensor array can be. Fix parameter.
% square - square sensor array with even distances to each sensor point
SensorArrayOptions.geometry = 'square';

% Sensor array square dimension. Fix parameter.
SensorArrayOptions.dimension = 8;

% Sensor array edge length in mm. Fix parameter.
SensorArrayOptions.edge = 2;

% Sensor array simulated supply voltage in volts. Fix parameter.
SensorArrayOptions.Vcc = 5;

% Sensor array simulated offset voltage for bridge outputs in volts. Fix
% parameter.
SensorArrayOptions.Voff = 2.5;

% Senor array voltage norm factor to recalculate norm bridge outputs to
% given supply voltage and offset voltage, current normin is mV/V which
% implements factor of 1e3. Fix paramter.
SensorArrayOptions.Vnorm = 1e3;
```

Dipole Options

Dipole options to calculate the magnetic field which stimulate the sensor array. The dipole is gained to sphere with additional z distance to the array by sphere radius. These parameters should be fix during generation a bulk of training or test data sets. The simulation function does not covers vectors yet.

```
disp('Set dipole options to calculate magnetic stimulus ...');
DipoleOptions = struct;

% Radius in mm of magnetic sphere in which the magnetic dipole is centered.
% So it can be seen as z-offset to the sensor array. Fix parameter.
DipoleOptions.sphereRadius = 2;

% H-field magnitude to multiply of generated and relative normed dipole
% H-fields, the norming is done in zero position of [0 0 z0 + sphere radius] for
% 0° due to the position of the magnetic moment [-1 0 0] x and y components
% are not relevant, norming without tilt. Magnitude in kA/m. The magnitude
% refers that the sphere magnet has this H-field magnitude in a certain distance
% z0 in example sphere with 2mm sphere radius has a H magnitude of 200kA/m in
% 5mm distance. Standard field strength for ferrite sphere magnets are between
% 180 and 200kA/m. Fix parameter.
DipoleOptions.H0mag = 200;

% Distance in zero position of the spherical magnet in which the imprinted
% H-field strength magnitude takes effect. Together with the sphere radius and
% and the imprinted field strength magnitude the distance in rest position
```

```
% characterizes the spherical magnet to later relative positions of the sensor
% array and generated dipole H-fields in rotation simulation. In mm. Fix
% parameter.
DipoleOptions.z0 = 1;

% Magnetic moment magnitude attach rotation to the dipole field at a
% certain position with x, y and z components. Choose a huge value to
% prevent numeric failures, by norming the factor is eliminated later. Fix
% parameter.
DipoleOptions.M0mag = 1e6;
```

Traning Options

Training options gives the software the needed information to generate training datasets by the sensor array simulation with a dipole magnet as stimulus which pushed with an z offset to a sphere.

```
disp('Set training options to generate dataset ...');
TrainingOptions = struct;

% Use case of options define what dataset it is and where to save resulting
% datasets by simulation function. Fix parameter.
TrainingOptions.useCase = 'Training';

% Sensor array relative position to dipole magnet as position vector with
% x, y and z posiotn in mm. Negative x for left shift, negative y for up
% shift and negative z to place the layer under the dipole decrease z to
% increase the distance. The z-position will be subtracted by dipole sphere
% radius in simulation. So there is an offset given by the sphere radius.
% Loop parameters.
TrainingOptions.xPos = [0,];
TrainingOptions.yPos = [0,];
TrainingOptions.zPos = [7,];

% Dipole tilt in z-axes in degree. Fix parameter.
TrainingOptions.tilt = 0;

% Resolution of rotaion in degree, use same resoultion in training and test
% datasets to have the ability to back reference the index to fullscale
% test data sets. In degree. Fix parameter.
TrainingOptions.angleRes = 0.5;

% Phase index applies a phase offset in the rotation, it is used as phase index
% to a down sampling to generate even distributed angles of a full scale
% rotation. Offset index of full rotation. In example a full scale rotation from
% 0° to 360° - angleRes returns 720 angles, if nAngles is set to 7 it returns 7
% angles [0, 51.5, 103, 154.5, 206, 257.5, 309]. To get a phase shift of 11° set
% phaseIndex to 22 a multiple of the resolution angleRes and get
% [11, 62.5, 114, 165.5, 217, 268.5, 320]. Must be positive integer. Fix
% parameter.
TrainingOptions.phaseIndex = 0;

% Number rotaion angles, even distribute between 0° and 360° with respect
% to the resolution, even down sampling. To generate full scale the number
% relatead to the resolution or fast generate but wrong number set it to 0 to
% generate full scale rotation too. Fix Parameter.
TrainingOptions.nAngles = 20;

% Charcterization datset to use in simulation. Current available datasets are
% TDK - for characterization dataset of TDK TAS2141 TMR sensor
% KMZ60 - for characterization dataset of NXP KMZ60 AMR sensor
TrainingOptions.BaseReference = 'TDK';
```

```
% Characteraztion field which should be load as refernce image from
% characterization data set, in TDK dataset are following fields. In the
% current dataset Rise has the widest linear plateau with a radius of ca.
% 8.5 kA/m. Fix parameter.
% Rise - Bridge outputs for rising stimulus amplituded
% Fall - Bridge outputs for falling stimulus amplitude
% All - Superimposed bridge outputs
% Diff - Differentiated bridge outputs
TrainingOptions.BridgeReference = 'Rise';
```

Test Options

Test options gives the software the needed information to generate test datasets by the sensor array simulation with a dipole magnet as stimulus which pushed with an z offset to a sphere.

```
disp('Set test options to generate dataset ...');
TestOptions = struct;

% Use case of options define what dataset it is and where to save resulting
% datasets by simulation function. Fix Parameter.
TestOptions.useCase = 'Test';

% Sensor array relative position to dipole magnet as position vector with
% x, y and z posiotn in mm. Negative x for left shift, negative y for up
% shift and negative z to place the layer under the dipole decrease z to
% increase the distance. The z-position will be subtracted by dipole sphere
% radius in simulation. So there is an offset given by the sphere radius.
% Loop parameter.
TestOptions.xPos = [0,];
TestOptions.yPos = [0,];
TestOptions.zPos = [7,];

% Dipole tilt in z-axes in degree. Fix parameter.
TestOptions.tilt = 0;

% Resolution of rotaion in degree, use same resoultion in training and test
% datasets to have the ability to back reference the index to fullscale
% test data sets. In degree. Fix parameter.
TestOptions.angleRes = 0.5;

% Phase index applies a phase offset in the rotation, it is used as phase index
% to a down sampling to generate even distributed angles of a full scale
% rotation. Offset index of full rotation. In example a full scale rotation from
% 0° to 360° - angleRes returns 720 angles, if nAngles is set to 7 it returns 7
% angles [0, 51.5, 103, 154.5, 206, 257.5, 309]. To get a phase shift of 11° set
% phaseIndex to 22 a multiple of the resolution angleRes and get
% [11, 62.5, 114, 165.5, 217, 268.5, 320]. Must be positive integer. Fix
% parameter.
TestOptions.phaseIndex = 0;

% Number rotaion angles, even distribute between 0° and 360° with respect
% to the resolution, even down sampling. To generate full scale the number
% relatead to the resolution or fast generate but wrong number to 0 to
% generate full scale rotation. Fix parameter.
TestOptions.nAngles = 720;

% Charcterization datset to use in simulation. Current available datasets are
% TDK - for characterization dataset of TDK TAS2141 TMR sensor
% KMZ60 - for characterization dataset of NXP KMZ60 AMR sensor
TestOptions.BaseReference = 'TDK';
```

```
% Characteraztion field which should be load as refernce image from
% characterization data set, in TDK dataset are following fields. In the
% current dataset Rise has the widest linear plateau with a radius of ca.
% 8.5 kA/m. Fix parameter.
% Rise - Bridge outputs for rising stimulus amplituded
% Fall - Bridge outputs for falling stimulus amplitude
% All - Superimposed bridge outputs
% Diff - Differentiated bridge outputs
TestOptions.BridgeReference = 'Rise';
```

GPR Options

Gaussian Process Regression options to generate a regression model for angular prediction and analyzing the accuracy of the prediction. The GPR model uses one certain covariance function to compute the prediction but it is possible to initiate as zero mean GPR without mean correction and a simple mean function which supports offset amplitude correction of the sinoid inputs for cosine and sine. The GPR uses a quadratic frobenius norm covariance function. That function has two kernel parameters s2f as variance parameter and sl as length scale parameter. Additional a noise variance s2n must be passed to GPR to compute noisy observations.

```
disp('Set test options to generate GPR model ...');
GPROptions = struct();

% Set kernel function to compute covariance matrix, vectors or test point
% covariance. Current available covariance functions are:
% QFC - Quadratic Frobenius Covariance with exact distance.
% QFCAPX - Quadratic Frobenius Covariance with approximated distance of triangle
%           inequation of matrix norm, minimizes training data to a vector.
GPROptions.kernel = 'QFC';

% Initial theta values as vector of [s2f, sl] variance and length scale
% parameter of the quadratic frobenius covariance function. Empirical
% tested start values are the sensor array dimension as length scale and a small
% value as variance factor. Set variance bounds to 1 for 1 on the diagonal of
% the covariance matrix. Only sl will be tuned in the process. Set sf2 not one
% it will be tuned both. Tuning both the vertical scale s2f and horizontal scale
% 2*sl^2 can lead to imbalance of cosine and sine prediction indicated by
% diverging log likelihoods for cosine and sine prediction.
%           [s2f , sl]
GPROptions.theta = [1, 1];

% Set lower and upper bounds to optimize kernel parameters theta which is a
% vector of covariance parameter covariance variance parameter s2f and lenght
% scale parameter sl. These bounds must be set to prevent an overfitting in
% tuning the kernel parameter. If the bounds are to tight in relation of datset
% number in variation the prediction losses its generalization.
% If the bounds are to tight in generel the tuning and optimization procedure
% cannot dismiss bad set points and tries to reach them over and over,
% the causes a limitting which would be break through if the procedure reaches
% the point evaluated as bad set point. If the bound are to wide in relation of
% number in dataset variousity the mean error raises. The model is to complex
% then. Try to keep up simple modles.
GPROptions.s2fBounds = [0.1, 100];
GPROptions.slBounds = [0.1, 100];

% Set initial noise variance to add noise along the diagonal of th covariance
% matrix to predict noisy observation. Set to small values or even 0 to get
% noise free observations.
GPROptions.s2n = 1e-06;

% Set lower and upper bounds for noise adjustment in computing the covariance
```

```
% matrix for noisy observations. These bounds prevent the GPR of overfitting in
% the noise optimization procedure. The default noise at initialization is 1e-5.
GPROptions.s2nBounds = [1e-8, 1e-04];

% Set number of outer optimization runs. For wide parameter bounds it is
% recommended to set the number of runs to min 30 otherwise the bayes
% optimization runs to short in finding error bounds and left with not good
% optimized parameters.
GPROptions.OptimRuns = 50;

% Set standardized logarithmic loss for bayes optimization of s2n with MSLL.
% MSLL results as mean of chosen SLL.
% SLLA - loss by simulation angles
% SLLR - loss by radius = 1 (unit circle)
GPROptions.SLL = 'SLLA';

% Enables mean function and offset and amplitude correction.
% Set basis function to compute H matrix of training points and h vector of
% test point. Current available basis function are:
% zero - init GPR as zero mean GPR m(x) = 0
% poly - init GPR with mean correction m(x) = H' * beta, where H is a matrix
%         polynom mean vectors at each observation points
%         h(x) = [1; x; x^2; x^3; ...] and beta are coefficients of the polynom.
%         For QFC kernel x = ||X||_F
GPROptions.mean = 'zero';

% Polynom degree for mean poly degree option 0 for constant, 1 for 1 + x,
% 2 fo 1 + x + x^2 and so on. Takes only effects if mean = 'poly'. Maximum
% polynom degree is 7.
GPROptions.polyDegree = 1;
```

Save Configuration

Save section wise each config part as struct to standalone variables in config.mat use newest save format with no compression.
create config.mat with timestamp.

```
disp('Create config.mat ...');
timestamp = datestr(now, GeneralOptions.dateFormat);
save(PathVariables.configPath, ...
    'timestamp', ...
    'GeneralOptions', ...
    'PathVariables', ...
    'PublishOptions', ...
    'SensorArrayOptions', ...
    'DipoleOptions', ...
    'TrainingOptions', ...
    'TestOptions', ...
    'GPROptions', ...
    '-v7.3', '-nocompression');
```

generateSimulationDatasets

Generate sensor array simulation datasets for training and test applications. Loads needed configurations from config.mat and characterization data from defined characterization dataset (current: PathVariables.tdkDatasetPath). Simulated datasets are saved to data/training and data/test path. Generate dataset for a predefined configuration at once. Best use is to generate simulation data, do wish application or evaluation on it and save results. Delete datasets, edit configuration and rerun for a new set of datasets.

Requirements

- Other m-files required: simulateDipoleSquareSensorArray.m
- Subfunctions: None
- MAT-files required: config.mat, TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat

See Also

- [sensorArraySimulation](#)
- [simulateDipoleSquareSensorArray](#)
- [generateConfigMat](#)

Created on November 25, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Load Configuration and Characterization Dataset

Load configuration to generate dataset from config.mat and defined characterization dataset.

```
try
    clearvars;
    close all;
    disp('Load configuration ...');
    load('config.mat', 'GeneralOptions', 'PathVariables', ...
        'SensorArrayOptions', 'DipoleOptions', ...
        'TrainingOptions', 'TestOptions');
    disp('Load characterization dataset ...');
    switch TrainingOptions.BaseReference
        case 'TDK'
            TrainingCharDataset = load(PathVariables.tdkDatasetPath);
        case 'KMZ60'
            TrainingCharDataset = load(PathVariables.kmz60DatasetPath);
        otherwise
            error('Unknow characterization dataset in config.');
    end

    switch TestOptions.BaseReference
        case 'TDK'
            TestCharDataset = load(PathVariables.tdkDatasetPath);
        case 'KMZ60'
            TestCharDataset = load(PathVariables.kmz60DatasetPath);
        otherwise
            error('Unknow characterization dataset in config.');
    end

catch ME
    rethrow(ME)
end
```

Generate Training Datasets

Generate training dataset from configuration and characterization dataset.

```
disp('Generate training datasets ...');
simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, TrainingOptions, TrainingCharDataset)
```

Generate Test Datasets

Generate test dataset from configuration and characterization dataset.

```
disp('Generate test datasets ...');
simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, TestOptions, TestCharDataset)
```

Published with MATLAB® R2020b

deleteSimulationDatasets

Delete simulation dataset from data/training and data/test path at once.

Requirements

- Other m-files required: removeFilesFromDir.m
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [removeFilesFromDir](#)
- [gernerateConfigMat](#)
- [Project Structure](#)

Created on November 25. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Load Path to Clean Up

Load path from config.mat and where to find training and test datasets.

```
try
    clearvars;
    close all;
    load('config.mat', 'PathVariables')
    disp('Delete from ...')
    disp(PathVariables.trainingDataPath);
    disp(PathVariables.testDataPath);
catch ME
    rethrow(ME)
end
```

Delete Datasets

Delete datasets from training dataset path and test dataset path with certain file pattern.

```
answer = removeFilesFromDir(PathVariables.trainingDataPath, '*.mat');
fprintf('Delete training datasets: %s\n', string(answer));
answer = removeFilesFromDir(PathVariables.testDataPath, '*.mat');
fprintf('Delete test datasets: %s\n', string(answer));
```

Published with MATLAB® R2020b

deleteSimulationPlots

Delete plots of simulation dataset from figure and image path at once.

Requirements

- Other m-files required: removeFilesFromDir.m
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [removeFilesFromDir](#)
- [generateConfigMat](#)
- [Project Structure](#)

Created on November 02. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Load Path to Clean Up

Load path from config.mat and where to find training and test datasets.

```
try
    clearvars;
    close all;
    load('config.mat', 'PathVariables')
    disp('Delete from ...')
    path = PathVariables.saveImagesPath;
catch ME
    rethrow(ME)
end
```

Delete Dataset Plots

Delete datasets plots from image path and figure path with certain file pattern and extensions.

```
ext = ["fig" "svg" "eps" "pdf" "avi"];
pat = "**";

for e = ext
    asw = removeFilesFromDir(path, join([pat, e], "."));
    fprintf('Deleted pattern %s.%s %s\n', pat, e, string(asw));
end
```

Published with MATLAB® R2020b

exportPublishedToPdf

Export Matlab generated HTML documentation (publish) to pdf-files and combine them into a LaTeX index file ready compile to pdf manual. This script works on unix systems only or needs to be adjusted for windows systems for library path and wkhtmltopdf binary path.

Runs on Unix systems only!

Requirements

- Other m-files required: src/util/removeFilesFromDir.m
- Subfunctions: wkhtmltopdf (shell), pdflatex (shell)
- MAT-files required: data/config.mat

See Also

- [generateConfigMat](#)
- [system](#)
- [wkhtmltopdf](#)
- [publishProjectFilesToHTML](#)
- [Documentation Workflow](#)

Created on December 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Start Exporting Script, Clean Up and Load Config

At first clean up junk from workspace and clear prompt for new output. Set project root path to create absolute file path with fullfile function. Load absolute path variables and publishing options from config.mat

```
disp('Workspace cleaned up ...');
clearvars;
clc;
disp('Load configuration ...');
try
    load('config.mat', 'PathVariables');
catch ME
    rethrow(ME);
end
```

Define Manual TOC

The manual toc must be in the same order as in helptoc.xml in the publish html folder. The toc is used to generate a latex file to include for appendices.

```
toc = ["section",
       "section",
       "subsection",
       "subsection",
       "subsection",
       "subsection",
       "subsection",
       "subsection",
       "section",
       "subsection",
       "subsection",
       "subsection",
       "subsection"];
```

"GaussianProcessDipoleSimulation.pdf";
"Workflows.pdf";
"Project_Preparation.pdf";
"Project_Structure.pdf";
"Git_Feature_Branch_Workflow.pdf";
"Documentation_Workflow.pdf";
"Simulation_Workflow.pdf";
"Executable_Scripts.pdf";
"publishProjectFilesToHTML.pdf";
"generateConfigMat.pdf";
"generateSimulationDatasets.pdf";

```
"subsection", "deleteSimulationDatasets.pdf";
"subsection", "deleteSimulationPlots.pdf";
"subsection", "exportPublishedToPdf.pdf";
"subsection", "demoGPRModule.pdf";
"subsection", "investigateKernelParameters.pdf";
"subsection", "compareGPRKernels.pdf";
"section", "Source_Code.pdf";
"subsection", "sensorArraySimulation.pdf";
"subsubsection", "rotate3DVector.pdf";
"subsubsection", "generateDipoleRotationMoments.pdf";
"subsubsection", "generateSensorArraySquareGrid.pdf";
"subsubsection", "computeDipoleH0Norm.pdf";
"subsubsection", "computeDipoleHField.pdf";
"subsubsection", "simulateDipoleSquareSensorArray.pdf";
"subsection", "gaussianProcessRegression.pdf";
"subsubsection", "initGPR.pdf";
"subsubsection", "initGPROptions.pdf";
"subsubsection", "initTrainDS.pdf";
"subsubsection", "initKernel.pdf";
"subsubsection", "initKernelParameters.pdf";
"subsubsection", "tuneKernel.pdf";
"subsubsection", "computeTuneCriteria.pdf";
"subsubsection", "predFrame.pdf";
"subsubsection", "predDS.pdf";
"subsubsection", "lossDS.pdf";
"subsubsection", "optimGPR.pdf";
"subsubsection", "computeOptimCriteria.pdf";
"subsubsection", "kernelQFCAPX.pdf";
"paragraph", "QFCAPX.pdf";
"paragraph", "meanPolyQFCAPX.pdf";
"paragraph", "initQFCAPX.pdf";
"subsubsection", "kernelQFC.pdf";
"paragraph", "QFC.pdf";
"paragraph", "meanPolyQFC.pdf";
"paragraph", "initQFC.pdf";
"subsubsection", "basicMathFunctions.pdf";
"paragraph", "sinoids2angles.pdf";
"paragraph", "angles2sinoids.pdf";
"paragraph", "decomposeChol.pdf";
"paragraph", "frobeniusNorm.pdf";
"paragraph", "computeInverseMatrixProduct.pdf";
"paragraph", "computeTransposeInverseProduct.pdf";
"paragraph", "addNoise2Covariance.pdf";
"paragraph", "computeAlphaWeights.pdf";
"paragraph", "computeStdLogLoss.pdf";
"paragraph", "computeLogLikelihood.pdf";
"paragraph", "estimateBeta.pdf";
"subsection", "util.pdf";
"subsubsection", "removeFilesFromDir.pdf";
"subsubsection", "publishFilesFromDir.pdf";
"subsubsection", "plotFunctions.pdf";
"paragraph", "plotTDKCharDataset.pdf";
"paragraph", "plotTDKCharField.pdf";
"paragraph", "plotTDKTransferCurves.pdf";
"paragraph", "plotKMZ60CharDataset.pdf";
"paragraph", "plotKMZ60CharField.pdf";
"paragraph", "plotKMZ60TransferCurves.pdf";
"paragraph", "plotDipoleMagnet.pdf";
"paragraph", "plotSimulationDataset.pdf";
"paragraph", "plotSingleSimulationAngle.pdf";
"paragraph", "plotSimulationSubset.pdf";
"paragraph", "plotSimulationCosSinStats.pdf"
```

```

"paragraph",      "plotSimulationDatasetCircle.pdf";
"section",       "Datasets.pdf";
"subsection",    "TDK_TAS2141_Characterization.pdf";
"subsection",    "NXP_KMZ60_Characterization.pdf";
"subsection",    "Config_Mat.pdf";
"subsection",    "Training_and_Test_Datasets.pdf";
"section",       "Unit_Tests.pdf";
"subsection",    "runTests.pdf";
"subsection",    "removeFilesFromDirTest.pdf";
"subsection",    "rotate3DVectorTest.pdf";
"subsection",    "generateDipoleRotationMomentsTest.pdf";
"subsection",    "generateSensorArraySquareGridTest.pdf";
"subsection",    "computeDipoleHONormTest.pdf";
"subsection",    "computeDipoleHFieldTest.pdf";
"subsection",    "tiltRotationTest.pdf";];

nToc = length(toc);
fprintf("%d toc entries remarked ...\\n", nToc);

```

Scan for HTML Files

Scan for all published HTML files in the project publish directory.

```

disp('Scan for published files ...');
HTML = dir(fullfile(PathVariables.publishHtmlPath, '*.html'));
if nToc ~= length(HTML)
    warning(...,
        'TOC (%d) length and found HTML (%d) files are diverging.', ...
        nToc, length(HTML));
end

```

Export HTML to Pdf

Export found HTML files to Pdf files. Each file gets its own Pdf representation. Filename is kept with pdf extension. Write files into Manual folder under LaTeX subdirectory in docs path. Using wkhtmltopdf shell application. Get filename, add pdf extension new path to file. Create shell string to execute with system command. Get current library path (Matlab) and change it to system library path to execute wkhtmltopdf after that restor library back to Matlab.

```

disp('Change local library path to system path ...');
matlabLibPath = getenv('LD_LIBRARY_PATH');
systemLibPath = '/usr/lib/x86_64-linux-gnu';
setenv('LD_LIBRARY_PATH', systemLibPath);

disp('Export published HTML to Pdf ...');
fprintf('Source: %s\\n', HTML(1).folder);
fprintf('Destination: %s\\n', PathVariables.exportPublishPath);
for fhtml = HTML'
    disp(fhtml.name);
    [~, fName, ~] = fileparts(fhtml.name);
    sourcePath = fullfile(fhtml.folder, fhtml.name);
    destinationPath = fullfile(...,
        PathVariables.exportPublishPath, [fName '.pdf']);

    cmdStr = join(["wkhtmltopdf", ...
        "-B 47mm", ...
        "-L 27mm", ...
        "-R 27mm", ...
        "-T 37mm", ...
        "--minimum-font-size 12", ...

```

```

"--enable-local-file-access", ...
"--disable-external-links", ...
"--disable-internal-links", ...
... "--disable-smart-shrinking", ...
"--window-status finished", ...
"--no-stop-slow-scripts", ...
"--javascript-delay 2000", ...
"%s %s"]);
shellStr = sprintf(cmdStr, sourcePath, destinationPath);

try
    [status, cmdout] = system(shellStr);
    % disp(cmdout);
    if status ~= 0
        error('Export failure.');
    end
catch ME
    setenv('LD_LIBRARY_PATH', matlabLibPath);
    disp(cmdout);
    rethrow(ME)
end
end

disp('Restore local library path ...');
setenv('LD_LIBRARY_PATH', matlabLibPath);

```

Write TOC to LaTeX File

Wirete TOC to LaTeX file and generate for each pdf to include a toc content line with marked toc depth. Get the number of pages and add only page title first pdf page.

```

disp('Write TOC to Manual.tex ...');
addFirstPage = "\\\addtocounter{\\sec}{1}\\n" + ...
    "\\\includepdf[page=1," + ...
    "pagecommand={\\phantomsection\\\" + ...
    "\\addcontentsline{toc}{\\sec}{\\label{\\sec}}}" + ...
    "\\protect\\numberline{\\the\\sec}]{\\\" + ...
    "\\label{\\sec}}}]\\n";
addRestPages = "\\\includepdf[page=2-, pagecommand={\\phantomsection}]{\\\" + ...
    "\\label{\\sec}}]\\n";

fileID = fopen(fullfile(... ...
    PathVariables.exportPublishPath, 'Manual.tex'), 'w');
% fprintf(fileID, "% !TEX root = ./thesis.tex\\n");
fprintf(fileID, "% appendix software documentation\\n");
fprintf(fileID, "% @author Tobias Wulf\\n");
fprintf(fileID, ...
    "% Autogenerated LaTeX file. Generated by exportPublishedToPdf.\\n");
fprintf(fileID, ...
    "% Software manual with TOC generated in the same script.\\n");
fprintf(fileID, "% Generated on %s.\\n\\n", datestr(datetime('now')));

pat = regexpPattern("\\d+");
shellStr = "pdftk %s pages" + ...

for i = 1:nToc
    level = toc(i);
    fName = toc(i,2);
    [~, titleStr, ~] = fileparts(fName);
    titleStr = strrep(titleStr, '_', ' ');
    try
        [status, cmdout] = system(sprintf(shellStr, ...
            fullfile(PathVariables.exportPublishPath, fName)));

```

```
pages = double(extract(string(cmdout), pat));

fprintf(fileID, addFirstPage, level, level, level, ...
        titleStr, titleStr, fName);
if pages > 1, fprintf(fileID, addRestPages, fName); end

catch ME
    setenv('LD_LIBRARY_PATH', matlabLibPath);
    fclose(fileID);
    disp(cmdout);
    rethrow(ME)
end
fclose(fileID);
```

Published with MATLAB® R2020b

demoGPRModule

This script demonstrates the use of gaussianProcessRegression module. The demonstration shows single steps of use from initialization to optimization. For use generate a training and testdataset with corresponding position. So the sensor model is built on none diverging coordinates.

Requirements

- Other m-files required: gaussianProcessRegression module files
- Subfunctions: none
- MAT-files required: data/config.mat, corresponding Training and Test dataset

See Also

- [gaussianProcessRegression](#)
- [initGPR](#)
- [tuneGPR](#)
- [optimGPR.html](#)
- [generateConfigMat](#)

Created on February 25, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Start Script, Load Config and Read in Datasets

```
clc;
disp('Start GPR module demonstration ...');
clearvars;
close all;

disp('Load config ...');
load config.mat PathVariables GPROptions;

disp('Search for datasets ...');
TrainFiles = dir(fullfile(PathVariables.trainingDataPath, 'Training*.mat'));
TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
assert(~isempty(TrainFiles), 'No training datasets found.');
assert(~isempty(TestFiles), 'No test datasets found.';

disp('Load first found datasets ...');
try
    TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1).name));
    TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).name));
catch ME
    rethrow(ME)
end

disp('Check dataset coordinates corresponds ...');
assert(all(TrainDS.Data.X == TestDS.Data.X, 'all'), 'Wrong X grid.');
assert(all(TrainDS.Data.Y == TestDS.Data.Y, 'all'), 'Wrong Y grid.');
assert(all(TrainDS.Data.Z == TestDS.Data.Z, 'all'), 'Wrong Z grid.');
```

Compute Means of Test Dataset as Comparing Root

Compute mean sinoids and angular error by raw dataset values. offcos0 - cosine offset offsin0 - sine offset fcos0 - offset free and normed mean cosine from raw test values fsin0 - offset free and normed mean sine from raw test values frad0 - radius by mean

sinoids fang0 - angles by mean sinoids compute with angle function in predDS AAED0 - Absolute Angular Error in Degrees by mean sinoids

```
offcos0 = mean2(TestDS.Data.Vcos);
offsin0 = mean2(TestDS.Data.Vsin);
fcos0 = zeros(TestDS.Info.UseOptions.nAngles, 1);
fsin0 = zeros(TestDS.Info.UseOptions.nAngles, 1);
for n = 1:TestDS.Info.UseOptions.nAngles
    fcos0(n) = mean2(TestDS.Data.Vcos(:,:,n));
    fsin0(n) = mean2(TestDS.Data.Vsin(:,:,n));
end
fcos0 = fcos0 - offcos0;
fsin0 = fsin0 - offsin0;
frad0 = sqrt(fcos0.^2 + fsin0.^2);
fang0 = sinoids2angles(fsin0, fcos0, frad0);
AAED0 = abs(TestDS.Data.angles' - fang0 * 180 / pi);
```

Create GPR Model for Demonstartion

Create three GPR Modles by the same base configuration to compare bare initilized modle with and optimized generated modle with same root of configuration. Mdl1 - optimized modle generated by configuration settings enable free free tuning of variance and length scale by changing the theta(1) to not equal 1 and widining the parameter bounds.

```
disp('Create GPR modles ...');
Mdl1 = optimGPR(TrainDS, TestDS, GPROptions, 0);
```

Prediction on Test Dataset

Predict sinoids and angles on test dataset for each created GPR modle. fang1 - computed angle by predicted sinoids frad1 - computed radius by predicted sinoids fcos1 - predicted cosine fsin1 - predicted sine fcov1 - predictive variance s1 - standard deviation of prediction ciang1 - 95% confidence interval for angles cirad1 - 95% confidence interval for radius

```
[fang1, frad1, fcos1, fsin1, fcov1, s1, ciang1, cirad1] = predDS(Mdl1, TestDS);
```

Compute Losses and Errors on Test Dataset

Compute the loss and error on test dataset for each created GPR modle. AAED1 - Absolute Angular Error in Degrees SLLA1 - Squared Log Loss Angular SLLR1 - Squared Log Loss Radius SEA1 - Squared Error Angular SER1 - Squared Error Radius SEC1 - Squared Error Cosine SES1 - Squared Error Sine

```
[AAED1, SLLA1, SLLR1, SEA1, SER1, SEC1, SES1] = lossDS(Mdl1, TestDS);
```

Plot Area and Expand Model Results

Plot demo results in modle parameter view to show characteristics of covariance functions and modle generalization. Show full rotation on test dataset with angle error, predicted sinoids and confidence intervals.

```
% create general plot scalse and title
angles = TestDS.Data.angles';
ticks = Mdl1.Angles;
titleStr = "Kernel %s: $\sigma_f = %1.2f$, $\sigma_l = %1.2f$," + ...
    " $\sigma_n^2 = %1.2e$, $N = %d$\n" + ...
    "%d $\times$ Sensor-Array, Posistion: (%1.1f,%1.1f,-%1.1f) mm," + ...
    " Magnet Tilt: %2.1f$\circ$";
titleStr = sprintf(titleStr, ...
```

```

Md11.kernel, Md11.theta(1), Md11.theta(2), Md11.s2n, ...
Md11.N, Md11.D, Md11.D, ...
TestDS.Info.UseOptions.xPos, ...
TestDS.Info.UseOptions.yPos, ...
TestDS.Info.UseOptions.zPos, ...
TestDS.Info.UseOptions.tilt);

% create figure for model view
figure('Name', Md11.kernel, 'Units', 'normalize', 'OuterPosition', [0 0 1 1]);
t=tiledlayout(2,2);
title(t, titleStr, 'Interpreter', 'latex', 'FontSize', 24);

% plot covariance slice for first covariance sample
nexttile;
p1 = plot(Md11.Ky, 'Color', [0.8 0.8 0.8]);
hold on;
p2 = yline(mean2(Md11.Ky), 'Color', '#0072BD', 'LineWidth', 2.5);
p3 = plot(1:Md11.N, Md11.Ky(1,:), 'kx-.');
xlim([1, Md11.N]);
ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
legend([p1(1), p2, p3], {'$i \neq 0$', '$\mu(K)$', '$i = 0$'});
xlabel('$n$ Samples');
ylabel('Autocorrelation Coeff.');
title('a) $K$-Matrix $i$-th Row');

% plot covariance matrix
nexttile([2, 1]);
colormap('jet');
imagesc(Md11.Ky);
axis square;
colorbar;
xlabel('$j$');
ylabel('i');
title(sprintf('b) $K$-Matrix $\times %d$ Samples', Md11.N, Md11.N));

% plot modle adjust as squared logarithmic loss for angles and radius
nexttile;
plot(angles, SLLA1, 'x-.');
hold on;
plot(angles, SLLR1, 'x-.');
xlim([0 360]);
legend({'$SLLA$' , '$SLLR$'});
xlabel('$\alpha$ in $^\circ$');
ylabel('$SLL$');
title(sprintf('c) $MSLLA = %1.2f$, $MSLLR = %1.2f$', mean(SLLA1), mean(SLLR1)));

% create figure for rotation results of test dataset
figure('Name', 'Rotation and Errors', 'Units', 'normalize', ...
    'OuterPosition', [0 0 1 1]);
t = tiledlayout(2,2);
title(t, titleStr, 'Interpreter', 'latex', 'FontSize', 24);

% plot circle with gpr reuslts and pure mean results
nexttile;
polarplot(fang0, frad0, 'LineWidth', 3.5);
hold on;
polarplot(fang1, frad1, 'LineWidth', 3.5);
polarscatter(Md11.Angles * pi / 180, 1.2 *ones(Md11.N, 1), 52, [0.8 0.8 0.8], ...
    'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);
legend({'Mean', 'GPR', 'Ref. $\alpha$'});
rticklabels(["", "0.5", "1"]);
title('a) Rotation along $Z$-Axis in $^\circ$')

```

```
% plot predicted, ideal and none treated sinusoids
nexttile;
p1 = plot(angles, cosd(angles), 'k-.', 'LineWidth', 6.5);
hold on;
plot(angles, sind(angles), 'k-.', 'LineWidth', 6.5);
p2 = plot(angles, fcos0, 'Color', '#0072BD');
plot(angles, fsin0, 'Color', '#0072BD');
p3 = plot(angles, fcos1, 'Color', '#D95319');
plot(angles, fsin1, 'Color', '#D95319');
xlim([0 360]);
ylim([-1.1 1.1]);
xlabel('$\alpha$ in $\circ$');
legend([p1 p2 p3], {'Ideal', 'Mean', 'GPR'});
title('b) Sine and Cosine');

% plot absolut angle errors of gpr and mean results
nexttile;
plot(angles, AAED0);
hold on;
plot(angles, AAED1);
yline(mean(AAED1), 'k-.', 'LineWidth', 3.5)
[~, idx] = max(AAED1);
scatter(angles(idx), max(AAED1), 52, [0.8 0.8 0.8], ...
    'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5)
ylim([0 4]);
xlim([0 360]);
xlabel('$\alpha$ in $\circ$');
ylabel('$|\epsilon_{abs}|$ in $\circ$');
legend({'Mean', 'GPR', '$\mu(|\epsilon_{abs}|)$', '$\max|\epsilon_{abs}|$'});
tstr = 'c) $\mu(|\epsilon_{abs}|) = %1.2f$, $\max|\epsilon_{abs}| = %1.2f$';
tstr = sprintf(tstr, mean(AAED1), max(AAED1));
title(tstr);

% plot 95 percent confidence intervals for angles and radius
nexttile;
yyaxis left;
plot(angles, (ciang1-fang1) * 180/pi, '--', 'Color', '#0072BD');
ylabel('$CIA_{95\%} - \alpha$ in $\circ$');
yyaxis right;
plot(angles, (cirad1-frad1), '--', 'Color', '#D95319');
ylabel('$CIR_{95\%} - r$');
xlim([0 360]);
xlabel('$\alpha$ in $\circ$');
title('d) Centerd 95% GPR Confidence Intervals')
```

Published with MATLAB® R2020b

investigateKernelParameters

Sweep kernel parameters against inner tuning criteria which is built by the logarithmic likelihoods for cosine and sine fit on training datasets.

Requirements

- Other m-files required: gaussianProcessRegression module files
- Subfunctions: none
- MAT-files required: data/config.mat, corresponding Training and Test dataset

See Also

- [gaussianProcessRegression](#)
- [initGPR](#)
- [tuneGPR](#)
- [optimGPR.html](#)
- [generateConfigMat](#)

Created on March 13. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Start Script, Load Config and Read in Datasets

```
clc;
disp('Start GPR module demonstration ...');
clearvars;
%close all;

disp('Load config ...');
load config.mat PathVariables GPROptions;

disp('Search for datasets ...');
TrainFiles = dir(fullfile(PathVariables.trainingDataPath, 'Training*.mat'));
TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
assert(~isempty(TrainFiles), 'No training datasets found.');
assert(~isempty(TestFiles), 'No test datasets found.');

disp('Load first found datasets ...');
try
    TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1).name));
    TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).name));
catch ME
    rethrow(ME)
end

disp('Check dataset coordinates corresponds ...');
assert(all(TrainDS.Data.X == TestDS.Data.X, 'all'), 'Wrong X grid.');
assert(all(TrainDS.Data.Y == TestDS.Data.Y, 'all'), 'Wrong Y grid.');
assert(all(TrainDS.Data.Z == TestDS.Data.Z, 'all'), 'Wrong Z grid.');
```

Create GPR Model for Investigation

```
disp('Create GPR modules ...');
Md11 = optimGPR(TrainDS, TestDS, GPROptions, 0);
```

Sweep Title with Model Parameters

```

titleStr = "Kernel %s: $\\sigma_f = %1.2f$, $\\sigma_l = %1.2f$," + ...
" $\\sigma_n^2 = %1.2e$, $N = %d$\n" +...
"$%d \\times %d$ Sensor-Array, Posistion: (%1.1f,%1.1f,-%1.1f)$ mm," + ...
" Magnet Tilt: %%2.1f^\\circ";
titleStr = sprintf(titleStr, ...
Md11.kernel, Md11.theta(1), Md11.theta(2), Md11.s2n, ...
Md11.N, Md11.D, Md11.D, ...
TestDS.Info.UseOptions.xPos, ...
TestDS.Info.UseOptions.yPos, ...
TestDS.Info.UseOptions.zPos, ...
TestDS.Info.UseOptions.tilt);

```

Execute Parameter Sweep with Constant Noise

```

nEval = 300;
disp('Sweep kernel parameters with constant noise ...');
sweepKernelWithConstNoise(Md11, nEval, titleStr, PathVariables)

```

Execute Parameter Sweep with Constant Variance

```

nEval = 300;
disp('Sweep kernel parameters with constant variance ...');
sweepKernelWithConstVariance(Md11, nEval, titleStr, PathVariables)

```

Execute Parameter Sweep with Constant Lengthscale

```

nEval = 300;
disp('Sweep kernel parameters with constant lengthscale ...');
sweepKernelWithConstLengthscale(Md11, nEval, titleStr, PathVariables)

```

Sweep Kernel Parameters vs. Likelihood Criteria with Constant Noise

```

function sweepKernelWithConstNoise(Mdl, nEval, titleStr, PathVariables)

    % create sweep parameters for sweeping theta to given mode
    s2f = linspace(Mdl.s2fBounds(1) * 0.1, Mdl.s2fBounds(2) * 10, nEval);
    s1 = linspace(Mdl.s1Bounds(1) * 0.1, Mdl.s1Bounds(2) * 10, nEval);
    [s1, s2f] = meshgrid(s1, s2f);

    % allocate memory for inner tuning criteria, combined likelihoods for cosine
    % and sine fit on trainings data
    RLI = zeros(nEval, nEval);

    % run sweep in multiprocess pool to gain speed
    parfor i = 1:nEval
        for j = 1:nEval
            % compute sweep with tuning criteria of inner GPR optimization of
            % tuning GPR kernel parameters
            RLI(i,j) = computeTuneCriteria([s2f(i,j) s1(i,j)], Mdl);
        end
    end

    % plot results in countour plot
    fig = figure('Name', 'Sweep Kernel Parameters with Constant Noise', ...

```

```

'Units', 'normalize', 'OuterPosition', [0 0 1 1]);

% plot sweep with log axis
contourf(sl, s2f, RLI, linspace(min(RLI, [], 'all') + 1, 1, 10), ...
    'LineWidth', 1.5);
set(gca, 'YScale', 'log')
set(gca, 'XScale', 'log')
hold on;
grid on;

% plot bounds origin model parameters
p1 = yline(Mdl.s2fBounds(1), 'k-.', 'LineWidth', 2.5);
yline(Mdl.s2fBounds(2), 'k-.', 'LineWidth', 2.5);
yline(Mdl.theta(1), 'k', 'LineWidth', 2.5);
xline(Mdl.s1Bounds(1), 'k-.', 'LineWidth', 2.5);
xline(Mdl.s1Bounds(2), 'k-.', 'LineWidth', 2.5);
xline(Mdl.theta(2), 'k', 'LineWidth', 2.5);

% plot fmincon search area
p2 = patch( ...
    [Mdl.s1Bounds(1), Mdl.s1Bounds(2), ...
    Mdl.s1Bounds(2), Mdl.s1Bounds(1)], ...
    [Mdl.s2fBounds(1) Mdl.s2fBounds(1), ...
    Mdl.s2fBounds(2) Mdl.s2fBounds(2)], ...
    [0.8 0.8 0.8], 'FaceAlpha', 0.7);

% plot argmin fmincon result
p3 = scatter(Mdl.theta(2), Mdl.theta(1), 60, [0.8 0.8 0.8], ...
    'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);

% labels, titles, legends
xlabel('$\sigma_l$')
ylabel('$\sigma_f^2$')
title(titleStr);
stStr = "$\sigma_f^2, \sigma_l | \sigma_n^2 = " + ...
    "\arg\min\tilde{R} \mathcal{L}(I)" + ...
    " $(\sigma_f^2, \sigma_l | \sigma_n^2) \$ f. \sigma_n^2 = const.";
subtitle(stStr);
legend([p1, p2, p3], ...
    {"Parameter Bounds", "Search Area", ...
    sprintf("fmincon $\tilde{R} \mathcal{L}(I) (%1.2f,%1.2f|%1.2e)=%1.2f$", ...
    Mdl.theta, Mdl.s2n, -(Mdl.LMLcos + Mdl.LMLSin))}, ...
    'Location', 'South');

cb = colorbar;
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 24;
cbStr = "$\tilde{R} (\sigma_f^2, \sigma_l | \sigma_n^2) \$";
cb.Label.String = cbStr;

% save and close
%   fPath = fullfile(PathVariables.saveImagePath, 'Sweep_Kernel_Const_Noise');
%   print(fig, fPath, '-dsvg');
%   close(fig);
end

```

Sweep Kernel Parameters vs. Likelihood Criteria with Constant Variance

```
function sweepKernelWithConstVariance(Mdl, nEval, titleStr, PathVariables)
```

```
% keep s2n origin to plot later
s2nOrigin = Mdl.s2n;

% create sweep parameters for sweeping lengthscale and noise to given mode
s2n = linspace(Mdl.s2nBounds(1) * 0.1, Mdl.s2nBounds(2) * 10, nEval);
s1 = linspace(Mdl.s1Bounds(1) * 0.1, Mdl.s1Bounds(2) * 10, nEval);
s2f = Mdl.theta(1);

% allocate memory for inner tuning criteria, combined likelihoods for cosine
% and sine fit on trainings data
RLI = zeros(nEval, nEval);

% run sweep in multiprocess pool to gain speed
for i = 1:nEval
    % assign struct values to compute corresponding lenght scale row wise
    % due to parfor struct issue
    Mdl.s2n = s2n(i);
    parfor j = 1:nEval
        % compute sweep with tuning criteria of inner GPR optimization of
        % tuning GPR kernel parameters, variance is set to 1
        RLI(i,j) = computeTuneCriteria([s2f s1(j)], Mdl);
    end
end

% generate grid on vectors to plot results
[s1, s2n] = meshgrid(s1, s2n);

% plot results in countour plot
fig = figure('Name', 'Sweep Kernel Parameters with Constant Variance', ...
    'Units', 'normalize', 'OuterPosition', [0 0 1 1]);

% plot sweep with log axis
contourf(s1, s2n, RLI, linspace(min(RLI, []), 'all') + 1, 1, 10), ...
    'LineWidth', 1.5);
set(gca, 'YScale', 'log')
set(gca, 'XScale', 'log')
hold on;
grid on;

% plot bounds origin model parameters
p1 = yline(Mdl.s2nBounds(1), 'k-.', 'LineWidth', 2.5);
yline(Mdl.s2nBounds(2), 'k-.', 'LineWidth', 2.5);
yline(s2nOrigin, 'k', 'LineWidth', 2.5);
xline(Mdl.s1Bounds(1), 'k-.', 'LineWidth', 2.5);
xline(Mdl.s1Bounds(2), 'k-.', 'LineWidth', 2.5);
xline(Mdl.theta(2), 'k', 'LineWidth', 2.5);

% plot fmincon search area
p2 = patch( ...
    [Mdl.s1Bounds(1), Mdl.s1Bounds(2), ...
    Mdl.s1Bounds(2), Mdl.s1Bounds(1)], ...
    [Mdl.s2nBounds(1) Mdl.s2nBounds(1), ...
    Mdl.s2nBounds(2) Mdl.s2nBounds(2)], ...
    [0.8 0.8 0.8], 'FaceAlpha', 0.7);

% plot argmin fmincon result
p3 = scatter(Mdl.theta(2), s2nOrigin, 60, [0.8 0.8 0.8], ...
    'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);

% labels, titles, legends
xlabel('$\sigma_1$')
ylabel('$\sigma_n^2$')
```

```

title(titleStr);
stStr = "$\sigma_f^2, \sigma_l|\sigma_n^2 = " + ...
    "\arg\min\tilde{R}|\mathcal{L}I" + ...
    "(\sigma_f^2, \sigma_l|\sigma_n^2) f. \sigma_f^2 = const.$";
subtitle(stStr);
legend([p1, p2, p3], ...
    {"Parameter Bounds", "Search Area", ...}
    sprintf("fmincon $\tilde{R}|\mathcal{L}I (%1.2f,%1.2f|%1.2e)=%1.2f$", ...
        Mdl.theta, s2nOrigin, -(Mdl.LMLcos + Mdl.LMLsin))), ...
    'Location', 'South')

cb = colorbar;
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 24;
cbStr = "$\tilde{R}|\mathcal{L}I (\sigma_f^2, \sigma_l|\sigma_n^2)$";
cb.Label.String = cbStr;

% save and close
%   fPath = fullfile(PathVariables.saveImagesPath, 'Sweep_Kernel_Const_Var');
%   print(fig, fPath, '-dsvg');
%   close(fig);
end

```

Sweep Kernel Parameters vs. Likelihood Criteria with Constant Lengthscale

```

function sweepKernelWithConstLengthscale(Mdl, nEval, titleStr, PathVariables)

% keep s2n origin to plot later
s2nOrigin = Mdl.s2n;

% create sweep parameters for sweeping lengthscale and noise to given mode
s2n = linspace(Mdl.s2nBounds(1) * 0.1, Mdl.s2nBounds(2) * 10, nEval);
s2f = linspace(Mdl.s2fBounds(1) * 0.1, Mdl.s2fBounds(2) * 10, nEval);
s1 = Mdl.theta(2);

% allocate memory for inner tuning criteria, combined likelihoods for cosine
% and sine fit on trainings data
RLI = zeros(nEval, nEval);

% run sweep in multiprocess pool to gain speed
for i = 1:nEval
    % assign struct values to compute corresponding lenght scale row wise
    % due to parfor struct issue
    Mdl.s2n = s2n(i);
    parfor j = 1:nEval
        % compute sweep with tuning criteria of inner GPR optimization of
        % tuning GPR kernel parameters, variance is set to 1
        RLI(i,j) = computeTuneCriteria([s2f(j), s1], Mdl);
    end
end

% generate grid on vectors to plot results
[s2f, s2n] = meshgrid(s2f, s2n);

% plot results in countour plot
fig = figure('Name', 'Sweep Kernel Parameters with Constant Lengthscale',...
    'Units', 'normalize', 'OuterPosition', [0 0 1 1]);

% plot sweep with log axis
contourf(s2f, s2n, RLI, linspace(min(RLI, []), 'all') + 1, 1, 10), ...

```

```

'LineWidth', 1.5);
set(gca, 'YScale', 'log')
set(gca, 'XScale', 'log')
hold on;
grid on;

% plot bounds origin model parameters
p1 = yline(Mdl.s2nBounds(1), 'k-.', 'LineWidth', 2.5);
yline(Mdl.s2nBounds(2), 'k-.', 'LineWidth', 2.5);
yline(s2nOrigin, 'k', 'LineWidth', 2.5);
xline(Mdl.s2fBounds(1), 'k-.', 'LineWidth', 2.5);
xline(Mdl.s2fBounds(2), 'k-.', 'LineWidth', 2.5);
xline(Mdl.theta(1), 'k', 'LineWidth', 2.5);

% plot fmincon search area
p2 = patch( ...
    [Mdl.s2fBounds(1), Mdl.s2fBounds(2), ...
     Mdl.s2fBounds(2), Mdl.s2fBounds(1)], ...
    [Mdl.s2nBounds(1) Mdl.s2nBounds(1), ...
     Mdl.s2nBounds(2) Mdl.s2nBounds(2)], ...
    [0.8 0.8 0.8], 'FaceAlpha', 0.7);

% plot argmin fmincon result
p3 = scatter(Mdl.theta(1), s2nOrigin, 60, [0.8 0.8 0.8], ...
    'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);

% labels, titles, legends
xlabel('$\sigma_f^2$')
ylabel('$\sigma_n^2$')
title(titleStr);
stStr = "$\sigma_f^2, \sigma_l | \sigma_n^2 = " + ...
    "\arg\min\tilde{R}(\mathcal{L})" + ...
    "(\sigma_f^2, \sigma_l | \sigma_n^2) f. \sigma_l = const.$";
subtitle(stStr);
legend([p1, p2, p3], ...
    {"Parameter Bounds", "Search Area", ...
     sprintf("fmincon $\tilde{R}(\mathcal{L})(%1.2f,%1.2f| %1.2e)=%1.2f$", ...
         Mdl.theta, s2nOrigin, -(Mdl.LMLcos + Mdl.LMLsin))}, ...
    'Location', 'South');

cb = colorbar;
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 24;
cbStr = "$\tilde{R}(\mathcal{L})(\sigma_f^2, \sigma_l | \sigma_n^2)$";
cb.Label.String = cbStr;

% save and close
%   fPath = fullfile(PathVariables.saveImagePath, 'Sweep_Kernel_Const_Len');
%   print(fig, fPath, '-dsvg');
%   close(fig);
end

```

Published with MATLAB® R2020b

compareGPRKernels

This script compares the implemented kernel function with each and another. It initiates each kernel with different kernel parameters and in a row and compares them in two plots for kernel functions and covariance matrix.

Requirements

- Other m-files required: gaussianProcessRegression module files
- Subfunctions: none
- MAT-files required: data/config.mat, corresponding Training and Test dataset

See Also

- [gaussianProcessRegression](#)
- [initGPR](#)
- [generateConfigMat](#)

Created on April 11, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Load Datasets

```
clear all;
close all;
disp('Load config ...');
load config.mat PathVariables GPROptions;
TrainFiles = dir(fullfile(PathVariables.trainingDataPath, 'Training*.mat'));
TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
assert(~isempty(TrainFiles), 'No training datasets found.');
assert(~isempty(TestFiles), 'No test datasets found.');

disp('Load first found datasets ...');
try
    TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1).name));
    TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).name));

catch ME
    rethrow(ME)
end
```

Initiate Kernels with different Kernelparameters

```
ma11 = initGPR(TrainDS, GPROptions);

GPROptions.theta = [1 2];
ma12 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [1 0.5];
ma105 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [1 4];
ma14 = initGPR(TrainDS, GPROptions);

GPROptions.theta = [2 1];
mc21 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [0.5 1];
mc051 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [4 1];
```

```

mc41 = initGPR(TrainDS, GPROptions);

GPROptions.kernel = 'QFCAPX';
GPROptions.theta = [1 1];
mb11 = initGPR(TrainDS, GPROptions);

GPROptions.theta = [1 2];
mb12 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [1 0.5];
mb105 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [1 4];
mb14 = initGPR(TrainDS, GPROptions);

GPROptions.theta = [2 1];
md21 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [0.5 1];
md051 = initGPR(TrainDS, GPROptions);
GPROptions.theta = [4 1];
md41 = initGPR(TrainDS, GPROptions);

```

Plot Area

```

% plot covariance slice for first covariance sample
figure;
tiledlayout(2,2);

nexttile;
hold on;

p1 = plot(1:ma11.N, ma11.Ky(1,:), 'k');
p2 = plot(1:ma12.N, ma12.Ky(1,:), 'b-.');
p3 = plot(1:ma105.N, ma105.Ky(1,:), 'r-.');
p4 = plot(1:ma14.N, ma14.Ky(1,:), 'g-.');

xlim([1, ma11.N]);
%ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
legend([p1, p2, p3, p4], ...
    {'$\theta = (1,1)$', ...
    '$\theta = (1,2)$', ...
    '$\theta = (1,0.5)$', ...
    '$\theta = (1,4)$'}, 'Location', 'north');
xlabel('$j$-tes$ Sample$');
title('a) $k(X_1, X_j|\theta) f. d_E^2$, $\theta_1 = konst.$');

nexttile;
hold on;

p1 = plot(1:mb11.N, mb11.Ky(1,:), 'k');
p2 = plot(1:mb12.N, mb12.Ky(1,:), 'b-.');
p3 = plot(1:mb105.N, mb105.Ky(1,:), 'r-.');
p4 = plot(1:mb14.N, mb14.Ky(1,:), 'g-.');

xlim([1, mb11.N]);
%ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
legend([p1, p2, p3, p4], ...
    {'$\theta = (1,1)$', ...
    '$\theta = (1,2)$', ...
    '$\theta = (1,0.5)$', ...
    '$\theta = (1,4)$'}, 'Location', 'north');
xlabel('$j$-tes$ Sample$');
title('b) $k(X_1, X_j|\theta) f. d_E^2$, $\theta_1 = konst.$');

```

```

nexttile;
hold on;

p1 = plot(1:ma11.N, ma11.Ky(1,:), 'k');
p2 = plot(1:mc21.N, mc21.Ky(1,:), 'b-.');
p3 = plot(1:mc051.N, mc051.Ky(1,:), 'r-.');
p4 = plot(1:mc41.N, mc41.Ky(1,:), 'g-.');
xlim([1, ma11.N]);
%ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
legend([p1, p2, p3, p4], ...
    {'$\theta = (1,1)$', ...
    '$\theta = (2,1)$', ...
    '$\theta = (0.5,1)$', ...
    '$\theta = (4,1)$'}, 'Location', 'north');
xlabel('$j$-tes Sample');
title('c) $k(x_1, x_j|\theta) f. d_F^2$, $\theta_2 = konst.$');

nexttile;
hold on;

p1 = plot(1:mb11.N, mb11.Ky(1,:), 'k');
p2 = plot(1:md21.N, md21.Ky(1,:), 'b-.');
p3 = plot(1:md051.N, md051.Ky(1,:), 'r-.');
p4 = plot(1:md41.N, md41.Ky(1,:), 'g-.');
xlim([1, mb11.N]);
%ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
legend([p1, p2, p3, p4], ...
    {'$\theta = (1,1)$', ...
    '$\theta = (2,1)$', ...
    '$\theta = (0.5,1)$', ...
    '$\theta = (4,1)$'}, 'Location', 'north');
xlabel('$j$-tes Sample');
title('c) $k(x_1, x_j|\theta) f. d_E^2$, $\theta_2 = konst.$');

% plot covariance matrix
figure;
tiledlayout(1,2);
colormap('jet');

nexttile;

imagesc(ma11.Ky);
axis square;
xlabel('$j$');
ylabel('$i$');
title('a) $K(X, X|\theta) f. d_F^2$, $\theta = (1,1)$')

nexttile;

% colormap('jet');
imagesc(mb11.Ky);
axis square;
xlabel('$j$');
ylabel('$i$');
title('a) $K(X, X|\theta) f. d_E^2$, $\theta = (1,1)$')

c = colorbar;
c.TickLabelInterpreter = 'latex';

```

.....

Published with MATLAB® R2020b

Source Code

The project source code is clustered in modules where every subdirectory represents one certain module. Each module gathers functions and classes which are related to module specific themes or task fields. So the basic structured source code is located here. The combination of module functionality takes place in executable area of the project. So use the functions and classes in scripts and further on compiled binaries. Do not write bare executable source code here. For reproducible results and source code traceability each module has its own documentation entry where all underlaying functions and classes are listed. The best practice to develop new source code or modules is to do it in test driven way. This means write a test m-file for every new function or class m-file and test the functionality of the source code with assertion. This test driven development is called unittest and provides in combination with detailed documentation a high percentage of reusable source code.

sensorArraySimulation

Function space to solve sensor array simulation with a certain magnetic stimulus. The Array simulation is based on the TDK TAS2141 characterization dataset. A magnetic dipole is used as basic magnetic stimulus and moved as imaginary sphere magnet with a certain radius. The magnet rotates in z-direction counterclockwise.

util

Util function and classes to provide reuse for often upcomings tasks and functionality besides project kernel and module source code. Located under source code directory: **./src/util**.

Created on October 10, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

sensorArraySimulation

A spherical magnet is assumed to be used for stimulation of the sensor array. The far field of a spherical magnet can be approximately described by the magnetic field of a magnetic dipole. The magnetization of the sphere is assumed to be in y direction and the magnetic moment in rest position for 0° points in x direction. The magnet must be defined in a way that its field lines or field strengths own gradients sufficiently strong enough in the distance to the sensor array and so the rotation of the magnet generates a small scattering of the bridge outputs in the individual sensor points in the array. That all sensors in the array approximately perceive the same magnetic field gradients of the current rotation step and the sensors in the array run through approximately equal circular paths in the characterization field. This means the spherical magnet is characterized by a favorable matching of sphere radius and a certain distance in rest position in which a sufficiently high field strength takes effect. Here are neglected small necessary distances which are demanded in standard automotive applications. The focus here is on to generate simulation datasets, which are uniform and valid for angle detection. The modelling of suitable small magnets is not taking place of the work.

A good working magnet is found empirical for H-field magnitudes of 200 kA/m and a distance from surface of 1 mm. See below figure of used magnet.

To change settings for simulation edit the config script and rerun it. To generate training and test data set use simulation script. It generates dataset for all position known to TrainingOptions and TestOptions in config. Generate a set of dataset for one evaluation case. Evaluate datasets, save results for later clustering, edit config for next use case and rerun simulation.

The simulation bases on TDK TAS2141 "Rise" characterization field. It has the widest linear plateau for corresponding Hx and Hy field strengths.

See Also

- [generateConfigMat](#)
- [generateSimulationDatasets](#)
- [deleteSimulationDatasets](#)

simulateDipoleSquareSensorArray

Simulates a square sensor array with dipole magnet as stimulus for a certain setup of training or test options. Saves generated dataset to data/training or data/test.

computeDipoleHField

Computes the dipole field strength for meshgrids with additional ability to imprint a certain field strength in defined radius on resulting field.

computeDipoleH0Norm

Computes a norm factor to imprint a magnetic field strength to magnetic dipole fields with same magnetic moment magnitude and constant dipole sphere radius on which the imprinted field strength takes effect.

generateSensorArraySquareGrid

Generates a square sensor array grid in a 3D coordinate system with relative position to center of the system and an additional offset in z direction.

generateDipoleRotationMoments

Generates magnetic rotation moments to rotate a magnetic dipole in its z-axes with a certain tilt.

rotate3DVector

Rotates a vector with x-, y- and z-components in a 3D-coordinate system. Rotate one step of certain angles.

Created on November 04. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

.....

Published with MATLAB® R2020b

rotate3DVector

Rotates a 3 dimensional vector with x-, y- and z-components in a 3 dimensional coordinate system along the x-, y- and z-axes. Using rotation matrix for x-, y- and z-axes. Angle must be served in degree. Vector must be a column vector 3 x 1 or matrix related x-, y-, z-components 3 x N.

This function was originally created by Thorben Schüthe is ported into source code under improvements and including Matlab built-in functions. Function rewritten.

Syntax

```
rotated = rotate3DVector(vector, alphaX, betaY, gammaZ)
```

Description

rotated = rotate3DVector(vector, alphaX, betaY, gammaZ) returns a rotated vector which is rotated by given angles on related axes. alphaX rotates along the x-axes, betaY along the y-axes and gammaZ along the z-axes. Therfore each rotation is described by belonging rotation matrix. The resulting rotation of the vector is computed by the matrix and vector multiplacation of the rotation matrices and the input vecotor.

$$v' = Av = R_z(\gamma)R_y(\beta)R_x(\alpha)v$$

Examples

```
% rotate a vector along z-axes by 45°  
vector = [1; 0; 0]  
rotated = rotate3DVector(vector, 0, 0, 45)  
  
% rotate a vector along z-axes by 35° with a tilt in x-axes by 1°  
vector = [1; 0; 0]  
rotated = rotate3DVector(vector, 1, 0, 35)  
  
% rotate a vector along z-axes by 35° with a tilt in x-axes by 1° and a  
% tilt in y-axes by 5°  
vector = [1; 0; 0]  
rotated = rotate3DVector(vector, 1, 5, 35)
```

Input Arguments

vector is a 3 x N column vector of real numbers which represents the a vector in a 3D coordinate system with x-, y- and z-components.

alphaX is a scalar angular value in degree and rotates the vector in the x-axes.

betaY is a scalar angular value in degree and rotates the vector in the y-axes.

gammaZ is a scalar angular value in degree and rotates the vector in the z-axes.

Output Arguments

rotated is rotation of vector by passed axes related angles.

Requirements

- Other m-files required: None
- Subfunctions: rotx, roty, rotz
- MAT-files required: None

See Also

- [rotx](#)
- [roty](#)
- [rotz](#)
- [Wikipedia Drehmatrix](#)

Created on August 03. 2016 by Thorben Schüthe. Copyright Thorben Schüthe 2016.

```
function [rotated] = rotate3DVector(vector, alphaX, betaY, gammaZ)
    arguments
        % validate as vecotor or matrix of size 3 x N
        vector (3,:) double {mustBeReal}
        % validate angles as scalar
        alphaX (1,1) double {mustBeReal}
        betaY (1,1) double {mustBeReal}
        gammaZ (1,1) double {mustBeReal}
    end

    % rotate vector or vector field as 3 x N matrix counterclockwise by given
    % angles along axes, calculate rotation matrices for each axes and
    % multiplicate with input vector
    rotated = rotz(gammaZ) * roty(betaY) * rotx(alphaX) * vector(:, 1:end);
end
```

Published with MATLAB® R2020b

generateDipoleRotationMoments

Generate magnetic moments to perform a full rotation of a magnetic dipole in the z-axes with a certain tilt. The moments covers a rotation from 0° to 360° and are equal distributed between 0° and 360°. 0° and 360° are related to the first moment which is represented by the start vector of

$$\vec{m}_0 = |m_0| \cdot \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

Due to the start vector position the tilt of z-axes must be applied with a tilt angle in y-axes. So the rotated vector of the start moment is described by

$$\vec{m}_i = R_z(\theta_i)R_y(\phi)R_x(0^\circ)\vec{m}_0$$

The returning Moments matrix is 3 x N matrix where each moment vector

$$\vec{M} = [\vec{m}_1 \dots \vec{m}_N]$$

corresponds to an i-th angle in 1 x N thetas vector.

$$\vec{\theta} = [\theta_1 \dots \theta_N]$$

for

$$i = 1 \dots N$$

The resolution of the angles can be modified additionally. At first the full angle vector theta is fully generated with given resolution and downsampled afterwards to the defined number of angles. On the resulting theta vector is base of magnetical moments.

Syntax

```
M = generateDipoleRotationMoments(m0, nTheta)
[M, thetas] = generateDipoleRotationMoments(m0, nTheta)
[M, thetas] = generateDipoleRotationMoments(m0, nTheta, phi)
[M, thetas] = generateDipoleRotationMoments(m0, nTheta, phi, resolution)
[M, thetas, index] = generateDipoleRotationMoments(m0, nTheta, phi, resolution, phaseIndex)
```

Description

M = generateDipoleRotationMoments(m0, nTheta) generate magnetic moments for N numbers of rotation angles theta in 3 x N sized matrix. With a default angle resolution of 1° and a start angle of 0°.

[M, theta] = generateDipoleRotationMoments(m0, nTheta) returns so magnetic moments as before and related angles theta as 1 x N vector.

[M, theta] = generateDipoleRotationMoments(m0, nTheta, phi) generate magnetic moments for a rotation with a tilt angle phi.

[M, theta] = generateDipoleRotationMoments(m0, nTheta, phi, resolution) return moments and angles like described above but with given resolution in degree. The resolution is used in generation of full scale rotation angle base and sometime not visible in the output caused by the number of angles. So which angle are even picked from full scale rotation to compute a down sampled set of angles.

[M, theta, index] = generateDipoleRotationMoments(m0, nTheta, phi, resolution, phaseIndex) returns the moments, the

angles and index reprensentation of down sampled angles in the full scale rotation vector.

Examples

```
% choose a huge moment amplitude to withdraw numeric erros in later H-field
% strength calculations
m0 = 1e6;

% get a full scale (FS) rotation of with 0.5° resolution and no tilt
[MFS, thetaFS] = generateDipolRotationMoments(m0, 0, 0, 0.5);

% get down sampled (DS) rotation with equal distanced angles of the same full
% scale and refered index to the full scale. 8 angles.
[MDS, thetaDS, iFS] = generateDipolRotationMoments(m0, 8, 0, 0.5);

% check distribution to full scale must be true if distribution is correct
all(MFS(iFS) == MDS)
all(thetaFS(iFS) == thetaDS)

% now shift the sample pick by 22 samples (11° with resolution of 0.5°)
[MDSS, thetaDSS] = generateDipolRotationMoments(m0, 8, 0, 0.5, 22);

% check with index shift by 22 in iFS index
all(MFS(iFS + 22) == MDSS)
all(thetaFS(iFS + 22) == thetaDSS)
```

Input Arguments

m0 scalar value of magnetic moment magnitude. Choose huge value to prevent numeric failures in later field strength calculation. 1e6 is a proven value. Later normated in the field calculation process. Can be any real number.

nTheta scalar value and number of angles which are even picked from the full rotation to produce smaller rotation datasets. Must be a positive integer or zero. If zero the full scale rotation is returned.

phi scalar angle in degree to tilt the z-axes of the rotation. Can be any real number. Default is 0°.

resolution scalar angle resolution must be real positive number and probably smaller than 360°. Default is 1°.

phaseIndex scalar integer number to shift the start index of down sampling the full scale rotation. Therfore nTheta must be greater than 0. Default is 0.

Output Arguments

M matrix of magnetic moments related to vector theta. Matrix of size 3 x N.

theta related angles to calculated magnetic moments in a row vector of size 1 x N.

index reference to full scale angle vector. Empty if nTheta is zero and theta is the full scale vector.

Requirements

- Other m-files required: rotate3DVector.m
- Subfunctions: length, downsample, ismember, find
- MAT-files required: None

See Also

- [rotate3DVector](#)

- [downsample](#)
- [ismember](#)
- [find](#)

Created on November 06. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```

function [M, theta, index] = generateDipoleRotationMoments(m0, nTheta, ...
phi, resolution, phaseIndex)
arguments
    % validate amplitude of magnetic moment as real scalar value
    m0 (1,1) double {mustBeReal}
    % validate number of used angulars as positive integer, for 0 return all
    nTheta (1,1) double {mustBeNonnegative, mustBeInteger}
    % validate tilt angle as real value with default 0°
    phi (1,1) double {mustBeReal} = 0
    % validate angle resolution as real positive value
    resolution (1,1) double {mustBePositive} = 1
    % validate downsample phase as positive integer with default 0, no shift
    phaseIndex (1,1) double {mustBeNonnegative, mustBeInteger} = 0
end

% scale full rotation angle vector with given resolution from 0° to 360°
% so run to 360°-resolution because 0° == 360°, its a circle
fullScale = 0:resolution:(360 - resolution);

% if nThetas is greater than 0 downsample to nTheta else use full scale
if nTheta
    % get equal distribute distance of samples in thetas for nThetas
    sampleDistance = length(downsample(fullScale, nTheta));

    % downsample with equal sample distance and passed sample phase to shift
    % first sample in downsample vector from 1 to phaseIndex
    theta = downsample(fullScale, sampleDistance, phaseIndex);

    % find index members of down sampled angles in full scale vector
    members = ismember(fullScale, theta);
    index = find(members);

else
    % 0 is given for number of theta so it returns the full scale rotation
    % no index relations if full scale is returned
    nTheta = length(fullScale);
    theta = fullScale;
    index = [];
end

% create start moment with given magnetic moment amplitude basic moment to
% produce rotate moments
m0 = m0 * [-1; 0; 0];

% allocate memory for the moments Matrix of rotated basic moments by i-th
% theta and fixed tilt of phi and rotate of theta angulars
M = zeros(3, nTheta);
for i = 1:nTheta
    M(:,i) = rotate3DVector(m0, 0, phi, theta(i));
end
end

```

.....

Published with MATLAB® R2020b

generateSensorSquareArrayGrid

Generates a position grid of sensors in x, y and z dimension. So the function returns a grid in shape of a square in which all sensors have even distances to each and another in x and y direction z is constant due to that all sensor are in the same distance to the magnet.

The size of the sensor array is described by its edge length a

$$A = a^2$$

and the distance d of each coordinate to the next point in x and y direction

$$d = \frac{a}{N-1}$$

The coordinates of the array are scale from center of the square. So for the upper left corner position is described by

$$x_{1,1} = -\frac{a}{2} \quad y_{1,1} = \frac{a}{2} \quad z = \text{const.}$$

The coordinates of each dimension are placed in matrices of size N x N related to the number of sensors at one edge of the square Array. So position pattern in x dimension are returned as

$$X_0 = \begin{bmatrix} x_{1,1} & \cdots & x_{1,N} \\ \vdots & \ddots & \vdots \\ x_{N,1} & \cdots & x_{N,N} \end{bmatrix}$$

$$x_{i,j} = x_{1,1} + j \cdot d - d$$

same wise for y dimension but transposed

$$Y_0 = \begin{bmatrix} y_{1,1} & \cdots & y_{1,N} \\ \vdots & \ddots & \vdots \\ y_{N,1} & \cdots & y_{N,N} \end{bmatrix}$$

$$y_{i,j} = y_{1,1} - i \cdot d + d$$

$$Y_0 = -X_0^T$$

and z dimension

$$Z_0 = \begin{bmatrix} z_{1,1} & \cdots & z_{1,N} \\ \vdots & \ddots & \vdots \\ z_{N,1} & \cdots & z_{N,N} \end{bmatrix}$$

$$z_{i,j} = 0$$

for

$$i = 1, 2, \dots, N \quad j = 1, 2, \dots, N$$

A relative position shift can be performed by pass a position vector p with relativ position to center

$$\vec{p} = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix}$$

So that a left shift in x direction relative to the magnet in the center of the coordinate system is done by negative values for p(1) and an up shift in y direction is performed by positive values for p(2). To gain distance in z from center point so the magnet is above the z layer of the sensor array increase the z positive. In addition to the z shift an offset r sphere can be set. The offset represents the radius of a sphere magnet in which center the dipole is placed. The dipole is placed in the center of the coordinate system and sensor array position is relative to the dipole or center. So shifts are described by

$$X = X_0 + x_p \quad Y = Y_0 + y_p \quad Z = Z_0 - (z_p + r_{sp})$$

Syntax

```
[X, Y, Z] = generateSensorArrayGrid(N, a, p, r)
```

Description

[X, Y, Z] = generateSensorArrayGrid(N, a, p, r) returns a sensor array grid of size N x N with grid position matrices for x, y and z positions of each sensor in the array.

Examples

```
% generate a grid of 8 x 8 sensors with no shift in x or y direction
and a static position of 4mm under the center in z dimension with a
z offset of 2mm so (2 + 2)mm
N = 8;
p = [0, 0, 2]
r = 2;
[X, Y, Z] = generateSensorArrayGrid(N, a, p, r);

% same layer but left shift by 2mm and down shift in y by 1mm
p = [-2, 1, 2]
r = 2;
[X, Y, Z] = generateSensorArrayGrid(N, a, p, r);
```

Input Arguments

N positive integer scalar number of sensors at one edge of the square grid. So the resulting grid has dimensions N x N.

a positive real scalar value of sensor array edge length.

p relative position vector, relative sensor array position to center of the array. Place the array in 3D coorodinate system relative to the center of system.

r positive real scalar is offset in z dimension and represents the sphere radius in which center the magnetic dipole is placed.

Output Arguments

X x coordinates for each sensor in N x N matrix where each point has the same orientation as in y and z dimension.

Y y coordinates for each sensor in N x N matrix where each point has the same orientation as in x and z dimension.

Z z coordinates for each sensor in N x N matrix where each point has the same orientation as in x and y dimension.

Requirements

- Other m-files required: None
- Subfunctions: meshgrid
- MAT-files required: None

See Also

- [meshgrid](#)

Created on November 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function [X, Y, Z] = generateSensorArraySquareGrid(N, a, p, r)
    arguments
        % validate N as positive integer
        N (1,1) double {mustBePositive, mustBeInteger}
        % validate array edge length as positive scalar
        a (1,1) double {mustBeReal, mustBePositive}
        % validate p as column vector of real scalars
        p (3,1) double {mustBeReal, mustBeVector}
        % validate r as real scalar
        r (1,1) double {mustBeReal}
    end

    % half edge length for square corners
    aHalf = a / 2;

    % distance in x and y direction of each coordinate to next point
    d = a / (N - 1);

    % grid vector for x and y coordinates z is constant layer with shifts
    x = (-aHalf:d:aHalf) + p(1);
    y = (aHalf:-d:-aHalf) + p(2);
    z = -(p(3) + r);

    % scale grid in x, y dimension with constant z dimension
    [X, Y, Z] = meshgrid(x, y, z);
end
```

Published with MATLAB® R2020b

computeDipoleH0Norm

Compute the norm factor for magnetic field generated by an Dipole in its zero position. That means the maximum H-field magnitude in zero position with no position shifts in x or y direction. So that norm factor is related to the center point of the coordinate system in x and y direction and to the dipoles initial z position. Which can be seen as sphere magnet for far field of the sphere. The norm relates that a dipole magnet in center of a sphere with a radius has certain field strength in related distance. For example a sphere of 2 mm radius has in 5 mm distance a field strength of 200 kA/m

It is simplified computation for the dipole equation for one position in initial state without tilt in z-axes to bring on a free chosen field strength to define the magnet. Because far field of sphere can be seen as dipole.

$$\vec{H}_0(\vec{r}_0) = \frac{1}{4\pi} \cdot \left(\frac{3\vec{r}_0 \left(\vec{m}_0 \cdot \vec{r}_0 \right)}{|\vec{r}_0|^5} - \frac{\vec{m}_0}{|\vec{r}_0|^3} \right)$$

$$H_{0norm} = \frac{H_{mag}}{|H_0(r_0)|}$$

Syntax

```
H0norm = functionName(Hmag, m0, r0)
```

Description

H0norm = functionName(Hmag, m0, r0) computes scalar norm factor related to dipole rest position. Multiply that factor to dipole generated fields which are computed with the same magnetic moment magnitude to imprint a chosen magnetic field strength magnitude on the dipole field rotation.

Examples

```
% distance where the magnetic field strength is the value of wished  
% magnitude, in mm  
r0 = [0; 0; -5]  
% field strength to imprint in norm factor in kA/m  
Hmag = 200  
% magnetic moment magnitude which is used generate rotation moments  
m0 = [-1e6; 0; 0]  
% compute norm factor for dipole rest position  
H0norm = computeDipoleH0Norm(Hmag, m0, r0)
```

Input Arguments

Hmag real scalar of H-field strength magnitude to imprint in norm factor to define a dipole sphere with constant radius and field strength at this radius.

m0 vector of magnetic moment magnitude which must be same as for later rotation of the dipole.

r0 vector of distance in rest position of magnet center.

Output Arguments

H0norm real scalar of norm factor which relates to the zero position of the dipole sphere and can be multiplied to generated dipole H-field to imprint a magnetic field strength relative to the position of sensor array. The imprinted field strength magnitude relates to the rest position z0 + rsp.

Requirements

- Other m-files required: None

- Subfunctions: None
- MAT-files required: None

See Also

- [rotate3DVector](#)
- [generateDipoleRotationMoments](#)
- [Wikipedia Magnetic Dipole](#)

Created on November 11. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function [H0norm] = computeDipoleH0Norm(Hmag, m0, r0)
    arguments
        % validate inputs as real scalars
        Hmag (1,1) double {mustBeReal}
        m0 (3,1) double {mustBeReal, mustBeVector}
        r0 (3,1) double {mustBeReal, mustBeVector}
    end

    % calculate the magnitude of all positions
    r0abs = sqrt(sum(r0.^2, 1));

    % calculate the the unit vector of all positions
    r0hat = r0 ./ r0abs;

    % calculate field strength and magnitude at position
    H0 = (3 * r0hat .* (m0' * r0hat) - m0) ./ (4 * pi * r0abs.^3);
    H0abs = sqrt(sum(H0.^2, 1));

    % compute the norm factor like described in the equations
    H0norm = Hmag / H0abs;
end
```

Published with MATLAB® R2020b

computeDipoleHField

Computes the magnetic field strength H of a dipole magnet dependent of position and magnetic moment and imprint a field strength magnitude on the resulting field by passing a norm factor which relates to the rest position of the dipole magnet. The resulting field strength has field components in x, y and z direction.

The magnetic dipole moment w must be a column vector or shape

$$\vec{m} = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix}$$

so that the magnetic moment corresponds to a position vector

$$\vec{r} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

with coordinates for x, y and z in 3D coordinate system which can be taken part of its unit vector and its magnitude.

$$\hat{r} = \hat{r} \cdot |\vec{r}|$$

It computes the field strength at this position with the current magnetic moment for field components in the same orientation.

$$\vec{H}(\vec{r}) = \begin{bmatrix} H_x \\ H_y \\ H_z \end{bmatrix}$$

The originally equation of the magnetic dipole is known as

$$\vec{H}(\vec{r}) = \frac{\vec{B}(\vec{r})}{\mu_0}$$

$$\vec{H}(\vec{r}) = \frac{1}{4\pi} \cdot \frac{3\vec{r} \cdot (\vec{m}^T \cdot \vec{r}) - \vec{m}|\vec{r}|^2}{|\vec{r}|^5}$$

which can be simplified by putting in the unit vector of the position in into the equation.

$$\vec{H}(\vec{r}) = \frac{1}{4\pi|\vec{r}|^3} \cdot (3\hat{r} \cdot (\vec{m}^T \cdot \hat{r}) - \vec{m})$$

To imprint a certain field strength related to a rest position of the dipole the resulting field strength is multiplied with a norming factor. The factor must be computed with same magnitude of the magnetic dipole moments which is passed to this computation to get correct field strengths. To get fields without imprinting set the norming factor to 1.

$$\vec{H}(\vec{r}) \cdot H_{0norm}$$

Syntax

`H = computeDipoleHField(x, y, z, m, H0norm)`

Description

H = computeDipoleHField(x, y, z, m, H0norm)computes dipole field strength at passed position (x,y,z) with the magnetic dipole moment m. The resulting field strength is a vector with components in x, y and z direction. A field strength norming is imprinted on a rest position computation and multiplied on the result by multiplying a norm factor to the field. The normfactor must be relate to the same magnitude of the magnetic dipole moment which is used here and corresponds to the magnets rest position in defined distance of the magnets surface.

Examples

```
% compute a single point without norming
H = computeDipoleHField(1, 2, 3, [1; 0; 0], 1)

% compute a 3D grid of positions
x = linspace(-10, 10, 40);
y = linspace(10, -10, 40);
z = linspace(10, -10, 40);
[X, Y, Z] = meshgrid(x, y, z);

% allocate memory for field components in x,y,z
Hx = zeros(40, 40, 40);
Hy = zeros(40, 40, 40);
Hz = zeros(40, 40, 40);

% compute without norming for each z layer and reshape results into layer
% magnetic moments points in -x direction which implies north and south pole
% is in x direction and rotation axes in z
for i=1:40
    H = computeDipoleHField(X(:,:,:i),Y(:,:,:i),Z(:,:,:i),[-1;0;0],1);
    Hx(:,:,:i) = reshape(H(1,:),40,40);
    Hy(:,:,:i) = reshape(H(2,:),40,40);
    Hz(:,:,:i) = reshape(H(3,:),40,40);
end

% calculate magnitude in each point for better view the results
Habs = sqrt(Hx.^2+Hy.^2+Hz.^2);

% define a index to view only every 4th point for not overcrowded plot
idx = 1:4:40;

% downsample and norm
Xds = X(idx, idx, idx);
Yds = Y(idx, idx, idx);
Zds = Z(idx, idx, idx);
Hxds = Hx(idx, idx, idx) ./ Habs(idx, idx, idx);
Hyds = Hy(idx, idx, idx) ./ Habs(idx, idx, idx);
Hzds = Hz(idx, idx, idx) ./ Habs(idx, idx, idx);

% show results
quiver3(Xds, Yds, Zds, Hxds, Hyds, Hzds);
axis equal;
```

Input Arguments

x coordinates of positions at the field strength is calculated can be scalar, vector or matrix of coordinates. Must be same size as y and z.

y coordinates of positions at the field strength is calculated can be scalar, vector or matrix of coordinates. Must be same size as x and z.

z coordinates of positions at the field strength is calculated can be scalar, vector or matrix of coordinates. Must be same size as x and y.

m magnetic dipole moment as 3 x 1 vector. The magnetic field strength is calculated with the same moment for all passed positions.

H0norm scalar factor to imprint a field strength to the dipole field. Must be computed with the same magnitude of passed magnetic moment vector. Set 1 to disable imprinting.

Output Arguments

H computed magnetic field strength at passed positions with related magnetic moment. If passed position is a scalar H has size of 3 X 1 with its components in x, y and z direction. H(1) -> x, H(2) -> y and H(3) -> z. If passed positions are not scalar H has size of 3 x numel(x) with position relations in columns. So reshape rows to shapes of positions to keep orientation as origin.

Requirements

- Other m-files required: None
- Subfunctions: mustBeEqualSize
- MAT-files required: None

See Also

- [generateDipoleRotationMoments](#)
- [generateSensorArraySquareGrid](#)
- [computeDipoleH0Norm](#)

Created on June 11. 2019 by Thorben Schüthe. Copyright Thorben Schüthe 2019.

```
function [H] = computeDipoleHField(x, y, z, m, H0norm)
    arguments
        % validate position, can be any size but must be same size of
        x (:,:,:,:) double {mustBeReal}
        y (:,:,:,:) double {mustBeReal, mustBeEqualSize(x, y)}
        z (:,:,:,:) double {mustBeNumeric, mustBeReal, mustBeEqualSize(y, z)}
        % validate magnetic moment as 3 x 1 vector
        m (3,1) double {mustBeReal, mustBeVector}
        % validate norm factor as scalar
        H0norm (1,1) double {mustBeReal}
    end

    % unify positions to column vector or matrix of column vectors if positions
    % are not passed as column vectors or scalar, resulting size of position R
    % is 3 x length(X), a indication if is column vector is not needed because
    % x(:) is returning all content as column vector. Transpose to match shape.
    r = [x(:), y(:), z(:)]';

    % calculate the magnitude of all positions
    rabs = sqrt(sum(r.^2, 1));

    % calculate the the unit vector of all positions
    rhat = r ./ rabs;

    % calculate H-field of current magnetic moment for all passed positions
    % calculate constants in equation once in the first bracket term, all vector
    % products in the second term and finally divide by related magnitude ^3
    H = (H0norm / 4 / pi) * (3 * rhat .* (m' * rhat) - m) ./ rabs.^3;
end

% Custom validation function
function mustBeEqualSize(a,b)
    % Test for equal size
```

```
if ~isequal(size(a),size(b))
    eid = 'Size:notEqual';
    msg = 'X Y Z positions must be the same size and orientation.';
    throwAsCaller(MException(eid,msg))
end
end
```

Published with MATLAB® R2020b

simulateDipoleSquareSensorArray

Simulate a sensor array of square shape with dipole magnet as stimulus. Needs options loaded from config file or generated from config generation script. Characterization data must be loaded before and served as CharData struct. Loops through positions saves a data set for every supported position of UseOptions which is called TrainingOptions or TestOptions in config.

Syntax

```
simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, UseOptions, CharData)
```

Description

simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ... SensorArrayOptions, DipoleOptions, UseOptions, CharData) saves simulation datasets to data path specified in PathVariables and UseOptions.

Examples

```
% load config from mat-file
load('config.mat', 'GeneralOptions', 'PathVariables', 'SensorArrayOptions',
    'DipoleOptions', 'TrainingOptions', 'TestOptions');

% load characterization dataset
TDK = load(PathVariables.tdkDatasetPath);

% generate training dataset(s)
simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, TrainingOptions, TDK)

% generate test dataset(s)
simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, TestOptions, TDK)
```

Input Arguments

GeneralOptions struct of general options generated by config script, includes date format and so on.

PathVariables struct of project path generated by config script, includes data path for save and load data.

SensorArrayOptions struct of sensor array shape and behavior generated by config script.

DipoleOptions struct of dipole specification, defines magnet and stimulus, generated by config script.

UseOptions struct of implementation of use case, defines which kind of dataset will be generated. At current state test and training dataset are available options in config. In config generated structs are TestOptions and TrainingOptions.

CharData struct of characterization data. Therefore load characterization dataset as shown in examples into a struct.

Output Arguments

None

Requirements

- Other m-files required: computeDipoleH0Norm.m, computeDipoleHField.m, generateDipoleRotationMoments.m, generateSensorArraySquareGrid.m, rotate3DVector.m
- Subfunctions: reshape, interp2, sum
- MAT-files required: config.mat, TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat

See Also

- [computeDipoleH0Norm](#)
- [computeDipoleHField](#)
- [generateDipoleRotationMoments](#)
- [generateSensorArraySquareGrid](#)
- [rotate3DVector](#)

Created on June 11. 2019 by Thorben Schüthe. Copyright Thorben Schüthe 2019.

```
function simulateDipoleSquareSensorArray(GeneralOptions, PathVariables, ...
    SensorArrayOptions, DipoleOptions, UseOptions, CharData)
arguments
    % validate inputs as struct, structs generated in config.mat
    GeneralOptions struct {mustBeA(GeneralOptions, 'struct')}
    PathVariables struct {mustBeA(PathVariables, 'struct')}
    SensorArrayOptions struct {mustBeA(SensorArrayOptions, 'struct')}
    DipoleOptions struct {mustBeA(DipoleOptions, 'struct')}
    UseOptions struct {mustBeA(UseOptions, 'struct')}
    CharData struct {mustBeA(CharData, 'struct')}
end

% try to load relavant values in local variable space for better
% handling and short names second check if struct fields are reachable
try
    % general options needed to create filenames etc.
    dfStr = GeneralOptions.dateFormat;

    % number of sensors at edge of square array, dimension N x N
    N = SensorArrayOptions.dimension;
    % sensor array edge length, square edge a
    a = SensorArrayOptions.edge;
    % sensor array supply voltage used to generate bridge outputs from
    % characterization data in combination with bridge offset voltage
    % characterization data should be in mV/V so check norm factor
    Vcc = SensorArrayOptions.Vcc;
    Voff = SensorArrayOptions.Voff;
    Vnorm = SensorArrayOptions.Vnorm;
    switch CharData.Info.Units.SensorOutputVoltage
        case 'mV/V'
            if Vnorm ~= 1e3
                error('Wrong norming mV/V: %e', Vnorm);
            end
        otherwise
            error('Unknown norm voltage: %s', ...
                  CharData.Info.Units.SensorOutputVoltage)
    end

    % sphere radius for dipole approximation of spherical magnet
    rsp = DipoleOptions.sphereRadius;
    % H-field magnitude to imprint in certain distance from magnet
    % surface which sphere radius rsp plus distance z0
    H0mag = DipoleOptions.H0mag;
    % distance from magnet surface where to imprint the H0mag
    z0 = DipoleOptions.z0;
    % magnetic dipole moment magnitude which define origin moment of the
    % magnet in rest position
    M0mag = DipoleOptions.M0mag;

    % dataset type or use case in which later it is use in application
```

```

useCase = UseOptions.useCase;
% destination path and filename to save generated data sets with
% timestamps in filename, place timestamps with sprintf
switch useCase
    case 'Training'
        fPath = PathVariables.trainingDataPath;
        fNameFmt = 'Training_%s.mat';
    case 'Test'
        fPath = PathVariables.testDataPath;
        fNameFmt = 'Test_%s.mat';
    otherwise
        error('Unknown use case: %s', UseOptions.useCase);
end
% x, y and z positions in which pairing the datasets are generated
% position vectors are run through in all combinations with tilt
% and number of angles
xPos = UseOptions.xPos;
yPos = UseOptions.yPos;
zPos = UseOptions.zPos;
tilt = UseOptions.tilt;
nAngles = UseOptions.nAngles;
% constants for generated use case, angle resolution for generated
% rotation angles, phase index for a phase shift in generation of
% rotation angles
angleRes = UseOptions.angleRes;
phaseIndex = UseOptions.phaseIndex;
% which characterization reference should be load from CharData
% sensor output bridge fields (cos/sin)
refImage = UseOptions.BridgeReference;

% load values from characterization dataset
% scales of driven Hx and Hy amplitudes in characterization
% stimulus in kA/m
if ~strcmp(CharData.Info.Units.MagneticFieldStrength, 'kA/m')
    error('Wrong H-field unit: %s', ...
          CharData.Info.Units.MagneticFieldStrength);
end
HxScale = CharData.Data.MagneticField.hx;
HyScale = CharData.Data.MagneticField.hy;
% cosinus and sinus characterization images for corresponding field
% amplitudes, load and norm to Vcc and Voff, references of
% simulation, adjust reference to bridge gain for output volgates
gain = CharData.Info.SensorOutput.BridgeGain;
VcosRef = CharData.Data.SensorOutput.CosinusBridge.(refImage) ...
    .* (gain * Vcc / Vnorm) + Voff;
VsinfRef = CharData.Data.SensorOutput.SinusBridge.(refImage) ...
    .* (gain * Vcc / Vnorm) + Voff;
catch ME
    rethrow(ME)
end

% now everything is successfully loaded, execute further constants
% which are needs to be generated once for all following operations
% meshgrids for refernce images to query bridge reference with interp2
[HxScaleGrid, HyScaleGrid] = meshgrid(HxScale, HyScale);
% allocate memory for results of on setup run, speed up compute by 10
% fix allocations which are not changing by verring parameters like
% number of angles or positon, for all parameter depended memory size
% allocalte matlab automatically by function call or need reallocation
% in for loops
% H-field components for each rotation step
Hx = zeros(N, N, nAngles);

```

```

Hy = zeros(N, N, nAngles);
Hz = zeros(N, N, nAngles);
% H-field abs for each rotation step
Habs = zeros(N, N, nAngles);
% Bridge output voltages for each sensor in grid, H-fields, sensor
% grid, voltages all same orientation
Vcos = zeros(N, N, nAngles);
Vsint = zeros(N, N, nAngles);

% compute values which not changing by loop parameters
% magnetic dipole moments for each rotation step
% rotation angles to compute
% index corresponding to full scale rotation with angleRes
[m, angles, angleRefIndex] = generateDipoleRotationMoments(M0mag, ...
    nAngles, tilt, angleRes, phaseIndex);

% rotation angle step width on full rotation 360° with subset of angles
if length(angles) > 1
    angleStep = angles(2) - angles(1);
else
    angleStep = 0;
end

% compute dipole rest position norm to imprint a certain field
% strength magnitude with respect of tilt in y axes and magnetization
% in x direction as in generate Dipole rotation moments
r0 = rotate3DVector([0; 0; -(z0 + rsp)], 0, tilt, 0);
m0 = rotate3DVector([-M0mag; 0; 0], 0, tilt, 0);
H0norm = computeDipoleH0Norm(H0mag, m0, r0);

% prepare file header Info struct, overwrite certain fields in loop like x,
% y, z positions
Info = struct;
Info.SensorArrayOptions = SensorArrayOptions;
Info.SensorArrayOptions.SensorCount = N^2;
Info.DipoleOptions = DipoleOptions;
Info.UseOptions = UseOptions;
Info.CharData = join( ...
    [CharData.Info.SensorManufacturer, CharData.Info.Sensor]);
Info.Units.SensorOutputVoltage = 'V';
Info.Units.MagneticFieldStrength = 'kA/m';
Info.Units.Angles = 'degree';
Info.Units.Length = 'mm';

% collect relevant to Data struct for save to file
% header Info struct, overwrite position depended fields in loop before save
Data = struct;
Data.HxScale = HxScale;
Data.HyScale = HyScale;
Data.VcosRef = VcosRef;
Data.VsinRef = VsintRef;
Data.Gain = gain;
Data.r0 = r0;
Data.m0 = m0;
Data.H0norm = H0norm;
Data.m = m;
Data.angles = angles;
Data.angleStep = angleStep;
Data.angleRefIndex = angleRefIndex;

% generate dataset for all use case setup pairs in for loop and append

```

```
% generated dataset path to path struct for result
% outer to inner loop is positions to angles
% generate z layer wise
for z = zPos
    for x = xPos
        for y = yPos
            % generate sensor array grid according to current position
            % current position vector of sensor array relative to
            % magnet surface
            p = [x; y; z];
            % write current position in file header
            Info.UseOptions.xPos = x;
            Info.UseOptions.yPos = y;
            Info.UseOptions.zPos = z;
            % sensor array grid coordinates
            [X, Y, Z] = generateSensorArraySquareGrid(N, a, p, rsp);
            % save current sensor gird to Data struct
            Data.X = X;
            Data.Y = Y;
            Data.Z = Z;
            for i = 1:nAngles
                % calculate H-field of one rotation step for all
                % positions, the field is normed to zero position
                H = computeDipoleHField(X, Y, Z, m(:,i), H0norm);
                % separate parts or field in axes direction/ components
                Hx(:,:,:i) = reshape(H(1,:), N, N);
                Hy(:,:,:i) = reshape(H(2,:), N, N);
                Hz(:,:,:i) = reshape(H(3,:), N, N);
                Habs(:,:,:i) = reshape(sqrt(sum(H.^2, 1)), N, N);
                % get bridge outputs from references by cross pick
                % references from grid, the Hx and Hy queries can be
                % served as matrix as long they have same size and
                % orientation the nearest neighbor interpolation
                % returns of same size and related to orientation, for
                % outlayers return NaN, do this for every angle
                Vcos(:,:,:i) = interp2(HxScaleGrid, HyScaleGrid, VcosRef, ...
                    Hx(:,:,:i), Hy(:,:,:i), 'nearest', NaN);
                Vsins(:,:,:i) = interp2(HxScaleGrid, HyScaleGrid, VsinsRef, ...
                    Hx(:,:,:i), Hy(:,:,:i), 'nearest', NaN);
            end % angles
            % save rotation results to Data struct
            Data.Hx = Hx;
            Data.Hy = Hy;
            Data.Hz = Hz;
            Data.Habs = Habs;
            Data.Vcos = Vcos;
            Data.Vsin = Vsins;
            % save results to file
            fName = sprintf(fNameFmt, datestr(now, dfStr));
            Info.filePath = fullfile(fPath, fName);
            disp(Info.filePath)
            save(Info.filePath, 'Info', 'Data', '-v7.3', '-nocompression');
        end % y
    end % x
end % z
end
```

gaussianProcessRegression

Function module which implements regression models with Gaussian Process. Implemented regression models posses the ability to process training and test datasets by sensor array simulation. The model creation can be bind into scripts by use of initGPR and tuneKernel for simple optimized models. A fully generalized regression model is supported by use of optimGPR to create models which are tuned on training data and generalized on test data.

basicMathFunctions

Submodule which contains basic math function to module functionality.

kernelQFC

Submodule which contain quadratic fractional covariance implementation.

kernelQFCAPX

Submodule which contains approximated quadratic fractional covariance implementation.

initGPR

Initializes regression model by training dataset and config dataset. Resulting model is not optimized.

initGPROptions

Attaches configuration to regression model including default parameters and bounds.

initTrainDS

Initiates the training data, reference angles and regression targets on regression model.

initKernel

Initiates kernel submodules by made configuration.

initKernelParameters

Initiates the regression model by its set configuration done initiating steps before.

tuneKernel

Tunes initiated regression model hyperparameters.

computeTuneCriteria

Computes min criteria for tuneKernel.

predFrame

Predicts single test data frame.

predDS

Predicts a whole test dataset at once.

lossDS

Computes prediction losses and errors of a test dataset at once.

optimGPR

Computes optimized regression model.

computeOptimCriteria

Computes min criteria for optimGPR.

See Also

- [generateConfigMat](#)
- [demoGPRModule](#)
- [investigateKernelParameters.html](#)
- [generateSimulationDatasets.html](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Published with MATLAB® R2020b

initGPR

Initializes GPR model by passed trainings dataset and GPR options struct.

Syntax

```
Mdl = initGPR(TrainDS, GPROptions)
```

Description

Mdl = initGPR(TrainDS, GPROptions) sequential initializing.

Examples

```
load config.mat PathVariables GPROptions;
TrainFiles = dir(fullfile(PathVariables.trainingDataPath, 'Training*.mat'));
TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
assert(~isempty(TrainFiles), 'No training datasets found.');
assert(~isempty(TestFiles), 'No test datasets found.');
try
    TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1).name));
    TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).name));
catch ME
    rethrow(ME)
end
Mdl = initGPR(TrainDS, GPROptions);
[fang, frad, fcov, fsin, s, ciang, cirad] = predDS(Mdl, TestDS)
```

Input Arguments

TrainDS loaded training data by infront processesed sensor array simulation.

GPROptions loaded parameter group from config.mat. Struct with options.

Output Arguments

Mdl bare initialized model struct with no further optimization.

Requirements

- Other m-files required: None
- Subfunctions: initGPROptions, initTrainDS, initKernel, initKernelParameters
- MAT-files required: config.mat, Train_*.mat

See Also

- [initGPROptions](#)
- [initTrainDS](#)
- [initKernel](#)
- [initKernelParameters](#)

Created on February 20. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initGPR(TrainDS, GPROptions)

    % create model struct
    Mdl = struct();
```

```
% init GPROptions on model struct
Mdl = initGPROptions(Mdl, GPROptions);

% init training data on model
Mdl = initTrainDS(Mdl, TrainDS);

% init kernel, covariance function, mean function and input transformation
% function if needed, initGPROptions and initTrainDS must run before
% initKernel otherwise missing parameters causing an error
Mdl = initKernel(Mdl);

% init model kernel with current hyperparameters, kernel must be initiated
% before otherwise nonsens and errors
Mdl = initKernelParameters(Mdl);

end
```

Published with MATLAB® R2020b

initGPROptions

Initiates GPR options struct from config on GPR model and sets defaults if expected options are not available.

Syntax

```
Mdl = initGPROptions(Mdl, GPROptions)
```

Description

Mdl = initGPROptions(Mdl, GPROptions) initiates default configuration on model struct.

Input Arguments

Mdl model struct.

GPROptions options struct.

Output Arguments

Mdl model struct with attached configuration.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [initGPR](#)
- [generateConfigMat](#)

Created on February 20, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initGPROptions(Mdl, GPROptions)

    % set kernel function option
    if isfield(GPROptions, 'kernel')
        Mdl.kernel = GPROptions.kernel;
    else
        Mdl.kernel = 'QFC';
    end

    % attach hyperparameters to model and bounds for tuning and model
    % optimization
    % theta covariance function parameter theta = [s2f, s1]
    if isfield(GPROptions, 'theta')
        Mdl.theta = GPROptions.theta;
    else
        Mdl.theta = [1, 1];
    end

    % lower and upper bound for tuning theta
    if isfield(GPROptions, 's2fBounds')
        Mdl.s2fBounds = GPROptions.s2fBounds;
    else
        Mdl.s2fBounds = [1e-2, 1e2];
```

```
end
if isfield(GPROptions, 's1Bounds')
    Mdl.s1Bounds = GPROptions.s1Bounds;
else
    Mdl.s1Bounds = [1e-2, 1e2];
end

% noise variance s2n to predict noisy observations
if isfield(GPROptions, 's2n')
    Mdl.s2n = GPROptions.s2n;
else
    Mdl.s2n = 1e-5;
end

% lower and upper bounds for optimizing s2n
if isfield(GPROptions, 's2nBounds')
    Mdl.s2nBounds = GPROptions.s2nBounds;
else
    Mdl.s2nBounds = [1e-4, 10];
end

% enable disable mean function and correction
if isfield(GPROptions, 'mean')
    Mdl.mean = GPROptions.mean;
else
    Mdl.mean = 'zero';
end

% set polynom degree to model, default is 1 for linear correction
if isfield(GPROptions, 'polyDegree')
    Mdl.polyDegree = GPROptions.polyDegree;

    % limit poly degree, because higher polynomials as degree 7 causes
    % an error in cholesky decomposition
    if Mdl.polyDegree > 5
        Mdl.polyDegree = 5;
    end
else
    Mdl.polyDegree = 1;
end

end
```

Published with MATLAB® R2020b

initTrainDS

Initiates needed data from training dataset to GPR model struct. Builds GPR target vectors depending on which sensor type was used to process the training dataset.

Syntax

```
Mdl = initTrainDS(Mdl, TrainDS)
```

Description

Mdl = initTrainDS(Mdl, TrainDS) attaches regression relevant data information to model struct and initiates the training data with references and regression targets.

Input Arguments

Mdl model struct.

TrainDS training data struct which includes Info and Data struct.

Output Arguments

Mdl with attached dataset information, raw training data, reference angles and regression targets for cosine and sine predictions.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: Train_*.mat

See Also

- [initGPR](#)
- [Training and Test Datasets](#)

Created on February 20, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initTrainDS(Mdl, TrainDS)

    % set model parameters from training dataset and training data dependencies
    % N number of angles and references in degree
    Mdl.N = TrainDS.Info.UseOptions.nAngles;
    Mdl.Angles = TrainDS.Data.angles';

    % D sensor array square dimension of DxD sensor array
    Mdl.D = TrainDS.Info.SensorArrayOptions.dimension;

    % P number of predictors in sensor array
    Mdl.P = TrainDS.Info.SensorArrayOptions.SensorCount;

    % get sensor type from dataset
    Mdl.Sensor = TrainDS.Info.UseOptions.BaseReference;

    % choose period factor depending on sensor type
    % how many sinoid periods are abstract on a full rotation by 360°
    switch Mdl.Sensor
        case 'TDK'
            Mdl.PF = 1;
```

```
case 'KMZ60'
    Mdl.PF = 2;

otherwise
    error('Unkown Sensor %s.', Mdl.Sensor);
end

% get reference angles in degree and transpose to column vector
% get sinoid target vectors depending period factor,
% transpose because angles2sinoids works with row vectors
[Mdl.Ysin, Mdl.Ycos] = angles2sinoids(Mdl.Angles, ...
    false, Mdl.PF);

% attach training data fro cosine and sine to model
Mdl.Xcos = TrainDS.Data.Vcos;
Mdl.Xsin = TrainDS.Data.Vsin;
end
```

Published with MATLAB® R2020b

initKernel

Initiates kernel and chooses kernel implementation by initiated GPR options.

Syntax

```
Mdl = initKernel(Mdl)
```

Description

Mdl = **initKernel**(**Mdl**) loads kernel submodule by passed identifier.

Input Arguments

Mdl model struct.

Output Arguments

Mdl with attached kernel functionality.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [initGPR](#)
- [kernelQFC](#)
- [kernelQFCAPX](#)

Created on February 20. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initKernel(Mdl)

    % set covariance function and input function to respect covariance function
    % serve with right format of data, sets mean function which belongs to the
    % corresponding kernel model
    switch Mdl.kernel
        case 'QFC'
            Mdl = initQFC(Mdl);

        case 'QFCAPX'
            Mdl = initQFCAPX(Mdl);

        % end kernel select
        otherwise
            error('Unknown kernel function .', Mdl.kernel);
    end
end
```

initKernelParameters

Init GPR model on current kernel parameters, computes covariance matrix and depending kernel values means, mean coefficients, regression weights and likelihoods.

Syntax

```
Mdl = initKernelParameters(Mdl)
```

Description

Mdl = initKernelParameters(Mdl) initializes the regression model in final.

Input Arguments

Mdl model struct.

Output Arguments

Mdl initialized regression model.

Requirements

- Other m-files required: basicMathFunctions, kernelQFC, kernelQFCAPX
- Subfunctions: None
- MAT-files required: None

See Also

- [basicMathFunctions](#)
- [kernelQFC](#)
- [kernelQFCAPX](#)
- [initGPR](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function Mdl = initKernelParameters(Mdl)

    % compute noise free covariance matrix
    Mdl.Ky = Mdl.kernelFun(Mdl.Xcos, Mdl.Xcos, Mdl.Xsin, Mdl.Xsin, Mdl.theta);

    % add noise to covariance matrix along its diagonal, which is noise on
    % training observations with itself
    Mdl.Ky = addNoise2Covariance(Mdl.Ky, Mdl.s2n);

    % compute the cholesky decomposition of the covariance matrix and the log
    % determinate of the covariance matrix, computes lower triangle matrix
    [Mdl.L, Mdl.logDet] = decomposeChol(Mdl.Ky);

    % compute beta coefficients to fit H matrices of cosine and sine mean
    % function if none zero mean is set as model mean, for zero mean all beta
    % and related means are zero.
    switch Mdl.mean
        case 'zero'
            % set not needed kernel parameters to zero,
            % beta is not used in zero mean GPR and so all related means are
            % zero, as name lets expect
```

```
Mdl.BetaCos = 0;
Mdl.BetaSin = 0;
Mdl.meanFunCos = @(X) 0;
Mdl.meanFunSin = @(X) 0;

case 'poly'
    % estimate beta for none zero H matrices
    Mdl.BetaCos = estimateBeta(Mdl.basisFun(Mdl.Xcos), Mdl.L, Mdl.Ycos);
    Mdl.BetaSin = estimateBeta(Mdl.basisFun(Mdl.Xsin), Mdl.L, Mdl.Ysin);

    % mean function for polynom approximated mean H' * beta
    Mdl.meanFunCos = @(X) Mdl.basisFun(X)' * Mdl.BetaCos;
    Mdl.meanFunSin = @(X) Mdl.basisFun(X)' * Mdl.BetaSin;

otherwise
    error('Unsupported mean function %s in beta estimation.', Mdl.mean);
end

% compute weights for cosine and sine, angles in rads and radius
Mdl.AlphaCos = computeAlphaWeights(Mdl.L, Mdl.Ycos, ...
    Mdl.meanFunCos(Mdl.Xcos));
Mdl.AlphaSin = computeAlphaWeights(Mdl.L, Mdl.Ysin, ...
    Mdl.meanFunSin(Mdl.Xsin));

% compute log marginal likelihoods for each cosine and sine weights
Mdl.IMLCos = computeLogLikelihood(Mdl.Ycos, Mdl.meanFunCos(Mdl.Xcos), ...
    Mdl.AlphaCos, Mdl.logDet, Mdl.N);
Mdl.IMLSin = computeLogLikelihood(Mdl.Ysin, Mdl.meanFunSin(Mdl.Xsin), ...
    Mdl.AlphaSin, Mdl.logDet, Mdl.N);
end
```

Published with MATLAB® R2020b

tuneKernel

Tunes kernel hyperparameters of GPR model. Dismiss tuning for each kernel parameter by setting corresponding bounds to equal values.

Syntax

```
Mdl = tuneKernel(Mdl, verbose)
```

Description

Mdl = tuneKernel(Mdl, verbose) solves the negative marginal logarithmic likelihood criteria with fmincon solver.

Input Arguments

Mdl model struct.

verbose activates prompt for true or 1. Vice versa for false or 0.

Output Arguments

Mdl optimized hyperparameters and resulting regression model.

Requirements

- Other m-files required: None
- Subfunctions: fmincon, optimoptions, computeTuneCriteria, initKernelParameters
- MAT-files required: None

See Also

- [fmincon](#)
- [optimoptions](#)
- [computeTuneCriteria](#)
- [initKernelParameters](#)

Created on March 03, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = tuneKernel(Mdl, verbose)

    % define options for minimum search
    options = optimoptions('fmincon', 'Display', 'off', 'Algorithm', 'sqp', ...
        'PlotFcn', {@optimplotx, @optimplotfval});

    % display tuning
    if verbose, options.Display = 'iter'; end

    % setup problem for minimum solver use problem structure to feed fmincon
    problem.solver = 'fmincon';
    problem.options = options;

    % apply bounds to prevent overfitting
    problem.lb = [Mdl.s2fBounds(1) Mdl.s1Bounds(1)];
    problem.ub = [Mdl.s2fBounds(2) Mdl.s1Bounds(2)];

    % set sl start value
    problem.x0 = Mdl.theta;
```

```
% apply objective function and start values
problem.objective = @(x) computeTuneCriteria(x, Mdl);

% solve problem
[Mdl.theta] = fmincon(problem);

% reinit kernel with tuned parameters
Mdl = initKernelParameters(Mdl);
end
```

Published with MATLAB® R2020b

computeTuneCriteria

Objective function to solve minimum constraint problem, delivers negative function values to search minimum function evaluation. Estimates the minimum of the negative logarithmic marginal likelihoods for current model parameters. No assignments on model, just recalculate function evaluation minimum.

Syntax

```
feval = computeTuneCriteria(theta, Mdl)
```

Description

feval = computeTuneCriteria(theta, Mdl) sets new kernel parameter, reinitiates model and calculates min criteria by likelihoods.

Input Arguments

theta kernel parameter vector.

Mdl model struct to reinitiate.

Output Arguments

feval function evaluation value.

Requirements

- Other m-files required: None
- Subfunctions: initKernelParameters
- MAT-files required: None

See Also

- [tuneKernel](#)
- [initKernelParameters](#)

Created on March 03. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function feval = computeTuneCriteria(theta, Mdl)
    % reinit kernel on new theta kernel parameters
    Mdl.theta = theta;
    Mdl = initKernelParameters(Mdl);

    % return function evaluation as neg. likelihood of radius
    feval = -1 * (Mdl.LMLSin + Mdl.LMLcos);
end
```

Published with MATLAB® R2020b

predFrame

Predicts single test point and computes angle and radius by predicted sinoids. Delivers several quality criteria.

Syntax

```
[fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predFrame(Mdl, Xcos, Xsin)
```

Description

[fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predFrame(Mdl, Xcos, Xsin) predicts sinoids by passed regression model and test frame of raw data matrix. Comutes angle and radius by predicted results. Several quality creteria are setup based on predictive variance.

Input Arguments

Mdl model struct.

Xcos matrix frame of cosine test data.

Xsin matrix frame of sine test data.

Output Arguments

fang computed angle by predicted cosine and sine results.

frad computed radius by predicted cosine and sine results.

fcos predictive mean result of cosine regression.

fsin predictive mean result of sine regression.

fcov predictive variance for both predictive means.

s resulting standard deviation by predictive variance and noise level.

ciang confidence interval of computed angle.

cirad confidence interval of computed radius.

Requirements

- Other m-files required: None
- Subfunctions: `computeTransposeInverseProduct`, `sinoids2angles`
- MAT-files required: None

See Also

- [computeTransposeInverseProduct](#)
- [sinoids2angles](#)
- [initKernel](#)
- [initKernelParameters](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```

function [fang, frad, fcov, s, ciang, cirad] = predFrame(Mdl, ...
Xcos, Xsin)

% adjust inputs if needed
Xcos = Mdl.inputFun(Xcos);
Xsin = Mdl.inputFun(Xsin);

% compute covariance between observations and test point
k = Mdl.kernelFun(Mdl.Xcos, Xcos, Mdl.Xsin, Xsin, Mdl.theta);

% compute predictiv variance as the difference between test point covariance
% which should be Mdl.theta(1) = s2f product of the covariance between
% observations and test points
% compute the covariance of test point itself means distance is zero which
% implies that result must be the variance s2f
c1 = Mdl.kernelFun(Xcos, Xcos, Xsin, Xsin, Mdl.theta);
% assert(c1 == Mdl.theta(1));

% now add variance from additives
fcov = c1 - computeTransposeInverseProduct(Mdl.L, k);

% predict depending on model mean function
switch Mdl.mean
    case 'zero'
        % compute the predictive means directly by covariance vector and
        % alpha weights, mean is zero
        fcov = k' * Mdl.AlphaCos;
        fsin = k' * Mdl.AlphaSin;
    case 'poly'
        % compute
        fcov = Mdl.meanFunCos(Xcos) + k' * Mdl.AlphaCos;
        fsin = Mdl.meanFunSin(Xsin) + k' * Mdl.AlphaSin;
    otherwise
        error('Unsupported mean function %s in prediction.', Mdl.mean);
end

% compute radius from sinoid results
frad = sqrt(fcov^2 + fsin^2);

% compute angle in rad from sinoid results
fang = sinoids2angles(fsin, fcov, frad, true);

% sigma of the normal distribution over fradius
s = sqrt(fcov + Mdl.s2n);

% 95% confidence interval over fradius
ciang = [fang - asin(1.96 * s * sqrt(2)), fang + asin(1.96 * s * sqrt(2))];

% 95% confidence interval over fradius
cirad = [frad - 1.96 * s * sqrt(2), frad + 1.96 * s * sqrt(2)];
end

```

predDS

Predicts all frames of a test dataset at once.

Syntax

```
[fang, frad, fcov, fsin, s, ciang, cirad] = predDS(Mdl, TestDS)  
predicts whole dataset at once using predFrame in a loop.
```

Description

```
[fang, frad, fcov, fsin, s, ciang, cirad] = predDS(Mdl, TestDS)
```

Input Arguments

Mdl model struct.

TestDS struct of loaded test dataset.

Output Arguments

fang vector of computed angle by predicted cosine and sine results.

frad vector of computed radius by predicted cosine and sine results.

fcos vector of predictive mean result of cosine regression.

fsin vector of predictive mean result of sine regression.

fcov vector of predictive variance for both predictive means.

s vector of resulting standard deviation by predictive variance and noise level.

ciang vector of confidence interval of computed angle.

cirad vector of confidence interval of computed radius.

Requirements

- Other m-files required: None
- Subfunctions: predFrame
- MAT-files required: None

See Also

- [predFrame](#)
- [Training and Test Datasets](#)

Created on March 03. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function [fang, frad, fcov, fsin, s, ciang, cirad] = predDS(Mdl, TestDS)  
  
    % get number of angles in dataset  
    N = TestDS.Info.UseOptions.nAngles;  
  
    % allocate memory for results
```

```
fang = zeros(N, 1); % angle
frad = zeros(N, 1); % radius
fcos = zeros(N, 1); % cosine
fsin = zeros(N, 1); % sine
fcov = zeros(N, 1); % predictive covariance over radius
s = zeros(N, 1); % sigma standard deviation over radius
ciang = zeros(N, 2); % confidence 95% interval over angles lower and upper
cirad = zeros(N, 2); % confidence 95% interval over radius lower and upper

% predict angle by angle from dataset
for n = 1:N
    % get cosine and sine at n-th angle
    Xcos = TestDS.Data.Vcos(:, :, n);
    Xsin = TestDS.Data.Vsin(:, :, n);

    % predict frame
    [fang(n), frad(n), fcos(n), fsin(n), ...
     fcov(n), s(n), ciang(n, :), cirad(n, :)] = predFrame(Mdl, Xcos, Xsin);
end
end
```

Published with MATLAB® R2020b

lossDS

Predicts all angles of passed test dataset and computes logarithmic losses for radius and angles plus several squared errors.

Syntax

```
[AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl, TestDS)
```

Description

[AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl, TestDS) computes losses and prediction errors of a whole datasets

Examples

Enter example matlab code for each use case.

Input Arguments

positionalArg argument description.

optionalArg argument description.

Output Arguments

AAED Absolute Angular Error in Degrees **SLLA** Std. Log. Loss Angular **SLLR** Std. Log Loss Radius **SEA** Squared Error Angular
SER Squared Error Radius **SEC** Squared Error Cosine **SES** Squared Error Sine

Requirements

- Other m-files required: None
- Subfunctions: angles2sinoids, computeStdLogLoss
- MAT-files required: None

See Also

- [predDS](#)
- [Training and Test Datasets](#)
- [angles2sinoids](#)
- [computeStdLogLoss](#)

Created on March 03. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function [AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl, TestDS)

    % get number of angles in dataset
    N = TestDS.Info.UseOptions.nAngles;

    % get simulated cosin and sine references from dataset angles in degrees
    % and transpose to column vector, get sinoids and angles in rads
    [ysin, ycos, yang] = angles2sinoids(TestDS.Data.angles', false, Mdl.PF);

    % create reference radius of unit cricle, radius must be one for all angles
    yrad = ones(N, 1);

    % predict angles in rads not in degrees
    [fang, frad, fcos, fsin, ~, s, ~, ~] = predDS(Mdl, TestDS);
```

```
% compute log loss and squared error for angles in rad
[SLLA, SEA] = computeStdLogLoss(yang, fang, asin(s) * sqrt(2));

% compute absolute angular error in degrees
AAED = sqrt(SEA) * 180/pi;

% compute log loss and squared error for radius
[SLLR, SER] = computeStdLogLoss(yrad, frad, sqrt(2) * s);

% compute squared error of sinoids
SEC = (ycos - fcov).^2;
SES = (ysin - fsin).^2;

end
```

Published with MATLAB® R2020b

optimGPR

Noise level optimization that implements optimized model by embedded kernel tuning process.

Syntax

```
Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose)
```

Description

Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose) initiates regression model by training data and passed options. Solves min search via bayesopt for optimizing noise level. At each process step built model is reinitiated and tuned to fit best on training data. The noise optimization can be performed by SLLA and SLLR. Depends configuration of GPROptions. The loss computation is done on all forwarded test data.

Examples

```
load config.mat PathVariables GPROptions;
TrainFiles = dir(fullfile(PathVariables.trainingDataPath, 'Training*.mat'));
TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
assert(~isempty(TrainFiles), 'No training datasets found.');
assert(~isempty(TestFiles), 'No test datasets found.');
try
    TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1).name));
    TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).name));
catch ME
    rethrow(ME)
end
Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose);
[fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predDS(Mdl, TestDS)
[AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl, TestDS);
```

Input Arguments

TrainDS loaded training data by infront processes sensor array simulation.

TestDS loaded test data by infront processes sensor array simulation.

GPROptions loaded parameter group from config.mat. Struct with options.

verbose activates prompt for true or 1. Vice versa for false or 0.

Output Arguments

Mdl fully optimized model struct with tuned hyperparameters and optimized noise level.

Requirements

- Other m-files required: None
- Subfunctions: initGPR, tuneKernel, computeOptimCriteria, lossDS, optimizableVariable, bayesopt
- MAT-files required: None

See Also

- [bayesopt](#)
- [optimizablevariable](#)
- [initGPR](#)

- [tuneKernel](#)
- [computeOptimCriteria](#)
- [lossDS](#)

Created on March 05. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose)

    % init model by training data and initial options
    Mdl = initGPR(TrainDS, GPROptions);

    % create noise variance s2n used in GPR with bounds
    s2n = optimizableVariable('s2n', GPROptions.s2nBounds, 'Transform', 'log');

    % create function handle for bayes optimization
    SLL = GPROptions.SLL;
    fun = @(OptVar) computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose);

    % perform bayes noise optimization
    results = bayeso(fun, s2n, ...
        'Verbose', verbose, ...
        'MaxObjectiveEvaluations', GPROptions.OptimRuns, ...
        'AcquisitionFunctionName', 'expected-improvement-per-second');

    % update options with results and reinit model and tune to final model
    Mdl.s2n = results.XAtMinObjective.s2n;
    Mdl = tuneKernel(Mdl, verbose);

    % compute final loss and get mean log loss for angles and radius as
    % indicator of model total model fit
    [~, SLLA, SLLR] = lossDS(Mdl, TestDS);
    Mdl.MSLLA = mean(SLLA);
    Mdl.MSLLR = mean(SLLR);

end
```

Published with MATLAB® R2020b

computeOptimCriteria

Object function to compute the loss of a fully initialized and tuned regression model. Computes the mean std. log. loss of angles MSLA or radius MSLLR as function evaluation value for bayesopt. Perform noise adjustment in cycles in bayesopt.

Syntax

```
MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose)
```

Description

```
MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose)
```

Input Arguments

OptVar optimization variable. Noise level passed by bayesopt algorithm.

Mdl model struct.

TestDS loaded test data by infront processesed sensor array simulation.

SLL indicates which loss is used for MSLL. SLLA for angle and SLLR for radius.

verbose activates prompt for true or 1. Vice versa for false or 0.

Output Arguments

MSLL mean standardized logarithmic loss. Function evaluation value for optimGPR

Requirements

- Other m-files required: None
- Subfunctions: tuneKernel, lossDS, mean
- MAT-files required: None

See Also

- [optimGPR](#)
- [tuneKernel](#)
- [lossDS](#)
- [mean](#)

Created on March 05. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose)

    % push current variance value into GPR
    Mdl.s2n = OptVar.s2n;

    % tune kernel with new noise variance
    Mdl = tuneKernel(Mdl, verbose);

    % get loss on dataset for angular prediction
    switch SLL
        case 'SLLA'
            [~, SLL] = lossDS(Mdl, TestDS);
```

```
case 'SLLR'
    [~, ~, SLL] = lossDS(Mdl, TestDS);

otherwise
    error('Unknown SLL %s.', SLL);
end

% return mean loss to evaluate optimization run
MSLL = mean(SLL);
end
```

Published with MATLAB® R2020b

kernelQFCAPX

Kernel implementation for approximated quadratic fractional covariance (QFCAPX) kernel. The module contains the covariance function, a mean function to build up polynoms to approximation use and a init function to module into regression model.

QFCAPX

Covariance function to compute K matrix and k vector on vector data.

meanPolyQFCAPX

Builds up mean polynom on vector data.

initQFCAPX

Initiates kernel into regression model.

See Also

- [initGPR](#)
- [initKernel](#)
- [initKernelParameters](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Published with MATLAB® R2020b

QFCAPX

Approximates QFC with triangle inequation, norming is pulled out to input stage kernel is feeded with norm vectors or scalars instead of matrices.

Syntax

```
K = QFCAPX(ax, bx, ay, by, theta)
```

Description

K = QFCAPX(ax, bx, ay, by, theta) computes quadratic distances between data points and parametrize it with height and length scales. Computes distance with quadratic euclidian norm.

Input Arguments

ax vector of cosine simulation components.

bx vector of cosine simulation components.

ay vector of sine simulation components.

by vector of sine simulation components.

theta vector of kernel parameters.

Output Arguments

K noise free covarianc matrix.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [initQFCAPX](#)
- [meanPolyQFCAPX](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function K = QFCAPX(ax, bx, ay, by, theta)
    arguments
        % validate data as real vector of same size
        ax (:,:,:,:) double {mustBeReal}
        bx (:,:,:,:) double {mustBeReal, mustBeFitSize(ax,bx)}
        ay (:,:,:,:) double {mustBeReal, mustBeFitSize(ax,ay)}
        by (:,:,:,:) double {mustBeReal, mustBeFitSize(ax,by)}
        % validate kernel parameters as 1x2 vector
        theta (1,2) double {mustBeReal}
    end

    % get number of observations for each dataset, cosine and sine
    M = length(ax);
    N = length(bx);
```

```
% expand covariance parameters, variance and lengthscale
c2 = 2 * theta(2)^2; % 2*s1^2
c1 = theta(1) * c2;   % s2f * c

% allocate memory for K
K = zeros(M, N);

% loop through observation points and compute the covariance for each
% observation against another
for m = 1:M
    for n = 1:N
        % get distance between m-th and n-th observation
        % compute distance with quadratic frobenius normed vectors
        r2 = (ax(m) - bx(n))^2 + (ay(m) - by(n))^2;

        % engage lengthscale and variance on distance
        K(m,n) = c1 / (c2 + r2);
    end
end

function mustBeFitSize(a, b)
    % Test for equal size
    if ~isequal(size(a,1,2), size(b,1,2))
        eid = 'Size:notEqual';
        msg = 'Sizes of are not fitting.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

meanPolyQFCAPX

Basis or trend function to compute the H matrix as set of $h(x)$ vectors for each predictor to apply a mean feature space as polynom approximated mean with beta coefficients. Compute H matrix to estimate beta. Vectors instead of matrices norming is place at input stage.

Syntax

```
H = meanPolyQFCAPX(X, degree)
```

Description

H = meanPolyQFCAPX(X, degree) build polynom by passed data.

Input Arguments

X vector data.

degree polynom degree.

Output Arguments

H polynom.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [initQFCAPX](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function H = meanPolyQFCAPX(X, degree)
    % get number of observations
    N = length(X);

    % returns only ones if p = 0
    H = ones(degree + 1, N);

    % compute polynom for degrees > 0
    if degree > 0
        H(2,:) = X';
    end

    % compute none linear polynomials if degree > 1
    if degree > 1
        for p = 2:degree
            H(p+1,:) = X'.^p;
        end
    end
end
```

.....

Published with MATLAB® R2020b

initQFCAPX

Attaches QFCAPX kernel to model struct. Depending on mean options attach zero mean functions and sets all related kernel parameters and dependencies to zero. If mean is polynom fitting, attaches meanPolyQFCAPX as basis function to build polynom matrix H and sets a none zero mean function. Computes dataset inputs vectors or scalars. Kernel works on vector data.

Syntax

```
Mdl = initQFCAPX(Mdl)
```

Description

Mdl = initQFCAPX(Mdl) loads approximated quadratic fraction covariance function and basis function depending on mean in **Mdl** struct. Sets input function to Frobenius Norm. Reprocess training matrix data to vector data.

Input Arguments

Mdl struct with model parameter and training data.

Output Arguments

Mdl struct with attached kernel functionality

Requirements

- Other m-files required: None
- Subfunctions: QFCAPX, meanPolyQFCAPX, frobeniusNorm
- MAT-files required: None

See Also

- [initKernel](#)
- [meanPolyQFCAPX](#)
- [QFCAPX](#)
- [frobeniusNorm](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initQFCAPX(Mdl)

    % set QFC kernel function
    Mdl.kernelFun = @QFCAPX;

    % set input transformation function to apply adjustments to
    % covariance function, norm input matrice with frobenius norm to scalar or
    % vectors.
    Mdl.inputFun = @(X) frobeniusNorm(X, false);

    % transform traning data to vectors
    Ncos = zeros(Mdl.N, 1);
    Nsin = zeros(Mdl.N, 1);
    for n = 1:Mdl.N
        Ncos(n) = Mdl.inputFun(Mdl.Xcos(:,:,n));
        Nsin(n) = Mdl.inputFun(Mdl.Xsin(:,:,n));
    end

    % update training data with norm vectors
```

```
Mdl.Xcos = Ncos;
Mdl.Xsin = Nsin;

% set mean function to compute cosine and sine H matrix
switch Mdl.mean
    % zero mean m(x) = 0
    case 'zero'
        % set polyDegree to -1 for no polynom indication
        Mdl.polyDegree = -1;

        % set basis function
        Mdl.basisFun = @(X) 0;

    % mean by polynom m(x) = H' * beta
    case 'poly'
        % set basis function produces a (polyDeg+1)xN H matrix
        Mdl.basisFun = @(X) meanPolyQFCAPX(X, Mdl.polyDegree);

    % end mean select QFC kernel
    otherwise
        error('Unknown mean function %.', Mdl.mean);
end
end
```

Published with MATLAB® R2020b

kernelQFC

Kernel implementation for quadratic fractional covariance (QFC) kernel. The module contains the covariance function, a mean function to build up polynoms to approximation use and a init function to module into regression model.

QFC

Covariance function to compute K matrix and k vector on matrix data.

meanPolyQFC

Builds up mean polynom on matrix data.

initQFC

Initiates kernel into regression model.

See Also

- [initGPR](#)
- [initKernel](#)
- [initKernelParameters](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Published with MATLAB® R2020b

QFC

Quadratic Fractional Covariance function. Computes covariance matrix K. Works with raw matrix data. Precise solution.

Syntax

```
K = QFC(Ax, Bx, Ay, By, theta)
```

Description

K = QFC(Ax, Bx, Ay, By, theta) computes quadratic distances bewtween data points and parametrize it with height and length scales. Computes distance with quadratic Frobenius Norm.

Input Arguments

Ax matrix of cosine simulation components.

Bx matrix of cosine simulation components.

Ay matrix of sine simulation components.

By matrix of sine simulation components.

theta vector of kernel parameters.

Output Arguments

K noise free covarianc matrix.

Requirements

- Other m-files required: None
- Subfunctions: sum
- MAT-files required: None

See Also

- [initQFC](#)
- [meanPolyQFC](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function K = QFC(Ax, Bx, Ay, By, theta)
    arguments
        % validate data as real matrices of same size in 1st and 2nd dimension
        Ax (:,:,:,:) double {mustBeReal}
        Bx (:,:,:,:) double {mustBeReal, mustBeFitSize(Ax,Bx)}
        Ay (:,:,:,:) double {mustBeReal, mustBeFitSize(Ax,Ay)}
        By (:,:,:,:) double {mustBeReal, mustBeFitSize(Ax,By)}
        % validate kernel parameters as 1x2 vector
        theta (1,2) double {mustBeReal}
    end

    % get number of observations for each dataset, cosine and sine matrices have
    % equal sizes just extract size from one
    [~, ~, M] = size(Ax);
    [~, ~, N] = size(Bx);
```

```
% expand covariance parameters, variance and lengthscale
c2 = 2 * theta(2)^2; % 2*s1^2
c1 = theta(1) * c2; % s2f * c

% allocate memory for K
K = zeros(M, N);

% loop through observation points and compute the covariance for each
% observation against another
for m = 1:M
    for n = 1:N
        % get distance between m-th and n-th observation
        distCos = Ax(:,:,:m) - Bx(:,:,:n);
        distSin = Ay(:,:,:m) - By(:,:,:n);

        % compute quadratic frobenius norm distance as separated
        % distances of cosine and sine, norm of vector fields
        r2 = sum(distCos.^2, 'all') + sum(distSin.^2, 'all');

        % engage lengthscale and variance on distance
        K(m,n) = c1 / (c2 + r2);
    end
end

function mustBeFitSize(A, B)
    % Test for equal size
    if ~isequal(size(A,1,2), size(B,1,2))
        eid = 'Size:notEqual';
        msg = 'Sizes of A and B are not fitting.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

meanPolyQFC

Basis or trend function to compute the H matrix as set of $h(x)$ vectors for each predictor to apply a mean feature space as polynom approximated mean with beta coefficients. Compute H matrix to estimate beta.

Syntax

```
H = meanPolyQFC(X, degree)
```

Description

H = meanPolyQFC(X, degree) build polynom by passed data. Fires Frobenius Norm on matrix data.

Input Arguments

X matrix data.

degree polynom degree.

Output Arguments

H polynom.

Requirements

- Other m-files required: None
- Subfunctions: frobeniusNorm
- MAT-files required: None

See Also

- [initQFC](#)
- [frobeniusNorm](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function H = meanPolyQFC(X, degree)
    % get number of observations
    [~, ~, N] = size(X);

    % returns only ones if p = 0
    H = ones(degree + 1, N);

    % compute polynom for degrees > 0
    if degree > 0
        for n = 1:N
            H(2,n) = frobeniusNorm(X(:,:,n), false);
        end
    end

    % compute none linear polynomials if degree > 1
    if degree > 1
        for p = 2:degree
            H(p+1,:) = H(2,:).^p;
        end
    end
end
```

.....

Published with MATLAB® R2020b

initQFC

Attaches QFC kernel to model struct. Depending on mean options attach zero mean functions and sets all related kernel parameters and dependencies to zero. If mean is polynom fitting, attaches meanPolyQFC as basis function to build polynom matrix H and sets a none zero mean function. Bypasses dataset inputs as they are. Kernel works on matrix data.

Syntax

```
Mdl = initQFC(Mdl)
```

Description

Mdl = initQFC(Mdl) loads quadratic fraction covariance function and basis function depending on mean in **Mdl** struct. Sets input function as bypass.

Input Arguments

Mdl struct with model parameter and training data.

Output Arguments

Mdl struct with attached kernel functionality

Requirements

- Other m-files required: None
- Subfunctions: QFC, meanPolyQFC
- MAT-files required: None

See Also

- [initKernel](#)
- [meanPolyQFC](#)
- [QFC](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function Mdl = initQFC(Mdl)

    % set QFC kernel function
    Mdl.kernelFun = @QFC;

    % set input transformation function to apply adjustments to
    % covariance function, here bypass inputs as they are, no transformation of
    % training data needed
    Mdl.inputFun = @(X) X;

    % set mean function to compute cosine and sine H matrix
    switch Mdl.mean
        % zero mean m(x) = 0
        case 'zero'
            % set polyDegree to -1 for no polynom indication
            Mdl.polyDegree = -1;

            % set basis function
            Mdl.basisFun = @(X) 0;

            % mean by polynom m(x) = H' * beta
```

```
    case 'poly'
        % set basis function produces a (polyDeg+1)xN H matrix
        Md1.basisFun = @(X) meanPolyQFC(X, Md1.polyDegree);

        % end mean select QFC kernel
    otherwise
        error('Unknown mean function %.', Md1.mean);
    end
end
```

Published with MATLAB® R2020b

basicMathFunctions

Set of basic algebraic, analytic and trigonometric functions.

sinoids2angles

Converts sinoids to angles.

angles2sinoids

Converts angles to sinoids.

decomposeChol

Performs a Cholesky Decomposition of a matrix to its lower triangle matrix. Returns logarithmic determinate of the matrix as side product.

frobeniusNorm

Computes the Frobenius Norm of a matrix.

computeInverseMatrixProduct

Computes the product of an inverted matrix A and a vector b or a matrix B by the represented lower triangle matrix L of Cholesky decomposed A.

computeTransposeInverseProduct

Computes the both side product of an inverted matrix A with a vector b or matrix B (left product) and the transposed vector b or matrix B (right product).

addNoise2Covariance

Additive noise for noisy GPR observations. Add noise along covariance matrix diagonal.

computeAlphaWeights

Computes regression weights by residual of regression targets and regression mean values.

computeStdLogLoss

Computes standardized logarithmic loss of test data and predicted data.

computeLogLikelihood

Computes regression evidence as log marginal likelihood.

estimateBeta

Compute polynom coefficients for mean approximation on training data.

Created on February 14. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

Published with MATLAB® R2020b

sinoids2angles

Computes angles in rad or degree by passed sinoids and radius. The angle calculation is assigned in two steps. At first compute bare angles by acos and asin functions. Divide therfore the corresponding sinoid by the radius. Furhter on the result from acos is used with an interval correction given by the second results from the asin function. The amplitudes of the sinoids must be one or near to one.

So angle computation relates to the unit circle with dependencies of

$$f_{rad} = \sqrt{f_{sin}^2 + f_{cos}^2}$$

and computes intermediate angle results in rad of

$$f_c = \arccos\left(\frac{f_{cos}}{f_{rad}}\right)$$

$$f_s = \arcsin\left(\frac{f_{sin}}{f_{rad}}\right)$$

$$f_a = \text{atan2}\left(\frac{f_{sin}}{f_{cos}}\right)$$

The final angle result is computed by cosine intermediate result which uses the sine intermediate result als threshold to decide when angles must be enter the third quadrant of the unit circle.

$$f_{ang} = \begin{cases} f_c & f_s \geq 0 \\ -f_c + 2\pi & f_s < 0 \end{cases}$$

A the second angle reconstruction can be achieved by the atan2 function which has although an interval shift at 180 degree. It implies to use the sine results as threshold too.

$$f_{ang} = \begin{cases} f_a & f_s \geq 0 \\ f_a + 2\pi & f_s < 0 \end{cases}$$

Syntax

[fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad, rad)

Description

[fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad, rad) returns angles in rad or degrees given by corresponding sinoids and radius. Set rad flag to false if angles in degrees are needed.

Examples

```
phi = linspace(0, 2*pi, 100);
fsin = sin(phi);
fcos = cos(phi);
frad = sqrt(fsin.^2 + fcos.^2);
[fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad, true)
```

Input Arguments

fsin is a scalar or vector with sine information to corresponding angle.

fcos is a scalar or vector with cosine information to corresponding angle.

frad is a scalar or vector which represents the radius of each sine and cosine position.

rad is a boolean flag. If it is false the resulting angles are converted into degrees. If it is true fang is returned in rad. Default is true.

how is char vector which gives option how to reconstruct angles via acos or atan2 function. Default is atan2. Both methods use asin function as threshold to switch 180° intervall.

Output Arguments

fang is a scalar or vector with angles in rad or degree corresponding to the sine and cosine inputs.

fc is a scalar or vector with angles directly computed by cosine and radius.

fs is a scalar or vector with angles directly computed by sine and radius.

fa is a scalar or vector with angles directly computed by sine and cosine.

Requirements

- Other m-files required: None
- Subfunctions: acos, asin
- MAT-files required: None

See Also

- [angles2sinoids](#)

Created on December 31. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function [fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad, rad, how)
    arguments
        % validate sinoids and radius as scalar or vector of the same size
        fsin (:,1) double {mustBeReal}
        fcos (:,1) double {mustBeReal, mustBeEqualSize(fsin, fcos)}
        frad (:,1) double {mustBeReal, mustBeEqualSize(fsin, frad)}
        % validate rad as boolean flag with default true
        rad (1,1) logical {mustBeNumericOrLogical} = true
        % validate how as char option flag with default atan2
        how (1,:) char {mustBeText} = 'atan2'
    end

    % compute angles by cosine, sine and radius
    fc = acos(fcos ./ frad);
    fs = asin(fsin ./ frad);
    fa = atan2(fsin, fcos);

    % get indices for interval > 180°
    idx = fs < 0;

    switch how
        case 'acos'
            % angles from cosine
            fang = fc;

            % correct 180° interval
            fang(idx) = -1 * fang(idx) + 2 * pi;
    end
```

```
case 'atan2'
    fang = fa;

    % correct 180° interval
    fang(idx) = fang(idx) + 2 * pi;

    otherwise
        error('Unknow arc function for reconstruction: %s.', how)
    end

    % return degrees if not rad
    if ~rad, fang = 180 / pi * fang; end
end

% Custom validation function
function mustBeEqualSize(a,b)
    % Test for equal size
    if ~isequal(size(a),size(b))
        eid = 'Size:notEqual';
        msg = 'Size of first input must equal size of second input!';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

angles2sinoids

Converts angles (rad or degree) to sine and cosine waves with respect to a period factor which gives the ability to abstract higher periodicity. Additionally the angles are recalculated according to passed period factor.

Computes sine and cosine by product of angle in rad multiplied by period factor.

$$f_{\sin} = \sin(p_f \cdot f_{\text{ang}})$$

$$f_{\cos} = \cos(p_f \cdot f_{\text{ang}})$$

If needed a recomputation of the given angles takes place by computed sinoids.

Syntax

```
[fsin, fcos, fang] = angles2sinoids(fang, rad, pf)
```

Description

[fsin, fcos, fang] = angles2sinoids(fang, rad, pf) computes sinoids from passed angles in rad or degree with respect to periodicity of angles. The flag **rad** converts input angles from degree to rad if set to false.

Examples

```
fang = linspace(0, 360, 100);
[fsin, fcos, fang] = angles2sinoids(fang, true, 1)
```

Input Arguments

fang is a scalar or vector of angles in rad or degree.

rad is a boolean flag. Input angles are converted to rad if set to false.

pf is a positive integer factor. The period factor describes the periodicity of angles in data.

Output Arguments

fsin is a scalar or vector of sine values corresponding to passed angles with respect of the periodicity of angles.

fcos is a scalar or vector of cosine values corresponding to passed angles with respect of the periodicity of angles.

fang is a scalar or vector of recalculated angles with respect of periodicity.

Requirements

- Other m-files required: `sinoids2angles`
- Subfunctions: `sin`, `cos`
- MAT-files required: None

See Also

- [sinoids2angles](#)

Created on December 31. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function [fsin, fcos, fang] = angles2sinoids(fang, rad, pf)
```

```
arguments
    % validate angles as scalar or vector
    fang (:,1) double {mustBeReal}
    % validate rad as boolean flag with default true
    rad (1,1) logical {mustBeNumericOrLogical} = true
    % validate period factor as positive scalar with default 1
    pf (1,1) double {mustBeInteger, mustBePositive} = 1
end

% if rad flag is false and angles in degree convert to rad
if ~rad, fang = fang * pi / 180; end

% calculate sinoids
fsin = sin(pf * fang);
fcos = cos(pf * fang);

% compute radius
frad = sqrt(fcos.^2 + fsin.^2);

% recalculate angles to corrected sinoids in rad
if nargout > 2, fang = sinoids2angles(fsin, fcos, frad); end
end
```

Published with MATLAB® R2020b

decomposeChol

Computes the Cholesky Decomposition of a symmetric positive definite matrix A and calculate the log determinate as side product of the decomposition. Computes the lower triangle matrix L.

Syntax

```
[L, logDet] = decomposeChol(A)
```

Description

[L, logDet] = decomposeChol(A) computes lower triangle matrix of input matrix A using the Cholesky Decomposition. As side product of the decomposition the logarithmic determinante of A is returned too.

Examples

```
A = [1.0, 0.9, 0.8;
      0.9, 1.0, 0.9;
      0.8, 0.9, 1.0];
[L, logDet] = decomposeChol(A)
assert(all(L*L'==A, 'all'))
assert(log(det(A)) == logDet)
```

Input Arguments

A symmetric, pos. finite double matrix of size N x N.

Output Arguments

L is a lower trinagle matrix of size N x N. Multiply L with its transposed to get matrix A.

logDet is a scalar and the logarithmic determinante of A. Computed along the diagonal of L.

Requirements

- Other m-files required: None
- Subfunctions: chol, mustBeValidMatrix
- MAT-files required: None

See Also

- [chol](#)

Created on January 04. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function [L, logDet] = decomposeChol(A)
    arguments
        A (:,:) double {mustBeReal, mustBeValidMatrix(A)}
    end

    % decompose A to lower triangle matrix
    [L, flag]= chol(A, 'lower');
    if flag ~= 0
        eid = 'chol:Fails';
        msg = 'Cholesky Decomposition fails.';
        throwAsCaller(MException(eid,msg))
    end
```

```
% compute log determinate of A
if nargout > 1, logDet = 2 * sum(log(diag(L))); end
end

% Custom validation function
function mustBeValidMatrix(A)
    % test if is matrix
    if ~ismatrix(A)
        eid = 'Matrix:notMatrix';
        msg = 'A is not a matrix.';
        throwAsCaller(MException(eid,msg))
    end
    % Test for N x N
    if ~isequal(size(A,1),size(A,2))
        eid = 'Matrix:notNxN';
        msg = 'Size of matrix is not N x N.';
        throwAsCaller(MException(eid,msg))
    end
    % test if symmetric
    if ~issymmetric(A)
        eid = 'Matrix:notSymmetric';
        msg = 'Matrix is not symmetric.';
        throwAsCaller(MException(eid,msg))
    end
    % test if positive definite
    if ~all(eig(A) >= 0)
        eid = 'Matrix:notPosDefinite';
        msg = 'Matrix is not pos. definite.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

frobeniusNorm

Computes the Frobenius Norm of a matrix A.

Syntax

```
nv = frobeniusNorm(A, approx)
```

Description

frobeniusNorm(A, approx) computes Frobenius Norm of M x N matrix. If approx is true the Norm is approximated with mean2 function.

Examples

```
A = magic(8);  
nv = frobeniusNorm(A, approx)
```

Input Arguments

A is a M x N matrix of real values.

approx is boolean flag. If true the norm is approximated. Default is false.

Output Arguments

nv is a scalar norm value.

Requirements

- Other m-files required: None
- Subfunctions: mean2, sqrt, sum
- MAT-files required: None

See Also

- [QFCAPX](#)
- [meanPolyQFC](#)

Created on January 05, 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function nv = frobeniusNorm(A, approx)
    arguments
        % validate A as real matrix
        A (:,:) double {mustBeReal}
        % validate approx as flag with default false
        approx (1,1) logical {mustBeNumericOrLogical} = false
    end

    % norm matrix
    if approx
        % approximate frobenius with mean and multiply with radicant of RMS
        % frobenius norm is a RMS * sqrt(N x N), RMS >= mean
        nv = mean2(A) * sqrt(numel(A));
    else
        % norm with frobenius
        nv = sqrt(sum(A.^2, 'all'));
    end
```

```
end
```

Published with MATLAB® R2020b

computeInverseMatrixProduct

Computes the product of an inverted matrix A represented by its Cholesky decomposed lower triangle matrix L and a vector b or even another matrix B. Solving runs column by column if a Matrix B is passed and computes the linear system to an intermediate result with the lower triangle matrix L (inner solving) and finaly in an outer solving with the transposed lower triangle matrix L and the intermediate result to the final product x of the inverted matrix and vector(s).

Syntax

```
x = computeInverseMatrixProduct(L, b)
```

Description

x = computeInverseMatrixProduct(L, b) performs the inverse matrix product of matrix A and a vector b or even another matrix B. Then product is formed column by column of B. The not inverted matrix A is represented by its lower triangle matrix L (Cholesky Decomposition).

Examples

```
A = [1.0, 0.9, 0.8;
      0.9, 1.0, 0.9;
      0.8, 0.9, 1.0];
L = decomposeChol(A);
b = [5; 9; 0.5];
x = computeInverseMatrixProduct(L, b);
B = [5 , 9;
      0.5, 5;
      3 , -1];
X = computeInverseMatrixProduct(L, B);
```

Input Arguments

L is the lower triangle matrix of a matrix A.

b is a vector or matrix of real values.

Output Arguments

x is the product of the inverted matrix A and b.

Requirements

- Other m-files required: None
- Subfunctions: linsolve, mustBeLowerTriangle, mustBeFitSize
- MAT-files required: None

See Also

- [decomposeChol](#)
- [linsolve](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function x = computeInverseMatrixProduct(L, b)
    arguments
        % validate L as lower triangle matrix of size N x N
        L (:,:) double {mustBeReal, mustBeLowerTriangle(L)}
        % validate b as vecotr matrix with same row length as L
```

```
b (:,:) double {mustBeReal, mustBeFitSize(L, b)}
end

% set linsolve option for inner (lower triangle) and outer (upper triangle)
% solve, outer solve runs with intermediate result of inner solve
opts1.LT = true;
opts2.UT = true;

% get size of b, if b is a matrix solve column by column
[M, N] = size(b);

% allocate memory for product result
x = zeros(M, N);

% solve column by column
for n = 1:N
    % compute inner solve to intermediate result vecotor
    v = linsolve(L, b(:,n), opts1);

    % save final inverse product from outer solve
    x(:,n) = linsolve(L', v, opts2);
end
end

% Custom validation functions
function mustBeLowerTriangle(L)
    % Test for lower triangle matrix
    if ~istril(L)
        eid = 'Matrix:notLowerTriangle';
        msg = 'Matrix is not lower triangle.';
        throwAsCaller(MException(eid,msg))
    end
    % Test for N x N
    if ~isequal(size(L,1), size(L, 2))
        eid = 'Size:notEqual';
        msg = 'L is not size of N x N.';
        throwAsCaller(MException(eid,msg))
    end
end

function mustBeFitSize(L, b)
    % Test for equal size
    if ~isequal(size(L,1), size(b, 1))
        eid = 'Size:notEqual';
        msg = 'Size of rows are not fitting.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

computeTransposeInverseProduct

Computes the both side product of an inverted Matrix A and a vector b (left product) and the transposed of b (right product). If matrix B is passed instead of a vector b the computation is done column by column of B. The matrix A is represented by its Cholesky decomposed lower triangle matrix L. The computation is optimized so it does a linear solve with lower triangle matrix to intermediate result vector. The final both side product is now the transpose of intermediate result multiplied with intermediate results itself. So a outer linear solve is not needed anymore.

Syntax

```
x = computeTransposeInverseProduct(L, b)
```

Description

x = computeTransposeInverseProduct(L, b) linear solve the equation system to a intermediate result and get the both side transpose inverse product of A and b by multiply the transpose intermediate result with intermediate result itself.

Examples

```
A = [1.0, 0.9, 0.8;
      0.9, 1.0, 0.9;
      0.8, 0.9, 1.0];
L = decomposeChol(A);
b = [5; 9; 0.5];
x = computeTransposeInverseProduct(L, b);
B = [5, 9;
      0.5, 5;
      3, -1];
X = computeTransposeInverseProduct(L, B);
```

Input Arguments

L is the lower triangle matrix of a matrix A.

b is a vector or matrix of real values.

Output Arguments

x is the both side product of the inverted matrix A and b and transposed b.

Requirements

- Other m-files required: None
- Subfunctions: linsolve, mustBeLowerTriangle, mustBeFitSize
- MAT-files required: None

See Also

- [decomposeChol](#)
- [linsolve](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function x = computeTransposeInverseProduct(L, b)
    arguments
        % validate L as lower triangle matrix of size N x N
        L (:,:) double {mustBeReal, mustBeLowerTriangle(L)}
        % validate b as vecotr matrix with same row length as L
```

```
b (:,:) double {mustBeReal, mustBeFitSize(L, b)}
```

```
end
```

```
% set linsolve option to solve with lower triangle matrix
```

```
opts.LT = true;
```

```
% get size of b, if b is a matrix solve column by column
```

```
[M, N] = size(b);
```

```
% allocate memory for intermediate result
```

```
v = zeros(M, N);
```

```
% solve column by column
```

```
for n=1:N
```

```
    % save to intermediate result columns
```

```
    v(:,n) = linsolve(L, b(:,n), opts);
```

```
end
```

```
% get final product by multiply transposed intermediate result with itself
```

```
x = v' * v;
```

```
end
```

```
% Custom validation functions
```

```
function mustBeLowerTriangle(L)
```

```
    % Test for lower triangle matrix
```

```
    if ~istril(L)
```

```
        eid = 'Matrix:notLowerTriangle';
```

```
        msg = 'Matrix is not lower triangle.';
```

```
        throwAsCaller(MException(eid,msg))
```

```
    end
```

```
    % Test for N x N
```

```
    if ~isequal(size(L,1), size(L, 2))
```

```
        eid = 'Size:notEqual';
```

```
        msg = 'L is not size of N x N.';
```

```
        throwAsCaller(MException(eid,msg))
```

```
    end
```

```
end
```

```
function mustBeFitSize(L, b)
```

```
    % Test for equal size
```

```
    if ~isequal(size(L,1), size(b, 1))
```

```
        eid = 'Size:notEqual';
```

```
        msg = 'Size of rows are not fitting.';
```

```
        throwAsCaller(MException(eid,msg))
```

```
    end
```

```
end
```

Published with MATLAB® R2020b

addNoise2Covariance

Add noise to covarianc matrix for noisy observations. Add noise along matrix diagonal.

Syntax

```
Ky = addNoise2Covariance(K, s2n)
```

Description

Ky = **addNoise2Covariance(K, s2n)** witch on additive noise on covariance matrix diagonal. Uses eye matrix as mask.

Examples

```
addNoise2Covariance(zeros(4), 2)
```

Input Arguments

K N x N covariance matrix. Noise free.

s2n real scalar value.

Output Arguments

Ky covariance matrix for noisy observations.

Requirements

- Other m-files required: None
- Subfunctions: eye, mustBeSquareMatrix
- MAT-files required: None

See Also

- [initKernelParameters](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function Ky = addNoise2Covariance(K, s2n)
    arguments
        % validate K as square matrix
        K (:,:) double {mustBeReal, mustBeSquareMatrix(K)}
        % validate s2n as scalar value
        s2n (1,1) double {mustBeReal}
    end

    % add noise with eye matrix
    Ky = K + s2n * eye(size(K));
end

% Custom validation functions
function mustBeSquareMatrix(K)
    % Test for N x N
    if ~isequal(size(K,1), size(K, 2))
        eid = 'Size:notEqual';
        msg = 'K is not size of N x N.';
        throwAsCaller(MException(eid,msg))
    end
end
```

```
end
```

Published with MATLAB® R2020b

computeAlphaWeights

Computes alpha weights from feature space product $H^T \beta$ and target vector y as product with inverse covariance matrix with additive noise K^{-1} represented by its cholesky decomposed lower triangle matrix L . $K^{-1} * (y - m(x))$.

Syntax

```
alpha = computeAlphaWeights(L, y, m)
```

Description

alpha = computeAlphaWeights(L, y, m) prepare data and forward it to matrix computation.

Input Arguments

L lower triangle matrix of cholesky decomposed K matrix.

y regression target vector.

m regression mean vector.

Output Arguments

alpha regression weights.

Requirements

- Other m-files required: None
- Subfunctions: `computeInverseMatrixProduct`
- MAT-files required: None

See Also

- [decomposeChol](#)
- [computeInverseMatrixProduct](#)
- [initKernelParameters](#)

Created on November 06. 2019 by Klaus Jünemann. Copyright Klaus Jünemann 2019.

```
function alpha = computeAlphaWeights(L, y, m)
    % get residual
    residual = y - m;
    % L and residual is validated in computation below, get weights
    alpha = computeInverseMatrixProduct(L, residual);
end
```

Published with MATLAB® R2020b

computeStdLogLoss

Compute SLL loss between test targets and predictive mean dependend on predictive variance plus used variance for noisy covariance matrix as variance of normal distribution over predictive means $s^2 = fcov + s^2n$. The pred functions returns the standard deviation s so sqrt of the variance.

Syntax

```
[SLL, SE, s2] = computeStdLogLoss(y, fmean, s)
```

Description

[SLL, SE, s2] = computeStdLogLoss(y, fmean, s) compute standardized logarithmic loss by squared error of ideal test data predicted data and standard deviation of predicted data.

Input Arguments

y column vector of ideal data to compare with.

fmean column vector of predictive mean.

s column vector of standard deviation related to predictive mean.

Output Arguments

SLL column vector of standardized logarithmic loss.

SE squared error between **y** and **fmean**.

s2 variance column vector.

Requirements

- Other m-files required: None
- Subfunctions: log
- MAT-files required: None

See Also

- [predFrame](#)
- [predDS](#)
- [lossDS](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function [SLL, SE, s2] = computeStdLogLoss(y, fmean, s)
    arguments
        % validate inputs as real column vectors of same length
        y (:,1) double {mustBeReal, mustBeVector}
        fmean (:,1) double {mustBeReal, mustBeVector, mustBeEqualSize(y, fmean)}
        s (:,1) double {mustBeReal, mustBeVector, mustBeEqualSize(y, s)}
    end
    % squared error
    SE = (y - fmean).^2;

    % s as standard deviation of nomal distributed fmean so square it for
    % variance.
```

```
s2 = s.^2;

% logarithmic error
SLL = 0.5 * (log(2 * pi * s2) + SE ./ s2);
end

% Custom validation functions
function mustBeEqualSize(a, b)
    if ~isequal(length(a), length(b))
        eid = 'Size:notEqual';
        msg = 'Vectors must be the same length.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

computeLogLikelihood

Computes the marginal log likelihood as evidence of the current trained model parameter by solving the equation $\log p(y|X, \alpha, \log|K_y|) = -\frac{1}{2} * (y - m)^T \alpha - \frac{1}{2} \log|K_y| - N/2 \log(2\pi)$ where α is the inverse matrix product of $\alpha = K_y^{-1} * (y - m)$.

Syntax

```
computeLogLikelihood(y, m, alpha, logDet, N)
```

Description

```
computeLogLikelihood(y, m, alpha, logDet, N)
```

Input Arguments

y column vector of regression targets.

m column vector of regression means.

y column vector of regression weights.

logDet real scalar of K matrix log determinate.

N number of observations. Number of reference angles.

Output Arguments

lml real scalar of log marginal likelihood.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [decomposeChol](#)
- [computeAlphaWeights](#)
- [initKernelParameters](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function lml = computeLogLikelihood(y, m, alpha, logDet, N)
    arguments
        % validate inputs as real column vectors
        y (:,1) double {mustBeReal, mustBeVector}
        % m can be zero if zero gpr runs
        m (:,1) double {mustBeReal, mustBeVector}
        alpha (:,1) double {mustBeReal, mustBeVector, mustBeEqualSize(y, alpha)}
        % validate inputs as real scalar
        logDet (1,1) double {mustBeReal}
        N (1,1) double {mustBeReal}
    end

    % get residual of targets and mean
    residual = y - m;
```

```
% compute log marginal likelihood
lml = -0.5 * (residual' * alpha + logDet + N * log(2 * pi));
end

% Custom validation functions
function mustBeEqualSize(a, b)
    if ~isequal(length(a), length(b))
        eid = 'Size:notEqual';
        msg = 'Vectors must be the same length.';
        throwAsCaller(MException(eid,msg))
    end
end
```

Published with MATLAB® R2020b

estimateBeta

Find beta coefficients to basis matrix H and the current set of hyperparameters theta as vector of s2f and sl, s2n represented by the current inverse of noisy covariance matrix Ky^-1 and the feature target vector y of the observations. It calculates several inverse Matrix products so instead passing the current Ky the function uses the infront decomposed lower triangle matrix L of Ky.

Syntax

```
[beta, alpha0]= estimateBeta(H, L, y)
```

Description

[beta, alpha0]= estimateBeta(H, L, y) compute polynom coefficients to solve mean approximation.

Input Arguments

H basis matrix of training data. Polynomial represents of training data.

L lower triangle matrix of decomposed K matrix.

y regression targets.

Output Arguments

beta beta coefficients for polynomial approximation with basis matrix**H**.

alpha0 regression weights based on regression targets**y**.

Requirements

- Other m-files required: None
- Subfunctions: chol, computeInverseMatrixProduct, computeTransposeInverseProduct
- MAT-files required: None

See Also

- [computeInverseMatrixProduct](#)
- [computeTransposeInverseProduct](#)
- [initKernelParameters](#)

Created on February 15. 2021 by Tobias Wulf. Copyright Tobias Wulf 2021.

```
function [beta, alpha0]= estimateBeta(H, L, y)
    % Ky^-1 * y
    alpha0 = computeInverseMatrixProduct(L, y);

    % H * Ky^-1 * HT
    alpha1 = computeTransposeInverseProduct(L, H');

    % (H * Ky^-1 * HT)^-1 * H
    L1 = chol(alpha1, 'lower');
    alpha2 = computeInverseMatrixProduct(L1, H);

    % ((H * (Ky^-1 * HT))^-1 * H) * (Ky^-1 * y)
    beta = alpha2 * alpha0;
end
```

.....

Published with MATLAB® R2020b

util

The main property of this module is to contain functions and classes which are used in different scenarios or reused in different modules. So they are providing a more general use case and not a specific one e.g. like a certain algebra function that almost or always computes the same use case. The util classificated source code solve module unrelated tasks.

removeFilesFromDir

Remove files from passed directory and identifier. Return a operation status if files are removed successful or not.

publishFilesFromDir

Publish m-files with Matlab built-in publish mechanism, scan m-files from directory recursively.

plotFunctions

A submodule to contain plot functions for general and specific use on data or datasets.

Created on October 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

removeFilesFromDir

Remove files from passed directory.

Syntax

```
removeStatus = removeFilesFromDir(directory)
removeStatus = removeFilesFromDir(directory, filePattern)
```

Description

removeStatus = removeFilesFromDir(directory) removes all files that are located in the passed directory and returns a logical 1 if the operation was successful or 0 if not. The directory argument must be char vector of 1xN and valid path to a existing directory.

removeStatus = removeFilesFromDir(directory, filePattern) removes all files in the located directory which matches the passed file pattern. The filePattern argument must be be char vector of 1xN. It is an optional argument with a default value of ".:", valid file patterns can be filenames which part replace names by * character before the dot and existing file extensions e.g. myfile_*.m or *.txt and so on.

Examples

```
d = fullfile('rootPath', 'subfolder')
rs = removeFileFromDir(d)

d = fullfile('rootPath', 'subfolder')
rs = removeFileFromDir(d, '*.mat')
```

Input Arguments

directory char vector, path directory in which to scan for files with file pattern and to delete found files.

filePattern char vector of file pattern with extension. Default is to delete all files. Possible patterns can be passed with filename parts with start operator as place holder.

Output Arguments

removeStatus logical scalar which is true if all files which matches the file pattern are deleted successfully from passed directory path.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [fullfile](#)
- [dir](#)
- [delete](#)
- [isfile](#)
- [isempty](#)
- [ismember](#)
- [mustBeFolder](#)
- [mustBeText](#)

Created on October 10. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function [removeStatus] = removeFilesFromDir(directory, filePattern)
    arguments
        % validate directory
        directory (1,:) char {mustBeFolder}
        % validate filePattern
        filePattern (1,:) char {mustBeText} = '*.*'
    end
    % parse pattern for dir
    parsePattern = fullfile(directory, filePattern);
    % parse directory, returns struct
    filesToRemove = dir(parsePattern);
    % delete files, transpose to loop through struct
    for file = filesToRemove'
        % check before delete
        filePath = fullfile(file.folder, file.name);
        if isfile(filePath)
            delete(filePath);
        end
    end
    % check if dir returns an empty struct now
    check = dir(parsePattern);
    removeStatus = isempty(check(~ismember({check.name}, {'.', '..'})));
end
```

Published with MATLAB® R2020b

publishFilesFromDir

Publish m-files from given directory with passed publishing options.

Syntax

```
publishFilesFromDir(directory, PublishOptions)
publishFilesFromDir(directory, PublishOptions, recursivly)
publishFilesFromDir(directory, PublishOptions, recursivly, verbose)
```

Description

publishFilesFromDir(directory, PublishOptions) publish m-files which are located in the passed directory with options from passed publishing options struct which is must be strictly formatted after given example from Matlab documentation.

publishFilesFromDir(directory, PublishOptions, recursive) publishing like described before but scan the directory recursively for m-files. Default is false for do not recursively.

publishFilesFromDir(directory, PublishOptions, recursive, verbose) with optional verbose set to true the published html files will be displayed in the prompt. Default is false.

Examples

```
directory = 'src';
PublishOptions = struct;
PublishOptions.outputDir = 'src/html';
PublishOptions.evalCode = false;
publishFilesFromDir(directory, PublishOptions, true)

load('config.mat', 'srcPath', 'PublishOptions')
publishFilesFromDir(srcPath, PublishOptions, true, true)
```

Input Arguments

directory char vector, path to directory where m-files are located to publish.

PublishOptions struct which contains publishing options for the Matlab publish function.

recursive logical scalar which directs the function to scan recursively for m-files in passed directory if true. Default is false.

verbose logical scalar which determines to display the filenames and path to published file if true. Default is false.

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [dir](#)
- [fullfile](#)
- [publish](#)

Created on October 31. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function publishFilesFromDir(directory, PublishOptions, recursive, verbose)
    arguments
        % validate directory if exist
        directory (1,:) char {mustBeFolder}
        % validate Publish options if it is a struct
        PublishOptions (1,1) struct {mustBeA(PublishOptions, ["struct"])}
        % validate recursive if logical
        recursive (1,1) logical {mustBeNumericOrLogical} = false
        verbose (1,1) logical {mustBeNumericOrLogical} = false
    end

    % file extension is always m-file
    ext = '*.m';

    % recursive parsing for files with dir function needs a certain regex
    if recursive
        pathPattern = fullfile(directory, '***', ext);
    else
        pathPattern = fullfile(directory, ext);
    end

    % scan directory for files returns a column struct with path fields
    files = dir(pathPattern);

    % transpose files struct and loop through for publish
    for file = files'
        % if not dir must be file
        if ~file.isdir
            % build path by struct field for recursive tree
            written = publish(fullfile(file.folder, file.name), PublishOptions);
            if verbose
                disp(written);
            end
        end
    end
end
```

Published with MATLAB® R2020b

plotFunctions

Project related reusable plots for datasets and results.

plotTDKTransferCurves

Plot transfer curves for bridge output voltages of TDK TAS2141.

plotKMZ60TransferCurves

Plot transfer curves for bridge output voltages of NXP KMZ60.

plotKMZ60CharField

Plot NXP KMZ60 characterization field and slice around 0, 5, 10 and 15 kA/m.

plotKMZ60CharDataset

Explore the basic dataset of characterized NXP AMR sensor KMZ60 and plot the dataset content to visualize the base of dipole simulations.

plotSimulationDatasetCircle

Plot circular path of Hx, Hy and Vcos, Vsin at each sensor array position. Normed to max overall array positions and normed to max at each array position.

plotSimulationCosSinStats

Statistical compare plot of Vcos and Vsin output voltages for each sensor array members.

plotSimulationSubset

Plot subset of angles and sensor array position from training or test dataset.

plotSingleSimulationAngle

Plot single rotation step of test or training dataset.

plotSimulationDataset

Plot simulation test or training dataset created by sensor array simulation.

plotTDKCharField

Plot TDK TAS2141 characterization field and slice around 0, 5, 10 and 15 kA/m.

plotTDKCharDataset

Explore the basic dataset of characterized TDK TMR Sensor TAS2141 and plot the dataset content to visualize the base of dipole simulations.

plotDipoleMagnet

Plot dipole magnet and its approximation as spherical magnet from constants set in config file. Plot manget in rest position.

Created on October 24. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

plotTDKCharDataset

Explore TDK TAS2141 characterization dataset and plot its content.

Syntax

```
plotTDKCharDataset()
```

Description

plotTDKCharDataset() explores the dataset and plot its content in three docked figure windows. Loads dataset location from config.mat.

Examples

```
plotTDKCharDataset();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat, data/config.mat

See Also

- [plot](#)
- [imagsc](#)
- [polarplot](#)

Created on October 24, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotTDKCharDataset()
try
    % load dataset path and dataset content into function workspace
    load('config.mat', 'PathVariables');
    load(PathVariables.tdkDatasetPath, 'Data', 'Info');
    %
    % close all;
    catch ME
        rethrow(ME)
    end

    % figure save path for different formats
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fig1Filename = 'tdk_magnetic_stimulus';
    fig1Path = fullfile(PathVariables.saveImagesPath, fig1Filename);

    fig2Filename = 'tdk_bridge_charistic';
    fig2Path = fullfile(PathVariables.saveImagesPath, fig2Filename);
```

```
% load needed data from dataset in to local variables for better handling
%%%%%%%%%%%%%%%
%%%%% check if modulation fits to following reconstructioning
if ~strcmp("triang", Info.MagneticField.Modulation)
    error("Modulation function is not triang.");
end
if ~(strcmp("cos", Info.MagneticField.CarrierHx) && ...
    strcmp("sin", Info.MagneticField.CarrierHy))
    error("Carrier functions are not cos or sin.");
end

% modulation frequency
fm = Info.MagneticField.ModulationFrequency;
% carrier frequency
fc = Info.MagneticField.CarrierFrequency;
% max and min amplitude
Hmax = Info.MagneticField.MaxAmplitude;
Hmin = Info.MagneticField.MinAmplitude;
% step range or window size for output picking
Hsteps = Info.MagneticField.Steps;
% resolution of H steps
Hres = Info.MagneticField.Resolution;
% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
Hz = Info.Units.Frequency;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, Info.Sensor, ...
    Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Sch\"uthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);
    
% load characterization data
Vcos = Data.SensorOutput.CosinusBridge;
Vsin = Data.SensorOutput.SinusBridge;
gain = Info.SensorOutput.BridgeGain;

% clear dataset all loaded
clear Data Info;
disp('Info:');
disp([infoStr; dateStr]);

% reconstruct magnetic stimulus and reduce the view for example plot by 10
%%%%%%%%%%%%%%%
% number of periods reduced by factor 10
reduced = 10;
nPeriods = fc / fm / reduced;
% number of samples for good looking 40 times nPeriods
nSamples = nPeriods * 400;
% half number of samples
nHalf = round(nSamples / 2);
% generate angle base
phi = linspace(0, nPeriods * 2 * pi, nSamples);
% calculate modulated amplitude, triang returns a column vector, transpose
Hmag = Hmax * triang(nSamples)';
% calculate Hx and Hy stimulus
Hx = Hmag .* cos(phi);
```

```

Hy = Hmag .* sin(phi);
% index for rising and falling stimulus
idxR = 1:nHalf;
idxF = nHalf:nSamples;
% find absolute min and max values in bridge outputs for uniform colormap
A = cat(3, Vcos.Rise, Vcos.Fall, Vcos.All, Vcos.Diff, Vsins.Rise, ...
        Vsins.Fall, Vsins.All, Vsins.Diff);
Vmax = max(A, [], 'all');
Vmin = min(A, [], 'all');
clear A;

% figure 1 magnetic stimulus
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fig1 = figure('Name', 'Magnetic Stimulus');
tiledlayout(fig1, 2, 2);

% title and description
disp("Title: Magnetic Stimulus Reconstructed H_x-/ H_y-Stimulus" + ...
    "in Reduced View");
disp("Description: Stimulus for characterization in H_x and H_y in " + ...
    "reduced period view by factor 10");
disp([["a) Triangle modulated cosine carrier for H_x stimulus."; ...
    "b) Triangle modulated sine carrier for H_x stimulus."; ...
    "c) Modulation trajectory for rising stimulus"; ...
    "d) Modulation trajectory for falling stimulus"]]);

% Hx stimulus
nexttile;
p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hx(idxR), phi(idxF), Hx(idxF));
set(p, {'Color'}, {'k', 'k', 'b', 'r'});
legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location', 'NorthEast');
xticks((0:0.25*pi:2*pi) * nPeriods);
xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$', ...
    '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
xlim([0 phi(end)]);
ylim([Hmin Hmax]);
xlabel('$\phi$ in rad, Periode $\times 10$');
ylabel(sprintf('$H_x(\phi)$ in %s', kApm));
title(sprintf('a) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm, Hz, fc, Hz));

% Hy stimulus
nexttile;
p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hy(idxR), phi(idxF), Hy(idxF));
set(p, {'Color'}, {'k', 'k', 'b', 'r'});
legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location', 'NorthEast');
xticks((0:0.25*pi:2*pi) * nPeriods);
xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$', ...
    '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
xlim([0 phi(end)]);
ylim([Hmin Hmax]);
xlabel('$\phi$ in rad, Periode $\times 10$');
ylabel(sprintf('$H_y(\phi)$ in %s', kApm));
title(sprintf('b) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm, Hz, fc, Hz));

% polar for rising modulation
nexttile;
polarplot(phi(idxR), Hmag(idxR), 'b');
p = gca;
p.ThetaAxisUnits = 'radians';
title('c) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $0 < \phi < 320\pi$');

```

```
% polar for rising modulation
nexttile;
polarplot(phi(idxF), Hmag(idxF), 'r');
p = gca;
p.ThetaAxisUnits = 'radians';
title('d) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $320<\phi<640\pi$');

% figure 2 cosinus bridge outputs
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
fig2 = figure('Name', 'Cosine and Sine Bridge', 'Position', [0 0 33 30]);

tiledlayout(fig2, 2, 2);

% title and description
disp("Title: Cosine and Sine Bridge. Measured Bridge Outputs" + ...
    " of Corresponding H_x-/ H_y-Amplitudes");
disp("Description: " + sprintf("H_x, H_y in %s, %d Steps in %.4f %s", ...
    kApM, Hsteps, Hres, kApM));
disp(["a) Cosine Bridge Rising H-Amplitudes"; ...
    "b) Cosine Bridge Falling H-Amplitudes"; ...
    "c) Sine Bridge Rising H-Amplitudes"; ...
    "d) Sine Bridge Falling H-Amplitudes"]);

colormap('jet');

% cosinus bridge recorded during rising stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Rise);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos.Rise));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('a) $V_{\cos}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% cosinus bridge recorded during falling stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Fall);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos.Fall));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('b) $V_{\cos}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% sinus bridge recorded during rising stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vsins.Rise);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsins.Rise));
caxis([Vmin, Vmax]);
```

```

xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$_H_x$ in kA/m');
ylabel('$_H_y$ in kA/m');
title('c) $V_{\sin}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% sinus bridge recorded during falling stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Fall);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsin.Fall));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$_H_x$ in kA/m');
ylabel('$_H_y$ in kA/m');
title('d) $V_{\sin}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% add colorbar and place it overall plots
cb = colorbar;
cb.Layout.Tile = 'east';
cb.Label.String = sprintf(... 
    '$V(H_x, H_y)$ in %s, Gain $ = %.1f$', mV, gain);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 24;

%
yesno = input('Save? [y/n]: ', 's');
if strcmp(yesno, 'y')
    % save results of figure 1
    savefig(fig1, fig1Path);
    print(fig1, fig1Path, '-dsvg');
    print(fig1, fig1Path, '-depsc', '-tiff', '-loose');
    print(fig1, fig1Path, '-dpdf', '-loose', '-fillpage');

    %
    % save results of figure 2
    savefig(fig2, fig2Path);
    print(fig2, fig2Path, '-dsvg');
    print(fig2, fig2Path, '-depsc', '-tiff', '-loose');
    print(fig2, fig2Path, '-dpdf', '-loose', '-fillpage');
end
close(fig1)
close(fig2)
end

```

plotTDKCharField

Explore TDK TAS2141 characterization field.

Syntax

```
plotTDKCharField()
```

Description

plotTDKCharField() explore characterization field of TDK sensor.

Examples

```
plotTDKCharField();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat, data/config.mat

See Also

- [plotTDKCharDataset](#)

Created on October 28, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotTDKCharField()
    try
        % load dataset path and dataset content into function workspace
        load('config.mat', 'PathVariables');
        load(PathVariables.tdkDatasetPath, 'Data', 'Info');
        %      close all;
    catch ME
        rethrow(ME)
    end

    % load needed data from dataset in to local variables for better handling %%
    %%%%%%%%%%%%%%%%
    % get from user which field to investigate and limits for plateau
    fields = Info.SensorOutput.CosinusBridge.Determination;
    nFields = length(fields);
    fprintf('Choose 1 of %d fields ... \n', nFields);
    for i = 1:nFields
        fprintf('%s\t:\t(%d)\n', fields{i}, i);
    end

    iField = 1; % input('Choice: ');
    field = fields{iField};
```

```

pl = 5; % input('Plateu limit in kA/m: ');

Vcos = Data.SensorOutput.CosinusBridge.(field);
Vsin = Data.SensorOutput.SinusBridge.(field);
gain = Info.SensorOutput.BridgeGain;
HxScale = Data.MagneticField.hx;
HyScale = Data.MagneticField.hy;
Hmin = Info.MagneticField.MinAmplitude;
Hmax = Info.MagneticField.MaxAmplitude;

% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Sch\"uthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);

% clear dataset all loaded
clear Data Info;

% figure save path for different formats %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fName = sprintf("tdk_char_field_%s", field);
fPath = fullfile(PathVariables.saveImagesPath, fName);

% define slices and limits to plot %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
Hslice = [128 154 180 205]; % hit ca. 0, 5, 10, 15 kA/m
Hlims = [-pl pl];
mVpVlims = [-175 175];

% create figure for plots %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fig = figure('Name', 'Char Field', 'OuterPosition', [0 0 35 30]);
tiledlayout(fig, 2, 2);

% title and description
disp('Info:');
disp([infoStr; dateStr]);
fprintf('Title: TDK Characterization Field - %s\n', field);
disp('Description:');
disp(["a) Cosine Bridge Characteristic"; ...
    "b) Transfer slices for different const. H_y of Vcos"; ...
    "c) Sine Bridge Characteristic"; ...
    "d) Transfer slices for different const. H_x of Vsin"]);

% set colormap
colormap('jet');

% cosinus bridge %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(1);
im = imagesc(HxScale, HyScale, Vcos);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos));
xticks(-20:10:20);
yticks(-20:10:20);

```

```

axis square xy;

% plot lines for slice to investigate
hold on;
for i = Hslice
    yline(HyScale(i), 'k:', 'LineWidth', 3.5);
end
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('a) $V_{\cos}(H_x, H_y)$, Gain $ = %.1f$', gain));

cb = colorbar;
cb.Label.String = sprintf('$V_{\cos}$ in %s', mV);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 20;

% cosinus bridge slices %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(2);
% slices
p = plot(HxScale, Vcos(Hslice,:));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
    hold off;
end

legend(p, {'$H_y \approx 0$ kA/m', ...
           '$H_y \approx 5$ kA/m', ...
           '$H_y \approx 10$ kA/m', ...
           '$H_y \approx 15$ kA/m'}, ...
       'Location', 'SouthEast');
xlabel(sprintf('$H_x$ in %s', kApm));
title('b) $V_{\cos}(H_x, H_y)$, $H_y = $ const.');
ylim(mVpVlims);
xlim([Hmin Hmax]);

% sinus bridge %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(3);
im = imagesc(HxScale, HyScale, Vsin);
set(gca, 'Ydir', 'normal');
set(im, 'AlphaData', ~isnan(Vsin));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

% plot lines for slice to investigate
hold on;
for i = Hslice
    xline(HxScale(i), 'k:', 'LineWidth', 3.5);
end
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));

```

```

title(sprintf('c) $V_{sin}(H_x,H_y)$, Gain $ = %.1f$', gain));

cb = colorbar;
cb.Label.String = sprintf('$V_{sin}$ in %s', mV);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 20;

% sinus bridge slices %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(4);
% slices
p = plot(HxScale, Vsins(:,Hslice));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
    hold off;
end

legend(p, {'$H_x \approx 0$ kA/m', ...
            '$H_x \approx 5$ kA/m', ...
            '$H_x \approx 10$ kA/m', ...
            '$H_x \approx 15$ kA/m'},...
        'Location', 'SouthEast');
xlabel(sprintf('$H_y$ in %s', kApm));
title('d) $V_{sin}(H_x,H_y)$, $H_x = $ const.');
ylim(mVpVlims);
xlim([Hmin Hmax]);

% save results of figure %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     savefig(fig, fPath);
%     print(fig, fPath, '-dsvg');
%     print(fig, fPath, '-depsc', '-tiff', '-loose');
%     print(fig, fPath, '-dpdf', '-loose', '-fillpage');
% end
% close(fig)
end

```

Published with MATLAB® R2020b

plotTDKTransferCurves

Plot TDK TAS2141 characterization field transfer curves.

Syntax

```
plotTDKTransferCurves()
```

Description

plotTDKTransferCurves() plot characterization field of TDK sensor.

Examples

```
plotTDKTransferCurves();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat, data/config.mat

See Also

- [plotTDKCharField](#)

Created on December 05. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotTDKTransferCurves()
    try
        % load dataset path and dataset content into function workspace
        load('config.mat', 'PathVariables');
        load(PathVariables.tdkDatasetPath, 'Data', 'Info');
        %      close all;
    catch ME
        rethrow(ME)
    end

    % load needed data from dataset in to local variables for better handling %%
    %%%%%%%%%%%%%%%%
    % get from user which field to investigate and limits for plateau
    fields = Info.SensorOutput.CosinusBridge.Determination;
    nFields = length(fields);
    fprintf('Choose 1 of %d fields ... \n', nFields);
    for i = 1:nFields
        fprintf('%s\t:\t(%d)\n', fields{i}, i);
    end

    iField = 1; % input('Choice: ');
    field = fields{iField};
```

```

pl = 5; % input('Plateu limit in kA/m: ');

Vcos = Data.SensorOutput.CosinusBridge.(field);
Vsin = Data.SensorOutput.SinusBridge.(field);
gain = Info.SensorOutput.BridgeGain;
HxScale = Data.MagneticField.hx;
HyScale = Data.MagneticField.hy;
Hmin = Info.MagneticField.MinAmplitude;
Hmax = Info.MagneticField.MaxAmplitude;

% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Sch\"uthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);

% clear dataset all loaded
clear Data Info;

% figure save path for different formats %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fName = sprintf("tdk_transfer_curves_%s", field);
fPath = fullfile(PathVariables.saveImagesPath, fName);

% define slices and limits to plot %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
Hslice = 128; % hit ca. 0 kA/m
Hlims = [-pl pl];
mVpVlims = [-175 175];

% create figure for plots %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fig = figure('Name', 'Transfer Curves', 'OuterPosition', [0 0 33 30]);

tiledlayout(fig, 2, 2);

disp('Info:');
disp([infoStr; dateStr]);
disp('Title:');
fprintf('KMZ 60 Transfer Curves: %s\n', field);
disp(["a) Cosine Bridge Characteristic"; ...
    "b) Sine Bridge Characteristic"; ...
    "c) Transfer Curves for const. H_x = H_y = 0"]);;

% set colormap
colormap('jet');

% cosinus bridge %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(1);
im = imagesc(HxScale, HyScale, Vcos);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

```

```
% plot lines for slice to investigate
hold on;
yline(HyScale(Hslice), 'k:', 'LineWidth', 3.5);
plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth', 3.5);
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('a) $V_{\cos}(H_x,H_y)$, Gain $ = %.1f$', gain));

% sinus bridge %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(2);
im = imagesc(HxScale, HyScale, Vsinn);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsinn));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

% plot lines for slice to investigate
hold on;
xline(HxScale(Hslice), 'k:', 'LineWidth', 3.5);
plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth', 3.5);
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('b) $V_{\sin}(H_x,H_y)$, Gain $ = %.1f$', gain));

% colorbar for both %%%%%%%%%%%%%%
%%%%%%%%%%%%%
cb = colorbar;
cb.Label.String = sprintf('$V_{out}$ in %s', mV);
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 20;

% cosinus bridge slices %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile([1 2]);
% slices
p = plot(HxScale, Vcos(Hslice,:), HyScale, Vsinn(:, Hslice'));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 3.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 3.5);
    hold off;
end

legend(p, {sprintf('$V_{\cos}(H_x,H_y)$, $H_y \approx 0$ %s', kApm), ...
    sprintf('$V_{\sin}(H_x,H_y)$, $H_x \approx 0$ %s', kApm)}, ...
    'Location', 'SouthEast');
ylabel(sprintf('$V_{out}$ in %s', mV));
xlabel(sprintf('$H$ in %s', kApm));
title('c) Cosine and Sine Transfer Curves');
ylim(mVpVlims);
xlim([Hmin Hmax])
```

```
% save results of figure %%%%%%%%
%
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     savefig(fig, fPath);
%     print(fig, fPath, '-dsvg');
%     print(fig, fPath, '-depsc', '-tiff', '-loose');
%     print(fig, fPath, '-dpdf', '-loose', '-fillpage');
% end
% close(fig)

end
```

Published with MATLAB® R2020b

plotKMZ60CharDataset

Explore NXP KMZ60 characterization dataset and plot its content.

Syntax

```
plotKMZ60CharDataset()
```

Description

plotKMZ60CharDataset() explores the dataset and plot its content in three docked figure windows. Loads dataset location from config.mat.

Examples

```
plotKMZ60CharDataset();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat, data/config.mat

See Also

- [plotTDKCharDataset](#)

Created on December 05. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotKMZ60CharDataset()
    try
        % load dataset path and dataset content into function workspace
        load('config.mat', 'PathVariables');
        load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
    %
        close all;
    catch ME
        rethrow(ME)
    end

    % figure save path for different formats
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    fig1Filename = 'kmz60_magnetic_stimulus';
    fig1Path = fullfile(PathVariables.saveImagePath, fig1Filename);
    fig2Filename = 'kmz60_bridge_characteristic';
    fig2Path = fullfile(PathVariables.saveImagePath, fig2Filename);

    % load needed data from dataset in to local variables for better handling
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% check if modulation fits to following reconstructioning
if ~strcmp("triang", Info.MagneticField.Modulation)
    error("Modulation function is not triang.");
end
if ~(strcmp("cos", Info.MagneticField.CarrierHx) && ...
    strcmp("sin", Info.MagneticField.CarrierHy))
    error("Carrier functions are not cos or sin.");
end

% modulation frequency
fm = Info.MagneticField.ModulationFrequency;
% carrier frequency
fc = Info.MagneticField.CarrierFrequency;
% max and min amplitude
Hmax = Info.MagneticField.MaxAmplitude;
Hmin = Info.MagneticField.MinAmplitude;
% step range or window size for output picking
Hsteps = Info.MagneticField.Steps;
% resoulution of H steps
Hres = Info.MagneticField.Resolution;
% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
Hz = Info.Units.Frequency;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Schüthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);

% load characterization data
Vcos = Data.SensorOutput.CosinusBridge;
Vsin = Data.SensorOutput.SinusBridge;
gain = Info.SensorOutput.BridgeGain;

% clear dataset all loaded
clear Data Info;

disp('Info:');
disp([infoStr; dateStr]);

% reconstruct magnetic stimulus and reduce the view for example plot by 10
%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% number of periods reduced by factor 10
reduced = 10;
nPeriods = fc / fm / reduced;
% number of samples for good looking 40 times nPeriods
nSamples = nPeriods * 400;
% half number of samples
nHalf = round(nSamples / 2);
% generate angle base
phi = linspace(0, nPeriods * 2 * pi, nSamples);
% calculate modulated amplitude, triang returns a column vector, transpose
Hmag = Hmax * triang(nSamples)';
% calculate Hx and Hy stimulus
Hx = Hmag .* cos(phi);
Hy = Hmag .* sin(phi);
% index for rising and falling stimulus
```

```

idxR = 1:nHalf;
idxF = nHalf:nSamples;
% find absolute min and max values in bridge outputs for uniform colormap
A = cat(3, Vcos.Rise, Vcos.Fall, Vcos.All, Vcos.Diff, VsIn.Rise, ...
        VsIn.Fall, VsIn.All, VsIn.Diff);
Vmax = max(A, [], 'all');
Vmin = min(A, [], 'all');
clear A;

% figure 1 magnetic stimulus
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fig1 = figure('Name', 'Magnetic Stimulus');
tiledlayout(fig1, 2, 2);

% title and description
disp("Title: Magnetic Stimulus Reconstructed H_x-/ H_y-Stimulus" + ...
    "in Reduced View");
disp("Description: Stimulus for characterization in H_x and H_y in " + ...
    "reduced period view by factor 10");
disp([["a) Triangle modulated cosine carrier for H_x stimulus."; ...
    "b) Triangle modulated sine carrier for H_x stimulus."; ...
    "c) Modulation trajectory for rising stimulus"; ...
    "d) Modulation trajectory for falling stimulus"]]);

% Hx stimulus
nexttile;
p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hx(idxR), phi(idxF), Hx(idxF));
set(p, {'Color'}, {'k', 'k', 'b', 'r'});
legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location', 'NorthEast');
xticks((0:0.25*pi:2*pi) * nPeriods);
xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$', ...
    '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
xlim([0 phi(end)]);
ylim([Hmin Hmax]);
xlabel('$\phi$ in rad, Periode $\times 10$');
ylabel(sprintf('$H_x(\phi)$ in %s', kApm));
title(sprintf('a) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm, Hz, fc, Hz));

% Hy stimulus
nexttile;
p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hy(idxR), phi(idxF), Hy(idxF));
set(p, {'Color'}, {'k', 'k', 'b', 'r'});
legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location', 'NorthEast');
xticks((0:0.25*pi:2*pi) * nPeriods);
xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$', ...
    '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
xlim([0 phi(end)]);
ylim([Hmin Hmax]);
xlabel('$\phi$ in rad, Periode $\times 10$');
ylabel(sprintf('$H_y(\phi)$ in %s', kApm));
title(sprintf('b) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm, Hz, fc, Hz));

% polar for rising modulation
nexttile;
polarplot(phi(idxR), Hmag(idxR), 'b');
p = gca;
p.ThetaAxisUnits = 'radians';
title('c) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $0 < \phi < 320\pi$');

% polar for falling modulation
nexttile;

```

```

polarplot(phi(idxF), Hmag(idxF), 'r');
p = gca;
p.ThetaAxisUnits = 'radians';
title('d) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $320<\phi<640\pi$');

% figure 2 cosinus bridge outputs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fig2 = figure('Name', 'Cosine and Sine Bridge', 'Position', [0 0 33 30]);

tiledlayout(fig2, 2, 2);

% title and description
disp("Title: Cosine and Sine Bridge. Measured Bridge Outputs" + ...
    " of Corresponding H_x-/ H_y-Amplitudes");
disp("Description: " + sprintf("H_x, H_y in %s, %d Steps in %.4f %s", ...
    kApM, Hsteps, Hres, kApM));
disp(["a) Cosine Bridge Rising H-Amplitudes"; ...
    "b) Cosine Bridge Falling H-Amplitudes"; ...
    "c) Sine Bridge Rising H-Amplitudes"; ...
    "d) Sine Bridge Falling H-Amplitudes"]);

colormap('jet');

% cosinus bridge recorded during rising stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Rise);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos.Rise));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('a) $V_{\cos}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% cosinus bridge recorded during falling stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Fall);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos.Fall));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('b) $V_{\cos}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% sinus bridge recorded during rising stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], VsIn.Rise);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(VsIn.Rise));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);

```

```

axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('c) $V_{\sin}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% sinus bridge recorded during falling stimulus
nexttile;
im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Fall);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsin.Fall));
caxis([Vmin, Vmax]);
xlim([Hmin Hmax]);
ylim([Hmin Hmax]);
axis square xy;
xlabel('$H_x$ in kA/m');
ylabel('$H_y$ in kA/m');
title('d) $V_{\sin}(H_x, H_y)$');
yticks([-20 -10 0 10 20]);
xticks([-20 -10 0 10 20]);

% add colorbar and place it overall plots
cb = colorbar;
cb.Layout.Tile = 'east';
cb.Label.String = sprintf(...,
    '$V(H_x, H_y)$ in %s, Gain $ = %.1f$', mV, gain);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 24;

%
yesno = input('Save? [y/n]: ', 's');
%
if strcmp(yesno, 'y')
    %
    % save results of figure 1
    %
    savefig(fig1, fig1Path);
    %
    print(fig1, fig1Path, '-dsvg');
    %
    print(fig1, fig1Path, '-depsc', '-tiff', '-loose');
    %
    print(fig1, fig1Path, '-dpdf', '-loose', '-fillpage');
    %

    %
    % save results of figure 2
    %
    savefig(fig2, fig2Path);
    %
    print(fig2, fig2Path, '-dsvg');
    %
    print(fig2, fig2Path, '-depsc', '-tiff', '-loose');
    %
    print(fig2, fig2Path, '-dpdf', '-loose', '-fillpage');
%
end
%
close(fig1)
%
close(fig2)
end

```

plotKMZ60CharField

Explore NXP KMZ60 characterization field.

Syntax

```
plotKMZ60CharField()
```

Description

plotKMZ60CharField() explore characterization field of KMZ60 sensor.

Examples

```
plotKMZ60CharField();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat, data/config.mat

See Also

- [plotTDKCharField](#)

Created on December 05. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotKMZ60CharField()
    try
        % load dataset path and dataset content into function workspace
        load('config.mat', 'PathVariables');
        load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
    %
        close all;
    catch ME
        rethrow(ME)
    end

    % load needed data from dataset in to local variables for better handling %%
    %%%%%%%%%%%%%%%%
    % get from user which field to investigate and limits for plateau
    fields = Info.SensorOutput.CosinusBridge.Determination;
    nFields = length(fields);
    fprintf('Choose 1 of %d fields ... \n', nFields);
    for i = 1:nFields
        fprintf('%s\t:\t(%d)\n', fields{i}, i);
    end

    iField = 1; % input('Choice: ');
    field = fields{iField};
```

```

pl = 20; % input('Plateu limit in kA/m: ');

Vcos = Data.SensorOutput.CosinusBridge.(field);
Vsin = Data.SensorOutput.SinusBridge.(field);
gain = Info.SensorOutput.BridgeGain;
HxScale = Data.MagneticField.hx;
HyScale = Data.MagneticField.hy;
Hmin = Info.MagneticField.MinAmplitude;
Hmax = Info.MagneticField.MaxAmplitude;

% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Sch\"uthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);

% clear dataset all loaded
clear Data Info;

% figure save path for different formats %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fName = sprintf("kmz60_char_field_%s", field);
fPath = fullfile(PathVariables.saveImagePath, fName);

% define slices and limits to plot %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
Hslice = [128 154 180 205]; % hit ca. 0, 5, 10, 15 kA/m
Hlims = [-pl pl];
mVpVlims = [-8 8];

% create figure for plots %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fig = figure('Name', 'Char Field', 'OuterPosition', [0 0 35 30]);
tiledlayout(fig, 2, 2);

% title and description
disp('Info:');
disp([infoStr; dateStr]);
fprintf('Title: KMZ60 Characterization Field - %s\n', field);
disp('Description:');
disp(["a) Cosine Bridge Characteristic"; ...
    "b) Transfer slices for different const. H_y of Vcos"; ...
    "c) Sine Bridge Characteristic"; ...
    "d) Transfer slices for different const. H_x of Vsin"]);

% set colormap
colormap('jet');

% cosinus bridge %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(1);
im = imagesc(HxScale, HyScale, Vcos);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos));
xticks(-20:10:20);
yticks(-20:10:20);

```

```

axis square xy;

% plot lines for slice to investigate
hold on;
for i = Hslice
    yline(HyScale(i), 'k:', 'LineWidth', 3.5);
end
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('a) $V_{\cos}(H_x, H_y)$, Gain $ = %.1f$', gain));

cb = colorbar;
cb.Label.String = sprintf('$V_{\cos}$ in %s', mV);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 20;

% cosinus bridge slices %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(2);
% slices
p = plot(HxScale, Vcos(Hslice,:));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
    hold off;
end

legend(p, {'$H_y \approx 0$ kA/m', ...
           '$H_y \approx 5$ kA/m', ...
           '$H_y \approx 10$ kA/m', ...
           '$H_y \approx 15$ kA/m'}, ...
       'Location', 'SouthEast');
xlabel(sprintf('$H_x$ in %s', kApm));
title('b) $V_{\cos}(H_x, H_y)$, $H_y = $ const.');
ylim(mVpVlims);
xlim([Hmin Hmax]);

% sinus bridge %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(3);
im = imagesc(HxScale, HyScale, Vsin);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsin));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

% plot lines for slice to investigate
hold on;
for i = Hslice
    xline(HxScale(i), 'k:', 'LineWidth', 3.5);
end
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));

```

```

title(sprintf('c) $V_{sin}(H_x,H_y)$, Gain $ = %.1f$', gain));

cb = colorbar;
cb.Label.String = sprintf('$V_{sin}$ in %s', mV);
cb.Label.Interpreter = 'latex';
cb.TickLabelInterpreter = 'latex';
cb.Label.FontSize = 20;

% sinus bridge slices %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(4);
% slices
p = plot(HxScale, Vsins(:,Hslice));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
    hold off;
end

legend(p, {'$H_x \approx 0$ kA/m', ...
            '$H_x \approx 5$ kA/m', ...
            '$H_x \approx 10$ kA/m', ...
            '$H_x \approx 15$ kA/m'},...
        'Location', 'SouthEast');
xlabel(sprintf('$H_y$ in %s', kApm));
title('d) $V_{sin}(H_x,H_y)$, $H_x = $ const.');
ylim(mVpVlims);
xlim([Hmin Hmax]);

% save results of figure %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     savefig(fig, fPath);
%     print(fig, fPath, '-dsvg');
%     print(fig, fPath, '-depsc', '-tiff', '-loose');
%     print(fig, fPath, '-dpdf', '-loose', '-fillpage');
% end
% close(fig)
end

```

Published with MATLAB® R2020b

plotKMZ60TransferCurves

Plot NXP KMZ60 characterization field transfer curves.

Syntax

```
plotKMZ60TransferCurves()
```

Description

plotKMZ60TransferCurves() plot characterization field of KMZ 60 sensor.

Examples

```
plotKMZ60TransferCurves();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: none
- Subfunctions: none
- MAT-files required: data/NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat, data/config.mat

See Also

- [plotKMZ60CharField](#)

Created on December 05. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotKMZ60TransferCurves()
    try
        % load dataset path and dataset content into function workspace
        load('config.mat', 'PathVariables');
        load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
    %
        close all;
    catch ME
        rethrow(ME)
    end

    % load needed data from dataset in to local variables for better handling %%
    %%%%%%%%%%%%%%%%
    % get from user which field to investigate and limits for plateau
    fields = Info.SensorOutput.CosinusBridge.Determination;
    nFields = length(fields);
    fprintf('Choose 1 of %d fields ... \n', nFields);
    for i = 1:nFields
        fprintf('%s\t:\t(%d)\n', fields{i}, i);
    end

    iField = 1; % input('Choice: ');
    field = fields{iField};
```

```

pl = 20; % input('Plateu limit in kA/m: ');

Vcos = Data.SensorOutput.CosinusBridge.(field);
Vsin = Data.SensorOutput.SinusBridge.(field);
gain = Info.SensorOutput.BridgeGain;
HxScale = Data.MagneticField.hx;
HyScale = Data.MagneticField.hy;
Hmin = Info.MagneticField.MinAmplitude;
Hmax = Info.MagneticField.MaxAmplitude;

% get unit strings from
kApm = Info.Units.MagneticFieldStrength;
mV = Info.Units.SensorOutputVoltage;

% get dataset infos and format strings to place in figures
% subtitle string for all figures
infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
dateStr = join(["Created on", Info.Created, "by", 'Thorben Sch\"uthe', ...
    "and updated on", Info.Edited, "by", Info.Editor + "."]);

% clear dataset all loaded
clear Data Info;

% figure save path for different formats %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fName = sprintf("kmz60_transfer_curves_%s", field);
fPath = fullfile(PathVariables.saveImagesPath, fName);

% define slices and limits to plot %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
Hslice = 128; % hit ca. 0 kA/m
Hlims = [-pl pl];
mVpVlims = [-8 8];

% create figure for plots %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
fig = figure('Name', 'Transfer Curves', 'OuterPosition', [0 0 33 30]);

tiledlayout(fig, 2, 2);

disp('Info:');
disp([infoStr; dateStr]);
disp('Title:');
fprintf('KMZ 60 Transfer Curves: %s\n', field);
disp(["a) Cosine Bridge Characteristic"; ...
    "b) Sine Bridge Characteristic"; ...
    "c) Transfer Curves for const. H_x = H_y = 0"]);;

% set colormap
colormap('jet');

% cosinus bridge %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(1);
im = imagesc(HxScale, HyScale, Vcos);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vcos));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

```

```
% plot lines for slice to investigate
hold on;
yline(HyScale(Hslice), 'k:', 'LineWidth', 3.5);
plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth', 3.5);
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('a) $V_{\cos}(H_x,H_y)$, Gain $ = %.1f$', gain));

% sinus bridge %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(2);
im = imagesc(HxScale, HyScale, Vsinn);
set(gca, 'YDir', 'normal');
set(im, 'AlphaData', ~isnan(Vsinn));
xticks(-20:10:20);
yticks(-20:10:20);
axis square xy;

% plot lines for slice to investigate
hold on;
xline(HxScale(Hslice), 'k:', 'LineWidth', 3.5);
plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth', 3.5);
hold off;

xlabel(sprintf('$H_x$ in %s', kApm));
ylabel(sprintf('$H_y$ in %s', kApm));
title(sprintf('b) $V_{\sin}(H_x,H_y)$, Gain $ = %.1f$', gain));

% colorbar for both %%%%%%%%%%%%%%
%%%%%%%%%%%%%
cb = colorbar;
cb.Label.String = sprintf('$V_{out}$ in %s', mV);
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 20;

% cosinus bridge slices %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile([1 2]);
% slices
p = plot(HxScale, Vcos(Hslice,:), HyScale, Vsinn(:, Hslice'));

% plateau limits
if pl > 0
    hold on;
    xline(Hlims(1), 'k-.', 'LineWidth', 3.5);
    xline(Hlims(2), 'k-.', 'LineWidth', 3.5);
    hold off;
end

legend(p, {sprintf('$V_{\cos}(H_x,H_y)$, $H_y \approx 0$ %s', kApm), ...
    sprintf('$V_{\sin}(H_x,H_y)$, $H_x \approx 0$ %s', kApm)}, ...
    'Location', 'SouthEast');
ylabel(sprintf('$V_{out}$ in %s', mV));
xlabel(sprintf('$H$ in %s', kApm));
title('c) Cosine and Sine Transfer Curves');
ylim(mVpVlims);
xlim([Hmin Hmax])
```

```
% save results of figure %%%%%%%%
%
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     savefig(fig, fPath);
%     print(fig, fPath, '-dsvg');
%     print(fig, fPath, '-depsc', '-tiff', '-loose');
%     print(fig, fPath, '-dpdf', '-loose', '-fillpage');
% end
% close(fig)
%
end
```

Published with MATLAB® R2020b

plotDipoleMagnet

Plot dipole magnet which approximate a spherical magnet in its far field.

Syntax

`plotDipoleMagnet()`

Description

`plotDipoleMagnet()` load dipole constants from config.mat and construct magnet in its rest position in x and z layer for y = 0.

Examples

```
plotDipoleMagnet();
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files: generateDipoleRotationMoments.m, computeDipoleH0Norm.m, computeDipoleHField.m
 - Subfunctions: none
 - MAT-files required: data/config.mat

See Also

- `quiver`
 - `imagesc`
 - `streamslice`

Created on November 20. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```

rsp = DipoleOptions.sphereRadius;

% H-field magnitude to multiply of generated and relative normed dipole
Hmag = DipoleOptions.H0mag;

% Distance in zero position of the spherical magnet in which is imprinted
z0 = DipoleOptions.z0;

% Magnetic moment magnitude attach rotation to the dipole field
m0 = DipoleOptions.M0mag;

% clear dataset all loaded
clear DipoleOptions;

% set construction dipole magnet, all length in mm and areas mm^2
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
% number of samples for good looking
nSamples = 501;
% slice in view for quiver, every 25th point
slice = 25:25:nSamples-25;
% grid edge of meshgrid, square grid
xz = 15;
% y layer in coordinate system
y = 0;
% orientat of magnet along z axes
pz = pi/2:0.01:3*pi/2;
% distances magnet surface to display in plot
zd = -rsp:-z0:-xz;
xd = zeros(1, length(zd));
% scale grid to simulate
x = linspace(-xz, xz, nSamples);
z = linspace(xz, -xz, nSamples);
[X, Z, Y] = meshgrid(x, z, y);

% compute dipole and fetch to far field to approximate a sperical magnet
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
% generate dipole moment for 0°
m = generateDipoleRotationMoments(m0, 1);
% compute H-field norm factor imprieng H magnitude on dipole, rest position
H0norm = computeDipoleH0Norm(Hmag, m, [0; 0 ;-(z0 + rsp)]);
% compute dipole H-field for rest position in y = 0 layer
H = computeDipoleHField(X, Y, Z, m, H0norm);
% calculate magnitudes for each point in the grid
Habs = reshape(sqrt(sum(H.^2, 1)), nSamples, nSamples);
% split H-field in componets and reshape to meshgrid
Hx = reshape(H(1,:), nSamples, nSamples) ./ Habs;
Hy = reshape(H(2,:), nSamples, nSamples) ./ Habs;
Hz = reshape(H(3,:), nSamples, nSamples) ./ Habs;
% exculde value within the spherical magnet, < rsp
innerField = X.^2 + Z.^2 <= rsp.^2;
Habs(innerField) = NaN;
% find relevant magnitudes at anounced distances
Hd = interp2(X, Z, Habs, xd, zd, 'nearest', NaN);

% figure dipole magnet
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
%%%%%%%%%%%%%
fig = figure('Name', 'Dipole Magnet');

```

```
% plot magnitude as colormap
imagesc(x, z, log10(Habs), 'AlphaData', 1);
set(gca, 'YDir', 'normal');
colormap('jet');
shading flat;

% set colorbar to log10 scaling of map
cb = colorbar;
cb.Label.String = '$\log_{10}(|H|)$ in kA/m';
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 24;

hold on;
grid on;

% plot field lines
st = streamslice(X, Z, Hx, Hz, 'arrows', 'cubic');
set(st, 'Color', 'k');

% plot magnet with north and south pole
rectangle('Position', [-rsp -rsp 2*rsp 2*rsp], ...
    'Curvature', [1 1], 'LineWidth', 3.5);
semicrc = rsp.*[cos(pz); sin(pz)];
patch(semicrc(1,:), semicrc(2,:),'r');
patch(-semicrc(1,:), -semicrc(2,:),'g');
text(-1.25, 0, '\textbf{N}');
text(0.5, 0, '\textbf{S}'');

% additional figure text and lines
text(-(xz-1), -(xz-1), ...
    sprintf('$\mathbf{Y} = %.1f$ \textbf{mm}', y), 'Color', 'w');

% distance scale in -z direction for x = 0, distance from magnet surface
line(xd, zd, 'Marker', '_', 'LineStyle', '--', 'LineWidth', 3.5, ...
    'Color', 'w');

% place text along marker
for i = 2:length(zd)-1
    markstr = "\textbf{$\mathbf{mm}$ $ kA/m}";
    mark = sprintf(markstr, abs(zd(i))-rsp, Hd(i));
    text(0.5, zd(i), mark, 'Color', 'w');
end

% limits ticks and labels
xlim([-xz xz]);
ylim([-xz xz]);
xticks(-xz:xz);
yticks(-xz:xz);
labels = string(xticks);
labels(1:2:end) = "";
xticklabels(labels);
yticklabels(labels);
xlabel('$X$ in mm');
ylabel('$Z$ in mm');

% axis shape set
axis equal;
axis tight;
grid off;
```

```
% title and description
disp('Title: Approximated Spherical Magnet with Dipole Far Field');
disp("Description:");
fprintf("Sphere whith imprinted H-field magnitude of %.1f kA/m\n", Hmag); ...
fprintf("at distance d = %.1f mm with d_z = |z| - r\n", z0);
fprintf("and sphere radius r = %.1f mm\n", rsp);

% save results of figure
%   yesno = input('Save? [y/n]: ', 's');
%   if strcmp(yesno, 'y')
%     savefig(fig, figPath);
%     print(fig, figPath, '-dsvg');
%     print(fig, figPath, '-depsc', '-tiff', '-loose');
%     print(fig, figPath, '-dpdf', '-loose', '-fillpage');
%   end
%   close(fig)
end
```

Published with MATLAB® R2020b

plotSimulationDataset

Search for available trainings or test dataset and plot dataset. Follow user input dialog to choose which dataset and decide how many angles to plot. Save dataset content redered to an avi-file. Filename same as dataset.

Syntax

```
plotSimulationDataset()
```

Description

plotSimulationDataset() plot training or test dataset which are located in data/test or data/training. The function lists all datasets and the user must decide during user input dialog which dataset to plot and how many angles to visualize. It loads path from config.mat and scans for file automatically.

Examples

```
plotSimulationDataset()
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [generateSimulationDatasets](#)
- [sensorArraySimulation](#)
- [generateConfigMat](#)

Created on November 25. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotSimulationDataset()
    % scan for datasets and load needed configurations %%%%%%
    %%%
    try
        disp('Plot simulation dataset ...');
        close all;
        % load path variables
        load('config.mat', 'PathVariables');
        % scan for datasets
        TrainingDatasets = dir(fullfile(PathVariables.trainingDataPath, ...
            'Training_*.mat'));
        TestDatasets = dir(fullfile(PathVariables.testDataPath, 'Test_*.mat'));
        allDatasets = [TrainingDatasets; TestDatasets];
        % check if files available
        if isempty(allDatasets)
            error('No training or test datasets found.');
        end
    end
```

```

    catch ME
        rethrow(ME)
    end

    % display available datasets to user, decide which to plot %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    % number of datasets
    nDatasets = length(allDatasets);
    fprintf('Found %d datasets:\n', nDatasets)
    for i = 1:nDatasets
        fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
    end
    % get numeric user input to indicate which dataset to plot
    iDataset = input('Type number to choose dataset to plot to: ');
    % iDataset = 2;

    % load dataset and ask user which one and how many angles %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
try
    ds = load(fullfile(allDatasets(iDataset).folder, ...
        allDatasets(iDataset).name));
    % check how many angles in dataset and let user decide how many to
    % render in plot
    fprintf('Detect %d angles in dataset ... \n',...
        ds.Info.UseOptions.nAngles);
    nSubAngles = input('How many angles to you wish to plot: ');
    % nSubAngles = 120;
    % indices for data to plot, get sample distance for even distance
    sampleDistance = length(ds.Data.angles);
    % get subset of angles
    subAngles = downsample(ds.Data.angles, sampleDistance);
    nSubAngles = length(subAngles); % just ensure
    % get indices for subset data
    indices = find(ismember(ds.Data.angles, subAngles));
catch ME
    rethrow(ME)
end

    % create dataset figure for a subset or all angle %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
fig = figure('Name', 'Sensor Array', ...
    'NumberTitle', 'off', ...
    'WindowSize', 'normal', ...
    'MenuBar', 'none', ...
    'ToolBar', 'none', ...
    'Units', 'centimeters', ...
    'OuterPosition', [0 0 30 30], ...
    'PaperType', 'a4', ...
    'PaperUnits', 'centimeters', ...
    'PaperOrientation', 'landscape', ...
    'PaperPositionMode', 'auto', ...
    'DoubleBuffer', 'on', ...
    'RendererMode', 'manual', ...
    'Renderer', 'painters');

    tdl = tiledlayout(fig, 2, 2, ...
        'Padding', 'normal', ...
        'TileSpacing', 'compact');

    title(tdl, 'Sensor Array Simulation', ...

```

```

'FontWeight', 'normal', ...
'FontSize', 18, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

subline1 = "Sensor Array (%s) of $%d\\times%d$ sensors, an edge" + ...
    " length of $.1f$ mm, a rel. pos. to magnet surface of";
subline2 = " $(%.1f, %.1f, -(%.1f))$ in mm, a magnet" + ...
    " tilt of $.1f^\\circ$, a sphere radius of $.1f$ mm, a imprinted";
subline3 = "field strength of $.1f$ kA/m at $.1f$ mm" + ...
    " from sphere surface in z-axis, $%d$ rotation angles with a ";
subline4 = "step width of $.1f^\\circ$ and a resolution" + ...
    " of $.1f^\\circ$. Visualized is a subset of $%d$ angles in ";
subline5 = "sample distance of $%d$ angles. Based on %s" + ...
    " characterization reference %s.";
sub = [sprintf(subline1, ...
    ds.Info.SensorArrayOptions.geometry, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.edge); ...
sprintf(subline2, ...
    ds.Info.UseOptions.xPos, ...
    ds.Info.UseOptions.yPos, ...
    ds.Info.UseOptions.zPos, ...
    ds.Info.UseOptions.tilt, ...
    ds.Info.DipoleOptions.sphereRadius); ...
sprintf(subline3, ...
    ds.Info.DipoleOptions.H0mag, ...
    ds.Info.DipoleOptions.z0, ...
    ds.Info.UseOptions.nAngles); ...
sprintf(subline4, ...
    ds.Data.angleStep, ...
    ds.Info.UseOptions.angleRes, ...
    nSubAngles);
sprintf(subline5, ...
    sampleDistance, ...
    ds.Info.CharData, ...
    ds.Info.UseOptions.BridgeReference)];

subtitle(tdl, sub, ...
    'FontWeight', 'normal', ...
    'FontSize', 14, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

% get subset of needed data to plot, only one load %%%%%%%%%%%%%%
%%%%%
N = ds.Info.SensorArrayOptions.dimension;
X = ds.Data.X;
Y = ds.Data.Y;
Z = ds.Data.Z;

% calc limits of plot 1
maxX = ds.Info.UseOptions.xPos + ds.Info.SensorArrayOptions.edge;
maxY = ds.Info.UseOptions.yPos + ds.Info.SensorArrayOptions.edge;
minX = ds.Info.UseOptions.xPos - ds.Info.SensorArrayOptions.edge;
minY = ds.Info.UseOptions.yPos - ds.Info.SensorArrayOptions.edge;

% calculate colormap to identify scatter points
c=zeros(N,N,3);
for i = 1:N
    for j = 1:N

```

```

        c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
    end
end
c = squeeze(reshape(c, N^2, 1, 3));

% load offset voltage to subtract from cosinus, sinus voltage
Voff = ds.Info.SensorArrayOptions.Voff;

% plot sensor grid in x and y coordinates and constant z layer %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax1 = nexttile(1);
% plot each coordinate in loop to create a special shading constant
% reliable to orientation for all matrice
hold on;
scatter(X(:, Y(:, [], c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

% text and labels
text(minX+0.2, minY+0.2, ...
    sprintf('$Z = %.1f$ mm', z(1)), ...
    'Color', 'k', ...
    'FontSize', 16, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

xlabel('$X$ in mm', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

ylabel('$Y$ in mm', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

title(sprintf('Sensor Array %d\ttimes%d$', N, N), ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

hold off;

% plot rotation angles in polar view %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(2);
% plot all angles grayed out
polarscatter(ds.Data.angles/180*pi, ...
    ones(1, ds.Info.UseOptions.nAngles), ...
    [], [0.8 0.8 0.8], 'filled');

% radius ticks and label
rticks(1);

```

```

rticklabels("");
hold on;

% plot subset of angles
% polarscatter(subAngles/180*pi, ones(1, nSubAngles), ...
%   'k', 'LineWidth', 0.8);
ax2 = gca;

% axis shape
axis tight;

% text an labels
% init first rotation step label
tA = text(2/3*pi, 1.5, ...
    '$\backslash\theta$', ...
    'Color', 'b', ...
    'FontSize', 16, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

title('Rotation around Z-Axis in Degree', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

hold off;

% Cosinus bridge outputs for rotation step %%%%%%%%%%%%%%
%%%%%%%%%%%%%
ax3 = nexttile(3);
hold on;

% set colormap
colormap('gray');

% plot cosinus reference, set NaN values to white color, orient Y to normal
imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VcosRef);
set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;
axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('$H_x$ in kA/m', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

ylabel('$H_y$ in kA/m', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

title('$V_{\cos}(H_x, H_y)$', ...
    'FontWeight', 'normal', ...

```

```

'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% add colorbar and place it
cb1 = colorbar;
cb1.Label.String = sprintf(... 
    '$V_{cos}(H_x, H_y)$ in V, $V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
    ds.Info.SensorArrayOptions.Vcc, ds.Info.SensorArrayOptions.Voff);
cb1.Label.Interpreter = 'latex';
cb1.Label.FontSize = 12;

hold off;

% Sinus bridge outputs for rotation step %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax4 = nexttile(4);
hold on;

% set colormap
colormap('gray');

% plot sinus reference, set NaN values to white color, orient Y to normal
imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VsinRef);
set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;
axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('$H_x$ in kA/m', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

ylabel('$H_y$ in kA/m', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

title('$V_{sin}(H_x, H_y)$', ...
    'FontWeight', 'normal', ...
    'FontSize', 12, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

% add colorbar and place it
cb2 = colorbar;
cb2.Label.String = sprintf(... 
    '$V_{sin}(H_x, H_y)$ in V, $V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
    ds.Info.SensorArrayOptions.Vcc, ds.Info.SensorArrayOptions.Voff);
cb2.Label.Interpreter = 'latex';
cb2.Label.FontSize = 12;

hold off;

```

```
% zoom axes for scatter on cosinuns reference images %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(3);
ax5 = axes('Position', [0.07 0.02 0.19 0.19], 'XColor', 'r', 'YColor', 'r');
hold on;
axis square xy;
grid on;
hold off;

% draw everything prepared before start renewing frame wise and prepare for
% recording frames to video file %%%%%%%%%%%%%%
%%%%%%%%%%%%%
% draw frame
drawnow;

% get file path and change extension
[~, fName, ~] = fileparts(ds.Info.filePath);
fPath = PathVariables.saveImagesPath;

% string allows simple cat ops
VW = VideoWriter(fullfile(fPath, fName + ".avi"), ...
    "Uncompressed AVI");

% scale frame rate on 10 second movies, ensure at least 1 fps
fr = floor(nSubAngles / 10) + 1;
VW.FrameRate = fr;

% open video file, ready to record frames
open(VW)

% loop through subset angle dataset and renew plots %%%%%%%%%%%%%%
%%%%%%%%%%%%%
for i = indices
    % H load subset
    Hx = ds.Data.Hx(:,:,i);
    Hy = ds.Data.Hy(:,:,i);
    % get min max
    maxHx = max(Hx, [], 'all');
    maxHy = max(Hy, [], 'all');
    minHx = min(Hx, [], 'all');
    minHy = min(Hy, [], 'all');
    dHx = abs(maxHx - minHx);
    dHy = abs(maxHy - minHy);

    % load V subset
    Vcos = ds.Data.Vcos(:,:,i) - Voff;
    Vsin = ds.Data.Vsin(:,:,i) - Voff;
    angle = ds.Data.angles(i);

    % lock plots
    hold(ax1, 'on');
    hold(ax2, 'on');
    hold(ax3, 'on');
    hold(ax4, 'on');
    hold(ax5, 'on');

    % update plot 1
    qH = quiver(ax1, X, Y, Hx, Hy, 0.5, 'b');
    qV = quiver(ax1, X, Y, Vcos, Vsin, 0.5, 'r');
    legend([qH qV], {'$quiver(H_x,H_y)$', ...
        '$quiver(V_{cos}-V_{off},V_{sin}-V_{off})$'}, ...
        'Location', 'North');
    drawnow;
```

```

'FontWeight', 'normal', ...
'FontSize', 9, ...
'FontName', 'Times', ...
'Interpreter', 'latex', ...
'Location', 'NorthEast');

% update plot 2
tA.String = sprintf('$%.1f^\circ', angle);
pA = polarscatter(ax2, angle/180*pi, 1, 'b', 'filled', ...
    'MarkerEdgeColor', 'k', 'LineWidth', 0.8);

% update plot 3 and 4
sC = scatter(ax3, Hx(:, ), Hy(:, ), 5, c, 'filled', ...
    'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);
sS = scatter(ax4, Hx(:, ), Hy(:, ), 5, c, 'filled', ...
    'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);

% calc position of scatter area frame and reframe
pos = [minHx - 0.3 * dHx, minHy - 0.3 * dHy, 1.6 * dHx, 1.6 * dHy];
rtC = rectangle(ax3, 'Position', pos, 'LineWidth', 1, ...
    'EdgeColor', 'r');
rts = rectangle(ax4, 'Position', pos, 'LineWidth', 1, ...
    'EdgeColor', 'r');

% update plot 5 (zoom)
sZ = scatter(ax5, Hx(:, ), Hy(:, ), [], c, 'filled', ...
    'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);
xlim(ax5, [pos(1) maxHx + 0.3 * dHx])
ylim(ax5, [pos(2) maxHy + 0.3 * dHy])

% release plots
hold(ax1, 'off');
hold(ax2, 'off');
hold(ax3, 'off');
hold(ax4, 'off');
hold(ax5, 'off');

% draw frame
drawnow;

% record frame to file
frame = getframe(fig);
writeVideo(VW, frame);

% delete part of plots to renew for current angle, delete but last
if i ~= indices(end)
    delete(qH);
    delete(qV);
    delete(pA);
    delete(rtC);
    delete(rts);
    delete(sC);
    delete(sS);
    delete(sZ);
end
end
% close video file
close(VW)
close(fig)

```

```
end
```

Published with MATLAB® R2020b

plotSingleSimulationAngle

Search for available trainings or test dataset and plot dataset. Follow user input dialog to choose which dataset and decide how many angles to plot. Plot single Angle and save figure to file. File name same as dataset with attach angle index.

Syntax

```
plotSingleSimulationAngle()
```

Description

plotSingleSimulationAngle() plot training or test dataset which are located in data/test or data/training. The function lists all datasets and the user must decide during user input dialog which dataset to plot and which angle to visualize to. It loads path from config.mat and scans for file automatically.

Examples

```
plotSingleSimulationAngle()
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [generateSimulationDatasets](#)
- [sensorArraySimulation](#)
- [generateConfigMat](#)

Created on November 28. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotSingleSimulationAngle()
    % scan for datasets and load needed configurations %%%%%%
    %%%
    try
        disp('Plot single simulation angle ...');
        close all;
        % load path variables
        load('config.mat', 'PathVariables');
        % scan for datasets
        TrainingDatasets = dir(fullfile(PathVariables.trainingDataPath, ...
            'Training_*.mat'));
        TestDatasets = dir(fullfile(PathVariables.testDataPath, 'Test_*.mat'));
        allDatasets = [TrainingDatasets; TestDatasets];
        % check if files available
        if isempty(allDatasets)
            error('No training or test datasets found.');
        end
    end
```

```

    catch ME
        rethrow(ME)
    end

    % display available datasets to user, decide which to plot %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    % number of datasets
    nDatasets = length(allDatasets);
    fprintf('Found %d datasets:\n', nDatasets)
    for i = 1:nDatasets
        fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
    end
    % get numeric user input to indicate which dataset to plot
    iDataset = input('Type number to choose dataset to plot to: ');
    % iDataset = 2;

    % load dataset and ask user which one and how many angles %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
try
    ds = load(fullfile(allDatasets(iDataset).folder, ...
        allDatasets(iDataset).name));
    % check how many angles in dataset and let user decide how many to
    % render in plot
    fprintf('Detect %d angles ([1:%d]) in dataset ... \n', ...
        ds.Info.UseOptions.nAngles, ds.Info.UseOptions.nAngles);
    fprintf('Resolution\t:\t%.1f\n', ds.Info.UseOptions.angleRes);
    fprintf('Step width\t:\t%.1f\n', ds.Data.angleStep);
    fprintf('Start angle\t:\t%.1f\n', ds.Data.angles(1))
    idx = input('Which angle do you wish to plot (enter index): ');
    angle = interp1(ds.Data.angles, idx, 'nearest');
catch ME
    rethrow(ME)
end

    % figure save path for different formats %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
fPath = PathVariables.saveImagesPath;

    % create dataset figure for a subset or all angle %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
fig = figure('Name', 'Sensor Array', ...
    'NumberTitle', 'off', ...
    'WindowStyle', 'normal', ...
    'Position', [4381 15 1244 983], ...
    'Units', 'pixels', ...
    'WindowState', 'maximized');

    tdl = tiledlayout(fig, 2, 2, ...
        'Padding', 'normal', ...
        'TileSpacing', 'compact');

    disp('Sensor Array Simulation');

    subline1 = "Sensor Array (%s) of %dx%d sensors, " + ...
        "an edge length of %.1f mm, a rel. pos. to magnet surface of";
    subline2 = " (%.1f, %.1f, -(%.1f)) in mm, a magnet tilt" + ...
        " of %.1f°, a sphere radius of %.1f mm, a imprinted";
    subline3 = "field strength of %.1f kA/m at %.1f mm from" + ...
        " sphere surface in z-axis, %d rotation angles with a ";
    subline4 = "step width of %.1f° and a resolution of" + ...

```

```

    " %.1f°. Visualized is rotation angle %d (%.1f°)$.";
subline5 = "Based on %s characterization reference %s.";
sub = [sprintf(subline1, ...
    ds.Info.SensorArrayOptions.geometry, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.edge); ...
sprintf(subline2, ...
    ds.Info.UseOptions.xPos, ...
    ds.Info.UseOptions.yPos, ...
    ds.Info.UseOptions.zPos, ...
    ds.Info.UseOptions.tilt, ...
    ds.Info.DipoleOptions.sphereRadius); ...
sprintf(subline3, ...
    ds.Info.DipoleOptions.H0mag, ...
    ds.Info.DipoleOptions.z0, ...
    ds.Info.UseOptions.nAngles); ...
sprintf(subline4, ...
    ds.Data.angleStep, ...
    ds.Info.UseOptions.angleRes, ...
    idx, angle)
sprintf(subline5, ...
    ds.Info.CharData, ...
    ds.Info.UseOptions.BridgeReference)];

disp(sub);

% get subset of needed data to plot, only one load %%%%%%%%%%%%%%
%%%%%%%%%%%%%
N = ds.Info.SensorArrayOptions.dimension;
X = ds.Data.X;
Y = ds.Data.Y;
Z = ds.Data.Z;

% calc limits of plot 1
maxX = ds.Info.UseOptions.xPos + ds.Info.SensorArrayOptions.edge;
maxY = ds.Info.UseOptions.yPos + ds.Info.SensorArrayOptions.edge;
minX = ds.Info.UseOptions.xPos - ds.Info.SensorArrayOptions.edge;
minY = ds.Info.UseOptions.yPos - ds.Info.SensorArrayOptions.edge;

% calculate colormap to identify scatter points
c=zeros(N,N,3);
for i = 1:N
    for j = 1:N
        c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
    end
end
c = squeeze(reshape(c, N^2, 1, 3));

% load offset voltage to subtract from cosinus, sinus voltage
Voff = ds.Info.SensorArrayOptions.Voff;

% plot sensor grid in x and y coordinates and constant z layer %%%%%%%%%%%%%%
%%%%%%%%%%%%%
ax1 = nexttile(1);
% plot each coordinate in loop to create a special shading constant
% reliable to orientation for all matrice
hold on;
scatter(X(:, Y(:, [], c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);

% axis shape and ticks

```

```

axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

% text and labels
text(minX+0.2, minY+0.2, ...
    sprintf('$Z = %.1f$ mm', Z(1)), ...
    'Color', 'k', ...
    'FontSize', 20, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

xlabel('$X$ in mm');

ylabel('$Y$ in mm');

title(sprintf('a) Sensor-Array %d\times%d', N, N));

hold off;

% plot rotation angles in polar view %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile(2);
% plot all angles grayed out
polarscatter(ds.Data.angles/180*pi, ones(1, ds.Info.UseOptions.nAngles), ...
    [], [0.8 0.8 0.8], 'filled');

% radius ticks and label
rticks(1);
rticklabels("");
hold on;

% plot subset of angles
% polarscatter(subAngles/180*pi, ones(1, nSubAngles),...
%     'k', 'LineWidth', 0.8);
ax2 = gca;

% axis shape
axis tight;

% text an labels
% init first rotation step label
tA = text(2/3*pi, 1.5, ...
    '$\backslash theta$', ...
    'Color', 'b', ...
    'FontSize', 20, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

title('b) Rotation Angle');

hold off;

% Cosinus bridge outputs for rotation step %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax3 = nexttile(3);
hold on;

% set colormap
colormap('gray');

```

```
% plot cosinus reference, set NaN values to white color, orient Y to normal
imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VcosRef);
set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;
axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('H_x in kA/m');

ylabel('H_y in kA/m');

title('c) V_{cos}(H_x, H_y) in V');

% add colorbar and place it
cb1 = colorbar;
cb1.Label.String = sprintf(
    '$V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
    ds.Info.SensorArrayOptions.Vcc, ds.Info.SensorArrayOptions.Voff);
cb1.TickLabelInterpreter = 'latex';
cb1.Label.Interpreter = 'latex';
cb1.Label.FontSize = 20;

hold off;

% Sinus bridge outputs for rotation step %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax4 = nexttile(4);
hold on;

% set colormap
colormap('gray');

% plot sinus reference, set NaN values to white color, orient Y to normal
imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VsinRef);
set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;
axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('H_x in kA/m');

ylabel('H_y in kA/m');

title('d) V_{sin}(H_x, H_y) in V');

% add colorbar and place it
cb2 = colorbar;
cb2.Label.String = sprintf(
    '$V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
    ds.Info.SensorArrayOptions.Vcc, ds.Info.SensorArrayOptions.Voff);
cb2.TickLabelInterpreter = 'latex';
```

```

cb2.Label.Interpreter = 'latex';
cb2.Label.FontSize = 20;

hold off;

% zoom axes for scatter on cosinus reference images %%%%%%%%%%%%%%
%%%%%%%%%%%%%
nexttile(3);
ax5 = axes('Position', [0.15 0.115 0.12 0.12], ...
    'XColor', 'r', 'YColor', 'r');
xticklabels(ax5, []);
yticklabels(ax5, []);
hold on;
axis square xy;
grid on;
hold off;

ax6 = axes('Position', [0.581 0.115 0.12 0.12], ...
    'XColor', 'r', 'YColor', 'r');
xticklabels(ax6, []);
yticklabels(ax6, []);
hold on;
axis square xy;
grid on;
hold off;

% plot angle into plots %%%%%%%%%%%%%%
%%%%%%%%%%%%%
% H load subset
Hx = ds.Data.Hx(:,:,idx);
Hy = ds.Data.Hy(:,:,idx);
% get min max
maxHx = max(Hx, [], 'all');
maxHy = max(Hy, [], 'all');
minHx = min(Hx, [], 'all');
minHy = min(Hy, [], 'all');
dHx = abs(maxHx - minHx);
dHy = abs(maxHy - minHy);

% load V subset
Vcos = ds.Data.Vcos(:,:,idx) - Voff;
Vsini = ds.Data.Vsin(:,:,idx) - Voff;
angle = ds.Data.angles(idx);

% lock plots
hold(ax1, 'on');
hold(ax2, 'on');
hold(ax3, 'on');
hold(ax4, 'on');
hold(ax5, 'on');
hold(ax6, 'on');

% update plot 1
qH = quiver(ax1, X, Y, Hx, Hy, 0.7, 'b');
qV = quiver(ax1, X, Y, Vcos, Vsini, 0.7, 'r');
legend([qH qV], {'$quiver(H_x,H_y)$', ...
    '$quiver(V_{cos}-V_{off},V_{sin}-V_{off})$'}, ...
    'FontSize', 14, ...
    'Location', 'NorthEast');

% update plot 2
tA.String = sprintf('$%.1f^\circ$', angle);

```

```

polarscatter(ax2, angle/180*pi, 1, 'b', 'filled', ...
    'MarkerEdgeColor', 'k', 'LineWidth', 0.8);

% update plot 3 and 4
scatter(ax3, Hx(:, ), Hy(:, ), 5, c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);
scatter(ax4, Hx(:, ), Hy(:, ), 5, c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);

% calc position of scatter area frame and reframe
pos = [minHx - 0.3 * dHx, minHy - 0.3 * dHy, 1.6 * dHx, 1.6 * dHy];
rectangle(ax3, 'Position', pos, 'LineWidth', 1.5, 'EdgeColor', 'r');
rectangle(ax4, 'Position', pos, 'LineWidth', 1.5, 'EdgeColor', 'r');

% update plot 5 (zoom)
scatter(ax5, Hx(:, ), Hy(:, ), [], c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);
xlim(ax5, [pos(1) maxHx + 0.3 * dHx])
ylim(ax5, [pos(2) maxHy + 0.3 * dHy])

% update plot 6 (zoom)
scatter(ax6, Hx(:, ), Hy(:, ), [], c, 'filled', 'MarkerEdgeColor', 'k', ...
    'LineWidth', 0.8);
xlim(ax6, [pos(1) maxHx + 0.3 * dHx])
ylim(ax6, [pos(2) maxHy + 0.3 * dHy])

% release plots
hold(ax1, 'off');
hold(ax2, 'off');
hold(ax3, 'off');
hold(ax4, 'off');
hold(ax5, 'off');
hold(ax6, 'off');

% save figure to file %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% get file path to save figure with angle index
[~, fName, ~] = fileparts(ds.Info.filePath);

% % save to various formats
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     fLabel = input('Enter file label: ', 's');
%     fName = fName + sprintf("_AnglePlot_%d_", idx) + fLabel;
%     savefig(fig, fullfile(fPath, fName));
%     print(fig, fullfile(fPath, fName), '-dsvg');
%     print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '-loose');
%     print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '-fillpage');
% end
% close(fig);
end

```

Published with MATLAB® R2020b

plotSimulationSubset

Search for available trainings or test dataset and plot dataset. Follow user input dialog to choose which dataset and decide which array elements to plot. Save created plot to file. Filename same as dataset with attached info.

Syntax

```
plotSimulationSubset()
```

Description

plotSimulationSubset() plot training or test dataset which are located in data/test or data/training. The function lists all datasets and the user must decide during user input dialog which dataset to plot and how many angles to visualize. It loads path from config.mat and scans for file automatically.

Examples

```
plotSimulationSubset()
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [generateSimulationDatasets](#)
- [sensorArraySimulation](#)
- [generateConfigMat](#)

Created on November 29. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

n

```
function plotSimulationSubset()
    % scan for datasets and load needed configurations %%%%%%
    %%%
try
    disp('Plot simulation dataset ...');
    close all;
    % load path variables
    load('config.mat', 'PathVariables');
    % scan for datasets
    TrainingDatasets = dir(fullfile(PathVariables.trainingDataPath, ...
        'Training_*.mat'));
    TestDatasets = dir(fullfile(PathVariables.testDataPath, 'Test_*.mat'));
    allDatasets = [TrainingDatasets; TestDatasets];
    % check if files available
```



```

    disp('Sensor Array Simulation');

    subline1 = "Sensor Array (%s) of %dx%d sensors," + ...
        " an edge length of %.1f mm, a rel. pos. to magnet surface of";
    subline2 = " (%.1f, %.1f, -(%.1f)) in mm, a magnet tilt" + ...
        " of %.1f°, a sphere radius of %.1f mm, a imprinted";
    subline3 = "field strength of %.1f kA/m at %.1f mm from" + ...
        " sphere surface in z-axis, %d rotation angles with a ";
    subline4 = "step width of %.1f° and a resolution" + ...
        " of %.1f°. Visualized is a subset.";
    subline5 = "Based on %s characterization reference %s.";
    sub = [sprintf(subline1, ...
        ds.Info.SensorArrayOptions.geometry, ...
        ds.Info.SensorArrayOptions.dimension, ...
        ds.Info.SensorArrayOptions.dimension, ...
        ds.Info.SensorArrayOptions.edge); ...
        sprintf(subline2, ...
        ds.Info.UseOptions.xPos, ...
        ds.Info.UseOptions.yPos, ...
        ds.Info.UseOptions.zPos, ...
        ds.Info.UseOptions.tilt, ...
        ds.Info.DipoleOptions.sphereRadius); ...
        sprintf(subline3, ...
        ds.Info.DipoleOptions.H0mag, ...
        ds.Info.DipoleOptions.z0, ...
        ds.Info.UseOptions.nAngles); ...
        sprintf(subline4, ...
        ds.Data.angleStep, ...
        ds.Info.UseOptions.angleRes)
    sprintf(subline5, ...
        ds.Info.CharData, ...
        ds.Info.UseOptions.BridgeReference)];

    disp(sub);

    % get subset of needed data to plot, only one load %%%%%%%%
    %%%
    N = ds.Info.SensorArrayOptions.dimension;
    X = ds.Data.X;
    Y = ds.Data.Y;
    Z = ds.Data.Z;

    % calc limits of plot 1
    a= ds.Info.SensorArrayOptions.edge;
    maxx = ds.Info.UseOptions.xPos + a * 0.66;
    maxy = ds.Info.UseOptions.yPos + a * 0.66;
    minx = ds.Info.UseOptions.xPos - a * 0.66;
    miny = ds.Info.UseOptions.yPos - a * 0.66;
    dp = a / (ds.Info.SensorArrayOptions.dimension - 1);
    x1 = ds.Info.UseOptions.xPos - a/2;
    x2 = ds.Info.UseOptions.xPos + a/2;
    y1 = ds.Info.UseOptions.yPos - a/2;
    y2 = ds.Info.UseOptions.yPos + a/2;

    % calculate colormap to identify scatter points
    c=zeros(N,N,3);
    for i = 1:N
        for j = 1:N
            c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
        end
    end

```

```

end
c = squeeze(reshape(c, N^2, 1, 3));
% reshape RGB for picking single sensors
R = reshape(c(:,1), N, N);
G = reshape(c(:,2), N, N);
B = reshape(c(:,3), N, N);

% load offset voltage to subtract from cosinus, sinus voltage
Voff = ds.Info.SensorArrayOptions.Voff;
Vcc = ds.Info.SensorArrayOptions.Vcc;

% plot sensor grid in x and y coordinates and constant z layer %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax1 = nexttile(6,[2 1]);
% plot each coordinate in loop to create a special shading constant
% reliable to orientation for all matrice
hold on;

scatter(X(:, Y(:, 48, [0.8 0.8 0.8], 'filled', ...
    'MarkerEdgeColor', 'k', 'LineWidth', 0.8);

for k = 1:length(xIdx)
    i = xIdx(k); j = yIdx(k);
    scatter(X(i,j), Y(i,j), 96, [R(i,j), G(i,j), B(i,j)], 'filled', ...
        'MarkerEdgeColor', 'k', 'LineWidth', 0.8);
end

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);
xticks(x1:dp:x2);
xticklabels(1:ds.Info.SensorArrayOptions.dimension);
yticks(y1:dp:y2);
yticklabels(ds.Info.SensorArrayOptions.dimension:-1:1);

xlabel('$j$');
ylabel('$i$');

title(sprintf('c) Sensor Array %d\times%d', N, N));

hold off;

% Cosinus bridge outputs for rotation step %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
ax3 = nexttile(1, [2 2]);
hold on;

% set colormap
colormap('gray');

% plot cosinus reference, set NaN values to white color, orient Y to normal
imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VcosRef);
set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;

```

```

axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('$H_x$ in kA/m');

ylabel('$H_y$ in kA/m');

title('a) $V_{\cos}(H_x, H_y)$');

% add colorbar and place it
cb1 = colorbar;
cb1.Label.String = 'in V';
cb1.TickLabelInterpreter = 'latex';
cb1.Label.Interpreter = 'latex';
cb1.Label.FontSize = 20;
hold off;

% Sinus bridge outputs for rotation step %%%%%%%%%%%%%%
%%%%%%%%%%%%%
ax4 = nexttile(13, [2 2]);
hold on;

% set colormap
colormap('gray');

% plot sinus reference, set NaN values to white color, orient Y to normal
imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.VsinRef);
set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
set(gca, 'YDir', 'normal')

% axis shape and ticks
axis square xy;
axis tight;
yticks(xticks);
grid on;

% test and labels
xlabel('$H_x$ in kA/m');

ylabel('$H_y$ in kA/m');

title('d) $V_{\sin}(H_x, H_y)$');

% add colorbar and place it
cb2 = colorbar;
cb2.Label.String = 'in V';
cb2.TickLabelInterpreter = 'latex';
cb2.Label.Interpreter = 'latex';
cb2.Label.FontSize = 20;

hold off;

% plot Vcos Vsin over angles %%%%%%%%%%%%%%
%%%%%%%%%%%%%
% axes limits
xlimits = [0 360];
ylimits = [min(cat(...
    3, ds.Data.VsinRef, ds.Data.VcosRef), [], 'all') - 0.1*Vcc, ...
    max(cat(3, ds.Data.VsinRef, ds.Data.VcosRef), [], 'all') + 0.1*Vcc];

```

```
% Vcos
ax5 = nexttile(3, [2 3]);
yline(Voff, 'k-.', 'LineWidth', 2.5);
xlim(xlimits);
ylim(ylimits);
grid on;

xlabel('$\alpha$ in $^\circ$');
%ylabel('in V');

title(sprintf...
    "b) $V_{cos}(\alpha) f." + ...
    " $V_{cc} = .1f$ V, $V_{off} = .2f$ V", Vcc, Voff));

% Vsin
ax6 = nexttile(15, [2 3]);
yline(Voff, 'k-.', 'LineWidth', 2.5);
xlim(xlimits);
ylim(ylimits);
grid on;

xlabel('$\alpha$ in $^\circ$');
%ylabel('in V');

title(sprintf("e) $V_{sin}(\alpha) f." + ...
    " $V_{cc} = .1f$ V, $V_{off} = .2f$ V", Vcc, Voff));

% loop through subset of dataset and renew plots %%%%%%%%%%%%%%
%%%%%%%%%%%%%
% lock plots
hold(ax3, 'on');
hold(ax4, 'on');
hold(ax5, 'on');
hold(ax6, 'on');

% loop over indices
for k = 1:length(xIdx)
    i = xIdx(k); j = yIdx(k);
    % H load subset
    Hx = squeeze(ds.Data.Hx(i,j,:));
    Hy = squeeze(ds.Data.Hy(i,j,:));
    % get min max

    % load V subset
    Vcos = squeeze(ds.Data.Vcos(i,j,:));
    Vsin = squeeze(ds.Data.Vsin(i,j,:));

    % update plot 3, 4, 5 and 6
    scatter(ax3, Hx, Hy, 5, [R(i,j), G(i,j), B(i,j)], 'filled');
    scatter(ax4, Hx, Hy, 5, [R(i,j), G(i,j), B(i,j)], 'filled');
    scatter(ax5, ds.Data.angles, Vcos, 8, [R(i,j), G(i,j), B(i,j)], ...
        'filled');
    scatter(ax6, ds.Data.angles, Vsin, 8, [R(i,j), G(i,j), B(i,j)], ...
        'filled');
end

% release plots
hold(ax3, 'off');
hold(ax4, 'off');
hold(ax5, 'off');
```

```
hold(ax6, 'off');

% save figure to file %%%%%%
% get file path to save figure with angle index
% [~, fName, ~] = fileparts(ds.Info.filePath);
%
% save to various formats
% yesno = input('Save? [y/n]: ', 's');
% if strcmp(yesno, 'y')
%     fLabel = input('Enter file label: ', 's');
%     fName = fName + "_SubsetPlot_" + fLabel;
%     savefig(fig, fullfile(fPath, fName));
%     print(fig, fullfile(fPath, fName), '-dsvg');
%     print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '-loose');
%     print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '-fillpage');
% end
% close(fig);
end
```

Published with MATLAB® R2020b

plotSimulationCosSinStats

Search for available trainings or test dataset and plot dataset. Follow user input dialog to choose which dataset to plot and statistics of cos sin. Save created plot to file. Filename same as dataset with attached info.

Syntax

```
plotSimulationCosSinStats()
```

Description

plotSimulationCosSinStats() plot training or test dataset which are located in data/test or data/training. The function lists all datasets and the user must decide during user input dialog which dataset to plot. It loads path from config.mat and scans for file automatically.

Examples

```
plotSimulationCosSinStats()
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [generateSimulationDatasets](#)
- [sensorArraySimulation](#)
- [generateConfigMat](#)

Created on November 30. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotSimulationCosSinStats()
    % scan for datasets and load needed configurations %%%%%%
    %%%
    try
        disp('Plot simulation dataset ...');
        close all;
        % load path variables
        load('config.mat', 'PathVariables');
        % scan for datasets
        TrainingDatasets = dir(fullfile(PathVariables.trainingDataPath, ...
            'Training_*.mat'));
        TestDatasets = dir(fullfile(PathVariables.testDataPath, 'Test_*.mat'));
        allDatasets = [TrainingDatasets; TestDatasets];
        % check if files available
        if isempty(allDatasets)
            error('No training or test datasets found.');
        end
    end
```

```

    catch ME
        rethrow(ME)
    end

    % display available datasets to user, decide which to plot %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    % number of datasets
nDatasets = length(allDatasets);
fprintf('Found %d datasets:\n', nDatasets)
for i = 1:nDatasets
    fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
end
% get numeric user input to indicate which dataset to plot
iDataset = input('Type number to choose dataset to plot to: ');

% load dataset and ask user which one and how many angles %%%%%%%%%%%%%%
%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
try
    ds = load(fullfile(allDatasets(iDataset).folder, ...
        allDatasets(iDataset).name));
    % check how many angles in dataset and let user decide how many to
    % render in plot
    fprintf('Detect %d angles in dataset ... \n', ...
        ds.Info.UseOptions.nAngles);
    nSubAngles = input('How many angles do you wish to plot: ');
    % nSubAngles = 120;
    % indices for data to plot, get sample distance for even distance
    sampleDistance = length(ds.Data.angles);
    % get subset of angles
    subAngles = downsample(ds.Data.angles, sampleDistance);
    nSubAngles = length(subAngles); % just ensure
    % get indices for subset data
    indices = find(ismember(ds.Data.angles, subAngles));
catch ME
    rethrow(ME)
end

% figure save path for different formats %%%%%%%%%%%%%%
%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
fPath = fullfile(PathVariables.saveImagePath);

% create dataset figure for a subset or all angle %%%%%%%%%%%%%%
%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
fig = figure('Name', 'Sensor Array', ...
    'NumberTitle', 'off', ...
    'WindowSize', 'normal', ...
    'MenuBar', 'none', ...
    'ToolBar', 'none', ...
    'Units', 'centimeters', ...
    'OuterPosition', [0 0 37 29], ...
    'PaperType', 'a4', ...
    'PaperUnits', 'centimeters', ...
    'PaperOrientation', 'landscape', ...
    'PaperPositionMode', 'auto', ...
    'DoubleBuffer', 'on', ...
    'RendererMode', 'manual', ...
    'Renderer', 'painters');

tdl = tiledlayout(fig, 2, 1, ...
    'Padding', 'compact', ...
    'TileSpacing', 'compact');

```

```

title(tdl, 'Sensor Array Simulation', ...
    'FontWeight', 'normal', ...
    'FontSize', 18, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

subline1 = "Sensor Array (%s) of %d\times%d$ sensors, " + ...
    "an edge length of %.1f$ mm, a rel. pos. to magnet surface of";
subline2 = " $(%.1f, %.1f, -(%!.1f))$ in mm, a magnet tilt" + ...
    " of %.1f^\circ, a sphere radius of %.1f$ mm, a imprinted";
subline3 = "field strength of %.1f$ kA/m at %.1f$ mm from" + ...
    " sphere surface in z-axis, %d$ rotation angles with a ";
subline4 = "step width of %.1f^\circ and a resolution of" + ...
    " %.1f^\circ. Visualized is a subset of %d$ angles in ";
subline5 = "sample distance of %d$ angles. Based on %s" + ...
    " characterization reference %s.";
sub = [sprintf(subline1, ...
    ds.Info.SensorArrayOptions.geometry, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.dimension, ...
    ds.Info.SensorArrayOptions.edge); ...
    sprintf(subline2, ...
        ds.Info.UseOptions.xPos, ...
        ds.Info.UseOptions.yPos, ...
        ds.Info.UseOptions.zPos, ...
        ds.Info.UseOptions.tilt, ...
        ds.Info.DipoleOptions.sphereRadius); ...
    sprintf(subline3, ...
        ds.Info.DipoleOptions.H0mag, ...
        ds.Info.DipoleOptions.z0, ...
        ds.Info.UseOptions.nAngles); ...
    sprintf(subline4, ...
        ds.Data.angleStep, ...
        ds.Info.UseOptions.angleRes, ...
        nSubAngles);
    sprintf(subline5, ...
        sampleDistance, ...
        ds.Info.CharData, ...
        ds.Info.UseOptions.BridgeReference)];

subtitle(tdl, sub, ...
    'FontWeight', 'normal', ...
    'FontSize', 14, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');

% get subset of needed data to plot, only one load %%%%%%%%%%%%%%
%%%%%%%%%%%%%
M = ds.Info.SensorArrayOptions.dimension^2;
% N = ds.Info.UseOptions.nAngles;
res = ds.Info.UseOptions.angleRes;
%angles = ds.Data.angles;
anglesIP = 0:res:360-res;

% load V subset and reshape for easier computing statistics
Vcos = squeeze(reshape(ds.Data.Vcos(:,:,indices), 1, M, nSubAngles));
Vsin = squeeze(reshape(ds.Data.Vsin(:,:,indices), 1, M, nSubAngles));

% load offset voltage to subtract from cosinus, sinus voltage
Voff = ds.Info.SensorArrayOptions.Voff;

```

```

Vcc = ds.Info.SensorArrayOptions.Vcc;

% compute statistics of Vcos Vsin %%%%%%
%%%%%%% interpolate with makima makes best results, ensure to kill nans for
% interpolate with makima makes best results, ensure to kill nans for
% fill otherwise fill strokes, use linestyle none for fill without frame
interpM = 'makima';
VcosMean = mean(Vcos, 1);
VcosMeanIP = interp1(subAngles, VcosMean, anglesIP, interpM);

VcosStd = std(Vcos, 1, 1);
VcosVar = var(Vcos, 1, 1); % std^2

% meanvariation coefficient in percent
VcosMVCp = mean(VcosStd ./ VcosMean) * 100;

VcosUpper1 = VcosMean + VcosStd;
VcosUpper2 = VcosMean + VcosVar;
VcosLower1 = VcosMean - VcosStd;
VcosLower2 = VcosMean - VcosVar;

VcosUpper1IP = interp1(subAngles, VcosUpper1, anglesIP, interpM);
VcosUpper1IP = fillmissing(VcosUpper1IP, 'previous');

VcosLower1IP = interp1(subAngles, VcosLower1, anglesIP, interpM);
VcosLower1IP = fillmissing(VcosLower1IP, 'previous');

VcosUpper2IP = interp1(subAngles, VcosUpper2, anglesIP, interpM);
VcosUpper2IP = fillmissing(VcosUpper2IP, 'previous');

VcosLower2IP = interp1(subAngles, VcosLower2, anglesIP, interpM);
VcosLower2IP = fillmissing(VcosLower2IP, 'previous');

VsInMean = mean(Vsin, 1);
VsInMeanIP = interp1(subAngles, VsInMean, anglesIP, interpM);

VsInStd = std(Vsin, 1, 1);
VsInVar = var(Vsin, 1, 1); % std^2

% meanvariation coefficient in percent
VsInMVCp = mean(VsinStd ./ VsInMean) * 100;

VsInUpper1 = VsInMean + VsInStd;
VsInUpper2 = VsInMean + VsInVar;
VsInLower1 = VsInMean - VsInStd;
VsInLower2 = VsInMean - VsInVar;

VsInUpper1IP = interp1(subAngles, VsInUpper1, anglesIP, interpM);
VsInUpper1IP = fillmissing(VsInUpper1IP, 'previous');

VsInLower1IP = interp1(subAngles, VsInLower1, anglesIP, interpM);
VsInLower1IP = fillmissing(VsInLower1IP, 'previous');

VsInUpper2IP = interp1(subAngles, VsInUpper2, anglesIP, interpM);
VsInUpper2IP = fillmissing(VsInUpper2IP, 'previous');

VsInLower2IP = interp1(subAngles, VsInLower2, anglesIP, interpM);
VsInLower2IP = fillmissing(VsInLower2IP, 'previous');

% plot Vcos VsIn over angles %%%%%%
%%%%%

```

```
% Vcos
nexttile;
hold on;

fillStdX = [anglesIP, fliplr(anglesIP)];
fillStdY = [VcosLower1IP, fliplr(VcosUpper1IP)];
fill(fillStdX, fillStdY, [0.95 0.95 0.95], 'LineStyle', 'none');

fillVarX = [anglesIP, fliplr(anglesIP)];
fillVarY = [VcosLower2IP, fliplr(VcosUpper2IP)];
fill(fillVarX, fillVarY, [0.7 0.7 0.7], 'LineStyle', 'none');

yline(Voff, 'k--');
scatter(subAngles, VcosUpper1, [], 'r*');
plot(anglesIP, VcosUpper1IP, 'r-.');
scatter(subAngles, VcosMean, [], 'm*');
plot(anglesIP, VcosMeanIP, 'm-.');
scatter(subAngles, VcosLower1, [], 'b*');
plot(anglesIP, VcosLower1IP, 'b-.');

hold off;
xlim([-res 360-res]);
%ylim(ylimits);
grid on;

xlabel('$\theta$ in Degree', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$V\{\cos\}(\theta)$ in V', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title(sprintf(...
"Compare $V_{\cos}(\theta)$ for each Array Member $V_{cc} = %.1f$" + ...
"V, $V_{off} = %.2f$ V, $\bar{\sigma}_{\mu} = %.2f$ perc.", ...
Vcc, Voff, VcosMVC));
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% Vsin
nexttile;
hold on;

fillStdX = [anglesIP, fliplr(anglesIP)];
fillStdY = [VsInLower1IP, fliplr(VsinUpper1IP)];
l1 = fill(fillStdX, fillStdY, [0.95 0.95 0.95], 'LineStyle', 'none');

fillVarX = [anglesIP, fliplr(anglesIP)];
fillVarY = [VsInLower2IP, fliplr(VsinUpper2IP)];
l2 = fill(fillVarX, fillVarY, [0.7 0.7 0.7], 'LineStyle', 'none');

l3 = yline(Voff, 'k--');
l4 = scatter(subAngles, VsInUpper1, [], 'r*');
l5 = plot(anglesIP, VsInUpper1IP, 'r-.');
```

```

16 = scatter(subAngles, VsinMean, [], 'm*');
17 = plot(anglesIP, VsinMeanIP, 'm-.');
18 = scatter(subAngles, VsinLower1, [], 'b*');
19 = plot(anglesIP, VsinLower1IP, 'b-.');

hold off;
xlim([-res 360-res]);
grid on;

xlabel('$\theta$ in Degree', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$V_{\sin}(\theta)$ in V', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title(sprintf(...
"Compare $V_{\sin}(\theta)$ for each Array Member $V_{cc} = %.1f$" + ...
" V, $V_{off} = %.2f$ V, $\bar{\sigma}_{\mu} = %.2f$ perc.", ...
Vcc, Voff, VsinMCP), ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% plot legend %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
l = [11 12 13 14 15 16 17 18 19];
L = legend(l, {'$2\sigma$', ...
'$2\sigma^2$', ...
'$V_{off}$', ...
'$U_{lim} = \mu + \sigma$', ...
sprintf('$_{lim}(U)$', interpM), ...
'$\mu(V)$', ...
sprintf('$_{lim}(\mu)$', interpM), ...
'$L_{lim} = \mu - \sigma$', ...
sprintf('$_{lim}(L)$', interpM)}, ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

L.Layout.Tile = 'east';

% save figure to file %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% get file path to save figure with angle index
[~, fName, ~] = fileparts(ds.Info.filePath);

% save to various formats
yesno = input('Save? [y/n]: ', 's');
if strcmp(yesno, 'y')
    fLabel = input('Enter file label: ', 's');
    fName = fName + "_StatsPlot_" + fLabel;
    savefig(fig, fullfile(fPath, fName));
    print(fig, fullfile(fPath, fName), '-dsvg');
    print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '-loose');
    print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '-fillpage');
end

```

```
    close(fig);  
end
```

Published with MATLAB® R2020b

plotSimulationDatasetCircle

Search for available trainings or test dataset and plot dataset. Follow user input dialog to choose which dataset to plot. Save created plot to file. Filename same as dataset with attached info.

Syntax

```
plotSimulationDatasetCircle()
```

Description

plotSimulationDatasetCircle() plot training or test dataset which are located in data/test or data/training. The function lists all datasets and the user must decide during user input dialog which dataset to plot. It loads path from config.mat and scans for file automatically.

Examples

```
plotSimulationDatasetCircle()
```

Input Arguments

None

Output Arguments

None

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: config.mat

See Also

- [generateSimulationDatasets](#)
- [sensorArraySimulation](#)
- [generateConfigMat](#)

Created on December 02. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
function plotSimulationDatasetCircle()
    % scan for datasets and load needed configurations %%%%%%
    %%%
try
    disp('Plot simulation dataset ...');
    close all;
    % load path variables
    load('config.mat', 'PathVariables');
    % scan for datasets
    TrainingDatasets = dir(fullfile(PathVariables.trainingDataPath, ...
        'Training_*.mat'));
    TestDatasets = dir(fullfile(PathVariables.testDataPath, 'Test_*.mat'));
    allDatasets = [TrainingDatasets; TestDatasets];
    % check if files available
    if isempty(allDatasets)
        error('No training or test datasets found.');
    end
end
```

```

    catch ME
        rethrow(ME)
    end

    % display available datasets to user, decide which to plot %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    % number of datasets
    nDatasets = length(allDatasets);
    fprintf('Found %d datasets:\n', nDatasets)
    for i = 1:nDatasets
        fprintf('%s\t:%t(%d)\n', allDatasets(i).name, i)
    end
    % get numeric user input to indicate which dataset to plot
    iDataset = input('Type number to choose dataset to plot to: ');

    % load dataset and ask user which one and how many angles %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    try
        ds = load(fullfile(allDatasets(iDataset).folder, ...
            allDatasets(iDataset).name));
    catch ME
        rethrow(ME)
    end

    % figure save path for different formats %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    fPath = PathVariables.saveImagesPath;

    % create dataset figure for a subset or all angle %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%% %%%%%%%%%%%%%%
    fig = figure('Name', 'Sensor Array', ...
        'NumberTitle', 'off', ...
        'WindowSize', 'normal', ...
        'MenuBar', 'none', ...
        'ToolBar', 'none', ...
        'Units', 'centimeters', ...
        'OuterPosition', [0 0 30 30], ...
        'PaperType', 'a4', ...
        'PaperUnits', 'centimeters', ...
        'PaperOrientation', 'landscape', ...
        'PaperPositionMode', 'auto', ...
        'DoubleBuffer', 'on', ...
        'RendererMode', 'manual', ...
        'Renderer', 'painters');

    tdl = tiledlayout(fig, 2, 2, ...
        'Padding', 'compact', ...
        'TileSpacing', 'compact');

    title(tdl, 'Sensor Array Simulation', ...
        'FontWeight', 'normal', ...
        'FontSize', 18, ...
        'FontName', 'Times', ...
        'Interpreter', 'latex');

    subline1 = "Sensor Array (%s) of $%d\\times%d$ sensors," + ...
        " an edge length of $%.1f$ mm, a rel. pos. to magnet surface of";
    subline2 = " $(%.1f, %.1f, -(%!.1f))$ in mm, a magnet tilt" + ...
        " of $%.1f^\\circ$, a sphere radius of $%.1f$ mm, a imprinted";
    subline3 = " field strength of $%.1f$ kA/m at $%.1f$ mm from" + ...
        " sphere surface in z-axis, $%d$ rotation angles with a ";

```

```

subline4 = "step width of %.1f^\circ and a resolution of" + ...
          " %.1f^\circ. Visualized are circular path of each array position ";
subline5 = "Based on %s characterization reference %s.";
sub = [sprintf(subline1, ...
               ds.Info.SensorArrayOptions.geometry, ...
               ds.Info.SensorArrayOptions.dimension, ...
               ds.Info.SensorArrayOptions.dimension, ...
               ds.Info.SensorArrayOptions.edge); ...
        sprintf(subline2, ...
               ds.Info.UseOptions.xPos, ...
               ds.Info.UseOptions.yPos, ...
               ds.Info.UseOptions.zPos, ...
               ds.Info.UseOptions.tilt, ...
               ds.Info.DipoleOptions.sphereRadius); ...
        sprintf(subline3, ...
               ds.Info.DipoleOptions.H0mag, ...
               ds.Info.DipoleOptions.z0, ...
               ds.Info.UseOptions.nAngles); ...
        sprintf(subline4, ...
               ds.Data.angleStep, ...
               ds.Info.UseOptions.angleRes)
        sprintf(subline5, ...
               ds.Info.CharData, ...
               ds.Info.UseOptions.BridgeReference)];

subtitle(tdl, sub, ...
         'FontWeight', 'normal', ...
         'FontSize', 14, ...
         'FontName', 'Times', ...
         'Interpreter', 'latex');

% get subset of needed data to plot, only one load %%%%%%%%%%%%%%
%%%%%%%%%%%%%
N = ds.Info.SensorArrayOptions.dimension;
M = ds.Info.UseOptions.nAngles;
Voff = ds.Info.SensorArrayOptions.Voff;
Vcos = ds.Data.Vcos - Voff;
Vsinn = ds.Data.Vsin - Voff;
Hx = ds.Data.Hx;
Hy = ds.Data.Hy;

% calculate norm values to align circles around position only for x,y
% directition for each sensor dot over all angles.
Vmagn = sqrt(Vcos.^2 + Vsinn.^2);
Hmagn = sqrt(Hx.^2 + Hy.^2);
%Hmagn = ds.Data.Habs;

% related to position, multiply scale factor for circle diameter
diameterFactor = 2 * N / ds.Info.SensorArrayOptions.edge;
MaxVmagnPos = max(Vmagn, [], 3) * diameterFactor;
MaxHmagnPos = max(Hmagn, [], 3) * diameterFactor;

% Overall maxima, scalar, multiply scale factor for circle diameter
MaxVmagnOA = max(Vmagn, [], 'all') * diameterFactor;
MaxHmagnOA = max(Hmagn, [], 'all') * diameterFactor;

% norm and scale volatages and filed strengths
VcosNorm = Vcos ./ MaxVmagnPos;
VcosScaled = Vcos / MaxVmagnOA;
VsinnNorm = Vsinn ./ MaxVmagnPos;
VsinnScaled = Vsinn / MaxVmagnOA;

```

```

HxNorm = Hx ./ MaxHmagPos;
HxScaled = Hx / MaxHmagOA;
HyNorm = Hy ./ MaxHmagPos;
HyScaled = Hy / MaxHmagOA;

% sensor array grid
X = ds.Data.X;
Y = ds.Data.Y;
Z = ds.Data.Z;

% calc limits of plot 1
maxX = ds.Info.UseOptions.xPos + 0.7 * ds.Info.SensorArrayOptions.edge;
maxY = ds.Info.UseOptions.yPos + 0.7 * ds.Info.SensorArrayOptions.edge;
minX = ds.Info.UseOptions.xPos - 0.7 * ds.Info.SensorArrayOptions.edge;
minY = ds.Info.UseOptions.yPos - 0.7 * ds.Info.SensorArrayOptions.edge;

% plot sensor grid in x and y coordinates %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
% plot each coordinate in loop to create a special shading constant
% reliable to orientation for all matrice
% calculate colormap to identify scatter points
c=zeros(N,N,3);
for i = 1:N
    for j = 1:N
        c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
    end
end
c = squeeze(reshape(c, N^2, 1, 3));
% reshape RGB for picking single sensors
R = reshape(c(:,1), N, N);
G = reshape(c(:,2), N, N);
B = reshape(c(:,3), N, N);

% Field strength scaled to overall maxima %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile;
hold on;
for i = 1:N
    for j = 1:N
        plot(squeeze(HxScaled(i, j, :)) + X(i,j), ...
              squeeze(HyScaled(i, j, :)) + Y(i,j), ...
              'Color', [R(i,j) G(i,j) B(i,j)], ...
              'LineWidth' , 1.5)
        line([X(i,j), HxScaled(i,j,1) + X(i,j)], ...
              [Y(i,j), HyScaled(i,j,1) + Y(i,j)], ...
              'Color','k','LineWidth',1.5)
    end
end

% scatter magnet x,y position (0,0,z)
scatter(0, 0, 32, 'r', 'filled');

hold off;

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

xlabel('$X$ in mm', ...

```

```

'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$Y$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title('$H_x$, $H_y$ Normed to Max overall Positions', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% Cosinus, sinus voltage scaled to overall maxima %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%
nexttile;
hold on;
for i = 1:N
    for j = 1:N
        plot(squeeze(VcosScaled(i, j, :)) + X(i,j), ...
            squeeze(VsinScaled(i, j, :)) + Y(i,j), ...
            'Color', [R(i,j) G(i,j) B(i,j)], ...
            'LineWidth', 1.5)
        line([X(i,j), VcosScaled(i,j,1) + X(i,j)], ...
            [Y(i,j), VsinScaled(i,j,1) + Y(i,j)], ...
            'Color','k','LineWidth',1.5)
    end
end

% scatter magnet x,y position (0,0,z)
scatter(0, 0, 32, 'r', 'filled');

hold off;

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

xlabel('$X$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$Y$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title('$V_{\cos}$, $V_{\sin}$ Normed to Max overall Positions', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

```

```
% Field strength normed each maxima at position %%%%%%%%%%%%%%%%
%%%%%%%
nexttile;
hold on;
for i = 1:N
    for j = 1:N
        plot(squeeze(HxNorm(i, j, :)) + X(i,j), ...
              squeeze(HyNorm(i, j, :)) + Y(i,j), ...
              'Color', [R(i,j) G(i,j) B(i,j)], ...
              'LineWidth' , 1.5)
        line([X(i,j), HxNorm(i,j,1) + X(i,j)], ...
              [Y(i,j), HyNorm(i,j,1) + Y(i,j)], ...
              'Color','k','LineWidth',1.5)
    end
end

% scatter magnet x,y position (0,0,z)
scatter(0, 0, 32, 'r', 'filled');

hold off;

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

xlabel('$X$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$Y$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title('$H_x$, $H_y$ Normed to Max at each Position', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% Cosinus, sinus voltage normed to each maxima at position %%%%%%%%%%%%%%
%%%%%%%
nexttile;
hold on;
for i = 1:N
    for j = 1:N
        plot(squeeze(VcosNorm(i, j, :)) + X(i,j), ...
              squeeze(VsinNorm(i, j, :)) + Y(i,j), ...
              'Color', [R(i,j) G(i,j) B(i,j)], ...
              'LineWidth' , 1.5)
        line([X(i,j), VcosNorm(i,j,1) + X(i,j)], ...
              [Y(i,j), VsinNorm(i,j,1) + Y(i,j)], ...
              'Color','k','LineWidth',1.5)
    end
end
```

```
% scatter magnet x,y position (0,0,z)
scatter(0, 0, 32, 'r', 'filled');

hold off;

% axis shape and ticks
axis square xy;
axis tight;
grid on;
xlim([minX maxX]);
ylim([minY maxY]);

xlabel('$X$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

ylabel('$Y$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

title('$V_{\cos}$, $V_{\sin}$ Normed to Max at each Positions', ...
'FontWeight', 'normal', ...
'FontSize', 12, ...
'FontName', 'Times', ...
'Interpreter', 'latex');

% save figure to file %%%%%%%%%%%%%%
%%%%%%%%%%%%%
% get file path to save figure with angle index
[~, fName, ~] = fileparts(ds.Info.filePath);

% save to various formats
yesno = input('Save? [y/n]: ', 's');
if strcmp(yesno, 'y')
    fLabel = input('Enter file label: ', 's');
    fName = fName + "_CirclePlot_" + fLabel;
    savefig(fig, fullfile(fPath, fName));
    print(fig, fullfile(fPath, fName), '-dsvg');
    print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '-loose');
    print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '-fillpage');
end
close(fig);
end
```

Published with MATLAB® R2020b

Datasets

Datasets are an appreciated way to save and reach done work and reuse it in progress. The easiest way to build and to use proper datasets in matlab are mat-files. They are easy to load and can be build by an script or function it just needs to save the variables from workspace. So latery save datasets can be used for futher calculations or to load certain configuration in to workspace and to solve task in a unified way.

TDK TAS2141 Characterization

The characterization dataset of the TDK TMR angular sensor as base dataset for sensor array dipol simulation. The dataset contains information about the stimulus which was used for characterization, the magnetic resolution or the sensor bridge outputs for Hx and Hy fields and bridge outputs corresponding to stimulus amplitudes in Hx and Hy direction.

NXP KMZ60 Characterization

The characterization dataset of the NXP AMR angular sensor is second characterization dataset which was acquired in the same way as the TDK dataset. The dataset is integrated in the simulation software after finish for TDK and comes along with option choose between both dataset. Bridge gain is introduced to handle internal amplification of bridge outputs.

Config Mat

Configuration dataset to control the main program from centralized config file. Includes any kind of configuration and parameters to load in function or script workspaces.

Training and Test Datasets

Sensor array simulation datasets for training and test purpose for angle prediction via gaussian processes.

Created on October 27, 2020 Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

TDK TAS2141 Characterization

TDK characterization as base of the sensor array simulation was done before the dataset is just modified in its structure and not in its values. An additional info struct is added which contains information about how the dataset was acquired and a data struct which contains the magnetic field resolution and the cosine and sine bridge images for variable Hx and Hy fieldstrengths. The raw dataset was acquired after the method Thorben Schüthe described in his IEEE paper for two-dimensional characterization of TMR angular sensors. The sensor characterized for both bridges a cosine and sine bridge. The bridges have a physically phase shift of 90° so the sensor is able to reference a superimposed magnetic field in x- and y-direction. The field was generated by a cross coil setup.

The resulting TMR characterization field abstracts a full rotation for cosine and sine output voltages by representing one maximum and minimum in the characterization fields. So circular path on the characterization fields generates one sinoid output related on current angle position of stimulus magnetic field.

See Also

- [IEEE Document 8706125](#)

Magnetic Stimulus

The right stimulus is the keynote for characterization records. It needs to have the ability record slow enough for quasi static recordings but is not allowed to be real static so the magnetic field is not interrupted during the recording. Therefore slow sinoid carrier functions with even slower amplitude modulation is choosen to provide a quasi static stimulus.

The carrier function for the Hx-field stimulus is related to the cosine bridge and so:

$$c_1(t) = \cos(\phi(t))$$

Due to the physically phase shift the Hy-field stimulus is related to sine:

$$c_2(t) = \sin(\phi(t))$$

Both carrier runs with same carrier frequency:

$$f_c = 3.2\text{Hz}$$

so they are executed with the phase vector over time:

$$\phi(t) = 2\pi f_c t$$

The carrier functions are triangle modulated to generate rising and falling amplitudes. The modulation frequency is set to:

$$f_m = 0.01\text{Hz}$$

Which generates a stimulus with 320 periods where 160 periods feeds a rising and falling record each multiplied with maximum fieldstrength amplitude:

$$m(t) = H_{max} \cdot tri(t) = H_{max} \cdot tri(2(t - t_0)f_m)$$

$$t_0 = \frac{1}{2f_m}$$

So the Hx- and Hy-field stimulus is described by:

$$H_x(t) = m(t) \cdot c_1(t)$$

$$H_y(t) = m(t) \cdot c_1(t)$$

The stimulus amplitude depending on the phase in polar coordinates can be displayed for both parts by:

$$H_{x,y}(\phi) = |H_{x,y}(\phi)| \cdot e^{j\phi} = m(t) \cdot e^{j\phi(t)}$$

Where a rising spiral runs from center outwards for:

$$0 < t < t_0$$

And a falling spiral of amplitudes from outwards to center for:

$$t_0 < t < \frac{1}{f_m}$$

Cosine Bridge Output

The record characterization raw data are one dimensional time discrete vectors. To fieldstrength images like down below the recorded data must be referenced backwards to driven stimulus of Hx- and Hy-direction. But at first the image size must be determined. Here fix size is set to 256 pixel for each direction. So it spans a vector for Hx- and Hy-direction from minimum -25 kA/m to maximum 25 kA/m in 256 steps with a resolution of 0.1961 kA/m. So it results into a 256x256 image. Now it runs for each point on the Hx- and Hy-axes and gets the record index of the stimulus as backreference to the recorded bridge signal and sets the pixel. That runs for the rising modulation amplitude and falling amplitude until every pixel is hit and ended up into a dimensional function image as:

$$V_{cos}(H_x, H_y) = [mV/V]$$

The information of the image is built up in rows. Reference Hx for constant Hy in each row. The method is also comparable to a histogram of Hx matches in the recorded sensor signal for one constant Hy and so on next histogram appends on the next row for the next Hy.

Sine Bridge Output

The sine characterization field is built up similar to the cosine images but the information lays now in the columns so the data is collected in each column for a constant Hx and variable Hy:

$$V_{sin}(H_x, H_y) = [mV/V]$$

Operating Point

To determine an operating point in sensor array simulation the characterization fields needs some further investigations in static Hy and variable Hx field strength for cosine bridge and vice versa for sine bridge references. The best results supports the "Rise" field because it has a wide linear plateau between -8.5 kA/m and 8.5 kA/m. So Rise characterization field is used in sensor array simulation. It is not needed to drive the sensor in saturation.

Dataset Structure

Info:

The dataset is separated in two main structs. The first one is filled with meta data. So it represents the file header. The struct is called Info and contains information about how the dataset is acquired. So the stimulus is reconstructable from that meta data.

- Created - string, contains dataset creation date
- Creator - string, contains dataset creator
- Edited - string, contains last time edited date
- Editor - string, contains last time editor
- Senor - string, sensor identification name e.g. TAS2141

- SensorType - string, kind of sensor e.g. Angular
- SensorTechnology - string, bridge technology e.g. AMR, GMR, TMR
- SensorManufacturer - string, producer or supplier e.g. NXP, TDK
- **MagneticField** - struct, contains further information about Hx and Hy
- **SensorOutput** - struct, contains information about sensor produced output and gathered image information
- **Units** - struct, contains information about used si units in dataset

- **MagneticField:**

- Modulation - string, contains modulation equivalent Matlab function
- ModulationFrequency - double, contains frequency of modulation in Hz
- CarrierFrequency - double, carrier frequency for both Hx and Hy carrier in Hz
- MaxAmplitude - double, maximum Hx and Hy field amplitude in kA/m
- MinAmplitude - double, minimum Hx and Hy field amplitude in kA/m
- Steps - double, Hx- and Hy-field steps to build characterization images
- Resolution - double, resolution of one step in kA/m
- CarrierHx - string, contains Hx carrier equivalent Matlab function
- CarrierHy - string, contains Hy carrier equivalent Matlab function

- **SensorOutput:**

- **CosinusBridge** - struct, contains further information about sensor cosine bridge outputs
- **SinusBridge** - struct, contains further information about sensor sine bridge outputs
- BridgeGain - double, scalar factor of bridge gain for output voltage

- **CosinusBridge/ SinusBridge:**

- xDimension - double, image size in x-direction
- yDimension - double, image size in y-direction
- xDirection - string, x-axis label
- yDirection - string, y-axis label
- Orientation - string, orientation of varying data, row or column
- Determination - cell, images in data {"Rise", "Fall", "All", "Diff"}

- **Units:**

- MagneticFieldStrength - string, kA/m
- Frequency - string, Hz
- SensorOutputVoltage - string, mV/V

Data:

The second struct contains the preprocessed characterization data of the TDK TAS2141 TMR angular Sensor. It is divided into two main structs one for the magnetic field reference points of the characterization images and one for the characterization sensor output images.

- **MagneticField** - struct, contains Hx- and Hy-field vectors which are the resolution references to each pixel in the characterization images of the sensors preprocessed bridge outputs
- **SensorOutput** - struct, contains structs for cosine and sine bridge outputs preprocessed in images of size of 256x256 pixel

where each pixel references a bridge output in mV to a certain Hx- and Hy-fieldstrength amplitude

■ **MagneticField:**

- hx - array, Hx field axis of characterization images column vector of 1x256 double values from -25 kA/m to 25 kA/m with a resolution of 0.1961 kA/m
- hy - array, Hy field axis of characterization images column vector of 1x256 double values from -25 kA/m to 25 kA/m with a resolution of 0.1961 kA/m

■ **SensorOutput:**

- **CosinusBridge** - struct, contains preprocessed characterization results of the sensors cosine bridge outputs
- **SinusBridge** - struct, contains preprocessed characterization results of the sensors sine bridge outputs

■ **CosinusBridge:**

- Rise - array, double array of size 256x256 which references the cosine bridge outputs for rising modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- Fall - array, double array of size 256x256 which references the cosine bridge outputs for falling modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- All - array, double array of size 256x256 superimposed image of Rise and Fall
- Diff - array, double array of size 256x256 differentiated image of Rise and Fall

■ **SinusBridge:**

- Rise - array, double array of size 256x256 which references the sine bridge outputs for rising modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- Fall - array, double array of size 256x256 which references the sine bridge outputs for falling modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- All - array, double array of size 256x256 superimposed image of Rise and Fall
- Diff - array, double array of size 256x256 differentiated image of Rise and Fall

The edited raw dataset provided from Thorben Schüthe is saved with Matlabs built-in save function in a certain way to perform partial loads from the dataset.

```
save('data/TDK_TAS2141_Characterization_2020-10-22_18-12-16-827.mat', ...
    'Info', 'Data', '-v7.3', '-nocompression')
```

Created on October 27. 2020 Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

NXP KMZ60 Characterization

NXP KMZ60 characterization as second base of the sensor array simulation was although done before and was manually modified to same structure as TDK dataset. With additional values for bridge gain TDK dataset is adjusted in the same way now. The raw dataset was acquired according to the method Thorben Schüthe described in his IEEE paper for two-dimensional characterization of TMR angular sensors. The sensor characterized for both bridges a cosine and sine bridge. The bridges have a phisically phase shift of 90° so the sensor is able to reference a superimposed magnetic field in x- and y-direction. The field was generated by a cross coil setup.

The resulting AMR characterization field abstracts a full rotation for cosine and sine output voltages by representing two maximum and minimum areas in the characterization fields. So circular path on the charcterization fields generates two sinoid periods related to current angle position between 0° and 180° or 180° and 360°.

See Also

- [IEEE Document 8706125](#)

Magnetic Stimulus

The right stimulus is the keynote for characterization records. It needs to have the ability record slow enough for quasi static recordings but is not allowed to be real static so the magnetic field is not interrupted during the recording. Therefore slow sinoid carrier functions with even slower amplitude modulation is choosen to provide a quasi static stimulus.

The carrier function for the Hx-field stimulus is related to the cosine bridge and so:

$$c_1(t) = \cos(\phi(t))$$

Due to the physically phase shift the Hy-field stimulus is related to sine:

$$c_2(t) = \sin(\phi(t))$$

Both carrier runs with same carrier frequency:

$$f_c = 3.2\text{Hz}$$

so they are executed with the phase vector over time:

$$\phi(t) = 2\pi f_c t$$

The carrier functions are triangle modulated to generate rising and falling amplituded. The modulation frequency is set to:

$$f_m = 0.01\text{Hz}$$

Which generates a stimulus with 320 periods where 160 periods feeds a rising and falling record each multiplied with maximum fieldstrength amplitude:

$$m(t) = H_{max} \cdot tri(t) = H_{max} \cdot tri(2(t - t_0)f_m)$$

$$t_0 = \frac{1}{2f_m}$$

So the Hx- and Hy-field stimulus is described by:

$$H_x(t) = m(t) \cdot c_1(t)$$

$$H_y(t) = m(t) \cdot c_2(t)$$

The stimulus amplitude depending on the phase in polar coordinates can be displayed for both parts by:

$$H_{x,y}(\phi) = |H_{x,y}(\phi)| \cdot e^{j\phi} = m(t) \cdot e^{j\phi(t)}$$

Where a rising spiral runs from center outwards for:

$$0 < t < t_0$$

And a falling spiral of amplitudes from outwards to center for:

$$t_0 < t < \frac{1}{f_m}$$

Cosinuns Bridge Output

The record characterization raw data are one dimensional time discrete vecotrs. To fieldstrength images like down below the recorded data must be referenced backwards to driven stimulus of Hx- and Hy-direction. But at first the image size of must be determined. Here fix size is set to 256 pixel for each direction. So it spans a vector for Hx- and Hy-direction from minimum -25 kA/m to maximum 25 kA/m in 256 steps with a resolution of 0.1961 kA/m. So it results into a 256x256 image. Now it runs for each point on the Hx- and Hy-axes and get the record index of the stimulus as backreference to the recorded bridge signal and sets the pixel. That runs for the rising modulation amplitude and falling amplitude until every pixel is hit and ended up into a dimensional function image as:

$$V_{cos}(H_x, H_y) = [mV/V]$$

The information of the image is build up in row. Reference Hx for constant Hy in each row. The method is also comparable to a histogram of Hx matches in the recorded sensor signal for one constant Hy and so on next histogram append on the next row for the next Hy.

Sine Bridge Output

The sine characterization field is built up similar to the cosine images but the information lays now in the columns so the data is collected in each column for a constant Hx and variable Hy:

$$V_{sin}(H_x, H_y) = [mV/V]$$

Operating Point

To determine an operating point in sensor array simulation the characterization fields needs some further investigations in static Hy and variable Hx field strength for cosine bridge and vice versa for sine bridge references. In compare to the TDK TMR the NXP AMR sensor has clear linear plateau. It has a continuous non-linear areas divided in two maximum and minum areas. The best results for bridge outputs is supported by an operating point in saturation of the characterization fields so circular path on the fields should be described at 20 kA/m to 25 kA/m path radius.

Dataset Structure

Info:

The dataset is separated in two main structs. The first one is filled with meta data. So it represents the file header. The struct is called Info and contains information about how the dataset is acquired. So the stimulus is reconstructable from that meta data.

- Created - string, contains dataset creation date
- Creator - string, contains dataset creator
- Edited - string, contains last time edited date
- Editor - string, contains last time editor
- Senor - string, sensor identification name e.g. TAS2141
- SensorType - string, kind of sensor e.g. Angular

- SensorTechnology - string, bridge technology e.g. AMR, GMR, TMR
- SensorManufacturer - string, producer or supplier e.g. NXP, TDK
- **MagneticField** - struct, contains further information about Hx and Hy
- **SensorOutput** - struct, contains information about sensor produced output and gathered image information
- **Units** - struct, contains information about used si units in dataset

- **MagneticField:**

- Modulation - string, contains modulation equivalent Matlab function
- ModulationFrequency - double, contains frequency of modulation in Hz
- CarrierFrequency - double, carrier frequency for both Hx and Hy carrier in Hz
- MaxAmplitude - double, maximum Hx and Hy field amplitude in kA/m
- MinAmplitude - double, minimum Hx and Hy field amplitude in kA/m
- Steps - double, Hx- and Hy-field steps to build characterization images
- Resolution - double, resolution of one step in kA/m
- CarrierHx - string, contains Hx carrier equivalent Matlab function
- CarrierHy - string, contains Hy carrier equivalent Matlab function

- **SensorOutput:**

- **CosinusBridge** - struct, contains further information about sensor cosine bridge outputs
- **SinusBridge** - struct, contains further information about sensor sine bridge outputs
- BridgeGain - double, scalar factor of bridge gain for output voltage

- **CosinusBridge/ SinusBridge:**

- xDimension - double, image size in x-direction
- yDimension - double, image size in y-direction
- xDirection - string, x-axis label
- yDirection - string, y-axis label
- Orientation - string, orientation of varying data, row or column
- Determination - cell, images in data {"Rise", "Fall", "All", "Diff"}

- **Units:**

- MagneticFieldStrength - string, kA/m
- Frequency - string, Hz
- SensorOutputVoltage - string, mV/V

Data:

The second struct contains the preprocessed characterization data of the TDK TAS2141 TMR angular Sensor. It is divided into two main structs one for the magnetic field reference points of the characterization images and one for the characterization sensor output images.

- **MagneticField** - struct, contain Hx- and Hy-field vectors which are the resolution references to each pixel in the characterization images of the sensors preprocessed bridge outputs
- **SensorOutput** - struct, contains structs for cosine and sine bridge outputs preprocessed in images of size of 256x256 pixel where each pixel references a bridge output in mV to a certain Hx- and Hy-fieldstrength amplitude

■ **MagneticField:**

- hx - array, Hx field axis of characterization images column vector of 1x256 double values from -25 kA/m to 25 kA/m with a resolution of 0.1961 kA/m
- hy - array, Hy field axis of characterization images column vector of 1x256 double values from -25 kA/m to 25 kA/m with a resolution of 0.1961 kA/m

■ **SensorOutput:**

- **CosinusBridge** - struct, contains preprocessed characterization results of the sensors cosine bridge outputs
- **SinusBridge** - struct, contains preprocessed characterization results of the sensors sine bridge outputs

■ **CosinusBridge:**

- Rise - array, double array of size 256x256 which references the cosine bridge outputs for rising modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- Fall - array, double array of size 256x256 which references the cosine bridge outputs for falling modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- All - array, double array of size 256x256 superimposed image of Rise and Fall
- Diff - array, double array of size 256x256 differentiated image of Rise and Fall

■ **SinusBridge:**

- Rise - array, double array of size 256x256 which references the sine bridge outputs for rising modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- Fall - array, double array of size 256x256 which references the sine bridge outputs for falling modulated stimulus amplitude to each cross reference of vectors MagneticField.hx and MagneticField.hy
- All - array, double array of size 256x256 superimposed image of Rise and Fall
- Diff - array, double array of size 256x256 differentiated image of Rise and Fall

The edited raw dataset provided from Thorben Schüthe is save with Matlabs build-in save function in a certain way to perform partial loads from the dataset.

```
save('data/NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat', ...
    'Info', 'Data', '-v7.3', '-nocompression')
```

Created on December 05. 2020 Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Config Mat

The configuration mat-file is a script generated mat-file. Generated by generateConfigMat script. The mat-files contains program and software wide useful configuration like path variables or parameter settings for program tasks or functions. It centralizes the program controlling configuration at once and can be full or partial loaded at different program stages. The key point is the configuration can only be modified by the generating script so that the config values are truly constant. Variation should be saved to temp folder or a temp mat-file. The configuration should be generated after major changes to the program or an established regeneration flow at program startup. The config.mat file is located under data directory and to path variable. Just load into the needed workspace.

```
load('config.mat')
```

Requirements

- Other m-files scripts/generateConfigMat
- Subfunctions: None
- MAT-files required: None

See Also

- [generateConfigMat](#)

Created on October 31. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Training and Test Datasets

Training and test datasets are generated by sensor array simulation part of the software. One dataset contains the simulation results generated with current configuration of used magnet in simulation and a setup of position and sensor behavior. The simulation computes for configured angles with certain angle resolution the magnetic field strength at sensor array member position for a rotation of the magnet through the configured angles. With respect to positions and angles the simulation maps the field strength for each array member to specified characterization field (current TDK Rise) and interpolates (nearest neighbor) the sensor bridge output voltages for cosine and sine bridge for each sensor array member. The acquired data is saved in matrices with same orientation as sensor array member matrix or coordinate matrices of the sensor array, so it completes the rotation in related data matrices.

Training and test datasets filenames are build by a certain pattern.

[Training|Test]_YYYY-mm-dd-_HH-MM-SS-FFF.mat

They are saved under data path data/training and data/test.

A best practice can be seen in workflow topic of the documentation.

Dataset Structure

Info:

A training or test dataset is separated into two main structures the first one the Info struct contains information about the simulation configuration and setup in which the simulation constructed the dataset.

- **SensorArrayOptions** - struct, contains setting of sensor size and behavior
- **DipoleOptions** - struct, contains setting of used magnet which was used in the simulation
- **UseOptions** - struct, contains information about use of the dataset if it is constructed for training or test use, sensor array position, number of angles, tilt of magnet and so on.
- CharData - string, identifies the characterization data set which was used to simulate the array members.
- **Units** - struct, si units of data in datasets
- filePath - string, which points on the absolute path origin where the dataset was saved including filename.

- **SensorOptions:**
 - geometry - char, identifier string of which shape the sensor array geometry was constructed, geometry of used meshgrid in computation
 - dimension - double, number of sensors at one array edge for square geometry
 - edge - double, edge length in mm of sensor array
 - Vcc - double, supply voltage of the sensor array
 - Voff - double, bridge offset voltage off the sensor array
 - Vnorm - double, norm value to get voltage values from characterization fields in combination with Vcc and Voff, TDK dataset is normed in mV/V.
 - SensorCount, double - number of sensors in the sensor array for square geometry it is square or dimension

- **DipoleOptions:**
 - sphereRadius - double, radius in mm around dipole magnet to approximate a spherical magnet in simulation with far field approximation (dipole field equation)
 - H0mag - double, field strength magnitude in kA/m which is imprinted on the compute field strength of the used magnet in a certain distance from magnet surface to construct magnet with fitting characteristics for simulation.

- z0 - double, distance from surface in which H0mag is imprinted on field computed field strength of the used magnet. Imprinting respects magnet tilts so the distance is always set to the magnet z-axis with no shifts in x and y direction
- M0mag - double, magnetic dipole moment magnitude which is used to define the magnetization direction of the magnet in its rest position.

■ **UseOptions:**

- useCase - string, identifies the dataset if it is for training or test purpose
- xPos - double, relative sensor array position to magnet surface
- yPos - double, relative sensor array position to magnet surface
- zPos - double, relative sensor array position to magnet surface
- tilt - double, magnet tilt in z-axis
- angleRes - double, angle resolution of rotation angles in simulation
- phaseIndex - double, start phase of rotation as index of full scale rotation angles with angleRes
- nAngles - double, number of rotation angles in datasets
- BaseReference - char, identifier which characterization dataset was loaded
- BridgeReference - char, identifier which reference from characterization dataset was used to generate cosine and sine voltages

■ **Units:**

- SensorOutputVoltage - char, si unit of sensor bridge outputs
- MagneticFieldStrength - char, si unit of magnetic field strength
- Angles - char, si unit of angles
- Length - char, si unit of metric length

Data:

- HxScale - 1 x L double vector of Hx field strength amplitudes used in characterization to construct sensor characterization references, x scale for characterization reference
- HyScale - 1 x L double vector of Hy field strength amplitudes used in characterization to construct sensor characterization references, y scale for characterization reference
- VcosRef - L x L double matrix of cosine bridge characterization field corresponding to HxScale and HyScale
- VsInRef - L x L double matrix of sine bridge characterization field corresponding to HxScale and HyScale
- Gain - double, scalar gain factor for bridge outputs (internal amplification)
- r0 - 3 x 1 double vector of magnet rest position from magnet surface and respect to magnet magnet tilt, used in computation of H0norm to imprint a certain field strength on magnets H-field, respects sphere radius of magnet
- m0 - 3 x 1 vector of magnetic dipole moment in magnet rest position with respect of magnet tilt, used to compute H0norm to imprint a certain field strength on magnet H-field, the magnitude of this vector is equal to M0mag
- H0norm - double, scalar factor to imprint a certain field strength on magnet H-field in rest position with respect to magnet tilt in coordinate system
- m - 3 x M double vector of magnetic dipole rotation moments each 3 x 1 vector is related to i-th rotation angle
- angles - 1 x M double vector of i-th rotation angles in degree
- angleStep - double, scalar of angle step width in rotation
- angleRefIndex - 1 x M double vector of indices which refer to a full scale rotation of configuration angle resolution, so it abstracts a subset angle rotation to the same rotation with all angles given by angle resolution
- X - N x N double matrix of x coordinate positions of each sensor array member
- Y - N x N double matrix of y coordinate positions of each sensor array member
- Z - N x N double matrix of z coordinate positions of each sensor array member

- Hx - N x N x M double matrix of compute Hx-field strength at each sensor array member position for each rotation angle 1...M
- Hy - N x N x M double matrix of compute Hy-field strength at each sensor array member position for each rotation angle 1...M
- Hz - N x N x M double matrix of compute Hz-field strength at each sensor array member position for each rotation angle 1...M
- Habs - N x N x M double matrix of compute H-field strength magnitude at each sensor array member position for each rotation angle 1...M
- Vcos - N x N x M double matrix of computed cosine bridge outputs at each sensor array member position for each rotation angle 1...M
- Vsin - N x N x M double matrix of computed sine bridge outputs at each sensor array member position for each rotation angle 1...M

See Also

- [Simulation Workflow](#)
- [sensorArraySimulation](#)
- [simulateDipoleSensorArraySquareGrid](#)
- [generateSimulationDatasets](#)
- [generateConfigMat](#)

Created on December 03. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

Published with MATLAB® R2020b

Unit Tests

Unit Tests are providing a way to test core functionality of the written software components. Matlab supports various methods to apply Unit Tests. The designed tests are using script-based testing. So far each function or functionality needs to be tested in a own test script and further on gathered into a main test script where all standalone test scripts are combined to a test suite and executed at once.

runTests

Test suite script which executes all Unit Tests scripts at once and gathers the test results in a Matlab table.

removeFilesFromDirTest

Test of function removeFilesFromDir. Creates several files and directories and deletes them during testing.

rotate3DVectorTest

Test rotate3DVector function. Do some rotations and check results.

generateDipoleRotationMomentsTest

Test the generation of magnetic dipole moments for a full rotation between 0° and 360°.

generateSensorArraySquareGridTest

Test the meshgrid generation of the sensor array and shifting it in x and y direction.

computeDipoleH0NormTest

Test magnetic field norming function. Simple test of consistent data.

computeDipoleHFieldTest

Test the magnetic dipole equation to generate dipole fields in 3D meshgrid of data points. Test field characteristics like symmetry and so on.

tiltRotationTest

Test tilt rotation of a dipole magnetic. Tilt magnet and coordinate cross to fetch pole values during rotation.

Requirements

- Other m-files required: None
- Subfunctions: None
- MAT-files required: None

See Also

- [Script-Based Unit Tests](#)
- [Write Script-Based Unit Tests](#)
- [Write Script-Based Unit Tests Using Local Functions](#)
- [Analyze Test Case Result](#)

Created on December 14. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

runTests

```
% clean workspace
clearvars;

% build suite from test files
suite = testsuite({'removeFilesFromDirTest', 'rotate3DVectorTest', ...
    'generateDipoleRotationMomentsTest', ...
    'generateSensorArraySquareGridTest', ...
    'computeDipoleH0NormTest', ...
    'computeDipoleHFieldTest', ...
    'tiltRotationTest'});

% run tests
results = run(suite);

% show results
disp(results)
disp(table(results))
cd ..
```

Published with MATLAB® R2020b

removeFilesFromDirTest

```
% create test directory with files
cd(fileparts(which('removeFilesFromDirTest')));
mkdir('testDir');
fclose(fopen(fullfile('testDir', 'testFile1.txt'), 'w'));
fclose(fopen(fullfile('testDir', 'testFile2.txt'), 'w'));
fclose(fopen(fullfile('testDir', 'testFile3.txt'), 'w'));
```

Test 1: delete all files

```
removeStatus = removeFilesFromDir(fullfile('testDir'));
assert(removeStatus == true)

% create more files
fclose(fopen(fullfile('testDir', 'testFile1.txt'), 'w'));
fclose(fopen(fullfile('testDir', 'testFile2.txt'), 'w'));
fclose(fopen(fullfile('testDir', 'testFile3.txt'), 'w'));
```

Test 2: delete with pattern

```
removeStatus = removeFilesFromDir(fullfile('testDir'), '*.txt');
assert(removeStatus == true)

% clean up
rmdir('testDir');
```

Published with MATLAB® R2020b

rotate3DVectorTest

```
% create column vectors with simple direction for rotations along the axes  
% without tilts in other achses.  
x = [-1; 0; 0];  
y = [0; -1; 0];  
z = [0; 0; -1];  
  
% set angle step width in degree to rotate at choosen axes (x, y, or z)  
angle = 90;
```

Test 1: output dimensions

```
rotated = rotate3DVector(x, 0, 0, angle);  
assert(isequal(size(rotated), [3, 1]))  
rotated = rotate3DVector([x x x x x], 0, 0, angle);  
assert(isequal(size(rotated), [3, 6]))
```

Test 2: rotate vectors in x-axes

```
rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree  
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))  
  
rotated = rotate3DVector([x y z], angle, 0, 0); % 90 degree  
assert(isequal(rotated, [-1 0 0; 0 0 1; 0 -1 0]))  
  
rotated = rotate3DVector([x y z], 2 * angle, 0, 0); % 180 degree  
assert(isequal(rotated, [-1 0 0; 0 1 0; 0 0 1]))  
  
rotated = rotate3DVector([x y z], 3 * angle, 0, 0); % 270 degree  
assert(isequal(rotated, [-1 0 0; 0 0 -1; 0 1 0]))  
  
rotated = rotate3DVector([x y z], 4 * angle, 0, 0); % 360 degree  
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

Test 3: rotate vectors in y-axes

```
rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree  
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))  
  
rotated = rotate3DVector([x y z], 0, angle, 0); % 90 degree  
assert(isequal(rotated, [0 0 -1; 0 -1 0; 1 0 0]))  
  
rotated = rotate3DVector([x y z], 0, 2 * angle, 0); % 180 degree  
assert(isequal(rotated, [1 0 0; 0 -1 0; 0 0 1]))  
  
rotated = rotate3DVector([x y z], 0, 3 * angle, 0); % 270 degree  
assert(isequal(rotated, [0 0 1; 0 -1 0; -1 0 0]))  
  
rotated = rotate3DVector([x y z], 0, 4 * angle, 0); % 360 degree  
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

Test 4: rotate vectors in z-axes

```
rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))

rotated = rotate3DVector([x y z], 0, 0, angle); % 90 degree
assert(isequal(rotated, [0 1 0; -1 0 0; 0 0 -1]))

rotated = rotate3DVector([x y z], 0, 0, 2 * angle); % 180 degree
assert(isequal(rotated, [1 0 0; 0 1 0; 0 0 -1]))

rotated = rotate3DVector([x y z], 0, 0, 3 * angle); % 270 degree
assert(isequal(rotated, [0 -1 0; 1 0 0; 0 0 -1]))

rotated = rotate3DVector([x y z], 0, 0, 4 * angle); % 360 degree
assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

Published with MATLAB® R2020b

generateDipoleRotationMomentsTest

```
% create full scale rotation with 0.5° resolution and no tilt,  
% return moments  
% and corresponding angles theta  
amp = 1e6;  
tilt = 0;  
res = 0.5;  
[MFS, tFS] = generateDipoleRotationMoments(amp, 0, tilt, res);  
  
% create same rotation but only a subset of angles N = 7  
% with equal distances to each and another, return additionally index which  
% reference to full scale  
[M, t, idx] = generateDipoleRotationMoments(amp, 7, tilt, res);  
  
% create shifted subset, shift by 22 positions in full scale theta,  
% so with 0.5° resolution it is phase shift by 11°  
[MSH, tSH, idxSH] = generateDipoleRotationMoments(amp, 7, tilt, res, 22);
```

Test 1: output dimensions

```
assert(isequal(size(MFS), [3 720]))  
assert(isequal(size(tFS), [1 720]))  
assert(isequal(size(M), [3 7]))  
assert(isequal(size(t), [1 7]))  
assert(isequal(size(idx), [1 7]))  
assert(isequal(size(MSH), [3 7]))  
assert(isequal(size(tSH), [1 7]))  
assert(isequal(size(idxSH), [1 7]))
```

Test 2: down sampling

```
assert(isequal(MFS(:,idx), M))  
assert(isequal(tFS(idx), t))  
assert(isequal(MFS(:,idxSH), MSH))  
assert(isequal(tFS(idxSH), tSH))
```

Test 3: phase shift

```
assert(isequal(tSH(1), 11))  
assert(isequal(idx, idxSH - 22))  
assert(isequal(MFS(:,idx + 22), MSH))  
assert(isequal(tFS(idx + 22), tSH))
```

Published with MATLAB® R2020b

generateSensorArraySquareGridTest

```
% create sensor array infos for size and position
% number of sensors at one edge
N = 8;

% sensor array edge length in mm
a = 2;

% relative position of the sensor array to the center of a 3D coordinate
% system (z inverse)
p = [0; 0; 2];

% z offset, later used as sphere radius of a dipole which is placed in the
% center of the coordinate system
r = 2;

% generate coordinates grid
[X, Y, Z] = generateSensorArraySquareGrid(N, a, p, r);

% create a shift in same layer
p2 = [-2; 3; 2];
[X2, Y2, Z2] = generateSensorArraySquareGrid(N, a, p2, r);
```

Test 1: output dimensions

```
assert(isequal(size(X), [N N]))
assert(isequal(size(Y), [N N]))
assert(isequal(size(Z), [N N]))
```

Test 2: equal x and y distances

```
assert(isequal(diff(Y), diff(-X, [], 2)'))
```

Test 3: constant z distances

```
assert(all(Z == -(p(3) + r), 'all'))
```

Test 4: position shift in x and y direction

```
assert(isequal(X + p2(1), X2))
assert(isequal(Y + p2(2), Y2))
assert(isequal(Z, Z2))
```

computeDipoleH0NormTest

```
% create a dipole with constant sphere radius in rest position and relative
% to sensor array with position x=0, y=0, z=0
% sphere radius 2mm
r = 2;
% distance in which the field strength is imprinted
z = 5;
% field strength magnitude to imprint in dipole field on sphere radius kA/m
Hmag = 8.5;
% magnetic moment magnitude which rotates the dipole without tilt
Mmag = 1e6;

% compute norm factor
H0norm = computeDipoleH0Norm(Hmag, [Mmag; 0; 0], [0; 0; -(z + r)]);
```

Test 1: positive scalar factor

```
assert(isscalar(H0norm))
assert(H0norm > 0)
```

Published with MATLAB® R2020b

computeDipoleHFieldTest

```
% compute a single point without norming
Hsingle = computeDipoleHField(1, 2, 3, [1; 0; 0], 1);

% compute a 3D grid of positions n+1 samples for even values
% in the grid and to
% include (0,0,0), in mm
x = linspace(-4, 4, 41);
y = linspace(4, -4, 41);
z = linspace(4, -4, 41);
[X, Y, Z] = meshgrid(x, y, z);

% magnetic dipole moment to define magnet orientation, no tilt
m = generateDipoleRotationMoments(-1e6, 1, 0);

% norm factor to imprint field strength in certain distance d = 1,
% r = 2 in mm,
% 200 kA/m, no tilt
r0 = rotate3DVector([0; 0; -3], 0, 0, 0);
H0norm = computeDipoleH0Norm(200, m, r0);

% allocate memory for field components in x,y,z
Hx = zeros(41, 41, 41);
Hy = zeros(41, 41, 41);
Hz = zeros(41, 41, 41);

% compute without norming for each z layer and reshape results into layer
for i=1:41
    Hgrid = computeDipoleHField(X(:,:,:,i), Y(:,:,:,i), Z(:,:,:,:,i), m, H0norm);
    Hx(:,:,:,:,i) = reshape(Hgrid(1,:,:), 41, 41);
    Hy(:,:,:,:,i) = reshape(Hgrid(2,:,:), 41, 41);
    Hz(:,:,:,:,i) = reshape(Hgrid(3,:,:), 41, 41);
end

% calculate magnitude in each point for better view the results
Habs = sqrt(Hx.^2+Hy.^2+Hz.^2);

% define a index to view only every 4th point for not overcrowded plot
idx = 1:4:41;

% downsample and norm
Xds = X(idx, idx, idx);
Yds = Y(idx, idx, idx);
Zds = Z(idx, idx, idx);
Hxds = Hx(idx, idx, idx) ./ Habs(idx, idx, idx);
Hyds = Hy(idx, idx, idx) ./ Habs(idx, idx, idx);
Hzds = Hz(idx, idx, idx) ./ Habs(idx, idx, idx);

% show results for test, comment out for regular unittest run, run suite
quiver3(Xds, Yds, Zds, Hxds, Hyds, Hzds);
xlabel('$X$ in mm', ...
    'FontWeight', 'normal', ...
    'FontSize', 16, ...
    'FontName', 'Times', ...
    'Interpreter', 'latex');
ylabel('$Y$ in mm', ...
    'FontWeight', 'normal', ...
```

```

'FontSize', 16, ...
'FontName', 'Times', ...
'Interpreter', 'latex');
zlabel('$Z$ in mm', ...
'FontWeight', 'normal', ...
'FontSize', 16, ...
'FontName', 'Times', ...
'Interpreter', 'latex');
title('Dipole H-Field - Equation Test', ...
'FontWeight', 'normal', ...
'FontSize', 18, ...
'FontName', 'Times', ...
'Interpreter', 'latex');
subtitle('$X$-, $Y$-, $Z$-Meshgrid from $-4 \dots 4$ mm', ...
'FontWeight', 'normal', ...
'FontSize', 14, ...
'FontName', 'Times', ...
'Interpreter', 'latex');
axis equal;

% pattern for logical indexing the center or opposite
p0 = false(1, 41);
p0(21) = true;
pN0 = true(41, 41, 41);
pN0(21,21, 21) = false;

% pattern for symmetry investigation
plu = [true(1, 20), false, false(1, 20)];
prl = [false(1, 20), false, true(1, 20)];

% compare values to check if fits in unit pairs of m and A/m
% and mm and kA/m
r0Apm = rotate3DVector([0; 0; -3e-3], 0, 0, 0);
H0normApm = computeDipoleH0Norm(200e3, m, r0Apm);
Xm = X * 1e-3;
Ym = Y * 1e-3;
Zm = Z * 1e-3;
HxApm = zeros(41, 41, 41);
HyApm = zeros(41, 41, 41);
HzApm = zeros(41, 41, 41);
for i=1:41
    HApm = computeDipoleHField(Xm(:,:,i),Ym(:,:,i),Zm(:,:,i),m,H0normApm);
    HxApm(:,:,:,i) = reshape(HApm(1,:),41,41);
    HyApm(:,:,:,i) = reshape(HApm(2,:),41,41);
    HzApm(:,:,:,i) = reshape(HApm(3,:),41,41);
end
HabsApm = sqrt(HxApm.^2+HyApm.^2+HzApm.^2);

```

Test 1: output dimensions

```

assert(isequal(size(Hsingle), [3, 1]))
assert(isequal(size(Hgrid), [3, 1681]))

```

Test 2: center of field

```

assert(X(p0,p0,p0) == 0)
assert(Y(p0,p0,p0) == 0)
assert(Z(p0,p0,p0) == 0)
assert(isnan(Hx(p0,p0,p0)))
assert(isnan(Hy(p0,p0,p0)))

```

```
assert(isnan(Hz(p0,p0,p0)))
assert(all(Hx(~p0,p0,p0) ~= 0, 'all'))
assert(all(Hx(p0,~p0,p0) ~= 0, 'all'))
assert(all(Hy(~p0,p0,p0) == 0, 'all'))
assert(all(Hy(p0,~p0,p0) == 0, 'all'))
assert(all(Hz(~p0,~p0,p0) == 0, 'all'))
```

Test 3: magnetization

```
assert(all(Hx(~p0,p0,~p0) ~= 0, 'all'))
assert(all(Hx(p0,~p0,~p0) ~= 0, 'all'))
assert(all(Hx(p0,p0,~p0) ~= 0, 'all'))
assert(all(Hy(~p0,p0,~p0) == 0, 'all'))
assert(all(Hy(p0,~p0,~p0) == 0, 'all'))
assert(all(Hy(p0,p0,~p0) == 0, 'all'))
assert(all(Hz(~p0,p0,~p0) == 0, 'all'))
assert(all(Hz(p0,~p0,~p0) == 0, 'all'))
```

Test 4: imprinting

index 6 is 3mm and 36 is -3mm from surface where 200 kA/m should be imprinted

```
assert(round(abs(Hx(p0,p0,6)),6) == 200)
assert(round(abs(Hx(p0,p0,36)),6) == 200)
```

Test 5: symmetry

```
assert(all((Hx(plu,:,:)) - flip(Hx(prl,:,:),1))==0, 'all'))
assert(all((Hx(:,plu,:)) - flip(Hx(:,prl,:),2))==0, 'all'))
assert(all((Hy(plu,:,:)) + flip(Hy(prl,:,:),1))==0, 'all'))
assert(all((Hy(:,plu,:)) + flip(Hy(:,prl,:),2))==0, 'all'))
assert(all((Hz(:,:,~p0)) + flip(Hz(:,:,~p0),2))==0, 'all'))
```

Test 6: units milli kilo

```
assert(all(round(HxApm(pN0) * 1e-3, 6) == round(Hx(pN0), 6), 'all'))
assert(all(round(HyApm(pN0) * 1e-3, 6) == round(Hy(pN0), 6), 'all'))
assert(all(round(HzApm(pN0) * 1e-3, 6) == round(Hz(pN0), 6), 'all'))
assert(all(round(HabsApm(pN0) * 1e-3, 6) == round(Habs(pN0), 6), 'all'))
```

Published with MATLAB® R2020b

tiltRotationTest

```
% clean
clearvars;

% relevant tilt in y axes
tilt = 0.5:0.5:90;

% magnetic dipole moment to define magnet orientation, no tilt
% rotate angles theta 0°, 90°, 180°, 270°
[mNoTilt, thetaNoTilt] = generateDipoleRotationMoments(-1e6, 4, 0);

% Habs for magnetization from north to south from -x to x
HabsMust = [400 400 200 200 200 200];

% norm factor to imprint field strength in certain distance d = 1,
% r = 2 in mm,
% 200 kA/m, no tilt
r0NoTilt = [0; 0; -3];
H0normNoTilt = computeDipoleH0Norm(200, mNoTilt(:,1), r0NoTilt);

% axes with no tilt, rest position
AxesNoTilt = [-3, 3, 0, 0, 0, 0;
               0, 0, -3, 3, 0, 0;
               0, 0, 0, 0, -3, 3];

% calc fields along coordinate cross and magnitudes
HNoTilt = zeros(3, 6, 4);
for i = 1:4
    % rotate axes same wise to check pole values
    RotateNoTiltAxes = rotate3DVector(AxesNoTilt, 0, 0, thetaNoTilt(i));
    HNoTilt(:,:,i) = computeDipoleHField(RotateNoTiltAxes(1,:), ...
                                           RotateNoTiltAxes(2,:), RotateNoTiltAxes(3,:), ...
                                           mNoTilt(:,i), H0normNoTilt);
end

% habs must be show imprinted field strength and double of it at poles
HabsNoTilt = sqrt(sum(HNoTilt.^2, 1));

% calc fields along tilt coordinate cross and magnitudes
HTilt = zeros(3, 6, 4, length(tilt));
for j = 1:length(tilt)
    % magnetic dipole moment to define magnet orientation, with tilt
    % rotate angles theta 0°, 90°, 180°, 270°
    [mTilt, thetaTilt] = generateDipoleRotationMoments(-1e6, 4, tilt(j));

    % norm factor to imprint field strength in certain distance d = 1,
    % r = 2 in mm,
    % 200 kA/m, no tilt
    r0Tilt = rotate3DVector(r0NoTilt, 0, tilt(j), 0);
    H0normTilt = computeDipoleH0Norm(200, mTilt(:,1), r0Tilt);

    % axes with tilt, rest position
    AxesTilt = rotate3DVector(AxesNoTilt, 0, tilt(j), 0);

    for i = 1:4
        % rotate axes same wise to check pole values
        RotateTiltAxes = rotate3DVector(AxesTilt, 0, 0, thetaTilt(i));
```

```
HTilt(:,:,:,i,j) = computeDipoleHField(RotateTiltAxes(1,:), ...
    RotateTiltAxes(2,:), RotateTiltAxes(3,:),
    mTilt(:,i), H0normTilt);
end
end

% habs must be show imprinted field strength and double of it at poles
HabsTilt = sqrt(sum(HTilt.^2, 1));
```

Test 1: rotation without tilt

```
assert(all(round(HabsNoTilt, 6) == round(HabsMust, 6), 'all'))
```

Test 2: rotation with tilt

```
assert(all(round(HabsTilt, 6) == round(HabsMust, 6), 'all'))
```

Published with MATLAB® R2020b

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort

Datum

Unterschrift im Original