

# 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

|  |           |
|--|-----------|
| <b>1 Motivation 0.0.1 17.02.2021</b>                                       | <b>1</b>  |
| 1.1 Stand der Vorarbeiten . . . . .  | 2         |
| 1.2 Zielstellung . . . . .   | 7         |
| <b>2 Grundlagen 0.0.4 25.04.2021</b>                                       | <b>8</b>  |
| 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 die Dipol-Feldgleichung . . . . .         | 25        |
| 2.7 Gauß-Prozesse für Regressionsverfahren . . . . .                       | 29        |
| <b>3 Software-Entwicklung für Optimierungsexperimente 0.0.4 25.04.2021</b> | <b>32</b> |
| 3.1 Aufbau und Funktion . . . . .  | 32        |
| 3.2 Simulationsprozesse und Ausführung . . . . .                           | 35        |
| 3.2.1 Sensor-Array-Simulation . . . . .                                    | 35        |
| 3.2.2 Gauß-Prozess-Regression . . . . .                                    | 37        |
| <b>4 Erprobungs- und Optimierungsexperimente 0.0.1 13.01.2021</b>          | <b>44</b> |
| 4.1 Festlegung des Startpunktes . . . . .                                  | 44        |
| 4.2 Festlegung des Verfahrweges ohne Verkippung . . . . .                  | 45        |
| 4.3 Simulationsdurchführung . . . . .                                      | 45        |
| <b>5 Auswertung 0.0.1 13.01.2021</b>                                       | <b>46</b> |
| 5.1 Gegenüberstellung der GPR-Modelle . . . . .                            | 46        |
| <b>6 Zusammenfassung und Bewertung 0.0.1 13.01.2021</b>                    | <b>47</b> |
| <b>Algorithmenverzeichnis</b>  | <b>51</b> |

|   |           |
|---|-----------|
| <b>Glossar</b>  | <b>52</b> |
| <b>Abkürzungen</b>  | <b>54</b> |
| <b>Literatur</b>  | <b>55</b> |
| <br>  |           |
| <b>Anhang</b>   | <b>57</b> |
| <br>  |           |
| <b>A TDK TAS2141-AAAB Kennfelddatensatz 0.0.1 29.03.2021</b>      | <b>57</b> |
| <br>  |           |
| <b>B Sensor-Array-Simulation Implementierung 0.0.1 07.04.2021</b> | <b>60</b> |
| <br>  |           |
| <b>C Gauß-Prozess-Regression Implementierung 0.0.1 13.04.2021</b> | <b>64</b> |
| C.1 Modellinitialisierung . . . . .                               | 66        |
| C.2 Modelloptimierung . . . . .                                   | 76        |
| C.3 Modellvorhersagen . . . . .                                   | 78        |
| C.4 Modellgeneralisierung . . . . .                               | 81        |
| <br>  |           |
| <b>D Genutzte Software 0.0.3 08.01.2021</b>                       | <b>83</b> |
| <br>  |           |
| <b>E Software-Dokumentation 0.0.7 18.04.2021</b>                  | <b>84</b> |
| E.1 GaussianProcessDipoleSimulation . . . . .                     | 85        |
| E.2 Workflows . . . . .   | 88        |
| E.2.1 Project Preparation . . . . .                               | 90        |
| E.2.2 Project Structure . . . . .                                 | 96        |
| E.2.3 Git Feature Branch Workflow . . . . .                       | 100       |
| E.2.4 Documentation Workflow . . . . .                            | 102       |
| E.2.5 Simulation Workflow . . . . .                               | 107       |
| E.3 Executable Scripts . . . . .                                  | 109       |
| E.3.1 publishProjectFilesToHTML . . . . .                         | 112       |
| E.3.2 generateConfigMat . . . . .                                 | 118       |
| E.3.3 generateSimulationDatasets . . . . .                        | 137       |
| E.3.4 deleteSimulationDatasets . . . . .                          | 140       |
| E.3.5 deleteSimulationPlots . . . . .                             | 142       |
| E.3.6 exportPublishedToPdf . . . . .                              | 144       |
| E.3.7 demoGPRModule . . . . .                                     | 151       |
| E.3.8 investigateKernelParameters . . . . .                       | 159       |
| E.3.9 compareGPRKernels . . . . .                                 | 172       |

|            |   |     |
|------------|---|-----|
| E.4        | Source Code . . . . .                     | 178 |
| E.4.1      | sensorArraySimulation . . . . .           | 180 |
| E.4.1.1    | rotate3DVector . . . . .                  | 183 |
| E.4.1.2    | generateDipoleRotationMoments . . . . .   | 187 |
| E.4.1.3    | generateSensorArraySquareGrid . . . . .   | 194 |
| E.4.1.4    | computeDipoleH0Norm . . . . .             | 200 |
| E.4.1.5    | computeDipoleHField . . . . .             | 204 |
| E.4.1.6    | simulateDipoleSquareSensorArray . . . . . | 211 |
| E.4.2      | gaussianProcessRegression . . . . .       | 223 |
| E.4.2.1    | initGPR . . . . .                         | 228 |
| E.4.2.2    | initGPROptions . . . . .                  | 232 |
| E.4.2.3    | initTrainDS . . . . .                     | 236 |
| E.4.2.4    | initKernel . . . . .                      | 239 |
| E.4.2.5    | initKernelParameters . . . . .            | 242 |
| E.4.2.6    | tuneKernel . . . . .                      | 246 |
| E.4.2.7    | computeTuneCriteria . . . . .             | 249 |
| E.4.2.8    | predFrame . . . . .                       | 251 |
| E.4.2.9    | predDS . . . . .                          | 255 |
| E.4.2.10   | lossDS . . . . .                          | 258 |
| E.4.2.11   | optimGPR . . . . .                        | 261 |
| E.4.2.12   | computeOptimCriteria . . . . .            | 265 |
| E.4.2.13   | kernelQFCAPX . . . . .                    | 268 |
| E.4.2.13.1 | QFCAPX . . . . .                          | 270 |
| E.4.2.13.2 | meanPolyQFCAPX . . . . .                  | 274 |
| E.4.2.13.3 | initQFCAPX . . . . .                      | 277 |
| E.4.2.14   | kernelQFC . . . . .                       | 281 |
| E.4.2.14.1 | QFC . . . . .                             | 283 |
| E.4.2.14.2 | meanPolyQFC . . . . .                     | 287 |
| E.4.2.14.3 | initQFC . . . . .                         | 290 |
| E.4.2.15   | basicMathFunctions . . . . .              | 293 |
| E.4.2.15.1 | sinoids2angles . . . . .                  | 296 |
| E.4.2.15.2 | angles2sinoids . . . . .                  | 301 |
| E.4.2.15.3 | decomposeChol . . . . .                   | 305 |
| E.4.2.15.4 | frobeniusNorm . . . . .                   | 309 |
| E.4.2.15.5 | computeInverseMatrixProduct . . . . .     | 312 |
| E.4.2.15.6 | computeTransposeInverseProduct . . . . .  | 316 |

|             |   |     |
|-------------|---|-----|
| E.4.2.15.7  | addNoise2Covariance . . . . .               | 320 |
| E.4.2.15.8  | computeAlphaWeights . . . . .               | 323 |
| E.4.2.15.9  | computeStdLogLoss . . . . .                 | 326 |
| E.4.2.15.10 | computeLogLikelihood . . . . .              | 329 |
| E.4.2.15.11 | estimateBeta . . . . .                      | 332 |
| E.4.3       | util . . . . .                              | 335 |
| E.4.3.1     | removeFilesFromDir . . . . .                | 337 |
| E.4.3.2     | publishFilesFromDir . . . . .               | 341 |
| E.4.3.3     | plotFunctions . . . . .                     | 345 |
| E.4.3.3.1   | plotTDKCharDataset . . . . .                | 348 |
| E.4.3.3.2   | plotTDKCharField . . . . .                  | 359 |
| E.4.3.3.3   | plotTDKTransferCurves . . . . .             | 368 |
| E.4.3.3.4   | plotKMZ60CharDataset . . . . .              | 376 |
| E.4.3.3.5   | plotKMZ60CharField . . . . .                | 387 |
| E.4.3.3.6   | plotKMZ60TransferCurves . . . . .           | 396 |
| E.4.3.3.7   | plotDipoleMagnet . . . . .                  | 404 |
| E.4.3.3.8   | plotSimulationDataset . . . . .             | 412 |
| E.4.3.3.9   | plotSingleSimulationAngle . . . . .         | 429 |
| E.4.3.3.10  | plotSimulationSubset . . . . .              | 444 |
| E.4.3.3.11  | plotSimulationCosSinStats . . . . .         | 458 |
| E.4.3.3.12  | plotSimulationDatasetCircle . . . . .       | 472 |
| E.5         | Datasets . . . . .                          | 487 |
| E.5.1       | TDK TAS2141 Characterization . . . . .      | 489 |
| E.5.2       | NXP KMZ60 Characterization . . . . .        | 498 |
| E.5.3       | Config Mat . . . . .                        | 507 |
| E.5.4       | Training and Test Datasets . . . . .        | 509 |
| E.6         | Unit Tests . . . . .                        | 514 |
| E.6.1       | runTests . . . . .                          | 517 |
| E.6.2       | removeFilesFromDirTest . . . . .            | 518 |
| E.6.3       | rotate3DVectorTest . . . . .                | 520 |
| E.6.4       | generateDipoleRotationMomentsTest . . . . . | 523 |
| E.6.5       | generateSensorArraySquareGridTest . . . . . | 525 |
| E.6.6       | computeDipoleH0NormTest . . . . .           | 527 |
| E.6.7       | computeDipoleHFieldTest . . . . .           | 528 |
| E.6.8       | tiltRotationTest . . . . .                  | 534 |

**Selbstständigkeitserklärung**

**537**

# 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.

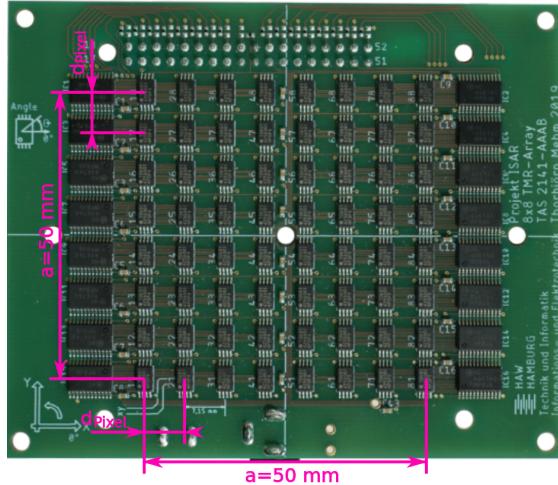


Abbildung 1.1: Platinen-Sensor-Array im Maßstab aufgebaut als  $8 \times 8$  Sensor-Array, dass als Aufsteckmodul für eine Mikrocontroller getriebene Signalverarbeitung bereitsteht. Die einzelnen Sensoren sind in Sensorbänken angeordnet. Die Anordnung erfolgt in eine linke und rechte Sensorbank pro Reihe 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.

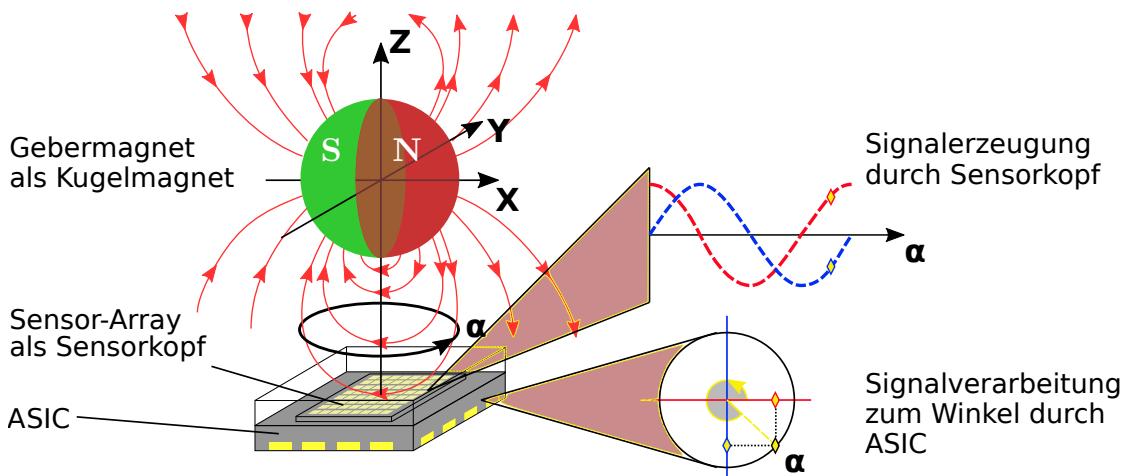


Abbildung 1.3: Veranschaulichung eines vollständigen Sensor-ICs für die Drehwinkel erfassung. Stark vereinfachte Darstellung eines Sensor-IC bestehend aus einem Sensorkopf und ASIC. Zu sehen sind die übergeordneten Aufgaben von Sensorkopf und ASIC. Der Sensorkopf erfasst die physikalische Stimulation (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 0.0.4 25.04.2021

Das Fundament für die Drehwinkel erfassung mittels magnetischem Sensor-Array und lernender Signalverarbeitung [15][19][20] 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 einem 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  $\alpha$  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  $90^\circ$  zur Cosinus-Funktion phasenverschobene Sinus-Funktion  $V_{sin}(H_x, H_y)$ . Genaue physikalische Größenzusammenhänge und technische Umsetzung sind dabei vorerst in den weiteren Darstellungen vernachlässigt.

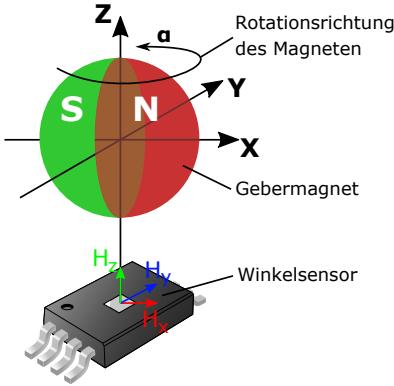


Abbildung 2.1: Klassischer Anwendungsfall für die Drehwinkelerfassung. Zeigt einen, um seine  $Z$ -Achse rotierenden, Gebermagneten und einen Winkelsensor in zentrierter und orthogonaler Ausrichtung zur  $Z$ -Achse des Magneten. Idealerweise befinden sich Magnet und Sensor, ohne Verkipplungen in  $X$ - 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  $\alpha$  des Gebermagneten [15].

Für eine beliebige Winkelmessung  $\mathbf{A}$ , die eine entsprechende Winkelstellung  $\alpha$  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 erhobene 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.

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

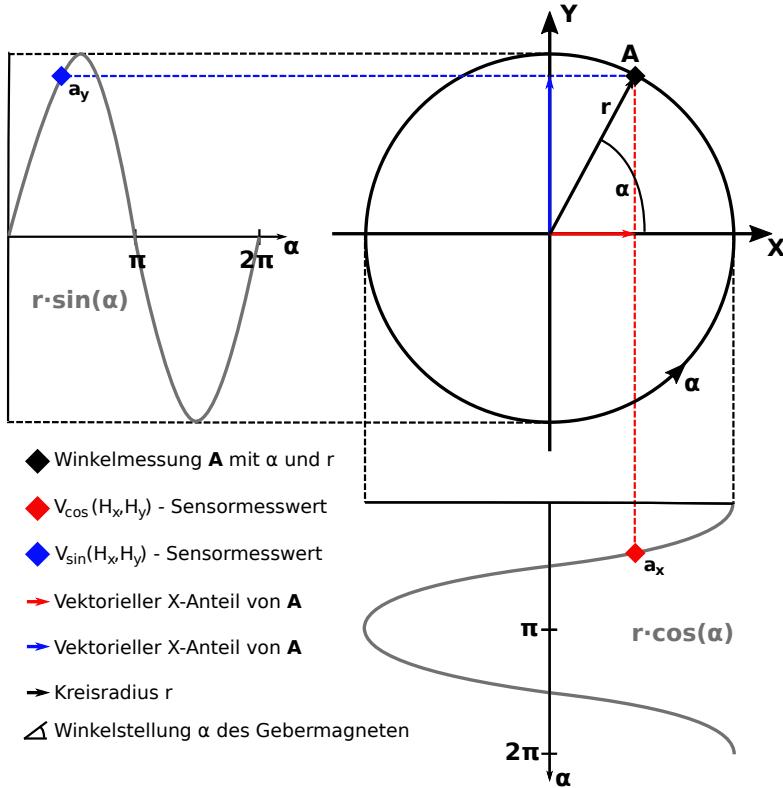


Abbildung 2.2: Kreisdarstellung der Winkelmessung. Als Abbildung der Winkelmessung  $\mathbf{A} \mapsto \alpha$  aus Gleichung 2.1. Die Zusammensetzung der Messung erfolgt durch die vom Winkelsensor gemessenen vektoriellen Anteile für die polare Darstellung der Gebermagnetwinkelstellung.

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 Funktion  $\text{arctan2}$  zuzuführen. Diese bildet einen Winkel von null bis  $\pi$  ab und besitzt eine Sprungstelle bei  $\pi$ . Der  $Y$ -Anteil kann dabei als Entscheider genutzt werden, um eine Abbildung des Winkels auf eine volle Kreisumdrehung ( $2\pi$ ) 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 Winkelstellungen  $\mathbf{A} \mapsto \alpha_1$  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 die 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.

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

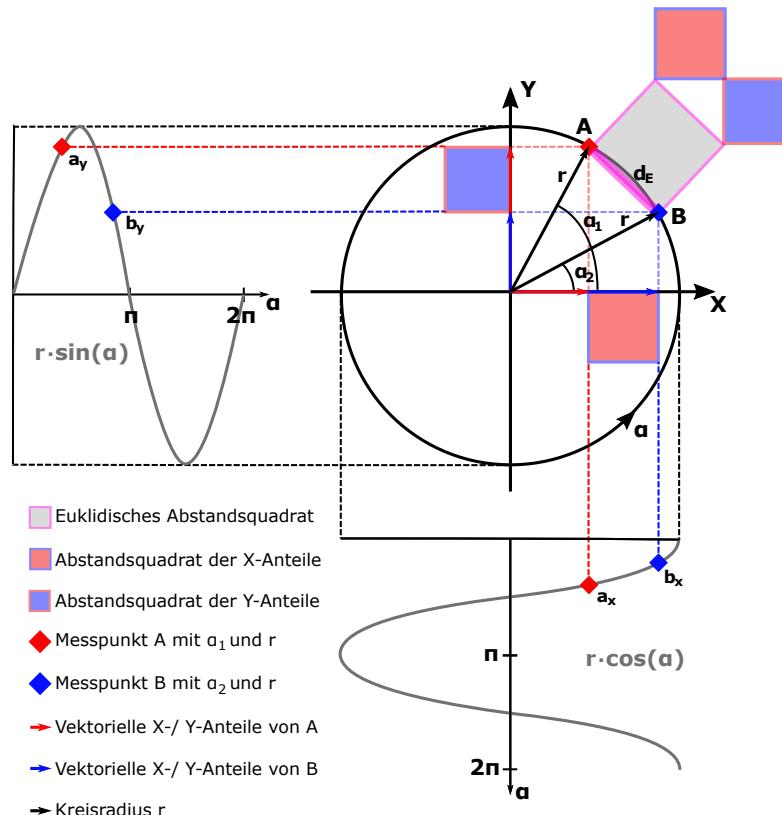


Abbildung 2.3: Allgemeine Kreisdarstellung des euklidischen Winkelabstands. Die Kreisdarstellung zeigt den euklidischen Winkelabstand zweier Winkelmesspunkte **A** und **B** mit gleichem Kreisradius  $r$ . Der euklidische Abstand, bzw. das Abstandsquadrat, zwischen den Winkelposition **A** und **B** ist zerlegt in Abstandsquadrateanteile. Die Abstandsquadrateanteile 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 der 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 höherem Normraum [4] und stellt somit die Grundlage für eine Adaptierung auf ein Sensor-Array dar.

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

## 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 Drehwinkel erfassung, besitzt daher zwei um 45° verdrehte Wheatstone-Brücken. Durch die schwache Widerstandsänderung des Materials ist eine nachgeschaltete 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 ließ [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 Verstärkerseinheiten [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  $360^\circ$  abgedeckt. Ein TMR-Sensor für die Drehwinkelerfassung besteht aus zwei um  $90^\circ$  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 dem Temperatureinfluss entgegenzuwirken, werden 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 Anregungsmagnetfeld wird durch die  $90^\circ$ -Verdrehung beider Brücken zueinander erreicht, dass die benötigte Cosinus- und Sinus-Funktion, mit entsprechender Phasenverschiebung um  $90^\circ$ , für den Anwendungsfall in Abschnitt 2.1 ausgeben werden [11].

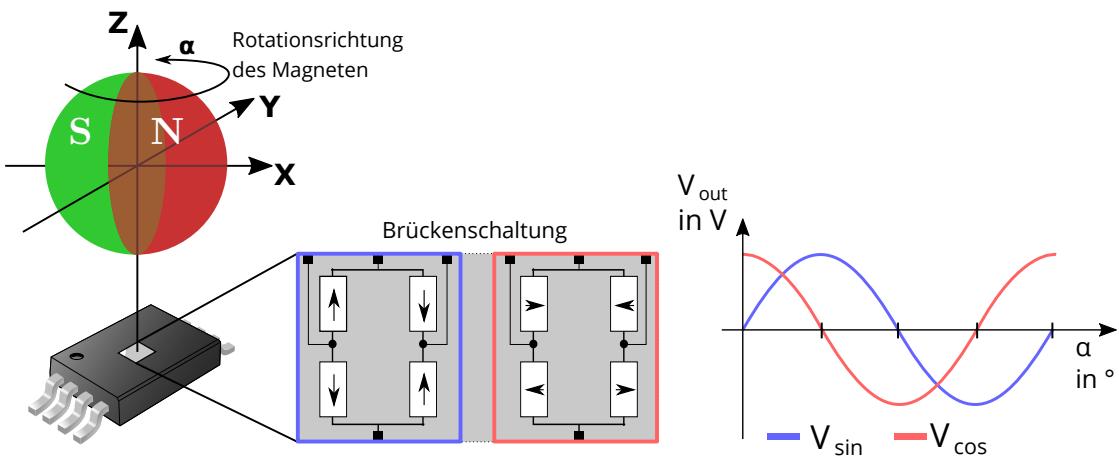


Abbildung 2.5: TMR Drehwinkelapplikation. Schematisch gezeigt für eine volle Rotation des Gebermagneten um  $360^\circ$ . Zu sehen sind die um  $90^\circ$  verdrehten Wheatstone-Brücken des Sensors. Die Brücken bilden, bei rotierendem Gebermagnetfeld, eine Sinus- und Cosinus-Funktion nach. Die Pfeile 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 Technologien 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 dreieckmodulierter Cosinus-Strom. Die  $H_y$ -Feldgenerierung nutzt einen mit gleicher Frequenz dreieckmodulierten Sinus-Strom. Es entstehen dabei steigende und fallende Modulationsflanken bzw. Messverläufe. Es ergeben sich in polarer Darstellung Trajektorien mit wachsender und abnehmenden 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 einem Kennfeld für eine entsprechende Winkelsensor-Wheatstone-Brücke, als Charakterisierungsergebnis zur Verfügung stehen [15].

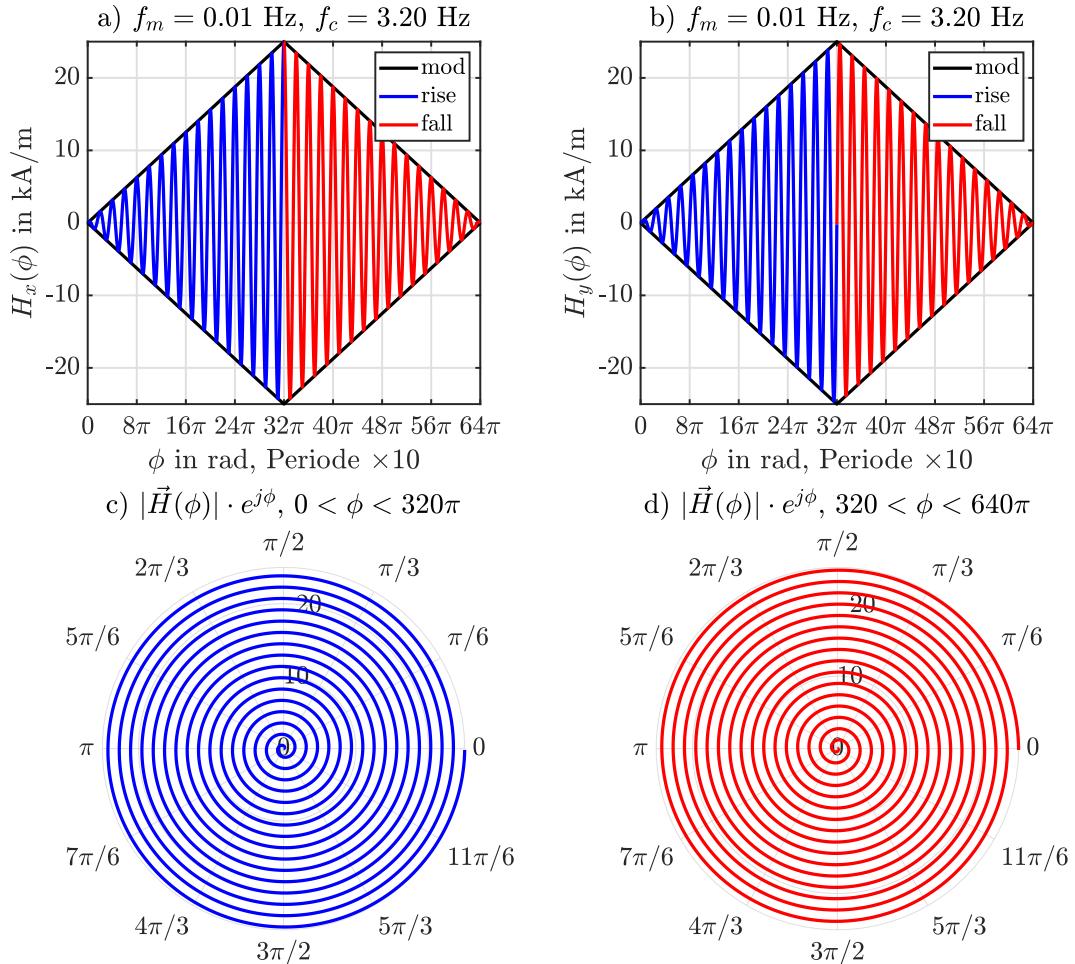


Abbildung 2.6: Magnetfeldstimuli zur Erzeugung von Sensorkennfeldern. Es sind die Bestandteile des magnetischen Sensorstimuli dargestellt, die zum Ausmessen des Sensorkennfeldes in  $H_x$ - und  $H_y$ -Richtung verwendet worden sind. Es ist das Prinzip des Verfahrens dargestellt. In a) und b) ist die 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 Betragsamplitude 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  $f_m = 0,1$  Hz und einer Trägerwellenfrequenz  $f_c = 3,2$  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.

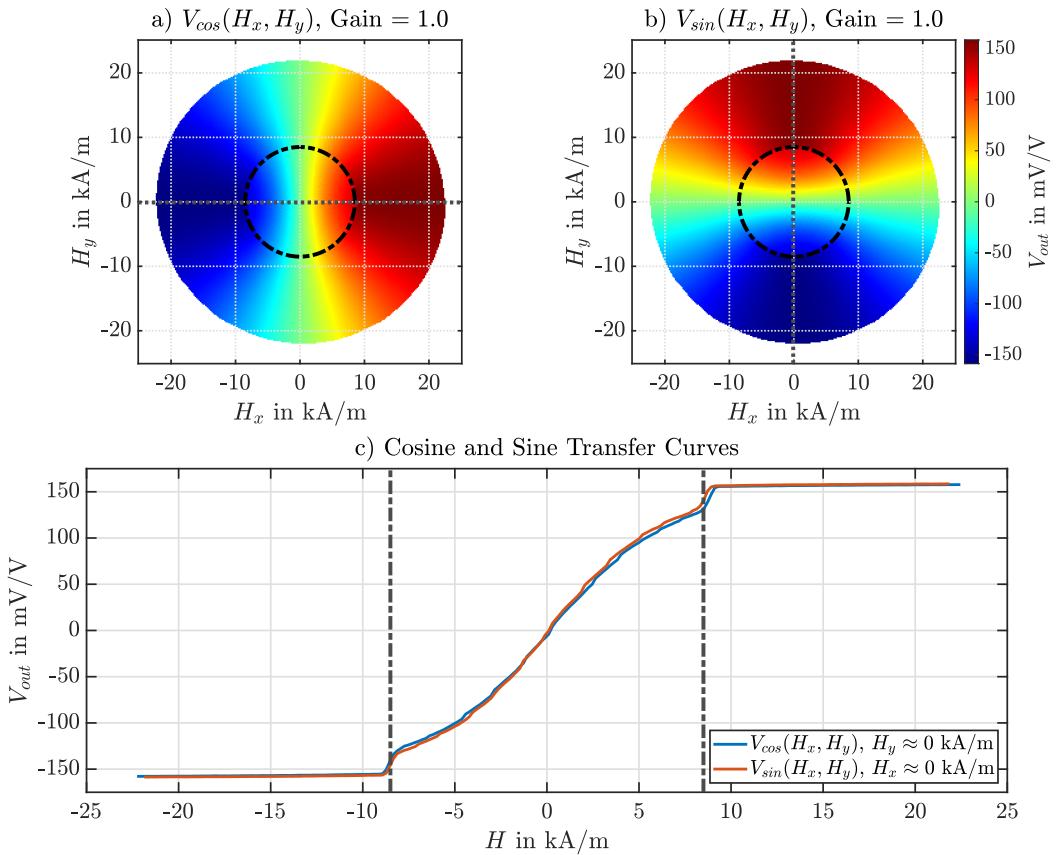


Abbildung 2.7: TDK TAS2141-AAAB Übertragungskennlinie. Es sind wieder die Kennfelder aus der steigenden Amplitudenmodulation in a) und b). In c) sind die Übertragungskennlinien für den Sensor gezeigt mit Kennzeichnung für den Betrieb auf dem linearen Plateau des Kennfeldes bei  $8,5 \text{ kA m}^{-1}$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

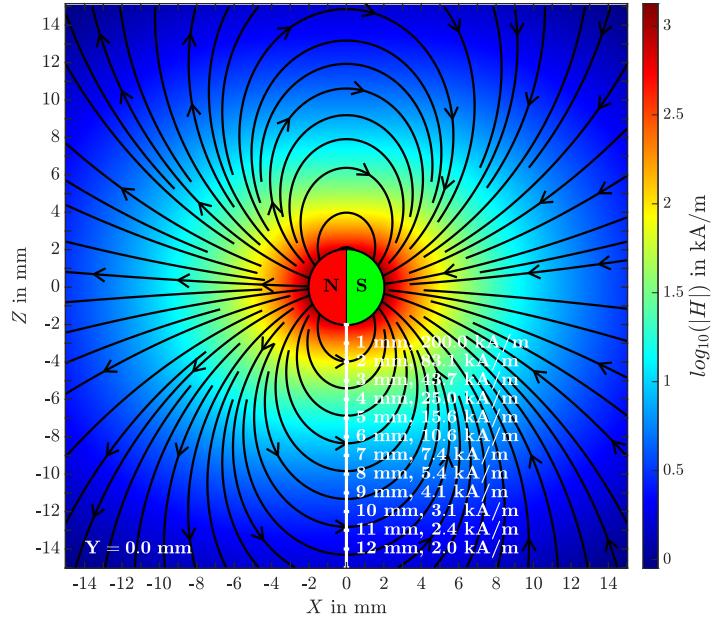


Abbildung 2.8: Approximierter Kugelmagnet. Die Approximation des Kugelmagneten erfolgt über die Dipol-Feldgleichung in Näherung des Kugelmagnetfernfeldes. Das Magnetfeld ist auf  $200 \text{ kA m}^{-1}$  bei einem Abstand von 1 mm zur Kugelmagnetoberfläche normiert. Der Radius des Kugelmagneten beträgt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

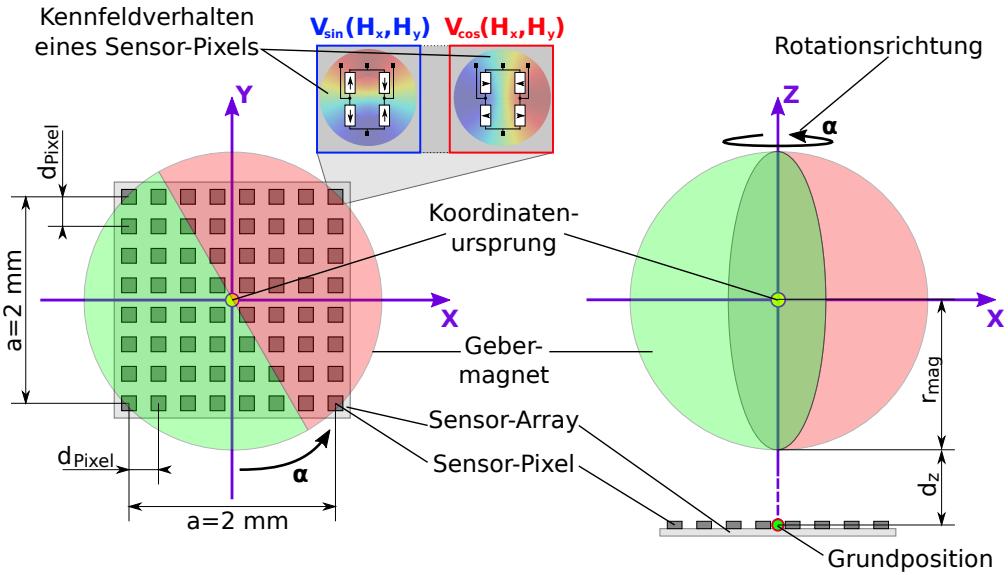


Abbildung 2.9: Geometrischer Aufbau und Ausrichtung des Sensor-Arrays. Quadratische Anordnung von Sensor-Pixeln zu einem  $8 \times 8$  Sensor-Array. Alle Pixel sind gleich verteilt auf der Array-Fläche. Sensor-Pixel-Verhalten ist aus Kennfeldern entnommen und ortsabhängig von Pixel-Position im Koordinatensystem. 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 im 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  $\alpha$ .

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 genau die Lösung und Projektion in den höheren Normraum [4][9] und somit die modifizierte Abstandsfunktion nach der Frobenius-Norm [19][18] in Gleichung 2.13. Es sind dargestellte 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 die 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.

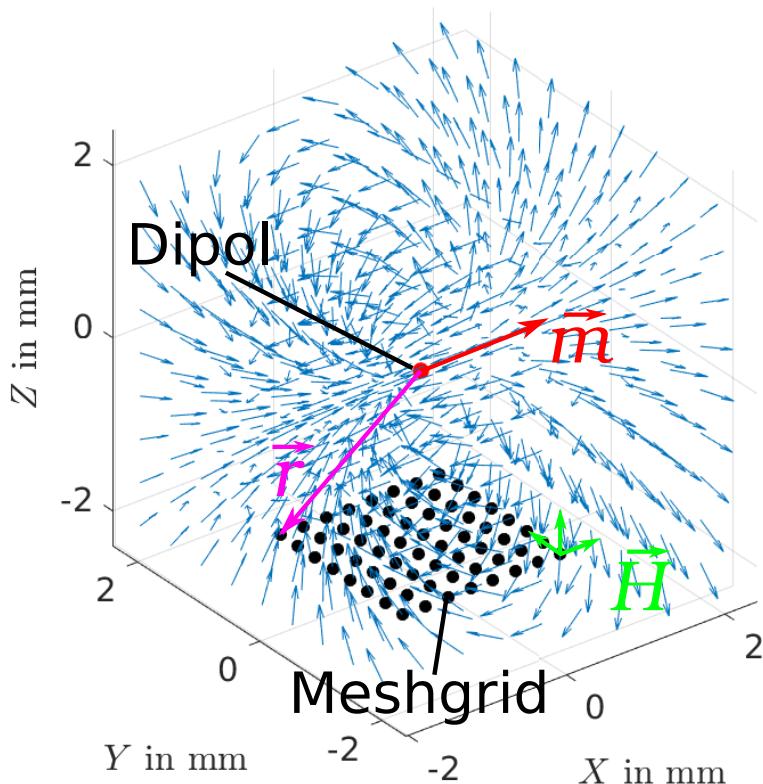


Abbildung 2.11: Simulation der Dipol-Feldgleichung. Als veranschaulichendes Beispiel ist ein Magnetfeld über die Dipol-Feldgleichung simuliert. Der Dipol bildet den Koordinatenursprung bei  $\vec{r} = (0, 0, 0)^T$ . In Relation zum Dipol ist ein Meshgrid der einzelnen Sensor-Pixel-Positionen  $\vec{r}$  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 dreidimensionales Koordinatensystem gelegt. Wie in Abbildung 2.11 zu sehen, liegt der Dipol im Koordinatenursprung bei  $\vec{r} = (0, 0, 0)^T$ . Die einzelne 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 dazugehörigem 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 Nullsetzen der Verkippungswinkel  $\alpha_x$  und  $\alpha_y$  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  $\alpha_i$  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  $m_0 > 1000 \text{ Am}^2$  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 im Weiteren zu manipulieren. Die Manipulation des Magnetfeldes erfolgt durch das Aufprägen einer Betragsfeldstärke  $H_{mag}$  für die Ruhelage des Magneten mit dazugehöriger Feldstärke  $\vec{H}_0$ . Dabei ist ein definierter Abstand zur Kugelmagnetoberfläche festzulegen, bei dem sich die aufzuprägende Betragsfeldstärke  $H_{mag}$  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  $\alpha_1$  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 entsprechendem Ruhemoment, der Normierungsposition folgend, nach Gleichung 2.21 auszurichten. Anschließend ist die Betragsfeldstärke  $|\vec{H}_0(\vec{r}_0, \vec{m}_0)|$  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 die in Abschnitt 2.5 beschriebenen Array-Datenformate ergeben. Diese können im Simulationsverlauf fortführend, direkt im Array-Format auf die Kennfelder, zur Entnahme von korrespondierenden Spannungsangaben angewandt und gespeichert werden. In der Sensor-Array-Simulation ist die Verkippung in der 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 dem mittelwertfreien 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].

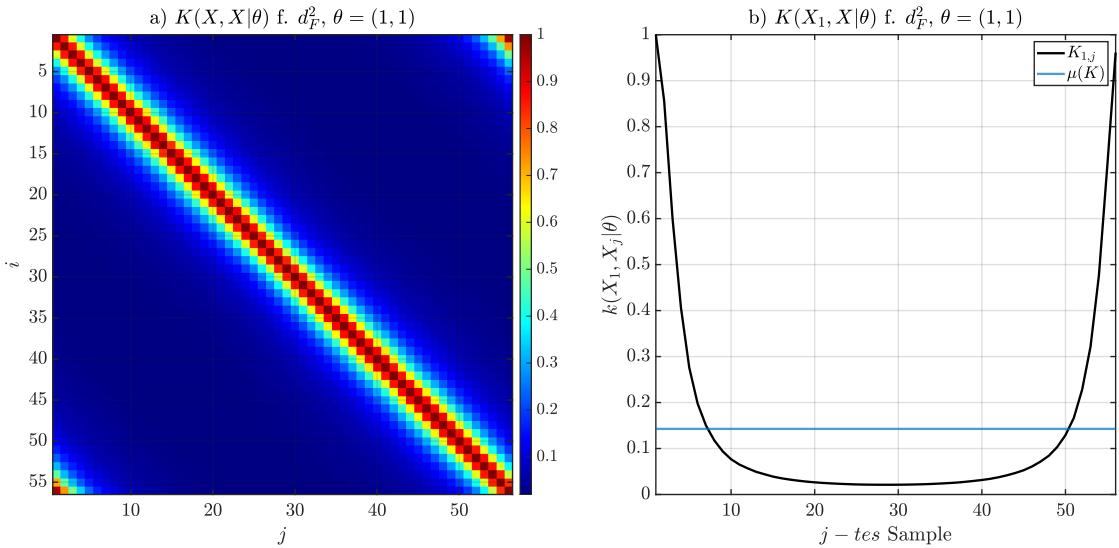


Abbildung 2.12: Kernel-Implementierung der Vorarbeiten mittels Kovarianzfunktion  $k(X_i, X_j)$  nach Gleichung C.4 f.  $d_F^2$  nach Gleichung 2.13 und einem Trainingsdatensatz  $X$  mit  $N_{Ref} = 56$  gleichverteilten Simulationswinkeln  $X_i \mapsto \alpha_i$ . Die Skalierung durch Kernel-Parameter 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 von Datensätzen im Regressionsverfahren zu interpretieren. Der genutzte Trainingsdatensatz  $X$  basiert auf erfolgter TMR-Sensor-Charakterisierung in Anhang A [11][15]. 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 verwandter 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. Ableitungen erfolgen für Winkelstellungen über orthogonal zueinander stehenden Cosinus- und Sinus-Funktionen. 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, das 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] bezogen, 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 Diagonalen 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 0.0.4**

## **25.04.2021**

Die Software-Entwicklung erfolgt unter dem Gesichtspunkt zur Durchführung von Versuchsreihen. Ziel ist dabei die Parameterfindung zur Modelloptimierung für die Winkelauswertung. Durchzuführende Simulationen und Erprobungsexperimente basieren teilweise auf Zwischenergebnissen, die strukturiert im Software-Projektverzeichnis zwischen gespeichert sind, siehe Unterabschnitt E.2.2. Für die Auswertung der Simulationen sind unterstützende Grafiken angefertigt worden. Diese sind als Funktionen im Unterabschnitt E.4.3.3 und in den ausführbaren Skripten im Abschnitt E.3 enthalten. Das Kapitel dient zur näheren Erläuterung des modularen Software-Aufbaus und der zur Verfügung stehenden Simulationsprozesse. Die Entwicklungs-Roadmap der Simulations-Software ist mit Aufführung der einzelnen Beiträge in Abschnitt E.1 einzusehen.

### **3.1 Aufbau und Funktion**

Als erster Schritt der hier stattfindenden Entwicklungsarbeiten wird ein Konzept aufgestellt, dass den erheblichen Simulationsaufwand praktisch benutzbar macht. Dafür sind die Simulationsbestandteile nach Aufgabenbereich und Charakter in Abbildung 3.1 zugeteilt. Die Implementierung erfolgt in zwei eigenständigen Simulationen. Für das Sensor-Array mittels Dipol-Feldgleichung in Anhang B und für die ASIC Simulation mit Gauß'scher Prozess-Regression in Anhang C. Zur Klassifizierung der Simulationen dienen hierbei Datendirektion und Datenbeschaffenheit in den einzelnen Simulationsabschnitten. Die Kopplung und Steuerung beider Simulationsabschnitte erfolgt über Datensätze. Die Datensatzbeschreibung und ihre Nutzung ist im Abschnitt E.5 nachzulesen.

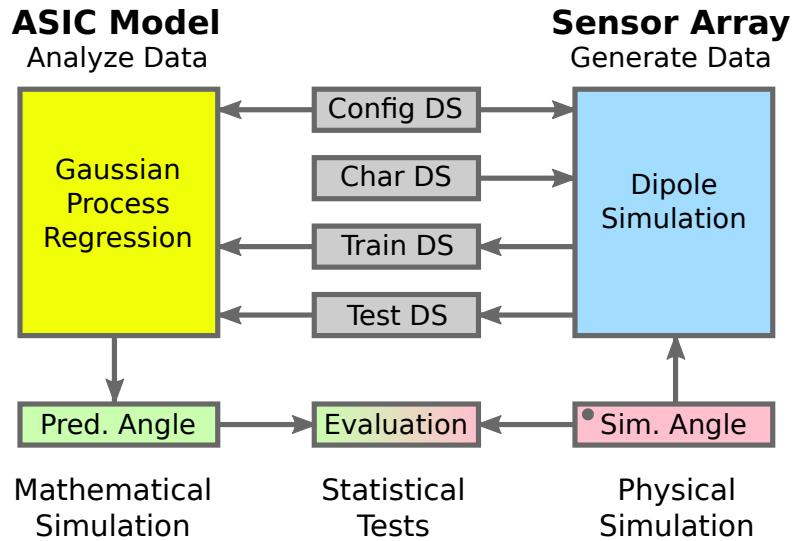


Abbildung 3.1: Simulationsaufbau im Überblick. Konzeptionelle Aufteilung der Simulationen für Sensor-Array und ASIC nach Simulationscharakter. Simulative Kopplung erfolgt, durch prozessierte Datensätze (DS). Eine statistische Auswertung bezieht sich auf angefahrene Simulationswinkel. Der Punkt kennzeichnet die Simulationswinkel eingabe und markiert den gedanklichen Startpunkt des Gesamtkonzepts.

Im ersten Simulationsschritt (Sensor-Array) werden Trainings- und Testdatensätze erzeugt. Die Datengenerierung in der Simulation basiert auf Charakterisierungsdatensätze. Diese stellen Technologieeigenschaften und Verhalten für die Simulation bereit, siehe Anhang A. Zur Datengenerierung sind physikalische Gleichungen aus Abschnitt 2.6 genutzt. Als zweiter Simulationsschritt (ASIC-Modell) folgt die Analyse der generierten Daten aus Schritt eins. Zur Analyse wird ein mathematisches Regressionsverfahren verwendet, siehe Abschnitt 2.7. Das Regressionsverfahren gewichtet Daten und macht entsprechende Vorhersagen gemäß eingestellter Regressionsziele und gewichteten Referenzdaten Anhang C. Das sind rein mathematische Vorhersagen für vorgegebene Funktionen. Ein physikalischer Gesamtbezug ist durch eine verbundene Auswertung beider Simulationsabschnitte herzustellen. Die Simulationssteuerung ist über einen gemeinsamen Konfigurationsdatensatz umgesetzt, siehe Tabelle B.1 und Tabelle C.1. Die jeweiligen Parametergruppen sind entsprechend ihrer Zugehörigkeit partiell in den Simulationsbetrieb eingebunden. Der Konfigurationsdatensatz kann je nach Bedarf erzeugt und manipuliert werden. Zur Konfigurationsgenerierung ist das Skript aus Unterabschnitt E.3.2 zu verwenden.

Die Zweischritt-Lösung bietet den Vorteil, dass zuerst verschiedene Trainings- und Testdatensätze generiert werden können. Nachfolgende und voneinander variierende Simulationen basieren hierbei auf gleichen Datensätzen. Sie sind damit vergleichbar für weitere Auswertungen, Diagnosen oder Optimierungen. Ein empfohlener Arbeitsablauf ist in Unterabschnitt E.2.5 festgehalten.

Zur Gestaltung der Simulations-Software ist ein modularer Ansatz nach Abbildung 3.2 verfolgt worden. Das modulare Konzept erhöht die Wiederverwendbarkeit des Quellcodes. Es ermöglicht einzelne Quellcodebestandteile miteinander zu kombinieren. Die Einbindung und Ausführung der Quellcodemodule aus Abschnitt E.4 erfolgt in Skripten des Abschnitt E.3. Die Software ist als Projekt in der Multi-Paradigmen-Programmiersprache Matlab umgesetzt, siehe Anhang D. Konzeptrichtlinien sind im Abschnitt E.2 festgehalten. Zusätzliche Anweisungen zur Arbeitsweise, Projektpflege, Dokumentation und Vorlagen für Skripte und Funktionen sind beigefügt. Alle Entwicklungsschritte sind mittels Git-Versionsverwaltung kommentiert und nachvollziehbar dokumentiert. Jede Quellcode- und Skript-Datei ist nach festgelegten Konventionen geschrieben worden [6]. Es ermöglicht eine automatisierte Dokumentation des gesamten Software-Projekts in Anhang E. Erzeugt ist diese mit Skripten aus Unterabschnitt E.3.1 und Unterabschnitt E.3.6.

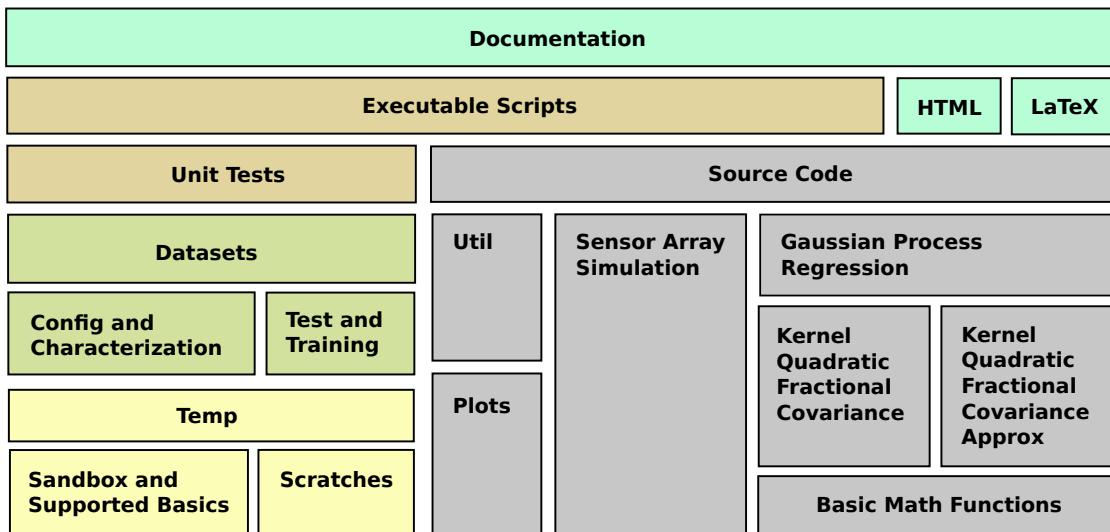


Abbildung 3.2: Blockschema Simulations-Software. Modularer Software-Gestaltung. Kernbestandteil ist funktionaler Quellcode. Quellcodemodule sind mit Datensätzen nach Bedarf in Skripten zu laden und ausführbar. Software-Dokumentation steht in HTML und LaTeX bereit. Als HTML ist diese in Matlab integriert. Entwurfsarbeiten sind nicht Bestandteil der Dokumentation, aber im Projektverzeichnis vorliegend.

## 3.2 Simulationsprozesse und Ausführung

In diesem Abschnitt der Arbeit werden Ausführungen zum Gesamt simulationsbetriebs und zur Moduleinbindung anhand von Prozessdarstellungen und Blockschemata erläutert. Jeder Simulationsabschnitt wird getrennt betrachtet. Prozesse und Schemata beziehen sich auf die Implementierung in den Anhängen und die dort beschriebenen Algorithmen. Gleichzeitig wird entsprechender Bezug zur Software-Dokumentation hergestellt, sodass ausgeführte Implementierungen zu umgesetztem Quellcode referenzierbar wird.

### 3.2.1 Sensor-Array-Simulation

Das Sensor-Array-Simulationsmodul Unterabschnitt E.4.1 ist funktional aufgebaut. Geschriebener Quellcode des Moduls mündet in eine Hauptsimulationsfunktion, siehe Unterabschnitt E.4.1.6. Diese ist mittels Skripten nach Abbildung 3.3 für den Simulationsbetrieb inklusive Konfiguration und Kennfelddatensatz zu laden.

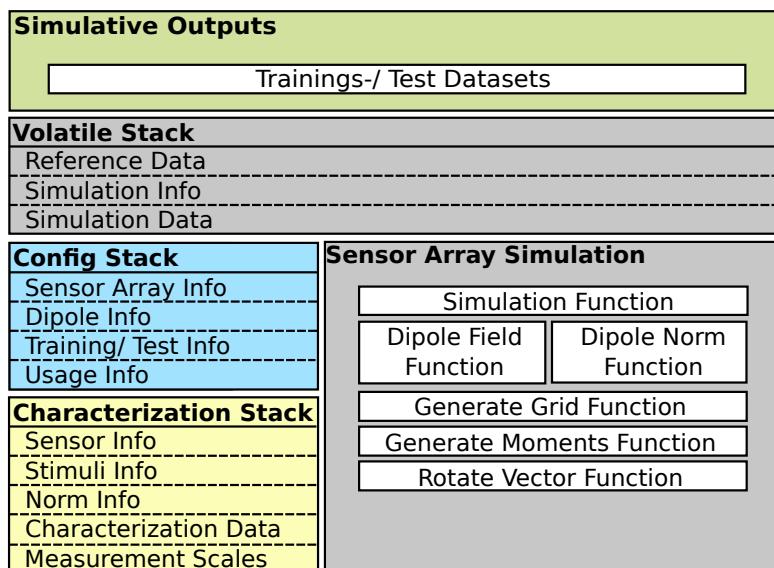


Abbildung 3.3: Blockschema Einbindung der Sensor-Array-Simulation. Das Schema beschreibt die Quellcodeeinbindung des Sensor-Array-Simulationsmodul für eine Ausführung mittels Skript-Dateien. Das Modul ist über die Modulsimulationsfunktion im Skript ausgeführt. Der Simulationsfunktion werden geladene Datensätze als entsprechende Simulations-Stacks zugeführt. Innerhalb die Funktions-Workspace sind alle Simulationsergebnisse prozessiert und zyklisch in Trainings-/ Testdatensätze zu speichern.

Das ausführende Skript der Simulation ist im Unterabschnitt E.3.3 zu finden. Es werden jeweils Trainings- und Testdatensätze gemäß eingestellter Konfiguration prozessiert. Die Sichtung der Datensätze und eine Analyse der Rohdaten ist mit Plot-Funktionen des Unterabschnitt E.4.3.3 durchzuführen. Die Plot-Funktionen können direkt ausgeführt werden. Eine Auswahl der zu sichtenden Daten ist über Nutzereingaben in der Konsole gesteuert. Erstellte Datensätze und Plots können mit Lösch-Skripten aus Unterabschnitt E.3.4 und Unterabschnitt E.3.5 entfernt werden.

Die Simulationsausführung nach Algorithmus 1 mittels Skript aus Unterabschnitt E.3.3 ist nochmals zur Verdeutlichung des Ablaufs als Prozess in Abbildung 3.4 dargestellt. Als erstes sind alle notwendigen Datensätze in den Skript-Workspace zu laden und als Eingabeargumente der Simulationsfunktion zuzuführen.

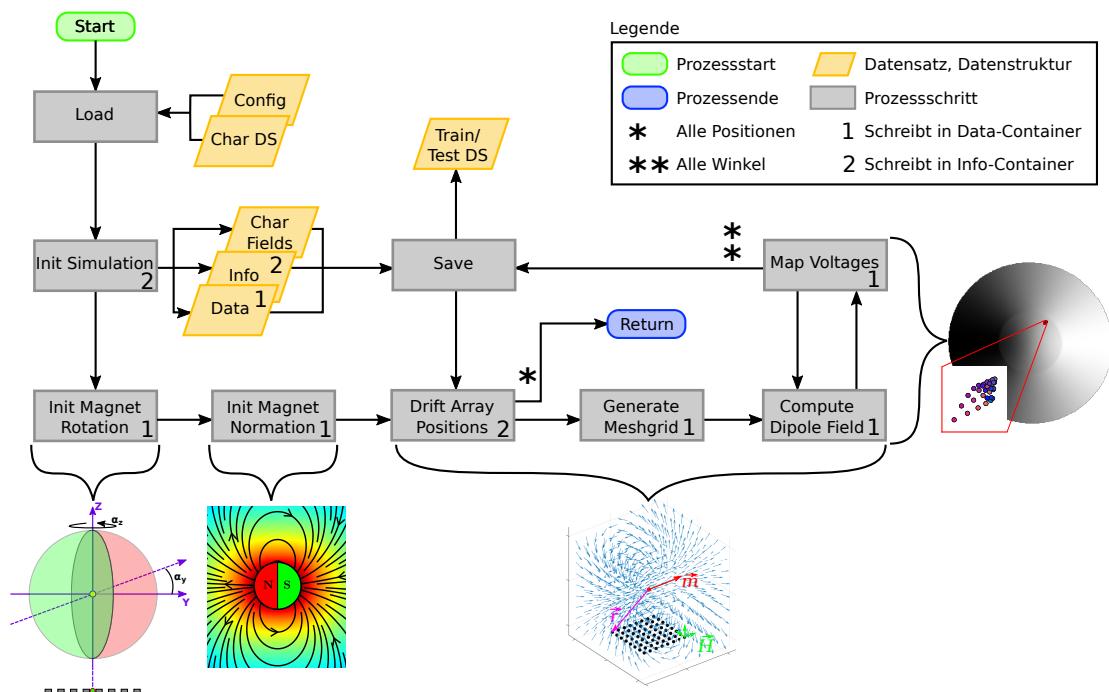


Abbildung 3.4: Sensor-Array-Simulation Prozessansicht. Prozessabfolge entsprechend der Implementierung aus Anhang B und Ausführung mit Skripten aus Abschnitt E.3 in Kombination mit Datensätzen (DS) nach Anhang A bzw. Abschnitt E.5.

Innerhalb der Funktion folgt eine schrittweise Initialisierung der Simulation nach einge stellter Konfiguration. Es werden Daten-Container angelegt, die Metadaten (Info) der Simulation und prozessierte Simulationsdaten (Data) beinhalten. Die Simulation erfolgt für fest eingestellte Sensor-, Magnet-, Rotationsparameter in Training-/ Testdaten und Verkippung. Die räumliche Lage des Sensor-Arrays ist dynamisch über Positionsvektoren in der Konfiguration gesteuert. So wird eine Positionsdrift des Arrays bei gleichbleibende Simulationsparameter umgesetzt. Die Simulation ist in verschachtelten For-Schleifen eingebettet. Die äußeren Schleifen setzen die Positionsdrift um. Die innere Schleife fährt die Magnetrotation durch. Für jede Position wird ein Datensatzpaar aus Trainings-/ Testdaten gespeichert, dass die volle Rotation enthält. Entsprechend des Schleifendurch laufs werden Info- und Data-Container aktualisiert. Anzahl der Rotationswinkel können für Trainings und Testdaten unterschiedlich eingestellt werden. Ebenfalls können unter schiedliche Rotationsauflösungen und Startphasen parametrisiert werden, siehe Unterab schnitt E.3.2.

### 3.2.2 Gauß-Prozess-Regression

Für die Verarbeitung von simulierten Sensor-Array-Datensätzen aus Unterabschnitt 3.2.1, dient das Quellcodemodule der Gauß-Prozess-Regression in Unterabschnitt E.4.2. Es ist funktional aufgebaut und erzeugt Regressionsmodelle in einer Trainingsphase. Ein erzeugtes Modell bildet zusammen mit einem Sensor-Array-Datensatz die Eingabeargumente für eine Arbeitsphase. In dieser werden Winkelvorhersagen auf Basis des Sensor-Array- Datensatzes berechnet. Das Quellcodemodul setzt sich aus drei weiteren Submodulen zusammen:

1. Mathematische Grundfunktionen, Unterabschnitt E.4.2.15
2. Kovarianzfunktion f.  $d_F^2$  Gleichung C.4 (Matrixdaten), Unterabschnitt E.4.2.14
3. Kovarianzfunktion f.  $d_E^2$  Gleichung C.4 (Vektordaten), Unterabschnitt E.4.2.13

Erläuterungen zur Einbindung von Trainingsphase und Arbeitsphase folgen zusammenfassend wie in Unterabschnitt 3.2.1 und beziehen sich auf das Demonstrationsskript für die Gauß-Prozess-Regression im Unterabschnitt E.3.7. Das Demonstrationsskript bedingt ein Trainings- und Testdatensatzpaar, dass mit gleichen Simulationsparametern in Bezug auf Position und Verkippung erstellt worden ist.

## Trainingsphase

Die Bildung des Regressionsmodell in der Trainingsphase ist modular nach Abbildung 3.5 umgesetzt. Einstiegspunkt für die optimierte Modellerstellung nach Algorithmus 7 ist die Funktion in Unterabschnitt E.4.2.11. Diese entspricht der Implementierung in Abschnitt C.2. Der Funktion werden zuvor geladene Konfigurationen sowie Trainings- und Testdaten zugeführt. Als Rückgabe erfolgt das optimierte Regressionsmodell mit der Fähigkeit zur Winkelvorhersage auf Basis von weiteren Testdatensätzen.

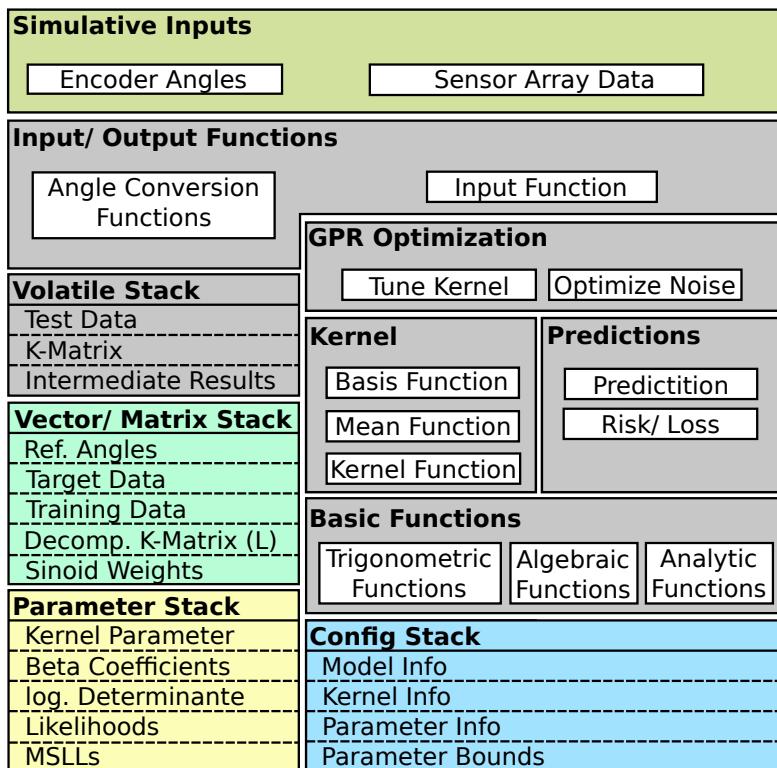


Abbildung 3.5: Blockschema Trainingsphase Regression. Veranschaulichung des Funktionsaufrufs für die Trainingsphase in einem Skript. Basierend auf übergebenen Konfigurations-Stack werden Modelfunktionalität aus Submodulen geladen und im Struct-Modell via Funktions-Handles verlinkt. Eingespeistes Trainings- und Testdatensatzpaar dient zur schrittweisen Modellinitialisierung und -Optimierung. Sich einstellende Parameterkonfigurationen werden mit Trainingsdaten als Stacks abgelegt. Resultierendes Struct-Modell enthält alle für Winkelvorhersagen nötigen Daten sowie Parameter und ihren Grenzen (engl. parameter bounds) aus der Optimierung.

Die Trainingsphase ist sehr aufwendig in ihrer Umsetzung und daher in Teilprozessschritte realisiert. Abbildung 3.7 zeigt den Hauptprozessablauf mit Weiterleitung zu den Teilprozessen in Abbildung 3.8 und Abbildung 3.9. Nach Übergabe der Regressionskonfiguration sowie der Trainings- und Testdaten, wird das Regressionsmodell in einem Struct initialisiert, siehe Funktion im Unterabschnitt E.4.2.1 und Abbildung 3.6.

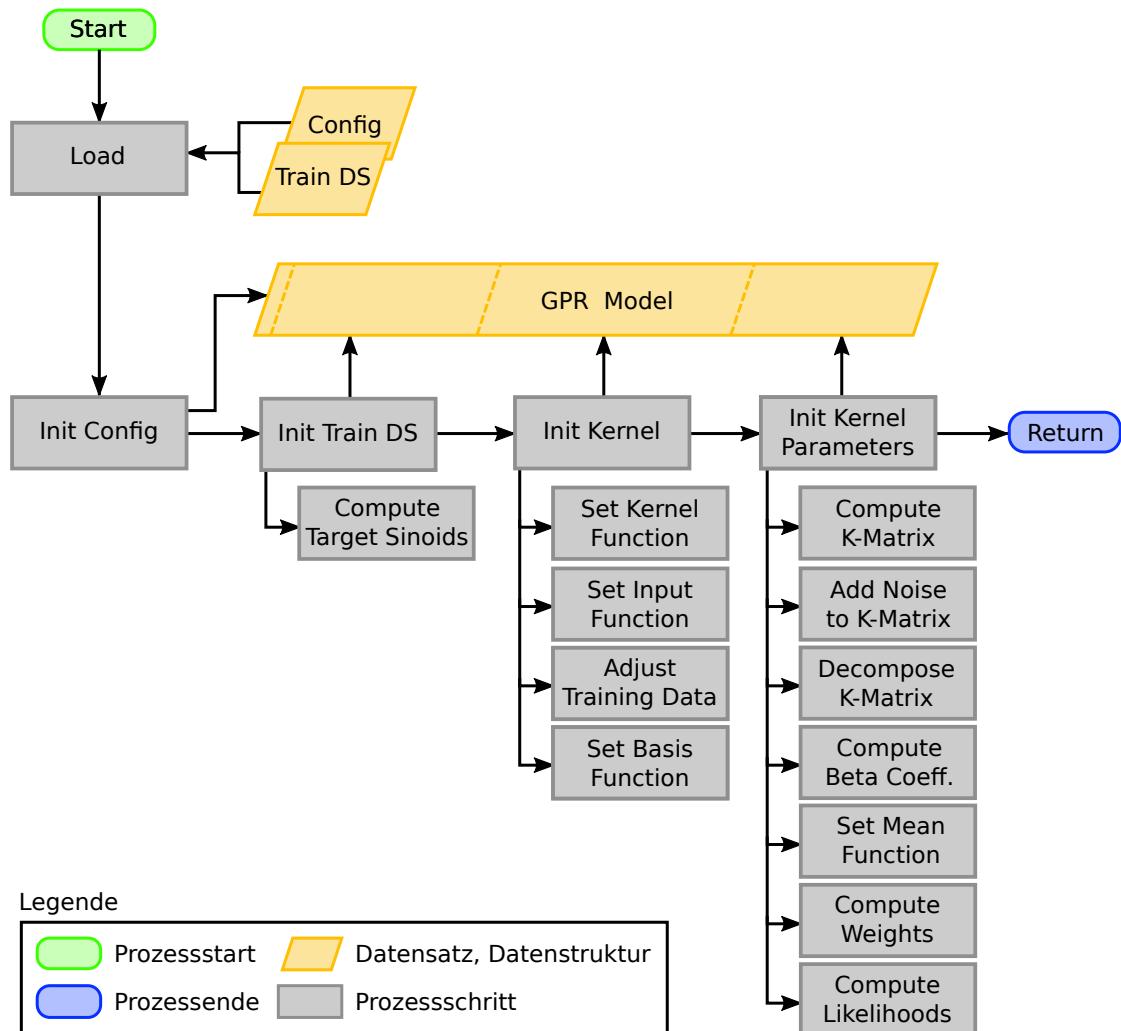


Abbildung 3.6: Regressionsinitialisierung Prozessansicht. Nach Algorithmus 2 in mehreren Teilschritten. Konfiguration, Trainingsdatensatz, Kernel-Module und abschließende Regressionsinitialisierung sind mit Funktionen aus Unterabschnitt E.4.2.2 bis Unterabschnitt E.4.2.5 realisiert.

Anschließend wird nach eingestellter Konfiguration die Rauschniveaumodellierung mittels der Matlab BayesOpt-Funktion initialisiert. Dazu werden entsprechende Berechnungen auf den Testdaten mit der Funktion aus Unterabschnitt E.4.2.12 umgesetzt. Für jedes ausprobierte Rauschniveau folgt eine Optimierung des Modells nach Abbildung 3.9 und eine darauffolgende Berechnung Modellverluste mittels Funktion aus Unterabschnitt E.4.2.10. Das Gesamtverfahren über die BayesOpt-Funktion ist ein Standardverfahren, dass durch Probieren sich einer optimalen Lösung annähert. Es werden ausprobierte Rauschniveaus und evaluierte Verluste nach Gleichung C.22 und Algorithmus 7 über alle Versuchsdurchläufe zwischengespeichert. Nach erreichen der eingestellten Durchlaufzahl für die Rauschniveaumodellierung mittels BayesOpt-Funktion, gibt die BayesOpt-Funktion alle ausprobierten Niveaus und errechneten Modellverluste in einem Result-Objekt zurück. Aus dem Result-Objekt werden für die finale Feinabstimmung das gefundene Rauschniveau am Minimum der aufgezeichneten Modellverluste entnommen.

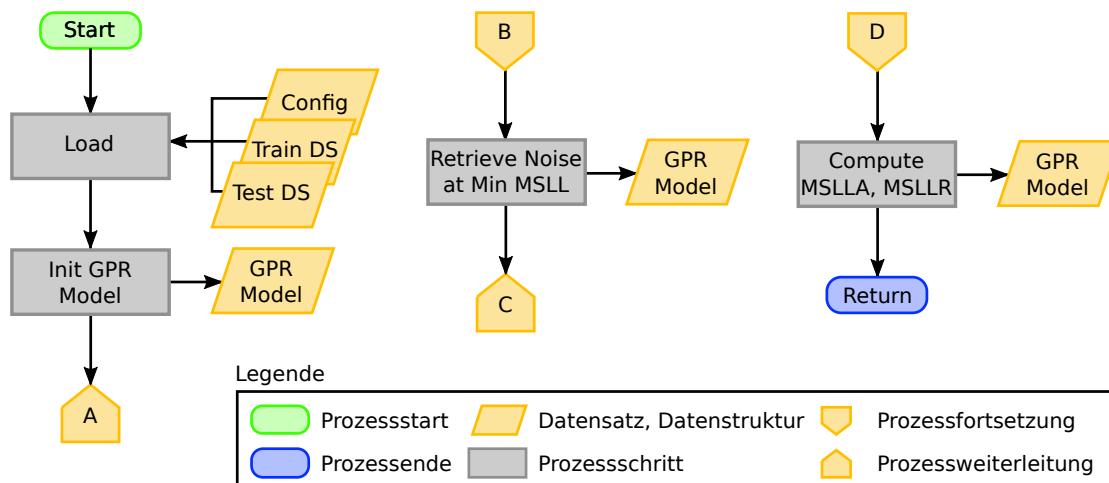


Abbildung 3.7: Regressionsoptimierung/-Generalisierung Prozessansicht. Darstellung Algorithmus 7 Teilprozessschritte. Modellinitialisierung in Abbildung 3.6. Rauschoptimierung (A, B in Abbildung 3.8) und finale Feinabstimmung (C, D in Abbildung 3.9).

Dieser Prozess wird innerhalb der BayesOpt-Funktion wiederholt, bis die eingestellte maximale Durchlaufzahl erreicht ist. Wird zu früh abgebrochen, kann bei falsch gewählten Parametergrenzen (engl. parameter bounds) keine hinreichend genaue Lösung gefunden werden. Daher ist die Durchlaufzahl bei weiten Parameter-Bounds entsprechend hoch zu wählen, sodass der Algorithmus genügend Versuche hat schlechte Wertebereiche auszuschließen.

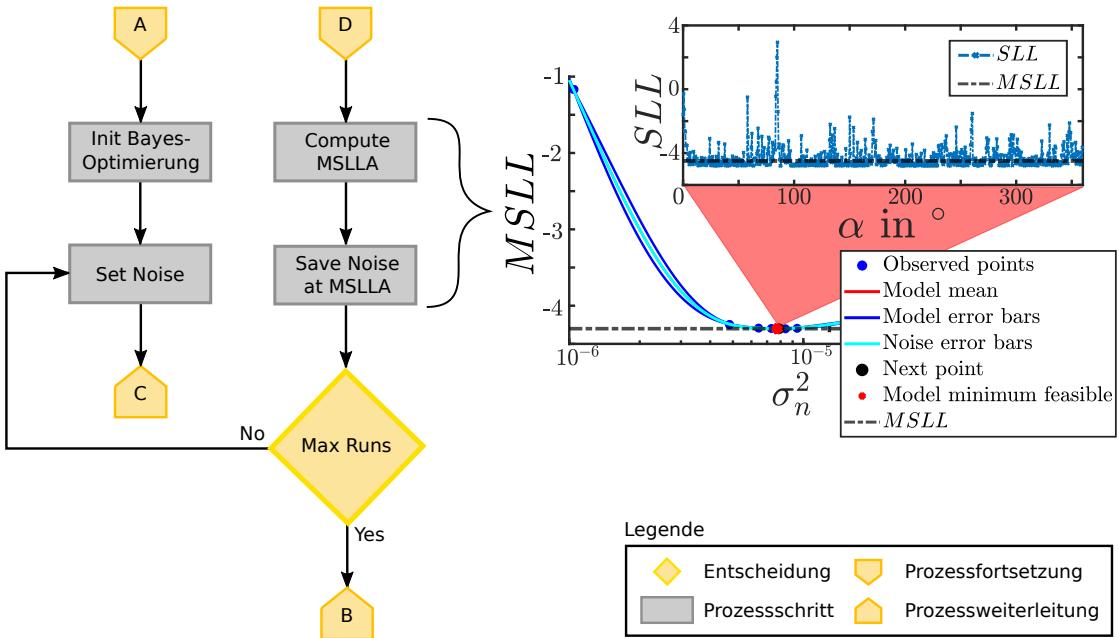


Abbildung 3.8: Rauschniveaumodellierung Prozessansicht. Kernprozess in Algorithmus 7 bzw. Abbildung 3.7. Löst Min-Kriterium nach Gleichung C.22. Die Berechnung ist als Funktions-Handle von Unterabschnitt E.4.2.12 der BayesOpt-Funktion zu übergeben. Die Rauschniveaumodellierung misst durch Probieren das Rauschniveau  $\sigma_n^2$  gegenüber dem mittleren Modellverlust  $MSLL$ . Zwischenprozess ist die Parameteroptimierung (C, D) in Abbildung 3.9

Der Zwischenprozessschritt zur inneren Parameteroptimierung nach Abbildung 3.9 ist in der Funktion Unterabschnitt E.4.2.6 implementiert. Diese bedient sich der Matlab Fmincon-Funktion um Algorithmus 5 zu lösen. Das aufzulösende Min-Kriterium aus Gleichung C.14 wird der Fmincon-Funktion als Funktions-Handle bereitgestellt. Das Min-Kriterium wird mittels der Funktion aus Unterabschnitt E.4.2.7 berechnet. Dabei wird das Regressionsmodell z.T. reinitialisiert. Dieser Prozessschritt wird auch zur finalen Feinabstimmung mit optimiertem Rauschniveau genutzt. Nach Abarbeitung aller Optimierungsprozesse steht das fertige Regressionsmodell als Eingabeargument für die Arbeitsphase bereit.

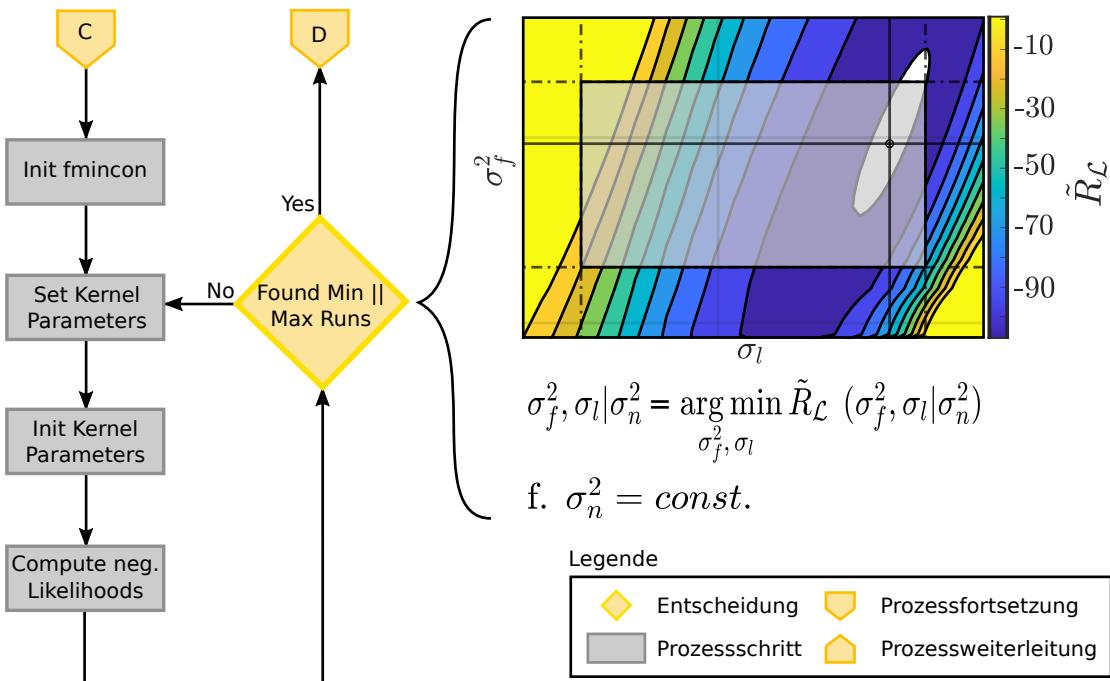


Abbildung 3.9: Regressionsparameteroptimierung Prozessansicht. Lösung des Algorithmus 5 mittels Matlab Fmincon-Funktion. Das Min-Kriterium aus Gleichung C.14 ist als Funktion in Unterabschnitt E.4.2.7 als Funktions-Handle an die Fmincon-Funktion zu übergeben. Die Parameteroptimierung für die Kovarianzfunktion  $\theta = (\sigma_f^2, \sigma_l)$  sucht, in einem durch Parameter-Bounds eingeschränkten Areal, nach lokalem Minimum  $\tilde{R}_{\mathcal{L}}$  bei logarithmisch gegeneinander aufgetragenen Parameterkombination für  $\sigma_f^2$  und  $\sigma_l$ .

## Arbeitsphase

Die Einbindung der Arbeitsphase ist im Vergleich zur Trainingsphase trivial. Das fertig optimierte Struct-Model wird einfach zusammen mit einem kompletten Testdatensatz der Vorhersage-Funktion aus Unterabschnitt E.4.2.9 und der Verlustfunktion aus Unterabschnitt E.4.2.10 übergeben. Berechnete Vorhersagen und Verluste stehen dann gemäß Abschnitt C.3 als Ergebnisvektoren bereit und können zur weiteren Auswertung genutzt werden.

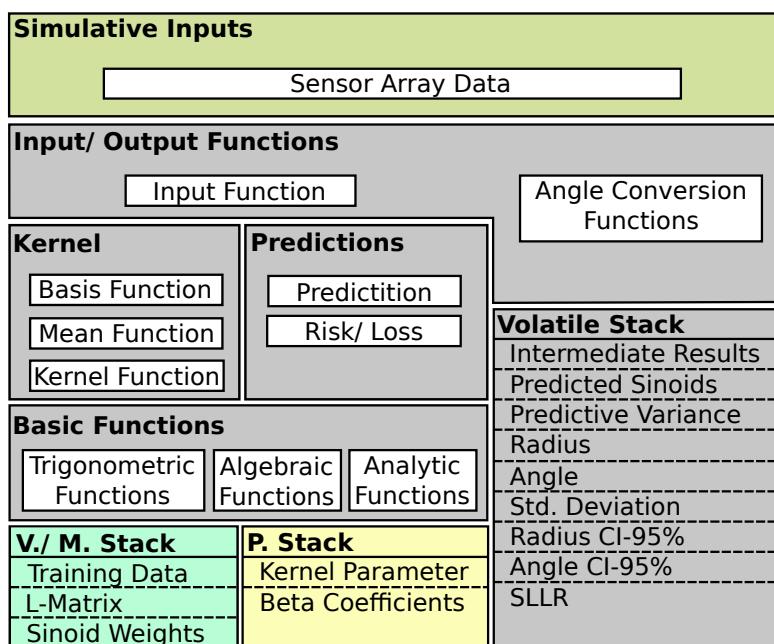


Abbildung 3.10: Blockschema Arbeitsphase Regression. Das fertig optimierte Modell wird mit minimalem Parameterbetrieb in Vorhersage- und Verlustfunktion als Argument zusammen mit den Testdaten übergeben. Einbindung ist funktional, siehe Unterabschnitt E.4.2.9 und Unterabschnitt E.4.2.10.

# **4 Erprobungs- und Optimierungsexperimente 0.0.1**

## **13.01.2021**

- Klassifizierung (Diagnose)
- Stabilitätskriterium
- Fehlererkennung Max. Mittelwert, Qualitätsmaß
- Allg. Vorgehen "Batch-Job"
- Konfigurierung der Simulationssoftware
- Kategorisieren von Fehllagen, standardisierbar, mediale, maximale
- Regressionsgrenzen durch Array-Technologie, Dämpfung Ellipsen

### **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  | Allgemeine Kreisdarstellung des euklidischen Winkelabstands . . . . .                              | 12 |
| 2.4  | Schichtmodelle dreier magnetoresistive Effekte . . . . .   | 13 |
| 2.5  | TMR Drehwinkelapplikation . . . . .  | 16 |
| 2.6  | Magnetfeldstimuli 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 . . . . .   | 33 |
| 3.2  | Blockschema Simulations-Software . . . . .   | 34 |
| 3.3  | Blockschema Einbindung der Sensor-Array-Simulation . . . . .                                       | 35 |
| 3.4  | Sensor-Array-Simulation Prozessansicht . . . . .   | 36 |
| 3.5  | Blockschema Trainingsphase Regression . . . . .  | 38 |
| 3.6  | Regressionsinitialisierung Prozessansicht . . . . .  | 39 |
| 3.7  | Regressionsoptimierung/-Generalisierung Prozessansicht . . . . .                                   | 40 |
| 3.8  | Rauschniveaumodellierung Prozessansicht . . . . .  | 41 |
| 3.9  | Regressionsparameteroptimierung Prozessansicht . . . . .   | 42 |
| 3.10 | Blockschema Arbeitsphase Regression . . . . .  | 43 |

## *Abbildungsverzeichnis*

---

|     |   |    |
|-----|---|----|
| A.1 | TDK TAS2141-AAAB Brückenkennfelder . . . . .    | 58 |
| A.2 | TDK TAS2141-AAAB Kennfeldquerschnitte . . . . . | 59 |
| B.1 | Kennfeld-Mapping . . . . .                      | 62 |
| B.2 | Sensor-Array-Datensatz Teilansicht . . . . .    | 63 |

# Tabellenverzeichnis

|     |  |    |
|-----|--|----|
| A.1 | Eckdaten TDK TAS2141-AAAB Kennfelder . . . . .         | 57 |
| B.1 | Sensor-Array-Simulationsparameter . . . . .            | 60 |
| C.1 | Gauß-Prozess-Regression-Simulationsparameter . . . . . | 65 |
| D.1 | Genutzte Software . . . . .                            | 83 |

# Algorithmenverzeichnis

|   |   |    |
|---|---|----|
| 1 | Sensor-Array-Simulation . . . . .   | 61 |
| 2 | Modellinitialisierung mit konst. Trainingsdaten und Parametern . . . . .              | 66 |
| 3 | Berechnung der Kovarianzmatrix $K(X, X \theta)$ . . . . .                             | 69 |
| 4 | Berechnung der $\beta$ Polynomkoeffizienten aus Gleichung C.10 . . . . .              | 72 |
| 5 | Modelloptimierung über Fmincon-Funktion f. $\sigma_n^2 = \text{konst.}$ . . . . .     | 76 |
| 6 | Modellvorhersage f. Sinoide eines Testwinkel mit $X_* \mapsto \alpha_*$ . . . . .     | 78 |
| 7 | Modellgeneralisierung über BayesOpt-Funktion f. alle $X_* \mapsto \alpha_*$ . . . . . | 82 |

# 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..

**Sensor-Array** Array aus geometrisch, gleichmäßig angeordneten Einzelsenoren, die messtechnische Aufgaben im Verbund bewältigen. Jeder einzelne Senor stellt dabei ein Senor-Pixel dar. Erhobene Messdaten sind entsprechend der Sensor-Array-Struktur in Matrixdatenformate zu bewerten und zu behandeln..

**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.

**Sensor-Pixel** Einzelsensor im Sensor-Array. Allein betrachtet als vollwertiger, analoger Sensor zu betrachten. Mehrere Sensor-Pixel zusammen verschaltet ergeben ein Sensor-Array..

**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.

**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](http://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](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 Unterabschnitt E.5.1 aufgeführt.

| Eigenschaft                | Wert             | Einheit            |
|----------------------------|------------------|--------------------|
| $H_x$ -Skala               | $-25 \dots 25$   | $\text{kA m}^{-1}$ |
| $H_y$ -Skala               | $-25 \dots 25$   | $\text{kA m}^{-1}$ |
| $H_x$ -Schrittweite        | 0,1961           | $\text{kA m}^{-1}$ |
| $H_y$ -Schrittweite        | 0,1961           | $\text{kA m}^{-1}$ |
| Auflösung                  | $256 \times 256$ | Pixel              |
| Wertebereich $V(H_x, H_y)$ | Normiert         | $\text{mV V}^{-1}$ |
| 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  $90^\circ$  verdreht. Die Kennfelder besitzen, je ein Minimum und Maximum, dass bei Abfahren eines Kreises auf einem Kennfeld zur  $360^\circ$  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.

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

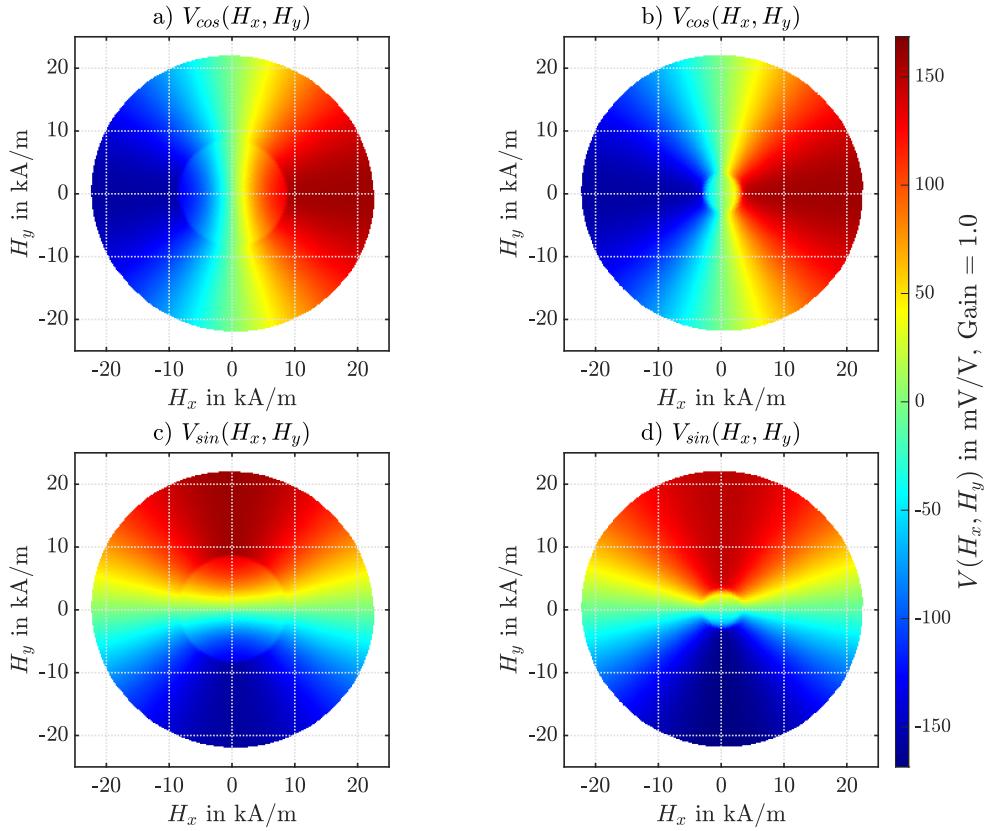


Abbildung A.1: TDK TAS2141-AAAB Brückenkennfelder. Kennfelder der Cosinus-Brücke a) und b). Kennfelder der Sinus-Brücke c) und d). a) und c) gewonnen aus steigenden Amplitudenmodulation. b) und d) gewonnen fallenden Modulation. Die Kennfelder sind normiert in  $\text{mV V}^{-1}$ . Grafik nachempfunden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

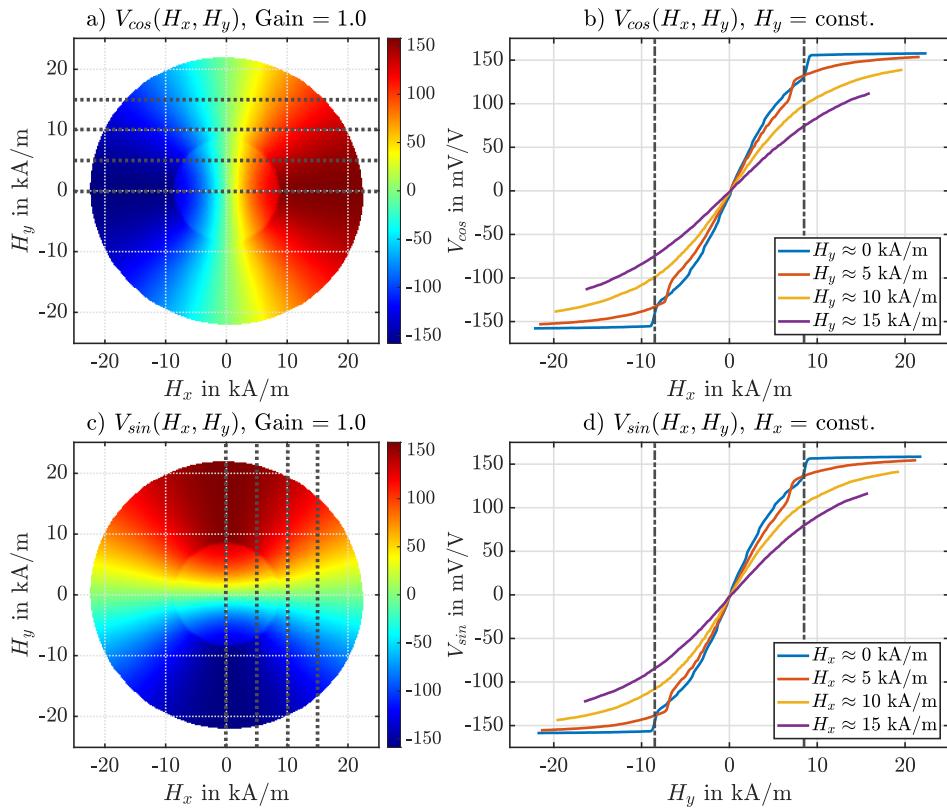


Abbildung A.2: TDK TAS2141-AAAB Kennfeldquerschnitte. Cosinus-Brücken-Kennfeld in a) und Sinus-Brücke in c). In b) und d) sind folgend der Verdrehung Querschnitte aus a) und c) aufgetragen. In b)  $V_{cos}$  f. variable  $H_x$ - und verschiedenen konst.  $H_y$ -Feldstärken. In d) vice versa für  $V_{sin}$  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 <sup>-1</sup> | Betragsfeldstärke Magnetfeldnormierung          |
|                        | $z_0$           | 1                   | mm                 | Z-Abstand Magnetfeldnormierung                  |
|                        | $m_{0mag}$      | $1 \cdot 10^6$      | A m <sup>2</sup>   | 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 angefahrenene 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**

---

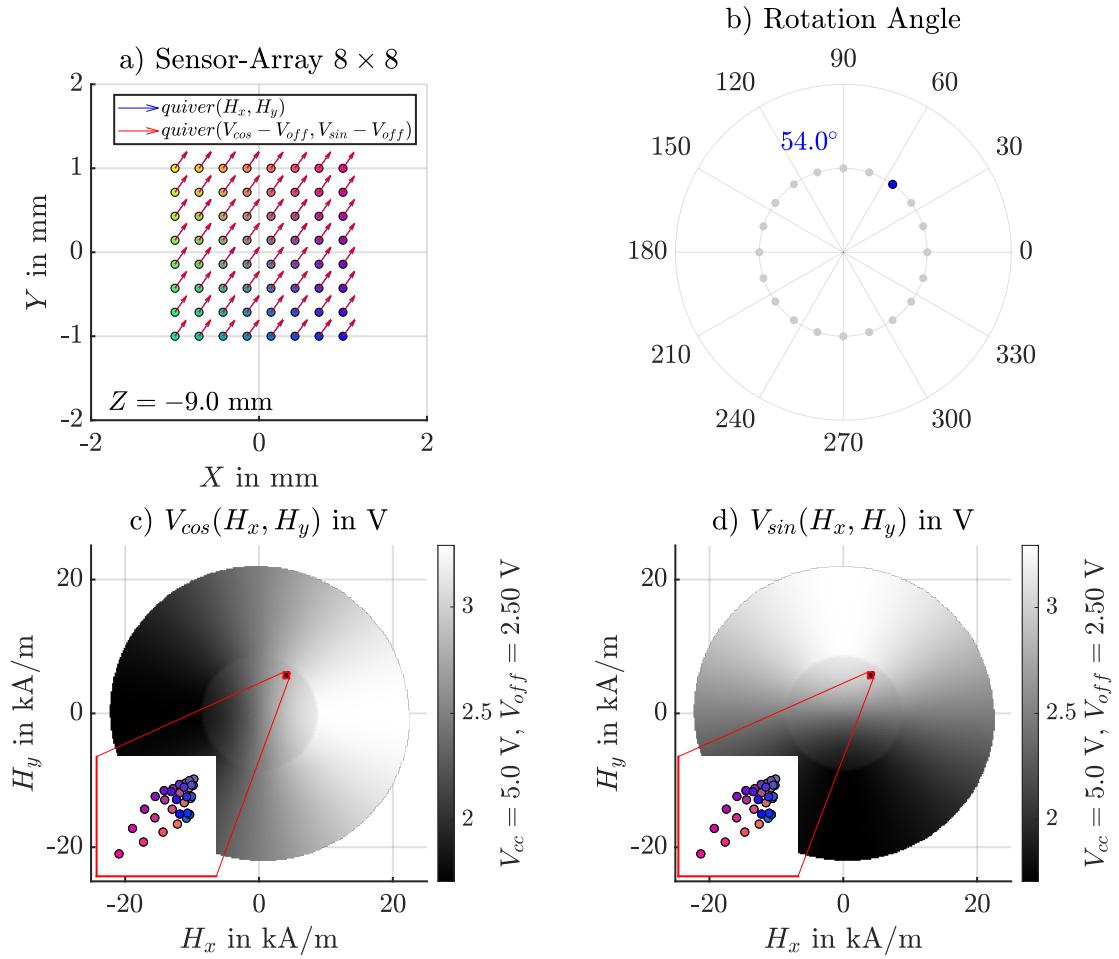


Abbildung B.1: Kennfeld-Mapping. Entnahme von Referenzspannungen aus Sensor-Kennfelder, gezeigt für einen beliebigen Simulationswinkel. In a) Meshgrid des Sensor-Arrays mit Grundposition  $(0, 0, -9)^T$  mm relativ zum Koordinatenursprung (mag. Dipol). Simulation ohne Dipol-Verkippung. b) Simulation für 20 gleich verteilte Winkel, gezeigt ist der vierte Winkel bei  $54^\circ$ . 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  $54^\circ$ . 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 des Funktionsmoduls ist in Unterabschnitt E.4.1 einzusehen.

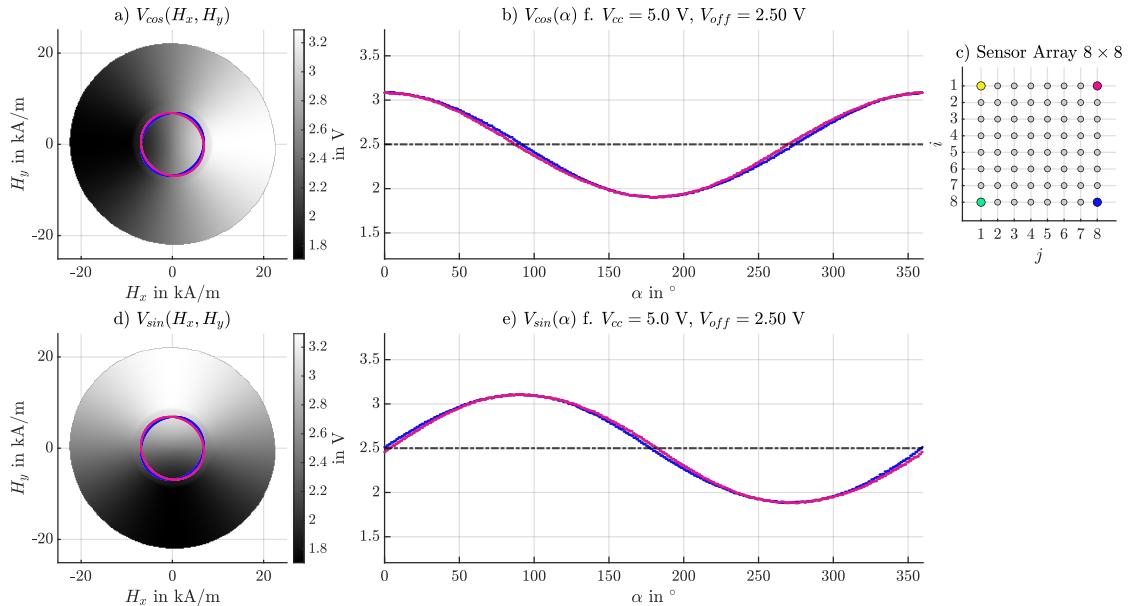


Abbildung B.2: Sensor-Array-Datensatz Teilansicht. Teilansicht der Simulationsergebnisse, entsprechend der Parametrierung nach Tabelle B.1. Es ist der gesamte Simulationsdurchlauf für die Eck-Sensor-Pixel c) gezeigt. a) und d) zeigen das Kennfeld-Mapping, für korrespondierende Feldstärken, bei einer 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 verschieden 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 Unterabschnitt E.4.2 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), ' $QFC \leftarrow d_F^2$ |
|                 | $\theta$             | (1, 1)                                 | -       | Kernel-Parametervektor $\theta$ (C.5)                     |
|                 | $\sigma_f^2$ -Bounds | (0.1, 100)                             | -       | Parameter-Bounds $\theta_1$ f. Algorithmus 5              |
|                 | $\sigma_l$ -Bounds   | (0.1, 100)                             | -       | Parameter-Bounds $\theta_2$ f. Algorithmus 5              |
|                 | $\sigma_n^2$         | $1 \cdot 10^{-6}$                      | -       | Rauschniveau, Rauschaufschaltung (C.6)                    |
|                 | $\sigma_n^2$ -Bounds | ( $1 \cdot 10^{-8}, 1 \cdot 10^{-4}$ ) | -       | Parameter-Bounds $\sigma_n^2$ 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  $X$ ,  $\alpha$  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  $\alpha_i$  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  $\sigma_f^2$  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  $\theta$

**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  $\sigma_n^2$  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].

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

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

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.

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

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

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

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

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

**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  $\beta$  Polynomkoeffizienten aus Gleichung C.10

---

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

**Result :**  $\beta$ -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,  $\approx 0$  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  $\theta$ . 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  $\sigma_n^2$  und verschiedenen Kernel-Parameter  $\theta$  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 ??, kann einen Winkelsatz oder komplette 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.  $X, \theta, \sigma_n^2$ , Testdatensatz (inkl. Testwinkel  $\alpha_*$ )  $X_* \mapsto \alpha_*$

**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  $\approx 0$ . 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  $\sigma_n^2$ . Das Bayes-Optimierungsverfahren ist in Matlab über die BayesOpt-Funktion implementiert und wird mit dem Probier-Algorithmus Improve-Per-2<sup>nd+</sup> betrieben. Entscheidend bei der Ausführung ist die Durchlaufzahl der Bayes-Optimierung, da sich das Verfahren durch Ausprobieren von  $\sigma_n^2$  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  $\geq 50$  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  $\sigma_n^2$

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  $\sigma_n^2$  f. jedes Ergebnis  $MSLL$ ;
  - end**
  6. Entnahme von  $\sigma_n^2$  bei  $\min MSLL$ ;
  7. Speichern von  $\sigma_n^2$  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  $\sigma_n^2$  zuzuführen. Das Minimierungsproblem induziert dabei für jedes ausprobierte  $\sigma_n^2$  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  $\sigma_n^2$  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<br>(Laptop OS)                               | 2 GHz Dual-Core-CPU<br>4 GB RAM<br>25 GB freier HDD-Speicher    | 18.04 LTS | 26.04.2018      |
| Windows 10 Enterprise | Windows-Betriebssystem<br>(Laptop OS)                             | 1 GHz Core-CPU<br>1 GB RAM<br>32 GB freier HDD-Speicher         | 1909      | 12.11.2020      |
| MATLAB                | Simulationssoftware<br>(Multi-Paradigmen Programmiersprache, IDE) | Intel/ AMD x86-64 CPU<br>4 GB RAM<br>3.5 GB freier HDD-Speicher | 2020b     | 17.09.2020      |
| Git                   | Versionierung<br>(Kommandozeilenprogramm)                         | -<br>-<br>-   | 2.29      | 29.10.2020      |
| Inkscape              | Vektorgrafikzeichenprogramm<br>(Grafikaufbereitung)               | 1 GHz CPU<br>256 MB RAM<br>302 MB freier HDD-Speicher           | 0.92.3    | 11.03.2018      |
| Texstudio             | Textbearbeitung f. LaTeX Dokumente (Editor)                       | -<br>-<br>24.7 MB freier HDD Speicher                           | 2.12.6    | 25.07.2020      |
| JabRef                | Literaturverwaltungsprogramm f.BibLaTeX (Editor)                  | -<br>-<br>-   | 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 0.0.7**

**18.04.2021**

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. Im Anschluss ist diese in Tex-Dateien. Als letzter Schritt sind diese zu einem LaTeX-Manualzusammengefasst 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 beigeftigt. 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. Die Software-Dokumentation ist automatisiert durch eigens dafür geschriebene Skripte erstellt worden. Diese sind in der Dokumentation enthalten.

Matlab software appendix auto generated on 18-Apr-2021 14:28:41.

## E.1 GaussianProcessDipoleSimulation

### 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 Simualtion 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 equauations to approximate a spherical magnet.

**November 06. 2019**

Prof. Dr. Klaus Jünemann supports the team around Prof. Dr.-Ing. Karl-Ragmar Riem-schneider and Thorben Schüthe with an apply of Gaussian Process learning to investigate on angle predictions for sensor array simualtion results. The attempt of the solution was working for tight set of parameter and was highly experimental with rare documentation and few set 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 guidline 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 establish 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 added TDK TAS2141 TMR characterization to the project. Thorben Schüthe provided a raw dataset which was manually modified by Tobias Wulf to dataset which is plotable and reconstructable in stimulus and characterization field investigations.

**October 31. 2020**

Tobias Wulf establish 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 unittest 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 gaussianProcess-Regression modul. Additionaly 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.

## E.2 Workflows

### 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 apobat frameworks e.g. Unitest.
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 with 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.**

### Contents

- See Also
- Project Preparation
- Project Structure
- Git Feature Branch Workflow
- Documentation Workflow
- Simulation Workflow

## 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 task 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.

### E.2.1 Project Preparation

## Project Preparation

The first steps to setup a scalable software project are none trivial and need a good structure 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 proper Matlab project structure in general. Each step is documented with screenshots to give a comprehensible explanation.

## Contents

- See Also
- Create Main Project Directory
- Create Matlab Project with Git Support
- Registrat Binaries to Git and Prepare Git Ignore Cases
- Checkout Project State and Do an Initial Commit
- Push to Remote and Backup
- Port Remote Repository to GitHub

## 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 automatacally 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 non text files because Git performs no automatic merges on file if they are not known text files. To keep the versioning Git makes a tagged copy of that file every time the file changes. That can be a very junk of memory and lets repository expand 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.

## 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 meachnism 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 direcory 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.

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.

## 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.

### **E.2.2 Project Structure**

## **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 Matlab provides strategies to add content to existing project structures and label it for script based execution of project tasks to manage project files. To add new content have a look at the links below.

## **Contents**

- See Also
- Directory Tasks
- Add New Elements

## **See Also**

- Specify Project Path
- Add Files to the Project
- Add Labels to Files

## **Directory Tasks**

### **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. Replaceable. 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/trainig** 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** 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 (HTML).

**./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.

**./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 Fractional Covariance kernel functions which bases on matrice training data.

**./src/gaussianProcessRegression/kernelQFCAPX** Approximated Quadratic Fractional 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 arround. Either made pre investigation and supported work basics are saved here.

## 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 directotry 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.

### E.2.3 Git Feature Branch Workflow

## 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.

## Contents

- See Also
- Examples

## 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.

#### E.2.4 Documentation Workflow

### 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.

### Contents

- See Also
- Script Template
- Function Template

## See Also

- Project Structure
- Display Custom Documentation
- publishProjectFilesToHTML
- exportPublishedToPdf

## Script Template

```
1 %% scriptName
2 % Detailed description of the script task and summary
3 % description of
4 % underlaying script sections.
5 %
6 %% Requirements
7 % * Other m-files required: None
8 % * Subfunctions: None
9 % * MAT-files required: None
10 %
11 %
12 %% See Also
13 % * Reference1
14 % * Reference2
15 % * Reference3
16 %
17 %
18 % Created on Month DD. YYYY by Creator. Copyright Creator YYYY.
19 %
20 % <html>
21 % <!--
22 % Hidden Clutter.
23 % Edited on Month DD. YYYY by Editor: Single line description.
```

```
24 % -->
25 % </html>
26 %
27 %
28 %% First Script Section
29 % Detailed section description of step by step executed script
   code.
30 disp("Prompt current step or meaningful information of
      variables.")
31 Enter section source code
```

```
1 %% Second Script Section
2 % Detailed section description of step by step executed script
   code.
3 disp("Prompt current step or meaningful information of
      variables.")
4 Enter section source code
```

## Function Template

```
1 %% functionName
2 % Single line summary.
3 %
4 %% Syntax
5 %   outputArg = functionName(positionalArg)
6 %   outputArg = functionName(positionalArg, optionalArg)
7 %
8 %
9 %% Description
10 % *outputArg = functionName(positionalArg)* detailed use case
   description.
11 %
12 % *outputArg = functionName(positionalArg, optionalArg)*
   detailed use case
13 % description.
```

```
14 %
15 %
16 %% Examples
17 % Enter example matlab code for each use case.
18 %
19 %
20 %% Input Argurments
21 % *positionalArg* argurment description.
22 %
23 % *optionalArg* argurment description.
24 %
25 %
26 %% Output Argurments
27 % *outputArg* argurment description.
28 %
29 %
30 %% Requirements
31 % * Other m-files required: None
32 % * Subfunctions: None
33 % * MAT-files required: None
34 %
35 %
36 %% See Also
37 % * Reference1
38 % * Reference2
39 % * Reference3
40 %
41 %
42 % Created on Month DD. YYYY by Creator. Copyright Creator YYYY.
43 %
44 % <html>
45 % <!--
46 % Hidden Clutter.
47 % Edited on Month DD. YYYY by Editor: Single line description.
48 % -->
49 % </html>
```

```
50 %
51 function [outputArg] = functionName(positionalArg, optionalArg
52 )
53     arguments
54         % validate positionalArg: dim class {validator}
55         positionalArg (1,:) double {mustBeNumeric}
56         % validate optionalArg: dim class {validator} =
57         % defaultValue
58         optionalArg (1,:) doubel {mustBeNumeric,
59                         mustBeEqualSize(positionalArg, optionalArg)} = 4
60     end
61     outputArg = positionalArg + optionalArg;
62 end
```

```
1 % Custom validation function
2 function mustBeEqualSize(a,b)
3     % Test for equal size
4     if ~ensuremath{\tilde{;}}isequal(size(a),size(b))
5         eid = 'Size:notEqual';
6         msg = 'Size of first input must equal size of second
7             input.';
8         throwAsCaller(MException(eid,msg))
9     end
10 end
```

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

### E.2.5 Simulation Workflow

## 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.

## Contents

- See Also

## See Also

- `generateConfigMat`
- `deleteSimulationDatasets`
- `generateSimulationDatasets`
- `deleteSimulationPlots`

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

## E.3 Executable Scripts

### 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.

### Contents

- compareGPRKernels
- investigateKernelParameters
- demoGPRModule
- exportPublishedToPdf
- deleteSimulationPlots
- deleteSimulationDatasets
- generateSimulationDatasets
- publishProjectFilesHTML
- generateConfigMat

#### **compareGPRKernels**

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**

Export published HTML files to a pdf manual.

### **deleteSimulationPlots**

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

### **deleteSimulationDatasets**

Delete generated simulation datasets from data path.

### **generateSimulationDatasets**

Generate simulation datasets from sensor array simulation configuration.

### **publishProjectFilesHTML**

Publish Matlab help browser integrated HTML documentation.

## **generateConfigMat**

Generate configuration for generic use or part use in different program layers.

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

### **E.3.1 publishProjectFilesToHTML**

## **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.

## **Contents**

- Requirements
- See Also
- Start Publishing Script, Clean Up and Load Config
- Remove Equation PNG Files
- Project Documentation Files
- Executable Script Files
- Source Code Functions and Classes
- Unit Test Scripts
- Build Documentation Database for Matlab Help Browser
- Open Generated Documentation.

## **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

```
1 disp('Workspace cleaned up ...');
2 clearvars;
3 clc;
4 disp('Load configuration ...');
5 try
6     load('config.mat', 'PathVariables', 'PublishOptions');
7 catch ME
8     rethrow(ME);
9 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.

```
1 yesno = input('Renew eqautions in docs [y/n]: ', 's');
2 if strcmp(yesno, 'y')
3     removeFilesFromDir(PublishOptions.outputDir, '*_eq*.png');
4 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.

```
1 disp('Publish project documentation files ...');
2 publishFilesFromDir(PathVariables.docsPath, PublishOptions,
    false, true);
3 PublishOptions.format = 'latex';
4 PublishOptions.stylesheet = PathVariables.
    publishStyleSheetPath2;
5 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 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.

```
1 disp('Publish executable scripts ...');
2 PublishOptions.evalCode = false;
3 PublishOptions.format = 'html';
4 PublishOptions.stylesheet = PathVariables.publishStyleSheetPath
    ;
5 publishFilesFromDir(PathVariables.scriptsPath, PublishOptions,
    true, true);
6 PublishOptions.format = 'latex';
7 PublishOptions.stylesheet = PathVariables.
    publishStyleSheetPath2;
8 publishFilesFromDir(PathVariables.scriptsPath, PublishOptions,
    true, 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/. Introcude 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 underlaying directory tree, verbose.

```
1 disp('Publish source code functions and classes ...');
2 PublishOptions.evalCode = false;
3 PublishOptions.format = 'html';
4 PublishOptions.stylesheet = PathVariables.publishStyleSheetPath
    ;
5 publishFilesFromDir(PathVariables.srcPath, PublishOptions, true
    , true);
6 PublishOptions.format = 'latex';
7 PublishOptions.stylesheet = PathVariables.
    publishStyleSheetPath2;
```

```
8 publishFilesFromDir(PathVariables.srcPath, PublishOptions, true  
, true);
```

## Unit Test Scripts

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

```
1 disp('Publish unit tests scripts ...');  
2 PublishOptions.evalCode = false;  
3 PublishOptions.format = 'html';  
4 PublishOptions.stylesheet = PathVariables.publishStyleSheetPath  
;  
5 publishFilesFromDir(PathVariables.unittestPath, PublishOptions,  
true, true);  
6 PublishOptions.format = 'latex';  
7 PublishOptions.stylesheet = PathVariables.  
publishStyleSheetPath2;  
8 publishFilesFromDir(PathVariables.unittestPath, PublishOptions,  
true, 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.

```
1 disp('Remove old search entries ...');
2 clearvars;
3 close all;
4 clc;
5 disp('Reload configuration after unit test execution ...');
6 try
7     load('config.mat', 'PathVariables', 'PublishOptions');
8 catch ME
9     rethrow(ME);
10 end
11 if removeFilesFromDir(PathVariables.helpsearchPath)
12     buildDocSearchDb(PublishOptions.outputDir);
13 else
14     disp('Could not remove old search entries ...');
15 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.

```
1 open(fullfile(PublishOptions.outputDir, ...
2     'GaussianProcessDipoleSimulation.html'));
3 disp('Done ...');
```

### E.3.2 generateConfigMat

## 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.

## Contents

- Requirements
- See Also
- Clean Up
- Default Plot Settings
- GeneralOptions
- Path Variables
- Publish Options
- Sensor Array Options
- Dipole Options
- Training Options
- Test Options
- GPR Options
- Save Configuration

## 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.

```
1 disp('Clean up workspace ...');
2 close all;
3 clearvars;
4 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 ...');

```
1 set(groot, 'DefaultTextInterpreter', 'latex');
2 set(groot, 'DefaultLegendInterpreter', 'latex');
3 set(groot, 'DefaultAxesTickLabelInterpreter', 'latex');
4 set(groot, 'DefaultAxesFontSize', 20);
5 set(groot, 'DefaultLineLineWidth', 2.5);
6 set(groot, 'DefaultAxesLineWidth', 1.5);
7 set(groot, 'DefaultAxesFontSize', 20);
8 % set(groot, 'DefaultAxesFontWeight', 'bold');
9 set(groot, 'DefaultTextFontSize', 20);
10 % set(groot, 'DefaultaxesFontName', 'Times new Roman')
11 % set(groot, 'DefaultlegendFontName', 'Times new Roman');
```

```
12 set(groot, 'DefaultAxesXGrid','on');
13 set(groot, 'DefaultAxesYGrid','on');
14 set(groot, 'DefaultFigureNumberTitle' , 'off');
15 set(groot, 'DefaultFigureWindowState', 'normal');
16 set(groot, 'DefaultFigureMenuBar', 'figure');
17 set(groot, 'DefaultFigureToolBar', 'figure');
18 % set(groot, 'DefaultFigureUnits', 'centimeters');
19 set(groot, 'DefaultFigurePosition', [100, 100, 800, 700]);
20 set(groot, 'DefaultFigureWindowState', 'normal')
21 set(groot, 'DefaultFigurePaperType', 'a4');
22 set(groot, 'DefaultFigurePaperUnits', 'centimeters');
23 set(groot, 'DefaultFigurePaperOrientation', 'landscape');
24 set(groot, 'DefaultFigurePaperPositionMode', 'auto');
25 set(groot, 'DefaultFigureDoubleBuffer', 'on');
26 set(groot, 'DefaultFigureRendererMode', 'manual');
27 set(groot, 'DefaultFigureRenderer', 'painters');
28 set(groot, 'DefaultTiledlayoutPadding', 'normal');
29 set(groot, 'DefaultTiledlayoutTileSpacing', 'compact');
30 set(groot, 'DefaultPolarAxesTickLabelInterpreter', 'latex');
31 set(groot, 'DefaultPolarAxesFontSize', 20);
32 % 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.

```
1 disp('Set general options ...');
2 GeneralOptions = struct;
3 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.

```
1 disp('Create current project instance to gather information ...');
2
3 % create current project instance to retrieve root information
4 projectInstance = matlab.project.currentProject;
5
6 disp('Set path variables ...');
7 PathVariables =struct;
8
9 % project root path, needs to be recreated generic to work on
10 % different machines
11 PathVariables.rootPath = projectInstance.RootFolder;
12
13 % path to data folder, which contains datasets and config.mat
14 PathVariables.dataPath = fullfile(PathVariables.rootPath, 'data
15 ');
16
17 % path to TDK TAS2141 TMR angular sensor characterization
18 % dataset
19 PathVariables.tdkDatasetPath = fullfile(PathVariables.dataPath,
20
21 ...
22
23 % path to TDK TAS2141 TMR angular sensor characterization
24 % dataset
25 PathVariables.kmz60DatasetPath = fullfile(PathVariables.
26
27 dataPath, ...
28
29
30 %NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.mat');
```

```
24 % path to config file dataset
25 PathVariables.configPath = fullfile(PathVariables.dataPath, ...
26     'config.mat');
27
28 % path to training dataset folder
29 PathVariables.trainingDataPath = fullfile(PathVariables.
30     dataPath, ...
31     'training');
32
33 % path to test dataset folder
34 PathVariables.testDataPath = fullfile(PathVariables.dataPath,
35     ...
36     'test');
37
38 % path to documentation and m-files only for documentation
39 PathVariables.docsPath = fullfile(PathVariables.rootPath, ...
40     'docs');
41
42 % path to publish html documentation output directory, helptoc.
43     xml location
44 PathVariables.publishHtmlPath = fullfile(PathVariables.docsPath
45     , 'html');
46
47 % path to save plots as images svg, eps, png, etc.
48 PathVariables.saveImagesPath = fullfile(PathVariables.
49     publishHtmlPath, ...
50     'images');
51
52 % path to latex docs folder
53 PathVariables.latexDocsPath = fullfile(PathVariables.docsPath,
54     ...
55     'latex');
56
57 % path to latex Thesis Tobias Wulf (take care if comment in)
58 % PathVariables.thesisTobiasWulf = fullfile(PathVariables.
59     latexDocsPath, ...
```

```
53 %     'BA_Thesis_Tobias_Wulf');
54
55 % path to docs export folder for Manual
56 PathVariables.exportPublishPath = fullfile(PathVariables.
57     latexDocsPath, ...
58     'Manual');
59
60 % path to style sheet for html documentation, Matlab provided
61 % style sheet
62 PathVariables.publishStyleSheetPath = fullfile(PathVariables.
63     publishHtmlPath, ...
64     'docsHtmlStyleSheet.xsl');
65
66 % path to style sheet for latex documentation, Matlab provided
67 % style sheet
68 PathVariables.publishStyleSheetPath2 = fullfile(PathVariables.
69     publishHtmlPath, ...
70     'docsLatexStyleSheet.xsl');
71
72 % path to documentation search database entries for Matlab help
73 % browser support
74 PathVariables.helpsearchPath = fullfile(PathVariables.
75     publishHtmlPath, ...
76     'helpsearch-v3');
77
78 % path to executable m-file scripts of the project
79 PathVariables.scriptsPath = fullfile(PathVariables.rootPath, 'scripts');
80
81 % path to source code files, function and class files
82 PathVariables.srcPath = fullfile(PathVariables.rootPath, 'src')
83 ;
84
85 % path to unittest files, scripts and script suite
86 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.

```
1 disp('Set publish options struct for publish function ...');
2 PublishOptions = struct('format', 'html');
3 PublishOptions.outputDir = PathVariables.publishHtmlPath;
4 PublishOptions.stylesheet = PathVariables.publishStyleSheetPath
    ;
5 PublishOptions.createThumbnail = false;
6 PublishOptions.figureSnapMethod = 'entireFigureWindow';
7 PublishOptions.imageFormat = 'png';
8 PublishOptions.maxHeight = 600;
9 PublishOptions.maxWidth = 600;
10 PublishOptions.useNewFigure = false;
11 PublishOptions.evalCode = false;
12 PublishOptions.catchError = true;
13 PublishOptions.codeToEvaluate = [];
14 PublishOptions.maxOutputLines = Inf;
15 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 cover vectors yet.

```
1 disp('Set sensor array option for geometry and behavior ...');
2 SensorArrayOptions = struct;
3
4 % Geometry of the sensor array current sensor array can be. Fix
5 % parameter.
6 % square - square sensor array with even distances to each
7 % sensor point
8 SensorArrayOptions.geometry = 'square';
9
10 % Sensor array square dimension. Fix parameter.
11 SensorArrayOptions.dimension = 8;
12
13 % Sensor array edge length in mm. Fix parameter.
14 SensorArrayOptions.edge = 2;
15
16 % Sensor array simulated supply voltage in volts. Fix parameter
17 %
18 SensorArrayOptions.Vcc = 5;
19
20 % Sensor array simulated offset voltage for bridge outputs in
21 % volts. Fix
22 % paramter.
23 SensorArrayOptions.Voff = 2.5;
24
25 % Senor array voltage norm factor to recalculate norm bridge
26 % outputs to
27 % given supply voltage and offset voltage, current normin is mV
28 % /V which
29 % implements factor of 1e3. Fix paramter.
30 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.

```
1 disp('Set dipole options to calculate magnetic stimulus ...');
2 DipoleOptions = struct;
3
4 % Radius in mm of magnetic sphere in which the magnetic dipole
5 % is centered.
6 % So it can be seen as z-offset to the sensor array. Fix
7 % parameter.
8 DipoleOptions.sphereRadius = 2;
9
10 % H-field magnitude to multiply of generated and relative
11 % normed dipole
12 % H-fields, the norming is done in zero position of [0 0 z0 +
13 % sphere radius] for
14 % 0 deg due to the position of the magnetic moment [-1 0 0] x
15 % and y components
16 % are not relevant, norming without tilt. Magnitude in kA/m.
17 % The magnitude
18 % refers that the sphere magnet has this H-field magnitude in a
19 % certain distance
20 % z0 in example sphere with 2mm sphere radius has a H magnitude
21 % of 200kA/m in
22 % 5mm distance. Standard field strength for ferrite sphere
23 % magnets are between
24 % 180 and 200kA/m. Fix parameter.
25 DipoleOptions.H0mag = 200;
26
27 % Distance in zero position of the spherical magnet in which
28 % the imprinted
```

```
19 % H-field strength magnitude takes effect. Together with the
20 % sphere radius and
21 % and the imprinted field strength magnitude the distance in
22 % rest position
23 % characterizes the spherical magnet to later relative
24 % positions of the sensor
25 % array and generated dipole H-fields in rotation simulation.
26 % In mm. Fix
27 % parameter.
28 DipoleOptions.z0 = 1;
29
30 % Magnetic moment magnitude attach rotation to the dipole field
31 % at a
32 % certain position with x, y and z components. Choose a huge
33 % value to
34 % prevent numeric failures, by norming the factor is eliminated
35 % later. Fix
36 % parameter.
37 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.

```
1 disp('Set training options to generate dataset ...');
2 TrainingOptions = struct;
3
4 % Use case of options define what dataset it is and where to
5 % save resulting
6 % datasets by simulation function. Fix parameter.
7 TrainingOptions.useCase = 'Training';
```

```
8 % Sensor array relative position to dipole magnet as position
  vector with
9 % x, y and z posiotn in mm. Negative x for left shift, negative
  y for up
10 % shift and negative z to place the layer under the dipole
  decrease z to
11 % increase the distance. The z-position will be subtracted by
  dipole sphere
12 % radius in simulation. So there is an offset given by the
  sphere radius.
13 % Loop parameters.
14 TrainingOptions.xPos = [0,];
15 TrainingOptions.yPos = [0,];
16 TrainingOptions.zPos = [7,];
17
18 % Dipole tilt in z-axes in degree. Fix parameter.
19 TrainingOptions.tilt = 0;
20
21 % Resolution of rotaion in degree, use same resoultion in
  training and test
22 % datasets to have the ability to back reference the index to
  fullscale
23 % test data sets. In degree. Fix parameter.
24 TrainingOptions.angleRes = 0.5;
25
26 % Phase index applies a phase offset in the rotation, it is
  used as phase index
27 % to a down sampling to generate even distributed angles of a
  full scale
28 % rotation. Offset index of full rotation. In example a full
  scale rotation from
29 % 0 to 360 - angleRes returns 720 angles, if nAngles is set to
  7 it returns 7
30 % angles [0, 51.5, 103, 154.5, 206, 257.5, 309]. To get a phase
  shift of 11 set
```

```
31 % phaseIndex to 22 a multiple of the resolution angleRes and
32 % get
33 % [11, 62.5, 114, 165.5, 217, 268.5, 320]. Must be positive
34 % integer. Fix
35 % parameter.
36 TrainingOptions.phaseIndex = 0;
37
38 % Number rotaion angles, even distribute between 0 and 360 with
39 % respect
40 % to the resolution, even down sampling. To generate full scale
41 % the number
42 % relatead to the resolution or fast generate but wrong number
43 % set it to 0 to
44 % generate full scale rotation too. Fix Parameter.
45 TrainingOptions.nAngles = 20;
46
47 % Charcterization datset to use in simulation. Current
48 % available datasets are
49 % TDK - for characterization dataset of TDK TAS2141 TMR sensor
50 % KMZ60 - for characterization dataset of NXP KMZ60 AMR sensor
51 TrainingOptions.BaseReference = 'TDK';
52
53 % Characteraztion field which should be load as refernce image
54 % from
55 % characterization data set, in TDK dataset are following
56 % fields. In the
57 % current dataset Rise has the widest linear plateau with a
58 % radius of ca.
59 % 8.5 kA/m. Fix parameter.
60 % Rise - Bridge outputs for rising stimulus amplituded
61 % Fall - Bridge outputs for falling stimulus amplitude
62 % All - Superimposed bridge outputs
63 % Diff - Differentiated bridge outputs
64 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.

```
1 disp('Set test options to generate dataset ...');
2 TestOptions = struct;
3
4 % Use case of options define what dataset it is and where to
   save resulting
5 % datasets by simulation function. Fix Parameter.
6 TestOptions.useCase = 'Test';
7
8 % Sensor array relative position to dipole magnet as position
   vector with
9 % x, y and z posiotn in mm. Negative x for left shift, negative
   y for up
10 % shift and negative z to place the layer under the dipole
    decrease z to
11 % increase the distance. The z-position will be subtracted by
    dipole sphere
12 % radius in simulation. So there is an offset given by the
    sphere radius.
13 % Loop parameter.
14 TestOptions.xPos = [0,];
15 TestOptions.yPos = [0,];
16 TestOptions.zPos = [7,];
17
18 % Dipole tilt in z-axes in degree. Fix parameter.
19 TestOptions.tilt = 0;
20
21 % Resolution of rotaion in degree, use same resoultion in
   training and test
22 % datasets to have the ability to back reference the index to
   fullscale
```

```
23 % test data sets. In degree. Fix parameter.  
24 TestOptions.angleRes = 0.5;  
25  
26 % Phase index applies a phase offset in the rotation, it is  
    used as phase index  
27 % to a down sampling to generate even distributed angles of a  
    full scale  
28 % rotation. Offset index of full rotation. In example a full  
    scale rotation from  
29 % 0 to 360 - angleRes returns 720 angles, if nAngles is set to  
    7 it returns 7  
30 % angles [0, 51.5, 103, 154.5, 206, 257.5, 309]. To get a phase  
    shift of 11 set  
31 % phaseIndex to 22 a multiple of the resolution angleRes and  
    get  
32 % [11, 62.5, 114, 165.5, 217, 268.5, 320]. Must be positive  
    integer. Fix  
33 % parameter.  
34 TestOptions.phaseIndex = 0;  
35  
36 % Number rotaion angles, even distribute between 0 and 360 with  
    respect  
37 % to the resolution, even down sampling. To generate full scale  
    the number  
38 % relatead to the resolution or fast generate but wrong number  
    to 0 to  
39 % generate full scale rotation. Fix parameter.  
40 TestOptions.nAngles = 720;  
41  
42 % Charcterization datset to use in simulation. Current  
    available datasets are  
43 % TDK - for characterization dataset of TDK TAS2141 TMR sensor  
44 % KMZ60 - for characterization dataset of NXP KMZ60 AMR sensor  
45 TestOptions.BaseReference = 'TDK';  
46
```

```

47 % Characteraztion field which should be load as reference image
48 % from
49 % characterization data set, in TDK dataset are following
50 % fields. In the
51 % current dataset Rise has the widest linear plateau with a
52 % radius of ca.
53 % 8.5 kA/m. Fix parameter.
54 % Rise - Bridge outputs for rising stimulus amplituded
55 % Fall - Bridge outputs for falling stimulus amplitude
56 % All - Superimposed bridge outputs
57 % Diff - Differentiated bridge outputs
58 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 frobnius 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.

```

1 disp('Set test options to generate GPR model ...');
2 GPROptions = struct();
3
4 % Set kernel function to compute covariance matrix, vectors or
5 % test point
6 % covariance. Current available covariance functions are:
7 % QFC      - Quadratic Frobenius Covariance with exact distance.
8 % QFCAPX - Quadratic Frobenius Covariance with approximated
%               distance of triangle
9 %               inequation of matrix norm, minimizes training data
%               to a vector.

```

```
9 GPROptions.kernel = 'QFC';
10
11 % Initial theta values as vector of [s2f, sl] variance and
12 % length scale
13 % parameter of the quadratic frobenius covariance function.
14 % Empirical
15 % tested start values are the sensor array dimension as length
16 % scale and a small
17 % value as variance factor. Set variance bounds to 1 for 1 on
18 % the diagonal of
19 % the covariance matrix. Only sl will be tuned in the process.
20 % Set sf2 not one
21 % it will be tuned both. Tuning both the vertical scale s2f and
22 % horizontal scale
23 % 2*sl^2 can lead to imbalance of cosine and sine prediction
24 % indicated by
25 % diverging log likelihoods for cosine and sine prediction.
26 % [s2f , sl]
27 GPROptions.theta = [1, 1];
28
29 % Set lower and upper bounds to optimize kernel parameters
30 % theta which is a
31 % vector of covariance parameter covariance variance parameter
32 % s2f and lenght
33 % scale parameter sl. These bounds must be set to prevent an
34 % overfitting in
35 % tuning the kernel parameter. If the bounds are to tight in
36 % relation of dataset
37 % number in variation the prediction losses its generalization.
38 % If the bounds are to tight in generel the tuning and
39 % optimization procedure
40 % cannot dismiss bad set points and tries to reach them over
41 % and over,
42 % the causes a limitting which would be break through if the
43 % procedure reaches
```

```
30 % the point evaluated as bad set point. If the bound are too
   wide in relation of
31 % number in dataset variousity the mean error raises. The model
   is to complex
32 % then. Try to keep up simple modles.
33 GPROptions.s2fBounds = [0.1, 100];
34 GPROptions.s1Bounds = [0.1, 100];
35
36 % Set initial noise variance to add noise along the diagonal of
   the covariance
37 % matrix to predict noisy observation. Set to small values or
   even 0 to get
38 % noise free observations.
39 GPROptions.s2n = 1e-06;
40
41 % Set lower and upper bounds for noise adjustment in computing
   the covariance
42 % matrix for noisy observations. These bounds prevent the GPR
   of overfitting in
43 % the noise optimization procedure. The default noise at
   initialization is 1e-5.
44 GPROptions.s2nBounds = [1e-08, 1e-04];
45
46
47 % Set number of outer optimization runs. For wide parameter
   bounds it is
48 % recommended to set the number of runs to min 30 otherwise the
   bayes
49 % optimization runs to short in finding error bounds and left
   with not good
50 % optimized parameters.
51 GPROptions.OptimRuns = 20;
52
53
54 % Set standardized logarithmic loss for bayes optimization of
   s2n with MSLL.
```

```

55 % MSLL resutls as mean of chosen SLL.
56 % SLLA - loss by simulation angles
57 % SLLR - loss by radius = 1 (unit circle)
58 GPROptions.SLL = 'SLLA';
59
60 % Enables mean function and offset and amplitude correction.
61 % Set basis function to compute H matrix of training points and
62 % h vector of
63 % test point. Current available basis function are:
64 % zero - init GPR as zero mean GPR m(x) = 0
65 % poly - init GPR with mean correction m(x) = H' * beta, where
66 % H is a matrix
67 % polynom mean vectors at each observation points
68 % h(x) = [1; x; x^2; x^3; ...] and beta are coefficients
69 % of the polynom.
70 % For QFC kernel x = ||X||_F
71 GPROptions.mean = 'zero';
72
73 % Polynom degree for mean poly degree option 0 for constanat, 1
74 % for 1 + x,
75 % 2 fo 1 + x + x^2 and so on. Takes only effects if mean = '
76 % poly'. Maximum
77 % polynom degree is 7.
78 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.

```

1 disp('Create config.mat ...');
2 timestamp = datestr(now, GeneralOptions.dateFormat);
3 save(PathVariables.configPath, ...
4     'timestamp', ...
5     'GeneralOptions', ...

```

```
6  'PathVariables', ...
7  'PublishOptions', ...
8  'SensorArrayOptions', ...
9  'DipoleOptions', ...
10 'TrainingOptions', ...
11 'TestOptions', ...
12 'GPROptions', ...
13 '-v7.3', '-nocompression');
```

### **E.3.3 generateSimulationDatasets**

## **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.

## **Contents**

- Requirements
- See Also
- Load Configuration and Characterization Dataset
- Generate Training Datasets
- Generate Test 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.

```
1 try
2     clearvars;
3     close all;
4     disp('Load configuration ...');
5     load('config.mat', 'GeneralOptions', 'PathVariables', ...
6           'SensorArrayOptions', 'DipoleOptions', ...
7           'TrainingOptions', 'TestOptions');
8     disp('Load characterization dataset ...');
9     switch TrainingOptions.BaseReference
10        case 'TDK'
11            TrainingCharDataset = load(PathVariables.
12                tdkDatasetPath);
13        case 'KMZ60'
14            TrainingCharDataset = load(PathVariables.
15                kmz60DatasetPath);
16        otherwise
17            error('Unknow characterization dataset in config.')
18            ;
19    end
20
21    switch TestOptions.BaseReference
```

```
19     case 'TDK'
20         TestCharDataset = load(PathVariables.tdkDatasetPath
21                             );
22     case 'KMZ60'
23         TestCharDataset = load(PathVariables.
24                             kmz60DatasetPath);
25     otherwise
26         error('Unknow characterization dataset in config.')
27     ;
28 end
29 catch ME
30     rethrow(ME)
31 end
```

## Generate Training Datasets

Generate training dataset from configuration and characterization dataset.

```
1 disp('Generate training datasets ...');
2 simulateDipoleSquareSensorArray(GeneralOptions, PathVariables,
3     ...
4     SensorArrayOptions, DipoleOptions, TrainingOptions,
5     TrainingCharDataset)
```

## Generate Test Datasets

Generate test dataset from configuration and characterization dataset.

```
1 disp('Generate test datasets ...');
2 simulateDipoleSquareSensorArray(GeneralOptions, PathVariables,
3     ...
4     SensorArrayOptions, DipoleOptions, TestOptions,
5     TestCharDataset)
```

### E.3.4 deleteSimulationDatasets

## deleteSimulationDatasets

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

### Contents

- Requirements
- See Also
- Load Path to Clean Up
- Delete Datasets

### Requirements

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

### See Also

- removeFilesFromDir
- generateConfigMat
- 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.

```
1 try
2     clearvars;
3     close all;
4     load('config.mat', 'PathVariables')
5     disp('Delete from ...')
6     disp(PathVariables.trainingDataPath);
7     disp(PathVariables.testDataPath);
8 catch ME
9     rethrow(ME)
10 end
```

## Delete Datasets

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

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

### **E.3.5 deleteSimulationPlots**

## **deleteSimulationPlots**

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

### **Contents**

- Requirements
- See Also
- Load Path to Clean Up
- Delete Dataset Plots

### **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.

```
1 try
2     clearvars;
3     close all;
4     load('config.mat', 'PathVariables')
5     disp('Delete from ...')
6     path = PathVariables.saveImagesPath;
7 catch ME
8     rethrow(ME)
9 end
```

## Delete Dataset Plots

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

```
1 ext = ["fig" "svg" "eps" "pdf" "avi"];
2 pat = "*";
3
4 for e = ext
5     asw = removeFilesFromDir(path, join([pat, e], "."));
6     fprintf('Deleted pattern %s.%s %s\n', pat, e, string(asw));
7 end
```

### E.3.6 exportPublishedToPdf

## exportPublishedToPdf

Export Matlab generated Tex documentation (publish) to combined LaTeX index file ready compile to appendix manual.

## Contents

- Requirements
- See Also
- Start Exporting Script, Clean Up and Load Config
- Define Manual TOC
- Scan for Tex Files
- Export Tex
- Write TOC to LaTeX File

## Requirements

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

## See Also

- generateConfigMat
- 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

```
1 disp('Workspace cleaned up ...');
2 clearvars;
3 clc;
4 disp('Load configuration ...');
5 try
6     load('config.mat', 'PathVariables');
7 catch ME
8     rethrow(ME);
9 end
```

## Define Manual TOC

The maual 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.

```
1 toc = ["section",
2         "tex";
3         "section"
4         "subsection",
5         "subsection",
6         "subsection",
7         "subsection",
8         "subsection",
9         "section",
10        "subsection",
11        "subsection",
12        "subsection",
13        "subsection",
14        "subsection"];
```

```
14      "subsection",           "exportPublishedToPdf.tex";
15      "subsection",           "demoGPRModule.tex";
16      "subsection",           "investigateKernelParameters.tex";
17      "subsection",           "compareGPRKernels.tex";
18      "section",              "Source_Code.tex";
19      "subsection",           "sensorArraySimulation.tex";
20      "subsubsection",         "rotate3DVector.tex";
21      "subsubsection",         "generateDipoleRotationMoments.tex
22          ";
22      "subsubsection",         "generateSensorArraySquareGrid.tex
23          ";
23      "subsubsection",         "computeDipoleH0Norm.tex";
24      "subsubsection",         "computeDipoleHField.tex";
25      "subsubsection",         "simulateDipoleSquareSensorArray.
26          tex";
26      "subsection",           "gaussianProcessRegression.tex";
27      "subsubsection",         "initGPR.tex";
28      "subsubsection",         "initGPROptions.tex";
29      "subsubsection",         "initTrainDS.tex";
30      "subsubsection",         "initKernel.tex";
31      "subsubsection",         "initKernelParameters.tex";
32      "subsubsection",         "tuneKernel.tex";
33      "subsubsection",         "computeTuneCriteria.tex";
34      "subsubsection",         "predFrame.tex";
35      "subsubsection",         "predDS.tex";
36      "subsubsection",         "lossDS.tex";
37      "subsubsection",         "optimGPR.tex";
38      "subsubsection",         "computeOptimCriteria.tex";
39      "subsubsection",         "kernelQFCAPX.tex";
40      "paragraph",            "QFCAPX.tex";
41      "paragraph",            "meanPolyQFCAPX.tex";
42      "paragraph",            "initQFCAPX.tex";
43      "subsubsection",         "kernelQFC.tex";
44      "paragraph",            "QFC.tex";
45      "paragraph",            "meanPolyQFC.tex";
46      "paragraph",            "initQFC.tex";
```

```
47     "subsubsection",      "basicMathFunctions.tex";
48     "paragraph",         "sinoids2angles.tex";
49     "paragraph",         "angles2sinoids.tex";
50     "paragraph",         "decomposeChol.tex";
51     "paragraph",         "frobeniusNorm.tex";
52     "paragraph",         "computeInverseMatrixProduct.tex";
53     "paragraph",         "computeTransposeInverseProduct.tex
54             ";
54     "paragraph",         "addNoise2Covariance.tex";
55     "paragraph",         "computeAlphaWeights.tex";
56     "paragraph",         "computeStdLogLoss.tex";
57     "paragraph",         "computeLogLikelihood.tex";
58     "paragraph",         "estimateBeta.tex";
59     "subsection",        "util.tex";
60     "subsubsection",      "removeFilesFromDir.tex";
61     "subsubsection",      "publishFilesFromDir.tex";
62     "subsubsection",      "plotFunctions.tex";
63     "paragraph",          "plotTDKCharDataset.tex";
64     "paragraph",          "plotTDKCharField.tex";
65     "paragraph",          "plotTDKTransferCurves.tex";
66     "paragraph",          "plotKMZ60CharDataset.tex";
67     "paragraph",          "plotKMZ60CharField.tex";
68     "paragraph",          "plotKMZ60TransferCurves.tex";
69     "paragraph",          "plotDipoleMagnet.tex";
70     "paragraph",          "plotSimulationDataset.tex";
71     "paragraph",          "plotSingleSimulationAngle.tex";
72     "paragraph",          "plotSimulationSubset.tex";
73     "paragraph",          "plotSimulationCosSinStats.tex";
74     "paragraph",          "plotSimulationDatasetCircle.tex";
75     "section",            "Datasets.tex";
76     "subsection",          "TDK_TAS2141_Characterization.tex";
77     "subsection",          "NXP_KMZ60_Characterization.tex";
78     "subsection",          "Config_Mat.tex";
79     "subsection",          "Training_and_Test_Datasets.tex";
80     "section",            "Unit_Tests.tex";
81     "subsection",          "runTests.tex";
```

```
82 "subsection",      "removeFilesFromDirTest.tex";
83 "subsection",      "rotate3DVectorTest.tex";
84 "subsection",      "generateDipoleRotationMomentsTest.
85           tex";
86 "subsection",      "generateSensorArraySquareGridTest.
87           tex";
88 "subsection",      "computeDipoleH0NormTest.tex";
89 "subsection",      "computeDipoleHFieldTest.tex";
90 "subsection",      "tiltRotationTest.tex";];
91
92 nToc = length(toc);
93 fprintf("%d toc entries remarked ...\\n", nToc);
```

## Scan for Tex Files

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

```
1 disp('Scan for published files ...');
2 TEX = dir(fullfile(PathVariables.publishHtmlPath, '*.tex'));
3 if nToc ~= length(TEX)
4     warning([
5         'TOC (%d) length and found Tex (%d) files are diverging
6         .', ...
7     nToc, length(TEX));
8 end
```

## Export Tex

Export found Tex files to Manual file. Each file gets its own representation. Filename is kept. Write files into Manual folder under LaTeX subdirectory in docs path. Get filename, move to new path. Write Manual.

```
1 disp('Export published Tex to Manual ...');
2 fprintf('Source: %s\n', TEX(1).folder);
3 fprintf('Destination: %s\n', PathVariables.exportPublishPath);
4 for ftex = TEX'
5     disp(ftex.name);
6     sourcePath = fullfile(ftex.folder, ftex.name);
7     destinationPath = fullfile(...,
8         PathVariables.exportPublishPath, ftex.name);
9     try
10         [status, msg] = movefile(sourcePath, destinationPath);
11         % disp(cmdout);
12         if status ~= 1
13             error('Export failure.');
14         end
15     catch ME
16         disp(msg);
17         rethrow(ME)
18     end
19 end
```

## Write TOC to LaTeX File

Wirete TOC to LaTeX file and generate for each file a subimport along toc content line with marked toc depth.

```
1 disp('Write TOC to Manual.tex ...');
2 fileID = fopen(fullfile(...,
3     PathVariables.exportPublishPath, 'Manual.tex'), 'w');
4 fprintf(fileID, "%% appendix software documentation\n");
5 fprintf(fileID, "%% @author Tobias Wulf\n");
6 fprintf(fileID, ...
7     "%% Autogenerated LaTeX file. Generated by
8     exportPublishedToPdf.\n");
8 fprintf(fileID, ...
```

```
9    %% Software manual with TOC generated in the same script.\n\n10   n");\n11   fprintf(fileID, "%% Generated on %s.\n\n", datestr(datetime('now')));\n12   fprintf(fileID, "\\documentclass[class=article, crop=false]{\n        standalone}\\n");\n13   fprintf(fileID, "\\usepackage[subpreambles=true]{standalone}\\n\n");\n14   fprintf(fileID, "\\usepackage{import}\\n\\n");\n15   fprintf(fileID, "\\begin{document}\\n");\n16   fprintf(fileID, "\\clearpage\\n");\n17   fprintf(fileID, ...,\n        "\\\ntextbf{Matlab software appendix auto generated on %s.}\\n\n", ...,\n        datestr(datetime('now', 'Format', 'y-MM-d')));\n18\n19\n20  for i = 1:nToc\n21      level = toc(i);\n22      fName = toc(i,2);\n23      [~, fstr, ~] = fileparts(fName);\n24      lstr = lower(strrep(fstr, '_', '-'));\n25      tstr = strrep(fstr, '_', ' ');\n26\n27      fprintf(fileID, "\\%s\\%s", level, tstr);\n28      fprintf(fileID, "\\label{mcode:\\%s}\\n", lstr);\n29      fprintf(fileID, "\\subimport{./}{\\%s}\\n", fName);\n30      fprintf(fileID, "\\clearpage\\n");\n31  end\n32\n33  fprintf(fileID, "\\end{document}\\n");\n34  fclose(fileID);
```

### E.3.7 demoGPRModule

## demoGPRModule

This script demonstrates the use of gaussianProcessRegression module. The demonstartions 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.

## Contents

- Requirements
- See Also
- Start Script, Load Config and Read in Datasets
- Compute Means of Test Dataset as Comparing Root
- Create GPR Model for Demonstartion
- Prediction on Test Dataset
- Compute Losses and Errors on Test Dataset
- Plot Area and Expand Model Results

## 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

```
1 clc;
2 disp('Start GPR module demonstration ...');
3 clearvars;
4 close all;
5
6 disp('Load config ...');
7 load config.mat PathVariables GPROptions;
8
9 disp('Search for datasets ...');
10 TrainFiles = dir(fullfile(PathVariables.trainingDataPath, [
11     'Training*.mat')));
11 TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
12 assert(~isempty(TrainFiles), 'No training datasets found.');
13 assert(~isempty(TestFiles), 'No test datasets found.');
14
15 disp('Load first found datasets ...');
16 try
17     TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1) ...
18         .name));
19     TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1) ...
20         .name));
```

```

19
20 catch ME
21     rethrow(ME)
22 end
23
24 disp('Check dataset coordinates corresponds ...');
25 assert(all(TrainDS.Data.X == TestDS.Data.X, 'all'), 'Wrong X
26 grid.');
27 assert(all(TrainDS.Data.Y == TestDS.Data.Y, 'all'), 'Wrong Y
28 grid.');
29 assert(all(TrainDS.Data.Z == TestDS.Data.Z, 'all'), 'Wrong Z
30 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 fc0s0 - 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

```

1 offcos0 = mean2(TestDS.Data.Vcos);
2 offsin0 = mean2(TestDS.Data.Vsin);
3 fc0s0 = zeros(TestDS.Info.UseOptions.nAngles, 1);
4 fsin0 = zeros(TestDS.Info.UseOptions.nAngles, 1);
5 for n = 1:TestDS.Info.UseOptions.nAngles
6     fc0s0(n) = mean2(TestDS.Data.Vcos(:,:,n));
7     fsin0(n) = mean2(TestDS.Data.Vsin(:,:,n));
8 end
9 fc0s0 = fc0s0 - offcos0;
10 fsin0 = fsin0 - offsin0;
11 frad0 = sqrt(fc0s0.^2 + fsin0.^2);
12 fang0 = sinoids2angles(fsin0, fc0s0, frad0);
13 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 initialized 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.

```
1 disp('Create GPR modles ...');
2 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 fcose1 - 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

```
1 [fang1, frad1, fcose1, 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

```
1 [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.

```

1 % create general plot scalses and title
2 angles = TestDS.Data.angles';
3 ticks = Md11.Angles;
4 titleStr = "Kernel %s: $\sigma_f = %1.2f$, $\sigma_l = %1.2f
$," +
5 " $\sigma_n^2 = %1.2e$, $N = %d\n" +
6 "$%d \times %d$ Sensor-Array, Posistion: $(%1.1f,%1.1f
,-%1.1f)$ mm," +
7 " Magnet Tilt: $%2.1f^\circ$";
8 titleStr = sprintf(titleStr, ...
9     Md11.kernel, Md11.theta(1), Md11.theta(2), Md11.s2n, ...
10    Md11.N, Md11.D, Md11.D, ...
11    TestDS.Info.UseOptions.xPos, ...
12    TestDS.Info.UseOptions.yPos, ...
13    TestDS.Info.UseOptions.zPos, ...
14    TestDS.Info.UseOptions.tilt);
15
16 % create figure for model view
17 figure('Name', Md11.kernel, 'Units', 'normalize', '
OuterPosition', [0 0 1 1]);
18 t=tiledlayout(2,2);
19 title(t, titleStr, 'Interpreter', 'latex', 'FontSize', 24);
20
21 % plot covariance slice for first covariance sample
22 nexttile;
23 p1 = plot(Md11.Ky, 'Color', [0.8 0.8 0.8]);
24 hold on;
25 p2 = yline(mean2(Md11.Ky), 'Color', '#0072BD', 'LineWidth',
2.5);
26 p3 = plot(1:Md11.N, Md11.Ky(1,:), 'kx-.');

```

```

27 xlim([1, Md11.N]);
28 ylim([min(Md11.Ky, [], 'all'), max(Md11.Ky, [], 'all')]);
29 legend([p1(1), p2, p3], {'$i \neq 0$', '$\mu(K)$', '$i = 0$'});
30 xlabel('$n$ Samples');
31 ylabel('Autocorrelation Coeff.');
32 title('a) $K$-Matrix $i$-th Row');

33
34 % plot covariance matrix
35 nexttile([2, 1]);
36 colormap('jet');
37 imagesc(Md11.Ky);
38 axis square;
39 colorbar;
40 xlabel('$j$');
41 ylabel('i');
42 title(sprintf('b) $K$-Matrix $\times %d$ Samples', Md11.N,
   Md11.N))

43
44 % plot modle adjust as squared logarithmic loss for angles and
   radius
45 nexttile;
46 plot(angles, SLLA1, 'x-');
47 hold on;
48 plot(angles, SLLR1, 'x-');
49 xlim([0 360]);
50 legend({'$SLLA$' , '$SLLR$'});
51 xlabel('$\alpha$ in $^\circ$');
52 ylabel('$SLL$');
53 title(sprintf('c) $MSLLA = %1.2f$, $MSLLR = %1.2f$', mean(SLLA1
   ), mean(SLLR1)));

54
55 % create figure for rotation results of test dataset
56 figure('Name', 'Rotation and Errors', 'Units', 'normalize', ...
   'OuterPosition', [0 0 1 1]);
57 t = tiledlayout(2,2);
58 title(t, titleStr, 'Interpreter', 'latex', 'FontSize', 24);

```

```

60
61 % plot circle with gpr results and pure mean results
62 nexttile;
63 polarplot(fang0, frad0, 'LineWidth', 3.5);
64 hold on;
65 polarplot(fang1, frad1, 'LineWidth', 3.5);
66 polarscatter(Mdl1.Angles * pi / 180, 1.2 *ones(Mdl1.N, 1), 52,
67 [0.8 0.8 0.8], ...
68 'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);
69 legend({'Mean', 'GPR', 'Ref. $\alpha$'});
70 rticklabels(["", "0.5", "1"]);
71 title('a) Rotation along Z-Axis in $\circ$')

72 % plot predicted, ideal and none treated sinusoids
73 nexttile;
74 p1 = plot(angles, cosd(angles), 'k-.', 'LineWidth', 6.5);
75 hold on;
76 plot(angles, sind(angles), 'k-.', 'LineWidth', 6.5);
77 p2 = plot(angles, fcose0, 'Color', '#0072BD');
78 plot(angles, fsine0, 'Color', '#0072BD');
79 p3 = plot(angles, fcose1, 'Color', '#D95319');
80 plot(angles, fsine1, 'Color', '#D95319');
81 xlim([0 360]);
82 ylim([-1.1 1.1]);
83 xlabel('$\alpha$ in $\circ$');
84 legend([p1 p2 p3], {'Ideal', 'Mean', 'GPR'})
85 title('b) Sine and Cosine')

86 % plot absolut angle errors of gpr and mean results
87 nexttile;
88 plot(angles, AAED0);
89 hold on;
90 plot(angles, AAED1);
91 yline(mean(AAED1), 'k-.', 'LineWidth', 3.5)
92 [~, idx] = max(AAED1);
93 scatter(angles(idx), max(AAED1), 52, [0.8 0.8 0.8], ...

```

```

95      'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5)
96 ylim([0 4]);
97 xlim([0 360]);
98 xlabel('$\alpha$ in $^\circ$');
99 ylabel('$\epsilon_{abs}$ in $^\circ$');
100 legend({'Mean', 'GPR', '$\mu(\epsilon_{abs})$', '$\max\epsilon_{abs}$'});
101 tstr = 'c) $\mu(\epsilon_{abs}) = %1.2f$, $\max\epsilon_{abs} = %1.2f$';
102 tstr = sprintf(tstr, mean(AAED1), max(AAED1));
103 title(tstr);
104
105 % plot 95 percent confidence intervals for angles and radius
106 nexttile;
107 yyaxis left;
108 plot(angles, (ciang1-fang1) * 180/pi, '--', 'Color', '#0072BD');
109 ylabel('$CIA_{95\%} - \alpha$ in $^\circ$');
110 yyaxis right;
111 plot(angles, (cirad1-frad1), '--', 'Color', '#D95319');
112 ylabel('$CIR_{95\%} - r$');
113 xlim([0 360]);
114 xlabel('$\alpha$ in $^\circ$');
115 title('d) Centered 95% GPR Confidence Intervalls')

```

### **E.3.8 investigateKernelParameters**

## **investigateKernelParameters**

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

### **Contents**

- Requirements
- See Also
- Start Script, Load Config and Read in Datasets
- Create GPR Model for Investigation
- Sweep Title with Model Parameters
- Execute Parameter Sweep with Constant Noise
- Execute Parameter Sweep with Constant Variance
- Execute Parameter Sweep with Constant Lengthscale
- Sweep Kernel Parameters vs. Likelihood Criteria with Constant Noise
- Sweep Kernel Parameters vs. Likelihood Criteria with Constant Variance
- Sweep Kernel Parameters vs. Likelihood Criteria with Constant Lengthscale

### **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

```
1 clc;
2 disp('Start GPR module demonstration ...');
3 clearvars;
4 %close all;
5
6 disp('Load config ...');
7 load config.mat PathVariables GPROptions;
8
9 disp('Search for datasets ...');
10 TrainFiles = dir(fullfile(PathVariables.trainingDataPath, [
11     'Training*.mat')));
11 TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat'));
12 assert(~isempty(TrainFiles), 'No training datasets found.');
13 assert(~isempty(TestFiles), 'No test datasets found.');
14
15 disp('Load first found datasets ...');
16 try
17     TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1) ...
18         .name));
19     TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1) ...
20         .name));
```

```
19
20 catch ME
21     rethrow(ME)
22 end
23
24 disp('Check dataset coordinates corresponds ...');
25 assert(all(TrainDS.Data.X == TestDS.Data.X, 'all'), 'Wrong X
26 grid.');
27 assert(all(TrainDS.Data.Y == TestDS.Data.Y, 'all'), 'Wrong Y
grid.');
28 assert(all(TrainDS.Data.Z == TestDS.Data.Z, 'all'), 'Wrong Z
grid.');
```

## Create GPR Model for Investigation

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

## Sweep Title with Model Parameters

```
1 titleStr = "Kernel %s: $\sigma_f = %1.2f$, $\sigma_l = %1.2f
2 $, " + ...
3 " $\sigma_n^2 = %1.2e$, N = %d\n" +
4 "%d \times %d$ Sensor-Array, Posistion: (%1.1f,%1.1f
5 ,-%1.1f)$ mm," + ...
6 " Magnet Tilt: %2.1f^\circ";
7 titleStr = sprintf(titleStr, ...
8 Mdl1.kernel, Mdl1.theta(1), Mdl1.theta(2), Mdl1.s2n, ...
9 Mdl1.N, Mdl1.D, Mdl1.D, ...
10 TestDS.Info.UseOptions.xPos, ...
11 TestDS.Info.UseOptions.yPos, ...
12 TestDS.Info.UseOptions.zPos, ...
13 TestDS.Info.UseOptions.tilt);
```

### Execute Parameter Sweep with Constant Noise

```
1 nEval = 300;
2 disp('Sweep kernel parameters with constant noise ...');
3 sweepKernelWithConstNoise(Mdl1, nEval, titleStr, PathVariables)
```

### Execute Parameter Sweep with Constant Variance

```
1 nEval = 300;
2 disp('Sweep kernel parameters with constant variance ...');
3 sweepKernelWithConstVariance(Mdl1, nEval, titleStr,
    PathVariables)
```

### Execute Parameter Sweep with Constant Lengthscale

```
1 nEval = 300;
2 disp('Sweep kernel parameters with constant lengthscale ...');
3 sweepKernelWithConstLengthscale(Mdl1, nEval, titleStr,
    PathVariables)
```

### Sweep Kernel Parameters vs. Likelihood Criteria with Constant Noise

```
1 function sweepKernelWithConstNoise(Mdl, nEval, titleStr,
    PathVariables)
2
3     % create sweep parameters for sweeping theta to given mode
4     s2f = linspace(Mdl.s2fBounds(1) * 0.1, Mdl.s2fBounds(2) *
            10, nEval);
5     sl = linspace(Mdl.slBounds(1) * 0.1, Mdl.slBounds(2) * 10,
        nEval);
6     [sl, s2f] = meshgrid(sl, s2f);
7
```

```

8      % allocate memory for inner tuning criteria, combined
9          likelihoods for cosine
10         % and sine fit on trainings data
11         RLI = zeros(nEval, nEval);
12
13         % run sweep in multiprocess pool to gain speed
14         parfor i = 1:nEval
15             for j = 1:nEval
16                 % compute sweep with tuning criteria of inner GPR
17                 % optimization of
18                 % tuning GPR kernel parameters
19                 RLI(i,j) = computeTuneCriteria([s2f(i,j) sl(i,j)],
20                                         Mdl);
21             end
22         end
23
24         % plot results in countour plot
25         fig = figure('Name', 'Sweep Kernel Parameters with Constant
26                         Noise', ...
27                         'Units', 'normalize', 'OuterPosition', [0 0 1 1]);
28
29         % plot sweep with log axis
30         contourf(sl, s2f, RLI, linspace(min(RLI, []), 'all') + 1, 1,
31                   10), ...
32             'LineWidth', 1.5);
33         set(gca, 'YScale', 'log')
34         set(gca, 'XScale', 'log')
35         hold on;
36         grid on;
37
38         % plot bounds origin model parameters
39         p1 = yline(Mdl.s2fBounds(1), 'k-.', 'LineWidth', 2.5);
40         yline(Mdl.s2fBounds(2), 'k-.', 'LineWidth', 2.5);
41         yline(Mdl.theta(1), 'k', 'LineWidth', 2.5);
42         xline(Mdl.slBounds(1), 'k-.', 'LineWidth', 2.5);
43         xline(Mdl.slBounds(2), 'k-.', 'LineWidth', 2.5);

```

```

39 xline(Mdl.theta(2), 'k', 'LineWidth', 2.5);
40
41 % plot fmincon search area
42 p2 = patch( ...
43 [Mdl.s1Bounds(1), Mdl.s1Bounds(2), ...
44 Mdl.s1Bounds(2), Mdl.s1Bounds(1)], ...
45 [Mdl.s2fBounds(1) Mdl.s2fBounds(1), ...
46 Mdl.s2fBounds(2) Mdl.s2fBounds(2)], ...
47 [0.8 0.8 0.8], 'FaceAlpha', 0.7);
48
49 % plot argmin fmincon result
50 p3 = scatter(Mdl.theta(2), Mdl.theta(1), 60, [0.8 0.8 0.8], ...
51 ...
52 'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);
53
54 % labels, titles, legends
55 xlabel('$\sigma_1$')
56 ylabel('$\sigma_f^2$')
57 title(titleStr);
58 stStr = "$\sigma_f^2, \sigma_1 | \sigma_n^2 = " + ...
59 "\arg\min\tilde{R} \mathcal{L} " + ...
60 "(\sigma_f^2, \sigma_1 | \sigma_n^2) \$ f. \sigma_n^2 = "
61 const.$;
62 subtitle(stStr);
63 legend([p1, p2, p3], ...
64 {"Parameter Bounds", "Search Area", ...
65 sprintf("fmincon $\tilde{R} \mathcal{L} (%1.2f,%1.2f ...
66 | %1.2e)=%1.2f", ...
67 Mdl.theta, Mdl.s2n, -(Mdl.LMLcos + Mdl.LMLSin))},
68 ...
69 'Location', 'South')
70
71 cb = colorbar;
72 cb.TickLabelInterpreter = 'latex';
73 cb.Label.Interpreter = 'latex';
74 cb.Label.FontSize = 24;

```

```

71 cbStr = "$\tilde{R}_\mathcal{LI}(\sigma_f^2, \sigma_l | \
72   \sigma_n^2)$";
73 cb.Label.String = cbStr;
74
75 % save and close
76 % fPath = fullfile(PathVariables.saveImagesPath, ' \
77 % Sweep_Kernel_Const_Noise');
78 % print(fig, fPath, '-dsvg');
79 % close(fig);
80 end

```

## Sweep Kernel Parameters vs. Likelihood Criteria with Constant Variance

```

1 function sweepKernelWithConstVariance(Mdl, nEval, titleStr,
2                                         PathVariables)
3
4     % keep s2n origin to plot later
5     s2nOrigin = Mdl.s2n;
6
7     % create sweep parameters for sweeping lengthscale and
8     % noise to given modle
9     s2n = linspace(Mdl.s2nBounds(1) * 0.1, Mdl.s2nBounds(2) *
10                  10, nEval);
11    sl = linspace(Mdl.slBounds(1) * 0.1, Mdl.slBounds(2) * 10,
12                  nEval);
13    s2f = Mdl.theta(1);
14
15    % allocate memory for inner tuning criteria, combined
16    % likelihoods for cosine
17    % and sine fit on trainings data
18    RLI = zeros(nEval, nEval);
19
20    % run sweep in multiprocess pool to gain speed
21    for i = 1:nEval

```

```
17 % assign struct values to compute corresponding lenght
18 % scale row wise
19 % due to parfor struct issue
20 Mdl.s2n = s2n(i);
21 parfor j = 1:nEval
22     % compute sweep with tuning criteria of inner GPR
23     % optimization of
24     % tuning GPR kernel parameters, variance is set to
25     % 1
26     RLI(i,j) = computeTuneCriteria([s2f sl(j)], Mdl);
27 end
28
29
30 % generate grid on vectors to plot results
31 [sl, s2n] = meshgrid(sl, s2n);
32
33
34 % plot results in countour plot
35 fig = figure('Name', 'Sweep Kernel Parameters with Constant
36 % Variance', ...
37     'Units', 'normalize', 'OuterPosition', [0 0 1 1]);
38
39
40 % plot sweep with log axis
41 contourf(sl, s2n, RLI, linspace(min(RLI, []), 'all') + 1, 1,
42             10), ...
43     'LineWidth', 1.5);
44 set(gca, 'YScale', 'log')
45 set(gca, 'XScale', 'log')
46 hold on;
47 grid on;
48
49
50 % plot bounds origin model parameters
51 p1 = yline(Mdl.s2nBounds(1), 'k-.', 'LineWidth', 2.5);
52 yline(Mdl.s2nBounds(2), 'k-.', 'LineWidth', 2.5);
53 yline(s2nOrigin, 'k', 'LineWidth', 2.5);
54 xline(Mdl.slBounds(1), 'k-.', 'LineWidth', 2.5);
55 xline(Mdl.slBounds(2), 'k-.', 'LineWidth', 2.5);
```

```

48 xline(Mdl.theta(2), 'k', 'LineWidth', 2.5);
49
50 % plot fmincon search area
51 p2 = patch( ...
52 [Mdl.s1Bounds(1), Mdl.s1Bounds(2), ...
53 Mdl.s1Bounds(2), Mdl.s1Bounds(1)], ...
54 [Mdl.s2nBounds(1) Mdl.s2nBounds(1), ...
55 Mdl.s2nBounds(2) Mdl.s2nBounds(2)], ...
56 [0.8 0.8 0.8], 'FaceAlpha', 0.7);
57
58 % plot argmin fmincon result
59 p3 = scatter(Mdl.theta(2), s2nOrigin, 60, [0.8 0.8 0.8],
60 ...
61 'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);
62
63 % labels, titles, legends
64 xlabel('$\sigma_1$')
65 ylabel('$\sigma_n^2$')
66 title(titleStr);
67 stStr = "$\sigma_f^2, \sigma_1 | \sigma_n^2 = " + ...
68 "\arg\min\tilde{R} \mathcal{L}" + ...
69 "(\sigma_f^2, \sigma_1 | \sigma_n^2) \$ f. \$\sigma_f^2 = "
70 const.$;
71 subtitle(stStr);
72 legend([p1, p2, p3], ...
73 {"Parameter Bounds", "Search Area", ...
74 sprintf("fmincon $\tilde{R} \mathcal{L} (%1.2f, %1.2f ...
75 | %1.2e) = %1.2f", ...
76 Mdl.theta, s2nOrigin, -(Mdl.LMLcos + Mdl.LMLSin))), ...
77 ...
78 'Location', 'South')
79
80 cb = colorbar;
81 cb.TickLabelInterpreter = 'latex';
82 cb.Label.Interpreter = 'latex';
83 cb.Label.FontSize = 24;

```

```

80 cbStr = "$\tilde{R}_\mathcal{LI}(\sigma_f^2, \sigma_l | \
81     \sigma_n^2)$";
82 cb.Label.String = cbStr;
83
84 % save and close
85 % fPath = fullfile(PathVariables.saveImagesPath, ' \
86 % Sweep_Kernel_Const_Var');
86 % print(fig, fPath, '-dsvg');
87 % close(fig);
87 end

```

## Sweep Kernel Parameters vs. Likelihood Criteria with Constant Lengthscale

```

1 function sweepKernelWithConstLengthscale(Mdl, nEval, titleStr,
2                                         PathVariables)
3
4     % keep s2n origin to plot later
5     s2nOrigin = Mdl.s2n;
6
7     % create sweep parameters for sweeping lengthscale and
8     % noise to given mode
9     s2n = linspace(Mdl.s2nBounds(1) * 0.1, Mdl.s2nBounds(2) *
10                  10, nEval);
11    s2f = linspace(Mdl.s2fBounds(1) * 0.1, Mdl.s2fBounds(2) *
12                  10, nEval);
13    sl = Mdl.theta(2);
14
15    % allocate memory for inner tuning criteria, combined
16    % likelihoods for cosine
17    % and sine fit on trainings data
18    RLI = zeros(nEval, nEval);
19
20    % run sweep in multiprocess pool to gain speed
21    for i = 1:nEval

```

```

17      % assign struct values to compute corresponding lenght
18      % scale row wise
19      % due to parfor struct issue
20      Mdl.s2n = s2n(i);
21      parfor j = 1:nEval
22          % compute sweep with tuning criteria of inner GPR
23          % optimization of
24          % tuning GPR kernel parameters, variance is set to
25          % 1
26          RLI(i,j) = computeTuneCriteria([s2f(j) sl], Mdl);
27      end
28
29
30      % generate grid on vectors to plot results
31      [s2f, s2n] = meshgrid(s2f, s2n);
32
33
34      % plot results in countour plot
35      fig = figure('Name', 'Sweep Kernel Parameters with Constant
36                  Lengthscale',...
37                  'Units', 'normalize', 'OuterPosition', [0 0 1 1]);
38
39
40      % plot sweep with log axis
41      contourf(s2f, s2n, RLI, linspace(min(RLI, []), 'all') + 1,
42                 1, 10), ...
43                 'LineWidth', 1.5);
44      set(gca, 'YScale', 'log')
45      set(gca, 'XScale', 'log')
46      hold on;
47      grid on;
48
49
50      % plot bounds origin model parameters
51      p1 = yline(Mdl.s2nBounds(1), 'k-.', 'LineWidth', 2.5);
52      yline(Mdl.s2nBounds(2), 'k-.', 'LineWidth', 2.5);
53      yline(s2nOrigin, 'k', 'LineWidth', 2.5);
54      xline(Mdl.s2fBounds(1), 'k-.', 'LineWidth', 2.5);
55      xline(Mdl.s2fBounds(2), 'k-.', 'LineWidth', 2.5);

```

```

48 xline(Mdl.theta(1), 'k', 'LineWidth', 2.5);
49
50 % plot fmincon search area
51 p2 = patch( ...
52 [Mdl.s2fBounds(1), Mdl.s2fBounds(2), ...
53 Mdl.s2fBounds(2), Mdl.s2fBounds(1)], ...
54 [Mdl.s2nBounds(1) Mdl.s2nBounds(1), ...
55 Mdl.s2nBounds(2) Mdl.s2nBounds(2)], ...
56 [0.8 0.8 0.8], 'FaceAlpha', 0.7);
57
58 % plot argmin fmincon result
59 p3 = scatter(Mdl.theta(1), s2nOrigin, 60, [0.8 0.8 0.8],
60 ...
61 'filled', 'MarkerEdgeColor', 'k', 'LineWidth', 1.5);
62
63 % labels, titles, legends
64 xlabel('$\sigma_f^2$')
65 ylabel('$\sigma_n^2$')
66 title(titleStr);
67 stStr = "$\sigma_f^2, \sigma_1 | \sigma_n^2 = " + ...
68 "\arg\min\tilde{R} \mathcal{L} " + ...
69 "(\sigma_f^2, \sigma_1 | \sigma_n^2) \$ f. \sigma_1 = const
70 .\$";
71 subtitle(stStr);
72 legend([p1, p2, p3], ...
73 {"Parameter Bounds", "Search Area", ...
74 sprintf("fmincon $\tilde{R} \mathcal{L} (%1.2f,%1.2f
75 |%1.2e)=%1.2f", ...
76 Mdl.theta, s2nOrigin, -(Mdl.LMLcos + Mdl.LMLSin))), ...
77 ...
78 'Location', 'South')
79
cb = colorbar;
cb.TickLabelInterpreter = 'latex';
cb.Label.Interpreter = 'latex';
cb.Label.FontSize = 24;

```

```
80 cbStr = "$\tilde{R}_\mathcal{L}(\sigma_f^2,\sigma_l|\sigma_n^2)$";
81 cb.Label.String = cbStr;
82
83 % save and close
84 % fPath = fullfile(PathVariables.saveImagesPath, 'Sweep_Kernel_Const_Len');
85 % print(fig, fPath, '-dsvg');
86 % close(fig);
87 end
```

### E.3.9 compareGPRKernels

## 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.

## Contents

- Requirements
- See Also
- Load Datasets
- Initiate Kernels with different Kernelparameters
- Plot Area

## 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

```
1 clear all;
2 close all;
3 disp('Load config ...');
4 load config.mat PathVariables GPROptions;
5 TrainFiles = dir(fullfile(PathVariables.trainingDataPath, [
6     'Training*.mat'));
7 TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat
8     '));
9 assert(~isempty(TrainFiles), 'No training datasets found.');
10 assert(~isempty(TestFiles), 'No test datasets found.');
11
12 disp('Load first found datasets ...');
13 try
14     TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1)
15         .name));
16     TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1) .
17         name));
18
19 catch ME
20     rethrow(ME)
21 end
```

## Initiate Kernels with different Kernelparameters

```
1 ma11 = initGPR(TrainDS, GPROptions);
2
3 GPROptions.theta = [1 2];
4 ma12 = initGPR(TrainDS, GPROptions);
5 GPROptions.theta = [1 0.5];
6 ma105 = initGPR(TrainDS, GPROptions);
7 GPROptions.theta = [1 4];
8 ma14 = initGPR(TrainDS, GPROptions);
9
```

```
10 GPROptions.theta = [2 1];
11 mc21 = initGPR(TrainDS, GPROptions);
12 GPROptions.theta = [0.5 1];
13 mc051 = initGPR(TrainDS, GPROptions);
14 GPROptions.theta = [4 1];
15 mc41 = initGPR(TrainDS, GPROptions);
16
17 GPROptions.kernel = 'QFCAPX';
18 GPROptions.theta = [1 1];
19 mb11 = initGPR(TrainDS, GPROptions);
20
21 GPROptions.theta = [1 2];
22 mb12 = initGPR(TrainDS, GPROptions);
23 GPROptions.theta = [1 0.5];
24 mb105 = initGPR(TrainDS, GPROptions);
25 GPROptions.theta = [1 4];
26 mb14 = initGPR(TrainDS, GPROptions);
27
28 GPROptions.theta = [2 1];
29 md21 = initGPR(TrainDS, GPROptions);
30 GPROptions.theta = [0.5 1];
31 md051 = initGPR(TrainDS, GPROptions);
32 GPROptions.theta = [4 1];
33 md41 = initGPR(TrainDS, GPROptions);
```

## Plot Area

```
1 % plot covariance slice for first covariance sample
2 figure;
3 tiledlayout(2,2);
4
5 nexttile;
6
7 hold on;
```

```

8
9 p1 = plot(1:ma11.N, ma11.Ky(1,:), 'k');
10 p2 = plot(1:ma12.N, ma12.Ky(1,:), 'b-.');
11 p3 = plot(1:ma105.N, ma105.Ky(1,:), 'r-.');
12 p4 = plot(1:ma14.N, ma14.Ky(1,:), 'g-.');
13 xlim([1, ma11.N]);
14 %ylim([min(Mdl1.Ky, [], 'all'), max(Mdl1.Ky, [], 'all')]);
15 legend([p1, p2, p3, p4], ...
16     {'$\theta = (1,1)$', ...
17     '$\theta = (1,2)$', ...
18     '$\theta = (1,0.5)$', ...
19     '$\theta = (1,4)$'}, 'Location', 'north');
20 xlabel('$j$-tes Sample');
21 title('a) $k(X_1, X_j|\theta)$ f. $d_F^2$, $\theta_1 = konst.$');
22
23 nexttile;
24 hold on;
25
26 p1 = plot(1:mb11.N, mb11.Ky(1,:), 'k');
27 p2 = plot(1:mb12.N, mb12.Ky(1,:), 'b-.');
28 p3 = plot(1:mb105.N, mb105.Ky(1,:), 'r-.');
29 p4 = plot(1:mb14.N, mb14.Ky(1,:), 'g-.');
30 xlim([1, mb11.N]);
31 %ylim([min(Mdl1.Ky, [], 'all'), max(Mdl1.Ky, [], 'all')]);
32 legend([p1, p2, p3, p4], ...
33     {'$\theta = (1,1)$', ...
34     '$\theta = (1,2)$', ...
35     '$\theta = (1,0.5)$', ...
36     '$\theta = (1,4)$'}, 'Location', 'north');
37 xlabel('$j$-tes Sample');
38 title('b) $k(X_1, X_j|\theta)$ f. $d_E^2$, $\theta_1 = konst.$');
39
40 nexttile;
41

```

```

42
43 hold on;
44
45 p1 = plot(1:ma11.N, ma11.Ky(1,:), 'k');
46 p2 = plot(1:mc21.N, mc21.Ky(1,:), 'b-.');
47 p3 = plot(1:mc051.N, mc051.Ky(1,:), 'r-.');
48 p4 = plot(1:mc41.N, mc41.Ky(1,:), 'g-.');
49 xlim([1, ma11.N]);
50 %ylim([min(Mdl1.Ky, [], 'all'), max(Mdl1.Ky, [], 'all')]);
51 legend([p1, p2, p3, p4], ...
52     {'$\theta = (1,1)$', ...
53     '$\theta = (2,1)$', ...
54     '$\theta = (0.5,1)$', ...
55     '$\theta = (4,1)$'}, 'Location', 'north');
56 xlabel('$j$-tes Sample');
57 title('c) $k(x_1, x_j|\theta)$ f. $d_F^2$, $\theta_2 = \text{konst.}$');
58
59 nexttile;
60
61 hold on;
62
63 p1 = plot(1:mb11.N, mb11.Ky(1,:), 'k');
64 p2 = plot(1:md21.N, md21.Ky(1,:), 'b-.');
65 p3 = plot(1:md051.N, md051.Ky(1,:), 'r-.');
66 p4 = plot(1:md41.N, md41.Ky(1,:), 'g-.');
67 xlim([1, mb11.N]);
68 %ylim([min(Mdl1.Ky, [], 'all'), max(Mdl1.Ky, [], 'all')]);
69 legend([p1, p2, p3, p4], ...
70     {'$\theta = (1,1)$', ...
71     '$\theta = (2,1)$', ...
72     '$\theta = (0.5,1)$', ...
73     '$\theta = (4,1)$'}, 'Location', 'north');
74 xlabel('$j$-tes Sample');
75 title('c) $k(x_1, x_j|\theta)$ f. $d_E^2$, $\theta_2 = \text{konst.}$');

```

```
76 % plot covariance matrix
77 figure;
78 tiledlayout(1,2);
79 colormap('jet');
80
81 nexttile;
82
83
84
85 imagesc(ma11.Ky);
86 axis square;
87 xlabel('$j$');
88 ylabel('$i$');
89 title('a) $K(X, X|\theta)$ f. $d_F^2$, $\theta = (1,1)$')
90
91 nexttile;
92
93 % colormap('jet');
94 imagesc(mb11.Ky);
95 axis square;
96 xlabel('$j$');
97 ylabel('$i$');
98 title('a) $K(X, X|\theta)$ f. $d_E^2$, $\theta = (1,1)$')
99
100 c = colorbar;
101 c.TickLabelInterpreter = 'latex';
```

## E.4 Source Code

### 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 classses 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.

### Contents

- sensorArraySimulation
- gaussianProcessRegression
- util

### 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.

## **gaussianProcessRegression**

Source code establish a struct based regression model which has the ability to work with datasets generated by sensorArraySimulation module. Generated regression models and datasets can be passed to prediction functions and loss computation as arguments. So it is possible to build up multi model evaluation in scripts at a time.

## **util**

Util function and classes to provide reuse for often upcomings tasks and functionality besides project kernel and module source code. Located are plot functions and file operations.

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

#### **E.4.1 sensorArraySimulation**

### **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 mating 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 positions 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.

## Contents

- See Also
- [simulateDipoleSquareSensorArray](#)
- [computeDipoleHField](#)
- [computeDipoleH0Norm](#)
- [generateSensorArraySquareGrid](#)
- [generateDipoleRotationMoments](#)
- [rotate3DVector](#)

## 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 dipol 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.

#### E.4.1.1 rotate3DVector

### 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.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 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

```
1 % rotate a vector along z-axes by 45
2 vector = [1; 0; 0]
3 rotated = rotate3DVector(vector, 0, 0, 45)
```

```
1 % rotate a vector along z-axes by 35 with a tilt in x-axes by 1
2 vector = [1; 0; 0]
3 rotated = rotate3DVector(vector, 1, 0, 35)
```

```
1 % rotate a vector along z-axes by 35 with a tilt in x-axes by 1
    and a
2 % tilt in y-axes by 5
3 vector = [1; 0; 0]
4 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 Schütte. Copyright Thorben Schütte 2016.

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

```
7      betaY (1,1) double {mustBeReal}
8      gammaZ (1,1) double {mustBeReal}
9  end
10
11 % rotate vector or vector field as 3 x N matrix
12 % counter-clockwise by given
13 % angles along axes, calculate rotation matrices for each
14 % axes and
15 % multiplificate with input vector
16 rotated = rotz(gammaZ) * roty(betaY) * rotx(alphaX) *
17     vector(:, 1:end);
18 end
```

#### E.4.1.2 generateDipoleRotationMoments

### 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

$$\vec{m}_0 = |m_0| \cdot [-1, , 0, ]^T$$

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

$$\vec{M} = [\vec{m}_i \cdots \vec{m}_N]$$

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

$$\vec{\theta} = [\theta_i \cdots \theta_N]$$

for

$$i = 1 \cdots N$$

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.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 M = generateDipoleRotationMoments(m0, nTheta)
2 [M, thetas] = generateDipoleRotationMoments(m0, nTheta)
3 [M, thetas] = generateDipoleRotationMoments(m0, nTheta, phi)
4 [M, thetas] = generateDipoleRotationMoments(m0, nTheta, phi,
      resolution)
5 [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 representation of down sampled angles in the full scale rotation vector.

## Examples

```
1 % choose a huge moment amplitude to withdraw numeric errors in  
  later H-field  
2 % strength calculations  
3 m0 = 1e6;
```

```
1 % get a full scale (FS) rotation of with 0.5 resolution and no  
  tilt  
2 [MFS, thetaFS] = generateDipoleRotationMoments(m0, 0, 0, 0.5);
```

```
1 % get down sampled (DS) rotation with equal distanced angles of
  the same full
2 % scale and refered index to the full scale. 8 angles.
3 [MDS, thetaDS, iFS] = generateDipolRotationMoments(m0, 8, 0,
  0.5);
```

```
1 % check distribution to full scale must be true if distribution
  is correct
2 all(MFS(iFS) == MDS)
3 all(thetaFS(iFS) == thetaDS)
```

```
1 % now shift the sample pick by 22 samples (11 with resolution
  of 0.5)
2 [MDSS, thetaDSS] = generateDipolRotationMoments(m0, 8, 0, 0.5,
  22);
```

```
1 % check with index shift by 22 in iFS index
2 all(MFS(iFS + 22) == MDSS)
3 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.

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

**phi** scalar angule 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.

```

1 function [M, theta, index] = generateDipoleRotationMoments(m0,
2 nTheta, ...
3 phi, resolution, phaseIndex)
4 arguments
5 % validate amplitude of magnetic moment as real scalar
6 % value
7 m0 (1,1) double {mustBeReal}
8 % validate number of used angulars as positive integer,
9 % for 0 return all
10 nTheta (1,1) double {mustBeNonnegative, mustBeInteger}
11 % validate tilt angle as real value with default 0
12 phi (1,1) double {mustBeReal} = 0
13 % validate angle resolution as real positive value
14 resolution (1,1) double {mustBePositive} = 1
15 % validate downsample phase as positive integer with
16 % default 0, no shift
17 phaseIndex (1,1) double {mustBeNonnegative,
18 % mustBeInteger} = 0
19 end
20
21 % scale full rotation angle vector with given resolution
22 % from 0 to 360
23 % so run to 360-resolution because 0 == 360, its a circle
24 fullScale = 0:resolution:(360 - resolution);
25
26 % if nThetas is greater than 0 downsample to nTheta else
27 % use full scale
28 if nTheta
29     % get equal distribute distance of samples in thetas
30     for nThetas
31         sampleDistance = length(downsampling(fullScale, nTheta));
32
33     % downsample with equal sample distance and passed
34     % sample phase to shift

```

```
26      % first sample in downsample vector from 1 to
27      % phaseIndex
28      theta = downsample(fullScale, sampleDistance,
29                          phaseIndex);
30
31      % find index members of down sampled angles in full
32      % scale vector
33      members = ismember(fullScale, theta);
34      index = find(members);
35
36  else
37      % 0 is given for number of theta so it returns the full
38      % scale rotation
39      % no index relations if full scale is returned
40      nTheta = length(fullScale);
41      theta = fullScale;
42      index = [];
43
44  end
45
46      % create start moment with given magnetic moment amplitude
47      % basic moment to
48      % produce rotate moments
49      m0 = m0 * [-1; 0; 0];
50
51      % allocate memory for the moments Matrix of rotated basic
52      % moments by i-th
53      % theta and fixed tilt of phi and rotate of theta angulars
54      M = zeros(3, nTheta);
55      for i = 1:nTheta
56          M(:,i) = rotate3DVector(m0, 0, phi, theta(i));
57      end
58  end
```

#### E.4.1.3 generateSensorArraySquareGrid

### 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 other 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 scaled from center of the square. So for the upper left corner position is described by

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

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  $X_0$  with

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

same wise for y dimension but transposed  $X_0 = Y_0^T$  with

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

and z dimension  $Z_0 = \text{const.}$  with  $z_{i,j} = 0$

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

$$\vec{p} = (x_p, y_p, z_p)^T$$

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

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

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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

```
1 % generate a grid of 8 x 8 sensors with no shift in x or y  
2   direction  
3 and a static position of 4mm under the center in z dimension  
4   with a  
5 z offset of 2mm so (2 + 2)mm  
6 N = 8;  
7 p = [0, 0, 2]  
8 r = 2;  
9 [X, Y, Z] = generateSensorArrayGrid(N, a, p, r);
```

```
1 % same layer but left shift by 2mm and down shift in y by 1mm  
2 p = [-2, 1, 2]  
3 r = 2;  
4 [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  $N \times N$ .

**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 coordinate 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  $N \times N$  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.

```
1 function [X, Y, Z] = generateSensorArraySquareGrid(N, a, p, r)
2     arguments
3         % validate N as positive integer
4         N (1,1) double {mustBePositive, mustBeInteger}
5         % validate array edge length as positive scalar
6         a (1,1) double {mustBeReal, mustBePositive}
7         % validate p as column vector of real scalars
8         p (3,1) double {mustBeReal, mustBeVector}
9         % validate r as real scalar
10        r (1,1) double {mustBeReal}
11    end
12
13    % half edge length for square corners
14    aHalf = a / 2;
15
```

```
16 % distance in x and y direction of each coordinate to next
  point
17 d = a / (N - 1);
18
19 % grid vector for x and y coordinates z is constant layer
  with shifts
20 x = (-aHalf:d:aHalf) + p(1);
21 y = (aHalf:-d:-aHalf) + p(2);
22 z = -(p(3) + r);
23
24 % scale grid in x, y dimension with constant z dimension
25 [X, Y, Z] = meshgrid(x, y, z);
26 end
```

#### E.4.1.4 computeDipoleH0Norm

### 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.

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

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

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments

- Requirements
- See Also

## Syntax

```
1 H0norm = computeDipoleH0Norm(Hmag, m0, r0)
```

## Description

**H0norm = computeDipoleH0Norm(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

```
1 % distance where the magnetic field strength is the value of
  % wished
2 % magnitude, in mm
3 r0 = [0; 0; -5]
4 % field strength to imprint in norm factor in kA/m
5 Hmag = 200
6 % magnetic moment magnitude which is used generate rotation
  % moments
7 m0 = [-1e6; 0; 0]
8 % compute norm factor for dipole rest position
9 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  $z_0 + r_{sp}$ .

## 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.

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

```
7     end
8
9     % calculate the magnitude of all positions
10    r0abs = sqrt(sum(r0.^2, 1));
11
12    % calculate the the unit vector of all positions
13    r0hat = r0 ./ r0abs;
14
15    % calculate field strength and magnitude at position
16    H0 = (3 * r0hat .* (m0' * r0hat) - m0) ./ (4 * pi *r0abs
17        .^3);
18    H0abs = sqrt(sum(H0.^2, 1));
19
20    % compute the norm factor like described in the equations
21    H0norm = Hmag / H0abs;
22
23 end
```

#### E.4.1.5 computeDipoleHField

### 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

$$\vec{m} = (m_x, m_y, m_z)^T$$

so that the magnetic moment corresponds to a position vector

$$\vec{r} = (x, y, z)^T$$

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

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

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

$$\vec{H}(\vec{r}) = (H_x, H_y, H_z)^T$$

The originally equation of the magnetic dipole is known as

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

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

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.

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

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 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

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

```

1 % compute a 3D grid of positions
2 x = linspace(-10, 10, 40);
3 y = linspace(10, -10, 40);
4 z = linspace(10, -10, 40);
5 [X, Y, Z] = meshgrid(x, y, z);

```

```

1 % allocate memory for field components in x,y,z
2 Hx = zeros(40, 40, 40);
3 Hy = zeros(40, 40, 40);
4 Hz = zeros(40, 40, 40);

```

```

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

```

```

1 % calculate magnitude in each point for better view the results
2 Habs = sqrt(Hx.^2+Hy.^2+Hz.^2);

```

```

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

```

```

1 % downsample and norm
2 Xds = X(idx,idx,idx);
3 Yds = Y(idx,idx,idx);
4 Zds = Z(idx,idx,idx);
5 Hxds = Hx(idx,idx,idx) ./ Habs(idx,idx,idx);

```

```
6 Hyds = Hy(idx, idx, idx) ./ Habs(idx, idx, idx);  
7 Hzds = Hz(idx, idx, idx) ./ Habs(idx, idx, idx);
```

```
1 % show results  
2 quiver3(Xds, Yds, Zds, Hxds, Hyds, Hzds);  
3 axis equal;
```

## Input Argurments

**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 Argurments

**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.

```
1 function [H] = computeDipoleHField(x, y, z, m, H0norm)
2     arguments
3         % validate position, can be any size but must be same
4         % size of
5         x (:,:,:,:) double {mustBeReal}
6         y (:,:,:,:) double {mustBeReal, mustBeEqualSize(x, y)}
7         z (:,:,:,:) double {mustBeNumeric, mustBeReal,
8             mustBeEqualSize(y, z)}
9         % validate magnetic moment as 3 x 1 vector
10        m (3,1) double {mustBeReal, mustBeVector}
11        % validate norm factor as scalar
12        H0norm (1,1) double {mustBeReal}
13    end
14
15    % unify positions to column vector or matrix of column
16    % vectors if positions
17    % are not passed as column vectors or scalar, resulting
18    % size of position R
19    % is 3 x length(X), a indication if is column vector is not
20    % needed because
```

```
16 % x(:) is returning all content as column vector. Transpose
17 % to match shape.
18 r = [x(:), y(:), z(:)]';
19
20 % calculate the magnitude of all positions
21 rabs = sqrt(sum(r.^2, 1));
22
23 % calculate the the unit vector of all positions
24 rhat = r ./ rabs;
25
26 % calculate H-field of current magnetic moment for all
27 % passed positions
28 % calculate constants in eqution once in the first bracket
29 % term, all vector
30 % products in the second term and finially divide by
31 % related magnitude ^3
32 H = (H0norm / 4 / pi) * (3 * rhat .* (m' * rhat) - m) ./
33 rabs.^3;
34 end
35
36 % Custom validation function
37 function mustBeEqualSize(a,b)
38 % Test for equal size
39 if ~isequal(size(a),size(b))
40     eid = 'Size:notEqual';
41     msg = 'X Y Z positions must be the same size and
42         orientation.';
43     throwAsCaller(MException(eid,msg))
44 end
45 end
```

#### E.4.1.6 simulateDipoleSquareSensorArray

## 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.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

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

## Description

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

## Examples

```
1 % load config from mat-file
2 load('config.mat', 'GeneralOptions', 'PathVariables', '
    SensorArrayOptions',
3     'DipoleOptions', 'TrainingOptions', 'TestOptions');
```

```
1 % load characterization dataset
2 TDK = load(PathVariables.tdkDatasetPath);
```

```
1 % generate training dataset(s)
2 simulateDipoleSquareSensorArray(GeneralOptions, PathVariables,
    ...
3     SensorArrayOptions, DipoleOptions, TrainingOptions, TDK)
```

```
1 % generate test dataset(s)
2 simulateDipoleSquareSensorArray(GeneralOptions, PathVariables,
    ...
3     SensorArrayOptions, DipoleOptions, TestOptions, TDK)
```

## Input Arguments

**GeneralOptions** struct of general options generate 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 characteriazation data. Therfore load characterization dataset as shown in examples into a struct.

## Output Argurments

**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
- genertateDipoleRotationMoments
- generateSensorArraySquareGrid
- rotate3DVector

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

```
1 function simulateDipoleSquareSensorArray(GeneralOptions,
2     PathVariables, ...
3     SensorArrayOptions, DipoleOptions, UseOptions, CharData)
4 arguments
5     % validate inputs as struct, structs generated in
6         config.mat
7     GeneralOptions struct {mustBeA(GeneralOptions, 'struct')
8         )}
9     PathVariables struct {mustBeA(PathVariables, 'struct')}
10    SensorArrayOptions struct {mustBeA(SensorArrayOptions,
11        'struct')}
12    DipoleOptions struct {mustBeA(DipoleOptions, 'struct')}
13    UseOptions struct {mustBeA(UseOptions, 'struct')}
14    CharData struct {mustBeA(CharData, 'struct')}
15 end
16
17 % try to load relavant values in local variable space for
18 % better
19 % handling and short names second check if struct fields
20 % are reachable
21 try
22     % general options needed to create filenames etc.
23     dfStr = GeneralOptions.dateFormat;
24
25     % number of sensors at edge of square array, dimension
26         N x N
27     N = SensorArrayOptions.dimension;
28     % sensor array edge length, square edge a
29     a = SensorArrayOptions.edge;
30     % sensor array supply voltage used to generate bridge
31         outputs from
32     % characterization data in combination with bridge
33         offset voltage
34     % characterization data should be in mv/V so check norm
35         factor
```

```
26     Vcc = SensorArrayOptions.Vcc;
27     Voff = SensorArrayOptions.Voff;
28     Vnorm = SensorArrayOptions.Vnorm;
29     switch CharData.Info.Units.SensorOutputVoltage
30         case 'mV/V'
31             if Vnorm ~= 1e3
32                 error('Wrong norming mV/V: %e', Vnorm);
33             end
34         otherwise
35             error('Unknown norm voltage: %s', ...
36                   CharData.Info.Units.SensorOutputVoltage)
37     end
38
39     % sphere radius for dipole approximation of spherical
40     % magnet
41     rsp = DipoleOptions.sphereRadius;
42     % H-field magnitude to imprint in certain distance from
43     % magnet
44     % surface which sphere radius rsp plus distance z0
45     H0mag = DipoleOptions.H0mag;
46     % distance from magnet surface where to imprint the
47     % H0mag
48     z0 = DipoleOptions.z0;
49     % magnetic dipole moment magnitude which define origin
50     % moment of the
51     % magnet in rest position
52     M0mag = DipoleOptions.M0mag;
53
54     % dataset type or use case in which later it is use in
55     % application
56     useCase = UseOptions.useCase;
57     % destination path and filename to save generated data
58     % sets with
59     % timestamps in filename, place timestamps with sprintf
60     switch useCase
61         case 'Training'
```

```
56         fPath = PathVariables.trainingDataPath;
57         fNameFmt = 'Training_%s.mat';
58     case 'Test'
59         fPath = PathVariables.testDataPath;
60         fNameFmt = 'Test_%s.mat';
61     otherwise
62         error('Unknown use case: %s', UseOptions.
63             useCase);
64
65     % x, y and z positions in which pairing the datasets
66     % are generated
67     % position vectors are run through in all combinations
68     % with tilt
69     % and number of angles
70     xPos = UseOptions.xPos;
71     yPos = UseOptions.yPos;
72     zPos = UseOptions.zPos;
73     tilt = UseOptions.tilt;
74     nAngles = UseOptions.nAngles;
75     % constants for generated use case, angle resolution
76     % for generated
77     % rotation angles, phase index for a phase shift in
78     % generation of
79     % rotation angles
80     angleRes = UseOptions.angleRes;
81     phaseIndex = UseOptions.phaseIndex;
82     % which characterization reference should be load from
83     % CharData
84
85     % sensor output bridge fields (cos/sin)
86     refImage = UseOptions.BridgeReference;
87
88     % load values from characterization dataset
89     % scales of driven Hx and Hy amplitudes in
90     % characteriazation
91     % stimulus in kA/m
```

```

84     if ~strcmp(CharData.Info.Units.MagneticFieldStrength, ' 
85         kA/m')
86         error('Wrong H-field unit: %s', ...
87             CharData.Info.Units.MagneticFieldStrength);
88     end
89     HxScale = CharData.Data.MagneticField.hx;
90     HyScale = CharData.Data.MagneticField.hy;
91     % cosinus and sinus characterization images for
92     % corresponding field
93     % amplitudes, load and norm to Vcc and Voff, references
94     % of
95     % simulation, adjust reference to bridge gain for
96     % output volgates
97     gain = CharData.Info.SensorOutput.BridgeGain;
98     VcosRef = CharData.Data.SensorOutput.CosinusBridge.( 
99         refImage) ...
100        .* (gain * Vcc / Vnorm) + Voff;
101     VsInRef = CharData.Data.SensorOutput.SinusBridge.( 
102         refImage) ...
103        .* (gain * Vcc / Vnorm) + Voff;
104     catch ME
105         rethrow(ME)
106     end
107
108     % now everything is successfully loaded, execute further
109     % constants
110
111     % which are needs to be generated once for all following
112     % operations
113
114     % meshgrids for refernece images to query bridge reference
115     % with interp2
116     [HxScaleGrid, HyScaleGrid] = meshgrid(HxScale, HyScale);
117     % allocate memory for results of on setup run, speed up
118     % compute by 10
119
120     % fix allocations which are not changing by varring
121     % parameters like

```

```
108 % number of angles or positon, for all parameter depended
109 % memory size
110 % allocalte matlab automatically by function call or need
111 % reallocation
112 % in for loops
113 % H-field components for each rotation step
114 Hx = zeros(N, N, nAngles);
115 Hy = zeros(N, N, nAngles);
116 Hz = zeros(N, N, nAngles);
117 % H-field abs for each rotation setp
118 Habs = zeros(N, N, nAngles);
119 % Bridge output voltages for each sensor in grid, H-fields,
120 % sensor
121 % grid, voltages all same orientation
122 Vcos = zeros(N, N, nAngles);
123 Vsini = zeros(N, N, nAngles);
124
125
126 % compute values which not changing by loop parameters
127 % magnetic dipole moments for each rotation step
128 % rotation angles to compute
129 % index corresponding to full scale rotation with angleRes
130 [m, angles, angleRefIndex] = generateDipoleRotationMoments(
131     M0mag, ...
132     nAngles, tilt, angleRes, phaseIndex);
133
134 % rotation angle step width on full rotation 360 with
135 % subset of angles
136 if length(angles) > 1
137     angleStep = angles(2) - angles(1);
138 else
139     angleStep = 0;
140 end
141
142 % compute dipole rest position norm to imprint a certain
143 % field
```

```
138 % strength magnitude with respect of tilt in y axes and
139 % magnetization
140 r0 = rotate3DVector([0; 0; -(z0 + rsp)], 0, tilt, 0);
141 m0 = rotate3DVector([-M0mag; 0; 0], 0, tilt, 0);
142 H0norm = computeDipoleH0Norm(H0mag, m0, r0);
143
144 % prepare file header Info struct, overwrite certain fields
145 % in loop like x,
146 % y, z positions
147 Info = struct;
148 Info.SensorArrayOptions = SensorArrayOptions;
149 Info.SensorArrayOptions.SensorCount = N^2;
150 Info.DipoleOptions = DipoleOptions;
151 Info.UseOptions = UseOptions;
152 Info.CharData = join( ...
153     [CharData.Info.SensorManufacturer, CharData.Info.Sensor
154         ]);
155 Info.Units.SensorOutputVoltage = 'V';
156 Info.Units.MagneticFieldStrength = 'kA/m';
157 Info.Units.Angles = 'degree';
158 Info.Units.Length = 'mm';
159
160 % collect relevant to Data struct for save to file with
161 % file
162 % header Info struct, overwrite position depended fields in
163 % loop before save
164 Data = struct;
165 Data.HxScale = HxScale;
166 Data.HyScale = HyScale;
167 Data.VcosRef = VcosRef;
168 Data.VsinRef = VsinRef;
169 Data.Gain = gain;
170 Data.r0 = r0;
171 Data.m0 = m0;
172 Data.H0norm = H0norm;
```

```
169 Data.m = m;
170 Data.angles = angles;
171 Data.angleStep = angleStep;
172 Data.angleRefIndex = angleRefIndex;
173
174 % generate dataset for all use case setup pairs in for loop
175 % and append
176 % generated dataset path to path struct for result
177 % outer to inner loop is positions to angles
178 % generate z layer wise
179 for z = zPos
180     for x = xPos
181         for y = yPos
182             % generate sensor array grid according to
183             % current position
184             % current position vector of sensor array
185             % relative to
186             % magnet surface
187             p = [x; y; z];
188             % write current position in file header
189             Info.UseOptions.xPos = x;
190             Info.UseOptions.yPos = y;
191             Info.UseOptions.zPos = z;
192             % sensor array grid coordinates
193             [X, Y, Z] = generateSensorArraySquareGrid(N, a,
194                 p, rsp);
195             % save current sensor gird to Data struct
196             Data.X = X;
197             Data.Y = Y;
198             Data.Z = Z;
199             for i = 1:nAngles
200                 % calculate H-field of one rotation step
201                 % for all
202                 % positions, the field is normed to zero
203                 % position
```

```
198     H = computeDipoleHField(X, Y, Z, m(:,i),
199                             H0norm);
200     % separate parts or field in axes direction
201     % / components
202     Hx (:,:,i) = reshape(H(1,:), N, N);
203     Hy (:,:,i) = reshape(H(2,:), N, N);
204     Hz (:,:,i) = reshape(H(3,:), N, N);
205     Habs (:,:,i) = reshape(sqrt(sum(H.^2, 1)), N
206                           , N);
207     % get bridge outputs from references by
208     % cross pick
209     % references from grid, the Hx and Hy
210     % queries can be
211     % served as matrix as long they have same
212     % size and
213     % orientation the nearest neighbor
214     % interpolation
215     % returns of same size and related to
216     % orientation, for
217     % outlayers return NaN, do this for every
218     % angle
219     Vcos (:,:,i) = interp2(HxScaleGrid,
220                            HyScaleGrid, VcosRef, ...
221                            Hx (:,:,i), Hy (:,:,i), 'nearest', NaN);
222     Vsinc (:,:,i) = interp2(HxScaleGrid,
223                               HyScaleGrid, VsincRef, ...
224                               Hx (:,:,i), Hy (:,:,i), 'nearest', NaN);
225     end % angles
226     % save rotation results to Data struct
227     Data.Hx = Hx;
228     Data.Hy = Hy;
229     Data.Hz = Hz;
230     Data.Habs = Habs;
231     Data.Vcos = Vcos;
232     Data.Vsin = Vsinc;
233     % save results to file
```

```
223         fName = sprintf(fNameFmt, datestr(now, dfStr));
224         Info.filePath = fullfile(fPath, fName);
225         disp(Info.filePath)
226         save(Info.filePath, 'Info', 'Data', '-v7.3', '-'
227             'nocompression');
228         end % y
229     end % x
230 end
```

#### E.4.2 gaussianProcessRegression

### 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.

- **Model struct:**

- kernel: Indicator which kernel implementation is used QFC or QFCAPX.
- theta: Kernel parameter vector.
- s2fBounds: Lower and upper bounds for theta(1).
- slBounds: Lower and upper bounds for theta(2).
- s2n: Noise level.
- s2nBounds: Lower and upper bounds for s2n.
- mean: Indicator if mean computation in GPR is active (poly) or not (zero).
- polyDegree: Polynom degree if mean is set to poly. Up degree of 4 is working valid.
- N: Number of reference angles.
- Angles: Column vector of reference angles.
- D: Number of sensor array pixels at each array edge.
- P: Number of predictors or number of sensor array pixels.
- Sensor: Indicator of which characterization was used sensor array datasets.
- PF: Periodicity of angular data depending on characterization dataset.
- Ysin: Column vector of sine regression targets by reference angles.
- Ycos: Column vector of cosine regression targets by reference angles.
- Xcos: Cosine training data.
- Xsin: Sine training data.
- kernelFun: Function handle to loaded covariance function by kernel indicator.

- inputFun: Function handle to loaded input function by kernel indicator. Prepares training and test data in front of regression computations.
- basisFun: Function handle to loaded polynom function by kernel indicator.
- Ky: Covariance Matrix for noisy observations.
- L: Lower triangle matrix of cholesky decomposed Ky.
- logDet: Logarithmic determinante of Ky.
- BetaCos: Polynom coefficients for polynomial mean approximation for cosine function as regression mean basis.
- BetaSin: Polynom coefficients for polynomial mean approximation for sine function as regression mean basis.
- meanFunCos: Function handle for cosine mean approximation by basisFun and BetaCos coefficients.
- meanFunSin: Function handle for sine mean approximation by basisFun and BetaSin coefficients.
- AlphaCos: Regression weights for cosine predictions.
- AlphaSin: Regression weights for sine predictions.
- LMLcos: Logarithmic marginal likelihood for cosine prediction.
- LMLsin: Logarithmic marginal likelihood for sine prediction.
- MSLLA: Mean standardized logarithmic loss for angles.
- MSLLR: Mean standardized logarithmic loss for radius.

## Contents

- basicMathFunctions
- kernelQFC
- kernelQFCAPX
- initGPR
- initGPROptions
- initTrainDS
- initKernel
- initKernelParameters

- tuneKernel
- computeTuneCriteria
- predFrame
- predDS
- lossDS
- optimGPR
- computeOptimCriteria
- See Also

### **basicMathFunctions**

Submodule which contains basic math function to module functionanlity.

### **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 singel 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.

#### E.4.2.1 initGPR

## initGPR

Initializes GPR model by passed trainings dataset and GPR options struct.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 Mdl = initGPR(TrainDS, GPOptions)
```

### Description

Mdl = initGPR(TrainDS, GPOptions) sequential initializing.

## Examples

```
1 load config.mat PathVariables GPROptions;
2 TrainFiles = dir(fullfile(PathVariables.trainingDataPath, [
3     'Training*.mat')));
4 TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat
5   '));
6 assert(~isempty(TrainFiles), 'No training
7 datasets found.');
8 assert(~isempty(TestFiles), 'No test
9 datasets found.');
10 try
11     TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1)
12         .name));
13     TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).
14         name));
15 catch ME
16     rethrow(ME)
17 end
18 Mdl = initGPR(TrainDS, GPROptions);
19 [fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predDS(Mdl,
20     TestDS)
```

## Input Arguments

**TrainDS** loaded training data by infront processeses 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.

```
1 function Mdl = initGPR(TrainDS, GPROptions)
2
3     % create model struct
4     Mdl = struct();
5
6     % init GPROptions on model struct
7     Mdl = initGPROptions(Mdl, GPROptions);
8
9     % init training data on model
10    Mdl = initTrainDS(Mdl, TrainDS);
11
12    % init kernel, covariance function, mean function and input
13        % transformation
14    % function if needed, initGPROptions and initTrainDS must
15        % run before
16    % initKernel otherwise missing parameters causing an error
17    Mdl = initKernel(Mdl);
```

```
17 % init model kernel with current hyperparameters, kernel  
18 % must be initiated  
19 % before otherwise nonesens and errors  
20 Mdl = initKernelParameters(Mdl);  
21 end
```

#### E.4.2.2 initGPROptions

## initGPROptions

Initiates GPR options struct from config on GPR model and sets defaults if expected options are not available.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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.

```
1 function Mdl = initGPROptions(Mdl, GPROptions)
2
3     % set kernel function option
4     if isfield(GPROptions, 'kernel')
5         Mdl.kernel = GPROptions.kernel;
6     else
7         Mdl.kernel = 'QFC';
8     end
9
10    % attach hyperparameters to model and bounds for tuning and
11    % model
12    % optimization
13    % theta covariance function parameter theta = [s2f, s1]
14    if isfield(GPROptions, 'theta')
15        Mdl.theta = GPROptions.theta;
16    else
```

```
16     Mdl.theta = [1, 1];
17 end
18
19 % lower and upper bound for tuning theta
20 if isfield(GPROptions, 's2fBounds')
21     Mdl.s2fBounds = GPROptions.s2fBounds;
22 else
23     Mdl.s2fBounds = [1e-2, 1e2];
24 end
25 if isfield(GPROptions, 's1Bounds')
26     Mdl.s1Bounds = GPROptions.s1Bounds;
27 else
28     Mdl.s1Bounds = [1e-2, 1e2];
29 end
30
31 % noise variance s2n to predict noisy observations
32 if isfield(GPROptions, 's2n')
33     Mdl.s2n = GPROptions.s2n;
34 else
35     Mdl.s2n = 1e-5;
36 end
37
38 % lower and upper bounds for optimizing s2n
39 if isfield(GPROptions, 's2nBounds')
40     Mdl.s2nBounds = GPROptions.s2nBounds;
41 else
42     Mdl.s2nBounds = [1e-4, 10];
43 end
44
45 % enable disable mean function and correction
46 if isfield(GPROptions, 'mean')
47     Mdl.mean = GPROptions.mean;
48 else
49     Mdl.mean = 'zero';
50 end
51
```

```
52 % set polynom degree to model, default is 1 for linear
53 % correction
54 if isfield(GPROptions, 'polyDegree')
55     Mdl.polyDegree = GPROptions.polyDegree;
56
57     % limit poly degree, because higher polynomials as degree
58     % 7 causes
59     % an error in cholesky decomposition
60     if Mdl.polyDegree > 5
61         Mdl.polyDegree = 5;
62     end
63 else
64     Mdl.polyDegree = 1;
65 end
```

#### E.4.2.3 initTrainDS

## 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.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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.

```
1 function Mdl = initTrainDS(Mdl, TrainDS)
2
3 % set model parameters from training dataset and training
4 % data dependencies
5 % N number of angles and references in degree
6 Mdl.N = TrainDS.Info.UseOptions.nAngles;
7 Mdl.Angles = TrainDS.Data.angles';
8
9 % D sensor array square dimension of DxD sensor array
10 Mdl.D = TrainDS.Info.SensorArrayOptions.dimension;
11
12 % P number of predictors in sensor array
13 Mdl.P = TrainDS.Info.SensorArrayOptions.SensorCount;
14 % get sensor type from dataset
```

```
15 Mdl.Sensor = TrainDS.Info.UseOptions.BaseReference;
16
17 % choose period factor depending on sensor type
18 % how many sinoid periods are abstract on a full rotation
19 % by 360
20 switch Mdl.Sensor
21     case 'TDK'
22         Mdl.PF = 1;
23
24     case 'KMZ60'
25         Mdl.PF = 2;
26
27     otherwise
28         error('Unkown Sensor %s.', Mdl.Sensor);
29 end
30
31 % get reference angles in degree and transpose to column
32 % vector
33 % get sinoid target vectors depending period factor,
34 % transpose because angles2sinoids works with row vectors
35 [Mdl.Ysin, Mdl.Ycos] = angles2sinoids(Mdl.Angles, ...
36     false, Mdl.PF);
37
38 % attach training data fro cosine and sine to model
39 Mdl.Xcos = TrainDS.Data.Vcos;
40 Mdl.Xsin = TrainDS.Data.Vsin;
41 end
```

#### E.4.2.4 initKernel

## initKernel

Initiates kernel and chooses kernel implementation by initiated GPR options.

### Contents

- Syntax
- Description
- Input Argurments
- Output Argurments
- Requirements
- See Also

### Syntax

```
1 Md1 = initKernel(Md1)
```

### Description

**Md1 = initKernel(Md1)** loads kernel submodule by passed identifier.

### Input Argurments

**Md1** 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.

```
1 function Mdl = initKernel(Mdl)
2
3     % set covariance function and input function to respect
4         % covariance function
5     % serve with right format of data, sets mean function which
6         % belongs to the
7     % corresponding kernel model
8     switch Mdl.kernel
9         case 'QFC'
10            Mdl = initQFC(Mdl);
11
12        case 'QFCAPX'
13            Mdl = initQFCAPX(Mdl);
14
15        % end kernel select
```

```
14     otherwise
15         error('Unknown kernel function %.', Mdl.kernel);
16     end
17 end
```

#### E.4.2.5 initKernelParameters

### initKernelParameters

Init GPR model on current kernel parameters, computes covariance matrix and depending kernel values means, mean coefficients, regression weights and likelihoods.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 Md1 = initKernelParameters(Md1)
```

### Description

**Md1 = initKernelParameters(Md1)** initializes the regresion model in final.

### Input Arguments

**Md1** 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 Juenemann. Copyright Klaus Juenemann 2019.

```
1 function Mdl = initKernelParameters(Mdl)
2
3 % compute noise free covariance matrix
4 Mdl.Ky = Mdl.kernelFun(Mdl.Xcos, Mdl.Xcos, Mdl.Xsin, Mdl.
5 Xsin, Mdl.theta);
6
7 % add noise to covariance matrix along its diagonal, which
8 % is noise on
9 % trainig observations with itself
10 Mdl.Ky = addNoise2Covariance(Mdl.Ky, Mdl.s2n);
11
12 % compute the cholesky decomposition of the covariance
13 % matrix and the log
```

```
11 % determinate of the covariance matrix, computes lower
12 % triangle matrix
13 [Mdl.L, Mdl.logDet] = decomposeChol(Mdl.Ky);
14
15 % compute beta coefficients to fit H matrices of cosine and
16 % sine mean
17 % function if none zero mean is set as model mean, for zero
18 % mean all beta
19 % and related means are zero.
20 switch Mdl.mean
21 case 'zero'
22     % set not needed kernel parameters to zero,
23     % beta is not used in zero mean GPR and so all
24     % related means are
25     % zero, as name lets expect
26     Mdl.BetaCos = 0;
27     Mdl.BetaSin = 0;
28     Mdl.meanFunCos = @(X) 0;
29     Mdl.meanFunSin = @(X) 0;
30
31 case 'poly'
32     % estimate beta for none zero H matrices
33     Mdl.BetaCos = estimateBeta(Mdl.basisFun(Mdl.Xcos),
34         Mdl.L, Mdl.Ycos);
35     Mdl.BetaSin = estimateBeta(Mdl.basisFun(Mdl.Xsin),
36         Mdl.L, Mdl.Ysin);
37
38     % mean function for polynom approximated mean H' *
39     % beta
40     Mdl.meanFunCos = @(X) Mdl.basisFun(X)' * Mdl.
41         BetaCos;
42     Mdl.meanFunSin = @(X) Mdl.basisFun(X)' * Mdl.
43         BetaSin;
44
45 otherwise
```

```
37     error('Unsupported mean function %s in beta
38             estimation.', Mdl.mean);
39
40     % compute weights for cosine and sine, angles in rads and
41     % radius
42     Mdl.AlphaCos = computeAlphaWeights(Mdl.L, Mdl.Ycos, ...
43                                         Mdl.meanFunCos(Mdl.Xcos));
44     Mdl.AlphaSin = computeAlphaWeights(Mdl.L, Mdl.Ysin, ...
45                                         Mdl.meanFunSin(Mdl.Xsin));
46
47     % compute log marginal likelihoods for each cosine and sine
48     % weights
49     Mdl.LMLcos = computeLogLikelihood(Mdl.Ycos, Mdl.meanFunCos(
50                                         Mdl.Xcos), ...
51                                         Mdl.AlphaCos, Mdl.logDet, Mdl.N);
52     Mdl.LMLSin = computeLogLikelihood(Mdl.Ysin, Mdl.meanFunSin(
53                                         Mdl.Xsin), ...
54                                         Mdl.AlphaSin, Mdl.logDet, Mdl.N);
55
56 end
```

#### E.4.2.6 tuneKernel

## tuneKernel

Tunes kernel hyperparameters of GPR model. Dismiss tuning for each kernel parameter by setting corresponding bounds to equal values.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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.

```
1 function Mdl = tuneKernel(Mdl, verbose)
2
3     % define options for minimum search
4     options = optimoptions('fmincon', 'Display', 'off', ...
5         'Algorithm', 'sqp', ...
6         'PlotFcn', {@optimplotx, @optimplotfval});
7
8     % display tuning
9     if verbose, options.Display = 'iter'; end
10
11    % setup problem for minimum solver use problem structure to
12    % feed fmincon
13    problem.solver = 'fmincon';
14    problem.options = options;
```

```
13
14     % apply bounds to prevent overfitting
15     problem.lb = [Mdl.s2fBounds(1) Mdl.slBounds(1)];
16     problem.ub = [Mdl.s2fBounds(2) Mdl.slBounds(2)];
17
18     % set sl start value
19     problem.x0 = Mdl.theta;
20
21     % apply objective function and start values
22     problem.objective = @(x) computeTuneCriteria(x, Mdl);
23
24     % solve problem
25     [Mdl.theta] = fmincon(problem);
26
27
28     % reinit kernel with tuned parameters
29     Mdl = initKernelParameters(Mdl);
30 end
```

#### E.4.2.7 computeTuneCriteria

## 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.

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 feval = computeTuneCriteria(theta, Mdl)
```

## Description

**feval = computeTuneCriteria(theta, Mdl)** sets new kernel parameter, reinitiates model and calculates min criteria by likelihoods.

## Input Argurments

**theta** kernel parameter vector.

**Mdl** model struct to reinitiate.

## Output Argurments

**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.

```
1 function feval = computeTuneCriteria(theta, Mdl)
2     % reinit kernel on new theta kernel parameters
3     Mdl.theta = theta;
4     Mdl = initKernelParameters(Mdl);
5
6     % return function evaluation as neg. likelihood of radius
7     feval = -1 * (Mdl.LMLsin + Mdl.LMLcos);
8 end
```

#### E.4.2.8 predFrame

## **predFrame**

Predicts single test point and computes angle and radius by predicted sinoids. Delivers several quality criteria.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 [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 Argurments

**Mdl** model struct.

**Xcos** matrix frame of cosine test data.

**Xsin** matrix frame of sine test data.

## Output Argurments

**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 Juenemann. Copyright Klaus Juenemann 2019.

```
1 function [fang, frad, fcos, fsin, fcov, s, ciang, cirad] =
2     predFrame(Mdl, ...
3     Xcos, Xsin)
4
5     % adjust inputs if needed
6     Xcos = Mdl.inputFun(Xcos);
7     Xsin = Mdl.inputFun(Xsin);
8
9     % compute covariance between observations and test point
10    k = Mdl.kernelFun(Mdl.Xcos, Xcos, Mdl.Xsin, Xsin, Mdl.theta
11        );
12
13    % compute predictiv variance as the difference between test
14        point covariance
15    % which should be Mdl.theta(1) = s2f product of the
16        covariance between
17    % observations and test points
18    % compute the covariance of test point itself means
19        distance is zero which
20    % implies that result must be the variance s2f
21    c1 = Mdl.kernelFun(Xcos, Xcos, Xsin, Xsin, Mdl.theta);
22    % assert(c1 == Mdl.theta(1));
23
24
25    % now add variance from additives
26    fcov = c1 - computeTransposeInverseProduct(Mdl.L, k);
```

```

21
22     % predict depending on model mean function
23     switch Mdl.mean
24         case 'zero'
25             % compute the predictive means directly by
26             % covariance vector and
27             % alpha weights, mean is zero
28             fcos = k' * Mdl.AlphaCos;
29             fsin = k' * Mdl.AlphaSin;
30         case 'poly'
31             % compute
32             fcos = Mdl.meanFunCos(Xcos) + k' * Mdl.AlphaCos;
33             fsin = Mdl.meanFunSin(Xsin) + k' * Mdl.AlphaSin;
34         otherwise
35             error('Unsupported mean function %s in prediction.'
36                   , Mdl.mean);
37     end
38
39     % compute radius from sinoid results
40     frad = sqrt(fcos^2 + fsin^2);
41
42     % compute angle in rad from sinoid results
43     fang = sinoids2angles(fsin, fcos, frad, true);
44
45     % sigma of the normal distribution over fradius
46     s = sqrt(fcov + Mdl.s2n);
47
48     % 95% confidence interval over fradius
49     ciang = [fang - asin(1.96 * s * sqrt(2)), fang + asin(1.96
50             * s * sqrt(2))];
51
52     % 95% confidence interval over fradius
53     cirad = [frad - 1.96 * s * sqrt(2), frad + 1.96 * s * sqrt
54             (2)];
55
56 end

```

#### E.4.2.9 predDS

## **predDS**

Predicts all frames of a test dataset at once.

### **Contents**

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### **Syntax**

```
1 [fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predDS(Mdl,  
    TestDS)  
2 predicts whole dataset at once using predFrame in a loop.
```

### **Description**

**[fang, frad, fcos, fsin, fcov, 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.

```
1 function [fang, frad, fcos, fsin, fcov, s, ciang, cirad] =
2     predDS(Mdl, TestDS)
3
4     % get number of angles in dataset
5     N = TestDS.Info.UseOptions.nAngles;
6
7     % allocate memory for results
8     fang = zeros(N, 1); % angle
9     frad = zeros(N, 1); % radius
10    fcos = zeros(N, 1); % cosine
11    fsin = zeros(N, 1); % sine
12    fcov = zeros(N, 1); % predictive covariance over radius
13    s = zeros(N, 1); % sigma standard deviation over radius
14    ciang = zeros(N, 2); % confidence 95% interval over angles
15        lower and upper
16    cirad = zeros(N, 2); % confidence 95% interval over raidus
17        lower and upper
18
19    % predict angle by angle from dataset
20    for n = 1:N
21        % get cosine and sine at n-th angle
22        Xcos = TestDS.Data.Vcos(:,:,n);
23        Xsin = TestDS.Data.Vsin(:,:,n);
24
25        % predict frame
26        [fang(n), frad(n), fcos(n), fsin(n), ...
27         fcov(n), s(n), ciang(n,:), cirad(n,:)] = predFrame(Mdl
28             , Xcos, Xsin);
29    end
30 end
```

#### E.4.2.10 lossDS

## lossDS

Predicts all angles of passed test dataset and computes logarithmic losses for radius and angles plus several squared errors.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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

```
1 Enter example matlab code for each use case.
```

## Input Argurments

**positionalArg** argurment description.

**optionalArg** argurment description.

## Output Argurments

**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.

```
1 function [AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl,
2 TestDS)
3
4 % get number of angles in dataset
5 N = TestDS.Info.UseOptions.nAngles;
6
7 % get simulated cosin and sine references from dataset
8 % angles in degrees
9 % and transpose to column vector, get sinoids and angles in
10 % rads
11 [ysin, ycos, yang] = angles2sinoids(TestDS.Data.angles',
12 false, Mdl.PF);
13
14 % create reference radius of unit cricle, radius must be
15 % one for all angles
16 yrad = ones(N, 1);
17
18 % predict angles in rads not in degrees
19 [fang, frad, fcos, fsin, ~, s, ~, ~] = predDS(Mdl, TestDS);
20
21 % compute log loss and squared error for angles in rad
22 [SLLA, SEA] = computeStdLogLoss(yang, fang, asin(s) * sqrt
23 (2));
24
25 % compute abslute angular error in degrees
26 AAED = sqrt(SEA) * 180/pi;
27
28 % compute log loss and squared error for radius
29 [SLLR, SER] = computeStdLogLoss(yrad, frad, sqrt(2) * s);
30
31 % compute squared error of sinoids
32 SEC = (ycos - fcos).^2;
33 SES = (ysin - fsin).^2;
34
35 end
```

#### E.4.2.11 optimGPR

## optimGPR

Noise level optimization that implements optimized model by embedded kernel tuning process.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 Md1 = optimGPR(TrainDS, TestDS, GPROptions, verbose)
```

### Description

**Md1 = 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

```
1 load config.mat PathVariables GPROptions;
2 TrainFiles = dir(fullfile(PathVariables.trainingDataPath, [
3     'Training*.mat')));
4 TestFiles = dir(fullfile(PathVariables.testDataPath, 'Test*.mat
5     '));
6 assert(~isempty(TrainFiles), 'No training
7     datasets found.');
8 assert(~isempty(TestFiles), 'No test
9     datasets found.');
10 try
11     TrainDS = load(fullfile(TrainFiles(1).folder, TrainFiles(1)
12         .name));
13     TestDS = load(fullfile(TestFiles(1).folder, TestFiles(1).
14         name));
15     catch ME
16         rethrow(ME)
17     end
18     Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose);
19     [fang, frad, fcos, fsin, fcov, s, ciang, cirad] = predDS(Mdl,
20         TestDS)
21     [AAED, SLLA, SLLR, SEA, SER, SEC, SES] = lossDS(Mdl, TestDS);
```

## Input Arguments

**TrainDS** loaded training data by infront processesesd sensor array simulation.

**TestDS** loaded test data by infront processesed 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.

```
1 function Mdl = optimGPR(TrainDS, TestDS, GPROptions, verbose)
2
3     % init model by training data and initial options
4     Mdl = initGPR(TrainDS, GPROptions);
5
6     % create noise variance s2n used in GPR with bounds
7     s2n = optimizableVariable('s2n', GPROptions.s2nBounds, '
8         Transform', 'log');
9
10    % create function handle for bayes optimization
```

```
10 SLL = GPROptions.SLL;
11 fun = @(OptVar) computeOptimCriteria(OptVar, Mdl, TestDS,
12                                     SLL, verbose);
13
14 % perform bayes noise optimization
15 results = bayesopt(fun, s2n, ...
16                     'Verbose', verbose, ...
17                     'MaxObjectiveEvaluations', GPROptions.OptimRuns, ...
18                     'AcquisitionFunctionName', 'expected-improvement-per-
19                     second');
20
21 % update options with results and reinit model and tune to
22 % final model
23 Mdl.s2n = results.XAtMinObjective.s2n;
24 Mdl = tuneKernel(Mdl, verbose);
25
26 % compute final loss and get mean log loss for angles and
27 % radius as
28 % indicator of model total model fit
29 [~, SLLA, SLLR] = lossDS(Mdl, TestDS);
30 Mdl.MSLLA = mean(SLLA);
31 Mdl.MSLLR = mean(SLLR);
32
33 end
```

#### E.4.2.12 computeOptimCriteria

## computeOptimCriteria

Object function to compute the loss of a fully initialized and tuned regression model. Computes the mean std. log. loss of angles MSLLA or radius MSLLR as function evaluation value for bayesopt. Perform noise adjustment in cycles in bayesopt.

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose)
```

## Description

**MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL, verbose)**

## Input Argurments

**OptVar** optimzation 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 Argurments

**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.

```
1 function MSLL = computeOptimCriteria(OptVar, Mdl, TestDS, SLL,
2                                         verbose)
3
4     % push current variance value into GPR
5     Mdl.s2n = OptVar.s2n;
6
7     % tune kernel with new noise variance
8     Mdl = tuneKernel(Mdl, verbose);
9
10    % get loss on dataset for angular prediction
11    switch SLL
12        case 'SLLA'
13            [~, SLL] = lossDS(Mdl, TestDS);
14
15        case 'SLLR'
16            [~, ~, SLL] = lossDS(Mdl, TestDS);
17
18        otherwise
19            error('Unknown SLL %s.', SLL);
20
21        end
22
23    MSLL = mean(SLL);
24end
```

#### **E.4.2.13 kernelQFCAPX**

### **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.

### **Contents**

- QFCAPX
- meanPolyQFCAPX
- initQFCAPX
- See Also

### **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.

#### E.4.2.13.1 QFCAPX

## QFCAPX

Approximates QFC with triangle inequation, norming is pulled out to input stage kernel is feeded with norm vectors or scalars instead of matrices.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 K = QFCAPX(ax, bx, ay, by, theta)
```

### Description

**K = QFCAPX(ax, bx, ay, by, theta)** computes quadratic distances bewtween data points and parametrize it with height and length scales. Computes distance with quadratic euclidian norm.

## Input Argurments

**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 Argurments

**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.

```

1 function K = QFCAPX(ax, bx, ay, by, theta)
2     arguments
3         % validate data as real vector of same size
4         ax (:,:) double {mustBeReal}
5         bx (:,:) double {mustBeReal, mustBeFitSize(ax,bx)}
6         ay (:,:) double {mustBeReal, mustBeFitSize(ax,ay)}
7         by (:,:) double {mustBeReal, mustBeFitSize(ax,by)}
8         % validate kernel parameters as 1x2 vector
9         theta (1,2) double {mustBeReal}
10    end
11
12    % get number of observations for each dataset, cosine and
13    % sine
14    M = length(ax);
15    N = length(bx);
16
17    % expand covariance parameters, variance and lengthscale
18    c2 = 2 * theta(2)^2; % 2*s1^2
19    c1 = theta(1) * c2; % s2f * c
20
21    % allocate memory for K
22    K = zeros(M, N);
23
24    % loop through observation points and compute the
25    % covariance for each
26    % observation against another
27    for m = 1:M
28        for n = 1:N
29            % get distance between m-th and n-th observation
30            % compute distance with quadratic frobenius normed
31            % vectors
32            r2 = (ax(m) - bx(n))^2 + (ay(m) - by(n))^2;
33
34            % engage lengthscales and variance on distance
35            K(m,n) = c1 / (c2 + r2);
36        end
37    end

```

```
33
34         end
35     end
36 end
37
38 function mustBeFitSize(a, b)
39     % Test for equal size
40     if ~isequal(size(a,2), size(b,2))
41         eid = 'Size:notEqual';
42         msg = 'Sizes of are not fitting.';
43         throwAsCaller(MException(eid,msg))
44     end
45 end
```

#### E.4.2.13.2 meanPolyQFCAPX

## 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.

### Contents

- Syntax
- Description
- Input Argurments
- Output Argurments
- Requirements
- See Also

### Syntax

```
1 H = meanPolyQFCAPX(X, degree)
```

### Description

**H = meanPolyQFCaPX(X, degree)** build polynom by passed data.

## Input Argurments

**X** vector data.

**degree** polynom degree.

## Output Argurments

**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.

```
1 function H = meanPolyQFCAPX(X, degree)
2     % get number of observations
3     N = length(X);
4
5     % returns only ones if p = 0
6     H = ones(degree + 1, N);
7
8     % compute polynom for degrees > 0
9     if degree > 0
10        H(2,:) = X';
```

```
11 end
12
13 % compute none linear polynoms if degree > 1
14 if degree > 1
15   for p = 2:degree
16     H(p+1,:) = X'.^p;
17   end
18 end
19 end
```

#### E.4.2.13.3 initQFCAPX

## 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.

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 Md1 = initQFCAPX(Md1)
```

## Description

**Md1 = initQFCAPX(Md1)** loads approximated quadratic fraction covariance function and basis function depending on mean in **Md1** struct. Sets input function to Frobenius Norm. Reprocess training matrix data to vector data.

## Input Argurments

**Mdl** struct with model parameter and training data.

## Output Argurments

**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.

```
1 function Mdl = initQFCAPX(Mdl)
2
3     % set QFC kernel function
4     Mdl.kernelFun = @QFCAPX;
5
6     % set input transformation function to apply adjustments to
7     % covariance function, norm input matrice with frobenius
8         norm to scalar or
% vectors.
```

```
9 Mdl.inputFun = @(X) frobeniusNorm(X, false);  
10  
11 % transform traning data to vectors  
12 Ncos = zeros(Mdl.N, 1);  
13 Nsin = zeros(Mdl.N, 1);  
14 for n = 1:Mdl.N  
15     Ncos(n) = Mdl.inputFun(Mdl.Xcos(:,:,n));  
16     Nsin(n) = Mdl.inputFun(Mdl.Xsin(:,:,n));  
17 end  
18  
19 % update training data with norm vectors  
20 Mdl.Xcos = Ncos;  
21 Mdl.Xsin = Nsin;  
22  
23 % set mean function to compute cosine and sine H matrix  
24 switch Mdl.mean  
25     % zero mean m(x) = 0  
26     case 'zero'  
27         % set polyDegree to -1 for no polynom indication  
28         Mdl.polyDegree = -1;  
29  
30         % set basis function  
31         Mdl.basisFun = @(X) 0;  
32  
33     % mean by polynom m(x) = H' * beta  
34     case 'poly'  
35         % set basis function produces a (polyDeg+1)xN H  
36             matrix  
37         Mdl.basisFun = @(X) meanPolyQFCAPX(X, Mdl.  
38             polyDegree);  
39  
40         % end mean select QFC kernel  
41         otherwise  
42             error('Unknown mean function .', Mdl.mean);  
43     end  
44 end
```

[REDACTED]

#### **E.4.2.14 kernelQFC**

### **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.

### **Contents**

- QFC
- meanPolyQFC
- initQFC
- See Also

### **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.

#### E.4.2.14.1 QFC

## QFC

Quadratic Fractional Covariance function. Computes covariance matrix K. Works with raw matrix data. Precise solution.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 K = QFC(Ax, Bx, Ay, By, theta)
```

### Description

**K = QFC(Ax, Bx, Ay, By, theta)** computes quadratic distances between data points and parametrize it with height and length scales. Computes distance with quadratic Frobenius Norm.

## Input Argurments

**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 Argurments

**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 Juenemann. Copyright Klaus Juenemann 2019.

```

1 function K = QFC(Ax, Bx, Ay, By, theta)
2     arguments
3         % validate data as real matrices of same size in 1st
4         % and 2nd dimension
5         Ax (:,:, :) double {mustBeReal}
6         Bx (:,:, :) double {mustBeReal, mustBeFitSize(Ax,Bx) }
7         Ay (:,:, :) double {mustBeReal, mustBeFitSize(Ax,Ay) }
8         By (:,:, :) double {mustBeReal, mustBeFitSize(Ax,By) }
9         % validate kernel parameters as 1x2 vector
10        theta (1,2) double {mustBeReal}
11    end
12
13    % get number of observations for each dataset, cosine and
14    % sine matrices have
15    % equal sizes just extract size from one
16    [~, ~, M] = size(Ax);
17    [~, ~, N] = size(Bx);
18
19    % expand covariance parameters, variance and lengthscale
20    c2 = 2 * theta(2)^2; % 2*s1^2
21    c1 = theta(1) * c2;   % s2f * c
22
23    % allocate memory for K
24    K = zeros(M, N);
25
26    % loop through observation points and compute the
27    % covariance for each
28    % observation against another
29    for m = 1:M
30        for n = 1:N
31            % get distance between m-th and n-th observation
32            distCos = Ax(:,: ,m) - Bx(:,: ,n);
33            distSin = Ay(:,: ,m) - By(:,: ,n);

```

```
32      % compute quadratic frobenius norm distance as
33      % separated
34      % distances of cosine and sine, norm of vector
35      % fields
36      r2 = sum(distCos .^ 2 , 'all') + sum(distSin .^ 2 ,
37          'all');
38
39      % engage lengthscale and variance on distance
40      K(m,n) = c1 / (c2 + r2);
41
42  end
43
44 function mustBeFitSize(A, B)
45     % Test for equal size
46     if ~isequal(size(A,1,2), size(B,1,2))
47         eid = 'Size:notEqual';
48         msg = 'Sizes of are not fitting.';
49         throwAsCaller(MException(eid,msg))
50     end
51 end
```

#### E.4.2.14.2 meanPolyQFC

## 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.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 H = meanPolyQFC(X, degree)
```

### Description

**H = meanPolyQFC(X, degree)** build polynom by passed data. Fires Frobenius Norm on matrix data.

## Input Argurments

**X** matrix data.

**degree** polynom degree.

## Output Argurments

**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.

```
1 function H = meanPolyQFC(X, degree)
2     % get number of observations
3     [~, ~, N] = size(X);
4
5     % returns only ones if p = 0
6     H = ones(degree + 1, N);
7
8     % compute polynom for degrees > 0
9     if degree > 0
```

```
10      for n = 1:N
11          H(2,n) = frobeniusNorm(X(:,:,n), false);
12      end
13  end
14
15 % compute none linear polynomials if degree > 1
16 if degree > 1
17     for p = 2:degree
18         H(p+1,:) = H(2,:).^p;
19     end
20 end
21 end
```

#### E.4.2.14.3 initQFC

## 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.

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 Md1 = initQFC(Md1)
```

## Description

**Md1 = initQFC(Md1)** loads quadratic fraction covariance function and basis function depending on mean in **Md1** struct. Sets input function as bypass.

## Input Argurments

**Mdl** struct with model parameter and training data.

## Output Argurments

**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.

```
1 function Mdl = initQFC(Mdl)
2
3     % set QFC kernel function
4     Mdl.kernelFun = @QFC;
5
6     % set input transformation function to apply adjustments to
7     % covariance function, here bypass inputs as they are, no
8     % transformation of
9     % training data needed
10    Mdl.inputFun = @(X) X;
```

```
10
11    % set mean function to compute cosine and sine H matrix
12    switch Mdl.mean
13        % zero mean m(x) = 0
14        case 'zero'
15            % set polyDegree to -1 for no polynom indication
16            Mdl.polyDegree = -1;
17
18        % set basis function
19        Mdl.basisFun = @ (X) 0;
20
21    % mean by polynom m(x) = H' * beta
22    case 'poly'
23        % set basis function produces a (polyDeg+1)xN H
24        % matrix
25        Mdl.basisFun = @ (X) meanPolyQFC(X, Mdl.polyDegree);
26
27    % end mean select QFC kernel
28    otherwise
29        error('Unknown mean function %.', Mdl.mean);
30    end
31 end
```

#### **E.4.2.15 basicMathFunctions**

### **basicMathFunctions**

Set of basic algebraic, analytic and trigonometric functions.

### **Contents**

- sinoids2angles
- angles2sinoids
- decomposeChol
- frobeniusNorm
- computeInverseMatrixProduct
- computeTransposeInverseProduct
- addNoise2Covariance
- computeAlphaWeights
- computeStdLogLoss
- computeLogLikelihood
- estimateBeta

#### **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.

#### E.4.2.15.1 sinoids2angles

### 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 therefore 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

$$f_c = \arccos \left( \frac{f_{cos}}{f_{rad}} \right)$$

$$f_s = \arcsin \left( \frac{f_{sin}}{f_{rad}} \right)$$

$$f_a = \arctan 2 \left( \frac{f_{sin}}{f_{cos}} \right)$$

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. 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.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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

```
1 phi = linspace(0, 2*pi, 100);
2 fsin = sin(phi);
3 fcos = cos(phi);
4 frad = sqrt(fsin.^{}2 + fcos.^{}2);
5 [fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad, true)
```

## Input Argurments

**fsin** is a scalar or vector with sine information to corresponding angle.

**fco**s 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 Argurments

**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.

```
1 function [fang, fc, fs, fa] = sinoids2angles(fsin, fcos, frad,
2     rad, how)
3     arguments
4         % validate sinoids and radius as scalar or vector of
5         % the same size
6         fsin (:,1) double {mustBeReal}
7         fcos (:,1) double {mustBeReal, mustBeEqualSize(fsin,
8             fcos)}
9         frad (:,1) double {mustBeReal, mustBeEqualSize(fsin,
10            frad)}
11        % validate rad as boolean flag with default true
12        rad (1,1) logical {mustBeNumericOrLogical} = true
13        % validate how as char option flag with default atan2
14        how (1,:) char {mustBeText} = 'atan2'
15
16    end
17
18    % compute angles by cosine, sine and radius
19    fc = acos(fcos ./ frad);
20    fs = asin(fsin ./ frad);
21    fa = atan2(fsin, fcos);
22
23    % get indices for interval > 180
24    idx = fs < 0;
25
26    switch how
27        case 'acos'
28            % angles from cosine
29            fang = fc;
30
31            % correct 180 interval
```

```
27         fang(idx) = -1 * fang(idx) + 2 * pi;
28
29     case 'atan2'
30         fang = fa;
31
32         % correct 180 interval
33         fang(idx) = fang(idx) + 2 * pi;
34
35     otherwise
36         error('Unknow arc function for reconstruction: %s.'
37               , how)
38
39     end
40
41     % return degrees if not rad
42     if ~rad, fang = 180 / pi * fang; end
43 end
44
45 % Custom validation function
46 function mustBeEqualSize(a,b)
47     % Test for equal size
48     if ~isequal(size(a),size(b))
49         eid = 'Size:notEqual';
50         msg = 'Size of first input must equal size of second
51         input.';
52         throwAsCaller(MException(eid,msg))
53     end
54 end
```

#### E.4.2.15.2 angles2sinoids

### 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.

$$f_{sin} = \sin(p_f \cdot f_{ang})$$

$$f_{cos} = \cos(p_f \cdot f_{ang})$$

If needed a recomputation of the given angels takes place by computed sinoids.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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

```
1 fang = linspace(0, 360, 100);
2 [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.

```
1 function [fsin, fcos, fang] = angles2sinoids(fang, rad, pf)
2     arguments
3         % validate angles as scalar or vector
4         fang (:,1) double {mustBeReal}
5         % validate rad as boolean flag with default true
6         rad (1,1) logical {mustBeNumericOrLogical} = true
7         % validate period factor as positive scalar with
8             default 1
9         pf (1,1) double {mustBeInteger, mustBePositive} = 1
10    end
11
12    % if rad flag is false and angles in degree convert to rad
13    if ~rad, fang = fang * pi / 180; end
14
15    % calculate sinoids
16    fsin = sin(pf * fang);
17    fcos = cos(pf * fang);
18
19    % compute radius
20    frad = sqrt(fcos.^2 + fsin.^2);
```

```
20
21     % recalculate angles to corrected sinoids in rad
22     if nargout > 2, fang = sinoids2angles(fsin, fcos, frad);
23         end
24 end
```

#### E.4.2.15.3 decomposeChol

## 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.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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

```
1 A = [1.0, 0.9, 0.8;
2     0.9, 1.0, 0.9;
3     0.8, 0.9, 1.0];
4 [L, logDet] = decomposeChol(A)
5 assert(all(L*L'==A, 'all'))
6 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.

```
1 function [L, logDet] = decomposeChol(A)
2     arguments
3         A (:,:) double {mustBeReal, mustBeValidMatrix(A)}
4     end
5
6     % decompose A to lower triangle matrix
7     [L, flag]= chol(A, 'lower');
8     if flag ~= 0
9         eid = 'chol:Fails';
10        msg = 'Cholesky Decomposition fails.';
11        throwAsCaller(MException(eid,msg))
12    end
13
14    % compute log determinate of A
15    if nargout > 1, logDet = 2 * sum(log(diag(L))); end
16 end
17
18 % Custom validation function
19 function mustBeValidMatrix(A)
20     % test if is matrix
21     if ~ismatrix(A)
22         eid = 'Matrix:notMatrix';
23         msg = 'A is not a matrix.';
24         throwAsCaller(MException(eid,msg))
25     end
26     % Test for N x N
27     if ~isequal(size(A,1),size(A,2))
28         eid = 'Matrix:notNxN';
29         msg = 'Size of matrix is not N x N.';
30         throwAsCaller(MException(eid,msg))
```

```
31 end
32 % test if symmetric
33 if ~issymmetric(A)
34     eid = 'Matrix:notSymmetric';
35     msg = 'Matrix is not symmetric.';
36     throwAsCaller(MException(eid,msg))
37 end
38 % test if positive definite
39 if ~all(eig(A) >= 0)
40     eid = 'Matrix:notPosDefinite';
41     msg = 'Matrix is not pos. definite.';
42     throwAsCaller(MException(eid,msg))
43 end
44 end
```

#### E.4.2.15.4 frobeniusNorm

## frobeniusNorm

Computes the Frobenius Norm of a matrix A.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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

```
1 A = magic(8);  
2 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.

```
1 function nv = frobeniusNorm(A, approx)
2     arguments
3         % validate A as real matrix
4         A (:,:) double {mustBeReal}
5         % validate approx as flag with default false
6         approx (1,1) logical {mustBeNumericOrLogical} = false
7     end
8
9     % norm matrix
```

```
10 if approx
11     % approximate frobenis with mean and multiply with
12     % radicant of RMS
13     % frobenius norm is a RMS * sqrt(N x N), RMS >= mean
14     nv = mean2(A) * sqrt(numel(A));
15 else
16     % norm with frobenius
17     nv = sqrt(sum(A.^2, 'all'));
18 end
```

#### E.4.2.15.5 computeInverseMatrixProduct

### 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).

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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

```
1 A = [1.0, 0.9, 0.8;
2     0.9, 1.0, 0.9;
3     0.8, 0.9, 1.0];
4 L = decomposeChol(A);
5 b = [5; 9; 0.5];
6 x = computeInverseMatrixProduct(L, b);
7 B = [5, 9;
8     0.5, 5;
9     3, -1];
10 x = computeInverseMatrixProduct(L, B);
```

## Input Argurments

**L** is the lower triangle matrix of a matrix A.

**b** is a vector or matrix of real values.

## Output Argurments

**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.

```
1 function x = computeInverseMatrixProduct(L, b)
2     arguments
3         % validate L as lower triangle matrix of size N x N
4         L (:,:) double {mustBeReal, mustBeLowerTriangle(L)}
5         % validate b as vecotr matrix with same row length as L
6         b (:,:) double {mustBeReal, mustBeFitSize(L, b)}
7     end
8
9     % set linsolve option for inner (lower triangle) and outer
10    % (upper triangle)
11    % solve, outer solve runs with intermediate result of inner
12    % solve
13    opts1.LT = true;
14    opts2.UT = true;
15
16    % get size of b, if b is a matrix solve column by column
17    [M, N] = size(b);
18
19    % allocate memory for product result
20    x = zeros(M, N);
21
22    % solve column by column
23    for n = 1:N
24        % compute inner solve to intermediate result vecotor
25        v = linsolve(L, b(:,n), opts1);
```

```
26         x(:,n) = linsolve(L', v, opts2);
27     end
28 end
29
30 % Custom validation functions
31 function mustBeLowerTriangle(L)
32     % Test for lower triangle matrix
33     if ~istril(L)
34         eid = 'Matrix:notLowerTriangle';
35         msg = 'Matrix is not lower triangle.';
36         throwAsCaller(MException(eid,msg))
37     end
38     % Test for N x N
39     if ~isequal(size(L,1), size(L, 2))
40         eid = 'Size:notEqual';
41         msg = 'L is not size of N x N.';
42         throwAsCaller(MException(eid,msg))
43     end
44 end
45
46 function mustBeFitSize(L, b)
47     % Test for equal size
48     if ~isequal(size(L,1), size(b, 1))
49         eid = 'Size:notEqual';
50         msg = 'Size of rows are not fitting.';
51         throwAsCaller(MException(eid,msg))
52     end
53 end
```

#### E.4.2.15.6 computeTransposeInverseProduct

### 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.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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

```
1 A = [1.0, 0.9, 0.8;
2     0.9, 1.0, 0.9;
3     0.8, 0.9, 1.0];
4 L = decomposeChol(A);
5 b = [5; 9; 0.5];
6 x = computeTransposeInverseProduct(L, b);
7 B = [5, 9;
8     0.5, 5;
9     3, -1];
10 X = computeTransposeInverseProduct(L, B);
```

## Input Argurments

**L** is the lower triangle matrix of a matrix **A**.

**b** is a vector or matrix of real values.

## Output Argurments

**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.

```
1 function x = computeTransposeInverseProduct(L, b)
2     arguments
3         % validate L as lower triangle matrix of size N x N
4         L (:,:) double {mustBeReal, mustBeLowerTriangle(L)}
5         % validate b as vecotr matrix with same row length as L
6         b (:,:) double {mustBeReal, mustBeFitSize(L, b)}
7     end
8
9     % set linsolve option to solve with lower triangle matrix
10    opts.LT = true;
11
12    % get size of b, if b is a matrix solve column by column
13    [M, N] = size(b);
14
15    % allocate memory for intermediate result
16    v = zeros(M, N);
17
18    % solve column by column
19    for n=1:N
20        % save to intermediate result columns
21        v(:,n) = linsolve(L, b(:,n), opts);
22    end
23
24    % get final product by multiply transposed intermediate
25    % result with itself
26    x = v' * v;
26 end
```

```
27 % Custom validation functions
28 function mustBeLowerTriangle(L)
29     % Test for lower triangle matrix
30     if ~istril(L)
31         eid = 'Matrix:notLowerTriangle';
32         msg = 'Matrix is not lower triangle.';
33         throwAsCaller(MException(eid,msg))
34     end
35     % Test for N x N
36     if ~isequal(size(L,1), size(L, 2))
37         eid = 'Size:notEqual';
38         msg = 'L is not size of N x N.';
39         throwAsCaller(MException(eid,msg))
40     end
41 end
42
43
44 function mustBeFitSize(L, b)
45     % Test for equal size
46     if ~isequal(size(L,1), size(b, 1))
47         eid = 'Size:notEqual';
48         msg = 'Size of rows are not fitting.';
49         throwAsCaller(MException(eid,msg))
50     end
51 end
```

#### E.4.2.15.7 addNoise2Covariance

## addNoise2Covariance

Add noise to covariance matrix for noisy observations. Add noise along matrix diagonal.

## Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 Ky = addNoise2Covariance(K, s2n)
```

## Description

**Ky = addNoise2Covariance(K, s2n)** witch on additive noise on covariance matrix diagonal. Uses eye matrix as mask.

## Examples

```
1 addNoise2Covariance(zeros(4), 2)
```

## Input Argurments

**K** N x N covariance matrix. Noise free.

**s2n** real scalar value.

## Output Argurments

**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.

```
1 function Ky = addNoise2Covariance(K, s2n)
2     arguments
3         % validate K as square matrix
4         K (:,:) double {mustBeReal, mustBeSquareMatrix(K) }
5         % validate s2n as scalar value
6         s2n (1,1) double {mustBeReal}
7     end
8
9     % add noise with eye matrix
```

```
10 Ky = K + s2n * eye(size(K));
11 end
12
13 % Custom validation functions
14 function mustBeSquareMatrix(K)
15 % Test for N x N
16 if ~isequal(size(K,1), size(K, 2))
17     eid = 'Size:notEqual';
18     msg = 'K is not size of N x N.';
19     throwAsCaller(MException(eid,msg))
20 end
21 end
```

#### E.4.2.15.8 computeAlphaWeights

## computeAlphaWeights

Computes alpha weights from feature space product  $H^T \beta$  and target vector  $y$  as product with inverse covariance matrix with additive noise  $K_y^{-1}$  represented by its cholesky decomposed lower triangle matrix  $L$ .  $K_y^{-1} * (y - m(x))$ .

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 alpha = computeAlphaWeights(L, y, m)
```

### Description

**alpha = computeAlphaWeights(L, y, m)** prepare data and forward it to matrix computation.

## Input Argurments

**L** lower triangle matrix of cholesky decomposed K matrix.

**y** regression target vector.

**m** regression mean vector.

## Output Argurments

**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 Juenemann. Copyright Klaus Juenemann 2019.

```
1 function alpha = computeAlphaWeights(L, y, m)
2     % get residual
3     residual = y - m;
4     % L and residual is validated in computation below, get
5         weights
6     alpha = computeInverseMatrixProduct(L, residual);
7 end
```

#### E.4.2.15.9 computeStdLogLoss

## 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.

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 [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 Argurments

**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 Argurments

**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.

```

1 function [SLL, SE, s2] = computeStdLogLoss(y, fmean, s)
2     arguments
3         % validate inputs as real column vectors of same length
4         y (:,1) double {mustBeReal, mustBeVector}
5         fmean (:,1) double {mustBeReal, mustBeVector,
6             mustBeEqualSize(y, fmean)}
7         s (:,1) double {mustBeReal, mustBeVector,
8             mustBeEqualSize(y, s)}
9     end
10    % squared error
11    SE = (y - fmean).^2;
12
13    % s as standard deviation of nomal distributed fmean so
14        square it for
15        % variance.
16    s2 = s.^2;
17
18    % logarithmic error
19    SLL = 0.5 * (log(2 * pi * s2) + SE ./ s2);
20
21 end
22
23 % Custom validation functions
24 function mustBeEqualSize(a, b)
25     if ~isequal(length(a), length(b))
26         eid = 'Size:notEqual';
27         msg = 'Vectors must be the same length.';
28         throwAsCaller(MException(eid, msg))
29     end
30 end

```

#### E.4.2.15.10 computeLogLikelihood

## computeLogLikelihood

Computes the marginal log likelihood as evidence of the current trained model parameter by solving the equation  $\log p(\mathbf{y}|\mathbf{X}, \alpha, \log|\mathbf{K}_y|) = -\frac{1}{2} * (\mathbf{y} - \mathbf{m})^T * \alpha - \frac{1}{2} \log|\mathbf{K}_y| - N/2 \log(2\pi)$  where  $\alpha$  is the inverse matrix product of  $\alpha = \mathbf{K}_y^{-1} * (\mathbf{y} - \mathbf{m})$ .

## Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

## Syntax

```
1 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.

```

1 function lml = computeLogLikelihood(y, m, alpha, logDet, N)
2     arguments
3         % validate inputs as real column vectors
4         y (:,1) double {mustBeReal, mustBeVector}
5         % m can be zero if zero gpr runs
6         m (:,1) double {mustBeReal, mustBeVector}
7         alpha (:,1) double {mustBeReal, mustBeVector,
8             mustBeEqualSize(y, alpha)}
9         % validate inputs as real scalar
10        logDet (1,1) double {mustBeReal}
11        N (1,1) double {mustBeReal}
12    end
13
14    % get residual of targets and mean
15    residual = y - m;
16
17    % compute log marginal likelihood
18    lml = -0.5 * (residual' * alpha + logDet + N * log(2 * pi))
19    ;
20
21    % Custom validation functions
22    function mustBeEqualSize(a, b)
23        if ~isequal(length(a), length(b))
24            eid = 'Size:notEqual';
25            msg = 'Vectors must be the same length.';
26            throwAsCaller(MException(eid, msg))
27        end
28    end

```

#### E.4.2.15.11 estimateBeta

### 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.

### Contents

- Syntax
- Description
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 [beta, alpha0]= estimateBeta(H, L, y)
```

### Description

**[beta, alpha0]= estimateBeta(H, L, y)** compute polynom coefficients to solve mean approximation.

## Input Argurments

**H** basis matrix of training data. Polynomial represents training data.

**L** lower triangle matrix of decomposed K matrix.

**y** regression targets.

## Output Argurments

**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.

```
1 function [beta, alpha0]= estimateBeta(H, L, y)
2     % Ky^-1 * y
3     alpha0 = computeInverseMatrixProduct(L, y);
4
5     % H * Ky^-1 * HT
6     alpha1 = computeTransposeInverseProduct(L, H');
7
8     % (H * Ky^-1 * HT) ^-1 * H
9     L1 = chol(alpha1, 'lower');
10    alpha2 = computeInverseMatrixProduct(L1, H);
11
12    % ((H * (Ky^-1 * HT)) ^-1 * H) * (Ky^-1 * y)
13    beta = alpha2 * alpha0;
14 end
```

### E.4.3 util

## 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 allways computes the same use case. The util classificated source code solve module unrelated tasks.

## Contents

- removeFilesFromDir
- publishFilesFromDir
- plotFunctions

### 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.

#### E.4.3.1 removeFilesFromDir

### removeFilesFromDir

Remove files from passed direcotory.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 removeStatus = removeFilesFromDir(directory)
2 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\_\*.\* or \*.txt and so on.

## Examples

```
1 d = fullfile('rootPath', 'subfolder')
2 rs = removeFileFromDir(d)
```

```
1 d = fullfile('rootPath', 'subfolder')
2 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.

```
1 function [removeStatus] = removeFilesFromDir(directory,
2     filePattern)
3     arguments
4         % validate directory
5         directory (1,:) char {mustBeFolder}
6         % validate filePattern
7         filePattern (1,:) char {mustBeText} = '*.*'
8     end
9     % parse pattern for dir
10    parsePattern = fullfile(directory, filePattern);
11    % parse directory, returns struct
12    filesToRemove = dir(parsePattern);
13    for file = filesToRemove'
14        % check before delete
15        filePath = fullfile(file.folder, file.name);
16        if isfile(filePath)
17            delete(filePath);
18        end
19    end
20    % check if dir returns an empty struct now
```

```
21     check = dir(parsePattern);  
22     removeStatus = isempty(check(~ismember({check.name}, {'.',  
23     '..'})));  
23 end
```

#### E.4.3.2 publishFilesFromDir

## publishFilesFromDir

Publish m-files from given directory with passed publishing options.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 publishFilesFromDir(directory, PublishOptions)
2 publishFilesFromDir(directory, PublishOptions, recursivly)
3 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

```
1 directory = 'src';
2 PublishOptions = struct;
3 PublishOptions.outputDir = 'src/html';
4 PublishOptions.evalCode = false;
5 publishFilesFromDir(directory, PublishOptions, true)
```

```
1 load('config.mat', 'srcPath', 'PublishOptions')
2 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.

```
1 function publishFilesFromDir(directory, PublishOptions,
2     recursive, verbose)
3     arguments
4         % validate directory if exist
5         directory (1,:) char {mustBeFolder}
6         % validate Publish options if it is a struct
7         PublishOptions (1,1) struct {mustBeA(PublishOptions, [
8             struct"])}
9         % validate recursive if logical
10        recursive (1,1) logical {mustBeNumericOrLogical} =
11            false
12        verbose (1,1) logical {mustBeNumericOrLogical} = false
13    end
14
15    % file extension is always m-file
16    ext = '*.m';
17
18    % recursive parsing for files with dir function needs a
19        certain regex
20    if recursive
```

```
17     pathPattern = fullfile(directory, '**', ext);
18 else
19     pathPattern = fullfile(directory, ext);
20 end
21
22 % scan directory for files returns a column struct with
23 % path fields
23 files = dir(pathPattern);
24
25 % transpose files struct and loop through for publish
26 for file = files'
27     % if not dir must be file
28     if ~file.isdir
29         % build path by struct field for recursive tree
30         written = publish(fullfile(file.folder, file.name),
31                         PublishOptions);
32         if verbose
33             disp(written);
34         end
35     end
36 end
```

#### **E.4.3.3 plotFunctions**

### **plotFunctions**

Project related reusable plots for datasets and results.

## **Contents**

- [plotTDKTransferCurves](#)
- [plotKMZ60TransferCurves](#)
- [plotKMZ60CharField](#)
- [plotKMZ60CharDataset](#)
- [plotSimulationDatasetCircle](#)
- [plotSimulationCosSinStats](#)
- [plotSimulationSubset](#)
- [plotSingleSimulationAngle](#)
- [plotSimulationDataset](#)
- [plotTDKCharField](#)
- [plotTDKCharDataset](#)
- [plotDipoleMagnet](#)

#### **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 magnet in rest position.

Created on October 24. 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

#### E.4.3.3.1 plotTDKCharDataset

## plotTDKCharDataset

Explore TDK TAS2141 characterization dataset and plot its content.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotTDKCharDataset()
```

### Description

**plotTDKCharDataset()** explores the dataset and plot its content in three docked figure windows. Loads dataset location from config.mat.

### Examples

```
1 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`
- `imagesc`
- `polarplot`

Created on October 24, 2020 by Tobias Wulf. Copyright Tobias Wulf 2020.

```
1 function plotTDKCharDataset()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.tdkDatasetPath, 'Data', 'Info');
7         %       close all;
8     catch ME
9         rethrow(ME)
```

```
9     end
10
11    % figure save path for different formats
12    %
13    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
14
15    fig1Filename = 'tdk_magnetic_stimulus';
16    fig1Path = fullfile(PathVariables.saveImagesPath,
17        fig1Filename);
18
19    fig2Filename = 'tdk_bridge_charistic';
20    fig2Path = fullfile(PathVariables.saveImagesPath,
21        fig2Filename);
22
23    % load needed data from dataset in to local variables for
24    % better handling
25    %
26    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27
28    %
29
30    % check if modulation fits to following reconstructioning
31    if ~strcmp("triang", Info.MagneticField.Modulation)
32        error("Modulation function is not triang.");
33    end
34
35    if ~(strcmp("cos", Info.MagneticField.CarrierHx) && ...
36        strcmp("sin", Info.MagneticField.CarrierHy))
37        error("Carrier functions are not cos or sin.");
38    end
39
40    %
41    % modulation frequency
42    fm = Info.MagneticField.ModulationFrequency;
```

```
34 % carrier frequency
35 fc = Info.MagneticField.CarrierFrequency;
36 % max and min amplitude
37 Hmax = Info.MagneticField.MaxAmplitude;
38 Hmin = Info.MagneticField.MinAmplitude;
39 % step range or window size for output picking
40 Hsteps = Info.MagneticField.Steps;
41 % resolution of H steps
42 Hres = Info.MagneticField.Resolution;
43 % get unit strings from
44 kApm = Info.Units.MagneticFieldStrength;
45 Hz = Info.Units.Frequency;
46 mV = Info.Units.SensorOutputVoltage;
47
48 % get dataset infos and format strings to place in figures
49 % subtitle string for all figures
50 infoStr = join([Info.SensorManufacturer, Info.Sensor, ...
51                 Info.SensorTechnology, ...
52                 Info.SensorType, "Sensor Characterization Dataset."]);
53 dateStr = join(["Created on", Info.Created, "by", 'Thorben
54 Sch\"uthe', ...
55                 "and updated on", Info.Edited, "by", Info.Editor +
56                 "."]);
```

```
57 % load characterization data
58 Vcos = Data.SensorOutput.CosinusBridge;
59 Vsincos = Data.SensorOutput.SinusBridge;
60 gain = Info.SensorOutput.BridgeGain;
61
62 % clear dataset all loaded
63 clear Data Info;
64 disp('Info:');
65 disp([infoStr; dateStr]);
```

```
66 % reconstruct magnetic stimulus and reduce the view for
       example plot by 10
```

```

67 %
68 %
69 % number of periods reduced by factor 10
70 reduced = 10;
71 nPeriods = fc / fm / reduced;
72 % number of samples for good looking 40 times nPeriods
73 nSamples = nPeriods * 400;
74 % half number of samples
75 nHalf = round(nSamples / 2);
76 % generate angle base
77 phi = linspace(0, nPeriods * 2 * pi, nSamples);
78 % calculate modulated amplitude, triang returns a column
79 % vector, transpose
80 Hmag = Hmax * triang(nSamples)';
81 % calculate Hx and Hy stimulus
82 Hx = Hmag .* cos(phi);
83 Hy = Hmag .* sin(phi);
84 % index for rising and falling stimulus
85 idxR = 1:nHalf;
86 idxF = nHalf:nSamples;
87 % find absolute min and max values in bridge outputs for
88 % uniform colormap
89 A = cat(3, Vcos.Rise, Vcos.Fall, Vcos.All, Vcos.Diff, Vsins.
90 Rise, ...
91 Vsins.Fall, Vsins.All, Vsins.Diff);
92 Vmax = max(A, [], 'all');
93 Vmin = min(A, [], 'all');
94 clear A;
95 %
96 % figure 1 magnetic stimulus

```

```

94 %
95 %
96 fig1 = figure('Name', 'Magnetic Stimulus');
97 tiledlayout(fig1, 2, 2);
98 %
99 % title and description
100 disp("Title: Magnetic Stimulus Reconstructed H_x-/ H_y-
101 Stimulus" + ...
102 "in Reduced View");
103 disp("Description: Stimulus for characterization in H_x and
104 H_y in " + ...
105 "reduced period view by factor 10");
106 disp(["a) Triangle modulated cosine carrier for H_x
107 stimulus."; ...
108 "b) Triangle modulated sine carrier for H_x stimulus
109 ."; ...
110 "c) Modulation trajectory for rising stimulus"; ...
111 "d) Modulation trajectory for falling stimulus"]);
112 %
113 % Hx stimulus
114 nexttile;
115 p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hx(idxR), phi(
116 idxF), Hx(idxF));
117 set(p, {'Color'}, {'k', 'k', 'b', 'r'});
118 legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location
119 ', 'NorthEast');
120 xticks((0:0.25*pi:2*pi) * nPeriods);
121 xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$',
122 '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
123 xlim([0 phi(end)]);
124 ylim([Hmin Hmax]);

```

```

119 xlabel('$\phi$ in rad, Periode $\times 10$');
120 ylabel(sprintf('$H_x(\phi)$ in %s', kApm));
121 title(sprintf('a) $f_m = %.2f %s, $f_c = %.2f %s', fm,
122 Hz, fc, Hz));
123
124 % Hy stimulus
125 nexttile;
126 p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hy(idxR), phi(
127 idxF), Hy(idxF));
128 set(p, {'Color'}, {'k', 'k', 'b', 'r'});
129 legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location
130 ', 'NorthEast');
131 xticks((0:0.25*pi:2*pi) * nPeriods);
132 xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi
133 $', ...
134 '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
135 xlim([0 phi(end)]);
136 ylim([Hmin Hmax]);
137 xlabel('$\phi$ in rad, Periode $\times 10$');
138 ylabel(sprintf('$H_y(\phi)$ in %s', kApm));
139 title(sprintf('b) $f_m = %.2f %s, $f_c = %.2f %s', fm,
140 Hz, fc, Hz));
141
142 % polar for rising modulation
143 nexttile;
144 polarplot(phi(idxR), Hmag(idxR), 'b');
145 p = gca;
146 p.ThetaAxisUnits = 'radians';
147 title('c) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $0 < \phi < 320\pi$');
148
149 % polar for rising modulation
150 nexttile;
151 polarplot(phi(idxF), Hmag(idxF), 'r');
152 p = gca;
153 p.ThetaAxisUnits = 'radians';

```

```

149 title('d) $|\vec{H}(\phi)| \cdot e^{j\phi}|$, $320<\phi
150 <640\pi$');
151 % figure 2 cosinus bridge outputs
152 %
153 %
154 fig2 = figure('Name', 'Cosine and Sine Bridge', 'Position',
155 [0 0 33 30]);
156 tiledlayout(fig2, 2, 2);
157 %
158 % title and description
159 disp("Title: Cosine and Sine Bridge. Measured Bridge
160 Outputs" + ...
161 " of Corresponding H_x-/ H_y-Amplitudes");
162 disp("Description: " + sprintf("H_x, H_y in %s, %d Steps in
163 %.4f %s", ...
164 kApm, Hsteps, Hres, kApm));
165 disp(["a) Cosine Bridge Rising H-Amplitudes"; ...
166 "b) Cosine Bridge Falling H-Amplitudes"; ...
167 "c) Sine Bridge Rising H-Amplitudes"; ...
168 "d) Sine Bridge Falling H-Amplitudes"]);
169 colormap('jet');
170 %
171 % cosinus bridge recorded during rising stimulus
172 nexttile;
173 im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Rise);
174 set(gca, 'YDir', 'normal');
175 set(im, 'AlphaData', ~isnan(Vcos.Rise));
176 caxis([Vmin, Vmax]);
177 xlim([Hmin Hmax]);

```

```
177     ylim([Hmin Hmax]);
178     axis square xy;
179     xlabel('$H_x$ in kA/m');
180     ylabel('$H_y$ in kA/m');
181     title('a) $V_{\cos}(H_x, H_y)$');
182     yticks([-20 -10 0 10 20]);
183     xticks([-20 -10 0 10 20]);
184
185 % cosinus bridge recorded during falling stimulus
186 nexttile;
187 im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Fall);
188 set(gca, 'YDir', 'normal');
189 set(im, 'AlphaData', ~isnan(Vcos.Fall));
190 caxis([Vmin, Vmax]);
191 xlim([Hmin Hmax]);
192 ylim([Hmin Hmax]);
193 axis square xy;
194 xlabel('$H_x$ in kA/m');
195 ylabel('$H_y$ in kA/m');
196 title('b) $V_{\cos}(H_x, H_y)$');
197 yticks([-20 -10 0 10 20]);
198 xticks([-20 -10 0 10 20]);
199
200 % sinus bridge recorded during rising stimulus
201 nexttile;
202 im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Rise);
203 set(gca, 'YDir', 'normal');
204 set(im, 'AlphaData', ~isnan(Vsin.Rise));
205 caxis([Vmin, Vmax]);
206 xlim([Hmin Hmax]);
207 ylim([Hmin Hmax]);
208 axis square xy;
209 xlabel('$H_x$ in kA/m');
210 ylabel('$H_y$ in kA/m');
211 title('c) $V_{\sin}(H_x, H_y)$');
212 yticks([-20 -10 0 10 20]);
```

```

213 xticks([-20 -10 0 10 20]);
214
215 % sinus bridge recorded during falling stimulus
216 nexttile;
217 im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Fall);
218 set(gca, 'YDir', 'normal');
219 set(im, 'AlphaData', ~isnan(Vsin.Fall));
220 caxis([Vmin, Vmax]);
221 xlim([Hmin Hmax]);
222 ylim([Hmin Hmax]);
223 axis square xy;
224 xlabel('H_x in kA/m');
225 ylabel('H_y in kA/m');
226 title('d) V_{sin}(H_x, H_y)');
227 yticks([-20 -10 0 10 20]);
228 xticks([-20 -10 0 10 20]);
229
230 % add colorbar and place it overall plots
231 cb = colorbar;
232 cb.Layout.Tile = 'east';
233 cb.Label.String = sprintf(...,
234     '$V(H_x, H_y)$ in %s, Gain $ = %.1f$', mV, gain);
235 cb.Label.Interpreter = 'latex';
236 cb.TickLabelInterpreter = 'latex';
237 cb.Label.FontSize = 24;
238
239 % yesno = input('Save? [y/n]: ', 's');
240 % if strcmp(yesno, 'y')
241 %     % save results of figure 1
242 %     savefig(fig1, fig1Path);
243 %     print(fig1, fig1Path, '-dsvg');
244 %     print(fig1, fig1Path, '-depsc', '-tiff', '-loose');
245 %     print(fig1, fig1Path, '-dpdf', '-loose', '-fillpage')
246 %
247 %         % save results of figure 2

```

```
248 %         savefig(fig2, fig2Path);
249 %         print(fig2, fig2Path, '-dsvg');
250 %         print(fig2, fig2Path, '-depsc', '-tiff', '-loose');
251 %         print(fig2, fig2Path, '-dpdf', '-loose', '-fillpage')
252 ;
253 %     end
254 %     close(fig1)
255 %     close(fig2)
256 end
```

#### E.4.3.3.2 plotTDKCharField

## plotTDKCharField

Explore TDK TAS2141 characterization field.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotTDKCharField()
```

### Description

**plotTDKCharField()** explore characterization field of TDK sensor.

### Examples

```
1 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.

```
1 function plotTDKCharField()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.tdkDatasetPath, 'Data', 'Info');
7     %     close all;
8     catch ME
9         rethrow(ME)
10    end
```

```
11 % load needed data from dataset in to local variables for
12 % better handling %%
13 %
14 % get from user which field to investigate and limits for
15 % plateau
16 fields = Info.SensorOutput.CosinusBridge.Determination;
17 nFields = length(fields);
18 fprintf('Choose 1 of %d fields ...\\n', nFields);
19 for i = 1:nFields
20     fprintf('%s\\t:\\t(%d)\\n', fields{i}, i);
21 end
22
23 iField = 1; % input('Choice: ');
24 field = fields{iField};
25 pl = 5; % input('Plateau limit in kA/m: ');
26
27 Vcos = Data.SensorOutput.CosinusBridge.(field);
28 Vsin = Data.SensorOutput.SinusBridge.(field);
29 gain = Info.SensorOutput.BridgeGain;
30 HxScale = Data.MagneticField.hx;
31 HyScale = Data.MagneticField.hy;
32 Hmin = Info.MagneticField.MinAmplitude;
33 Hmax = Info.MagneticField.MaxAmplitude;
34
35 % get unit strings from
36 kApm = Info.Units.MagneticFieldStrength;
37 mV = Info.Units.SensorOutputVoltage;
38
39 % get dataset infos and format strings to place in figures
40 % subtitle string for all figures
41 infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
```

```
42 dateStr = join(["Created on", Info.Created, "by", 'Thorben
43 Sch\"uthe', ...
44 "and updated on", Info.Edited, "by", Info.Editor +
45 "."]);  
46  
47 % clear dataset all loaded  
48 clear Data Info;  
49  
50 % figure save path for different formats  
51 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
53 % define slices and limits to plot  
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
55 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
56 Hslice = [128 154 180 205]; % hit ca. 0, 5, 10, 15 kA/m  
57 Hlims = [-pl pl];  
58 mVpVlims = [-175 175];  
59  
60 % create figure for plots  
61 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
62 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
63 fig = figure('Name', 'Char Field', 'OuterPosition', [0 0 35
64 30]);  
65 tiledlayout(fig, 2, 2);  
66  
67 % title and description  
68 disp('Info:');
```

```
66 disp([infoStr; dateStr]);
67 fprintf('Title: TDK Characterization Field - %s\n', field);
68 disp('Description:');
69 disp(["a) Cosine Bridge Characteristic"; ...
70     "b) Transfer slices for different const. H_y of Vcos
71     ";" ...
72     "c) Sine Bridge Characteristic"; ...
73     "d) Transfer slices for different const. H_x of Vsin
74     "]);
```

75 % set colormap  
76 colormap('jet');

77 % cosinus bridge  
%%%%%%%%%%%%%

78 %  
%%%%%%%%%%%%%

```
79 nexttile(1);
80 im = imagesc(HxScale, HyScale, Vcos);
81 set(gca, 'YDir', 'normal');
82 set(im, 'AlphaData', ~isnan(Vcos));
83 xticks(-20:10:20);
84 yticks(-20:10:20);
85 axis square xy;
```

86

```
87 % plot lines for slice to investigate
88 hold on;
89 for i = Hslice
90     yline(HyScale(i), 'k:', 'LineWidth', 3.5);
91 end
92 hold off;
```

93

```
94 xlabel(sprintf('$H_x$ in %s', kApm));
95 ylabel(sprintf('$H_y$ in %s', kApm));
```

```

96     title(sprintf('a) $V_{cos}(H_x,H_y)$, Gain $ = %.1f$', gain
97           ));
98
99     cb = colorbar;
100    cb.Label.String = sprintf('$V_{cos}$ in %s', mV);
101    cb.Label.Interpreter = 'latex';
102    cb.TickLabelInterpreter = 'latex';
103    cb.Label.FontSize = 20;
104
105    % cosinus bridge slices
106    %%%
107    %
108    nexttile(2);
109    % slices
110    p = plot(HxScale, Vcos(Hslice,:));
111
112    % plateau limits
113    if pl > 0
114        hold on;
115        xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
116        xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
117        hold off;
118    end
119
120    legend(p, {'$H_y \approx 0$ kA/m', ...
121             '$H_y \approx 5$ kA/m', ...
122             '$H_y \approx 10$ kA/m', ...
123             '$H_y \approx 15$ kA/m'}, ...
124             'Location', 'SouthEast');
125    xlabel(sprintf('$H_x$ in %s', kApm));
126    title('b) $V_{cos}(H_x,H_y)$, $H_y = $ const.');
127    ylim(mVpVlims);
128    xlim([Hmin Hmax])

```

```

128 % sinus bridge
129 %
130 nexttile(3);
131 im = imagesc(HxScale, HyScale, Vsin);
132 set(gca, 'YDir', 'normal');
133 set(im, 'AlphaData', ~isnan(Vsin));
134 xticks(-20:10:20);
135 yticks(-20:10:20);
136 axis square xy;
137
138 % plot lines for slice to investigate
139 hold on;
140 for i = Hslice
141     xline(HxScale(i), 'k:', 'LineWidth', 3.5);
142 end
143 hold off;
144
145 xlabel(sprintf('$H_x$ in %s', kApm));
146 ylabel(sprintf('$H_y$ in %s', kApm));
147 title(sprintf('c) $V_{sin}(H_x, H_y)$, Gain $ = %.1f$', gain
148 ));
149 cb = colorbar;
150 cb.Label.String = sprintf('$V_{sin}$ in %s', mV);
151 cb.Label.Interpreter = 'latex';
152 cb.TickLabelInterpreter = 'latex';
153 cb.Label.FontSize = 20;
154
155 % sinus bridge slices

```

```

156 %
157 nexttile(4);
158 % slices
159 p = plot(HxScale, Vsin(:,Hslice));
160
161 % plateau limits
162 if pl > 0
163     hold on;
164     xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
165     xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
166     hold off;
167 end
168
169 legend(p, {'$H_x \approx 0\$ kA/m', ...
170             '$H_x \approx 5\$ kA/m', ...
171             '$H_x \approx 10\$ kA/m', ...
172             '$H_x \approx 15\$ kA/m'}, ...
173             'Location', 'SouthEast');
174 xlabel(sprintf('$H_y\$ in %s', kApm));
175 title('d) $V_{\sin}(H_x,H_y)\$, $H_x = \$ \text{const.}');
176 ylim(mVpVlims);
177 xlim([Hmin Hmax])
178
179 % save results of figure
180 %
181 % yesno = input('Save? [y/n]: ', 's');
182 % if strcmp(yesno, 'y')
183 %     savefig(fig, fPath);
184 %     print(fig, fPath, '-dsvg');
185 %     print(fig, fPath, '-depsc', '-tiff', '-loose');
186 %     print(fig, fPath, '-dpdf', '-loose', '-fillpage');

```

```
187 %      end  
188 %      close(fig)  
189 end
```

#### E.4.3.3.3 plotTDKTransferCurves

## plotTDKTransferCurves

Plot TDK TAS2141 characterization field transfer curves.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotTDKTransferCurves()
```

### Description

**plotTDKTransferCurves()** plot characterization field of TDK sensor.

### Examples

```
1 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.

```
1 function plotTDKTransferCurves()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.tdkDatasetPath, 'Data', 'Info');
7         %       close all;
8         catch ME
9             rethrow(ME)
10    end
```

```
11 % load needed data from dataset in to local variables for
12 % better handling %%
13 %
14 % get from user which field to investigate and limits for
15 % plateau
16 fields = Info.SensorOutput.CosinusBridge.Determination;
17 nFields = length(fields);
18 fprintf('Choose 1 of %d fields ...\\n', nFields);
19 for i = 1:nFields
20     fprintf('%s\\t:\\t(%d)\\n', fields{i}, i);
21 end
22
23 iField = 1; % input('Choice: ');
24 field = fields{iField};
25 pl = 5; % input('Plateau limit in kA/m: ');
26
27 Vcos = Data.SensorOutput.CosinusBridge.(field);
28 Vsing = Data.SensorOutput.SinusBridge.(field);
29 gain = Info.SensorOutput.BridgeGain;
30 HxScale = Data.MagneticField.hx;
31 HyScale = Data.MagneticField.hy;
32 Hmin = Info.MagneticField.MinAmplitude;
33 Hmax = Info.MagneticField.MaxAmplitude;
34
35 % get unit strings from
36 kApm = Info.Units.MagneticFieldStrength;
37 mV = Info.Units.SensorOutputVoltage;
38
39 % get dataset infos and format strings to place in figures
40 % subtitle string for all figures
41 infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
```

```
42 dateStr = join(["Created on", Info.Created, "by", 'Thorben
43 Sch\"uthe', ...
44 "and updated on", Info.Edited, "by", Info.Editor +
45 "."]); 
46
47 % clear dataset all loaded
48 clear Data Info;
49
50 % figure save path for different formats
51 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
52 %
53 % define slices and limits to plot
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55 Hslice = 128; % hit ca. 0 kA/m
56 Hlims = [-pl pl];
57 mVpVlims = [-175 175];
58
59 % create figure for plots
60 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
61 fig = figure('Name', 'Transfer Curves', 'OuterPosition', [0
62 0 33 30]);
63 tiledlayout(fig, 2, 2);
64
65 disp('Info:');
```

```

66 disp([infoStr; dateStr]);
67 disp('Title:');
68 fprintf('KMZ 60 Transfer Curves: %s\n', field);
69 disp(["a) Cosine Bridge Characteristic"; ...
70 "b) Sine Bridge Characteristic"; ...
71 "c) Transfer Curves for const. H_x = H_y = 0"]);
```

72

```

73 % set colormap
74 colormap('jet');
```

75

```

76 % cosinus bridge
77 %%
```

78 nexttile(1);
79 im = imagesc(HxScale, HyScale, Vcos);
80 set(gca, 'YDir', 'normal');
81 set(im, 'AlphaData', ~isnan(Vcos));
82 xticks(-20:10:20);
83 yticks(-20:10:20);
84 axis square xy;

85

```

86 % plot lines for slice to investigate
87 hold on;
88 yline(HyScale(Hslice), 'k:', 'LineWidth', 3.5);
89 plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth',
90 3.5);
91 hold off;
```

92 xlabel(sprintf('\$H\_x\$ in %s', kApm));
93 ylabel(sprintf('\$H\_y\$ in %s', kApm));
94 title(sprintf('a) \$V\_{\cos}(H\_x, H\_y)\$, Gain \$ = %.1f\$', gain
95 ));

```

96    % sinus bridge
97
98    %
99
100   nexttile(2);
101   im = imagesc(HxScale, HyScale, Vsin);
102   set(gca, 'YDir', 'normal');
103   set(im, 'AlphaData', ~isnan(Vsin));
104   xticks(-20:10:20);
105   yticks(-20:10:20);
106   axis square xy;
107
108   % plot lines for slice to investigate
109   hold on;
110   xline(HxScale(Hslice), 'k:', 'LineWidth', 3.5);
111   plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth',
112       3.5);
113   hold off;
114
115   xlabel(sprintf('$H_x$ in %s', kApm));
116   ylabel(sprintf('$H_y$ in %s', kApm));
117   title(sprintf('b) $V_{sin}(H_x,H_y)$, Gain $ = %.1f$', gain
118         ));
119
120   % colorbar for both
121
122   cb = colorbar;
123   cb.Label.String = sprintf('$V_{out}$ in %s', mV);
124   cb.TickLabelInterpreter = 'latex';
125   cb.Label.Interpreter = 'latex';

```

```

123 cb.Label.FontSize = 20;
124
125 % cosinus bridge slices
126 %
127 nexttile([1 2]);
128 % slices
129 p = plot(HxScale, Vcos(Hslice,:), HyScale, Vsing(:, Hslice)
130   ');
131 % plateau limits
132 if pl > 0
133   hold on;
134   xline(Hlims(1), 'k-.', 'LineWidth', 3.5);
135   xline(Hlims(2), 'k-.', 'LineWidth', 3.5);
136   hold off;
137 end
138
139 legend(p, {sprintf('$V_{cos}(H_x,H_y)$, $H_y \\\approx 0$ %s
140   ', kApm), ...
141   sprintf('$V_{sin}(H_x,H_y)$, $H_x \\\approx 0$ %s
142   ', kApm) }, ...
143   'Location', 'SouthEast');
144 ylabel(sprintf('$V_{out}$ in %s', mV));
145 xlabel(sprintf('$H$ in %s', kApm));
146 title('c) Cosine and Sine Transfer Curves');
147 ylim(mVpVlims);
148 xlim([Hmin Hmax])
149 %

```

```
150 %     yesno = input('Save? [y/n]: ', 's');
151 %     if strcmp(yesno, 'y')
152 %         savefig(fig, fPath);
153 %         print(fig, fPath, '-dsvg');
154 %         print(fig, fPath, '-depsc', '-tiff', '-loose');
155 %         print(fig, fPath, '-dpdf', '-loose', '-fillpage');
156 %     end
157 %     close(fig)
158
159 end
```

#### E.4.3.3.4 plotKMZ60CharDataset

## plotKMZ60CharDataset

Explore NXP KMZ60 characterization dataset and plot its content.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotKMZ60CharDataset()
```

### Description

**plotKMZ60CharDataset()** explores the dataset and plot its content in three docked figure windows. Loads dataset location from config.mat.

### Examples

```
1 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.

```
1 function plotKMZ60CharDataset()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
7         %       close all;
8         catch ME
9             rethrow(ME)
10        end
11        % figure save path for different formats
```

```

12 %
13 %
14 fig1Filename = 'kmz60_magnetic_stimulus';
15 fig1Path = fullfile(PathVariables.saveImagesPath,
16     fig1Filename);
17 fig2Filename = 'kmz60_bridge_characteristic';
18 fig2Path = fullfile(PathVariables.saveImagesPath,
19     fig2Filename);

20 %
21 %
22 %
23 %
24 %
25 %
26 %
27 %
28 %
29 %
30 %
31 %
32 fm = Info.MagneticField.ModulationFrequency;
33 %
34 fc = Info.MagneticField.CarrierFrequency;
35 %
36 Hmax = Info.MagneticField.MaxAmplitude;

```

```
37 Hmin = Info.MagneticField.MinAmplitude;
38 % step range or window size for output picking
39 Hsteps = Info.MagneticField.Steps;
40 % resolution of H steps
41 Hres = Info.MagneticField.Resolution;
42 % get unit strings from
43 kApm = Info.Units.MagneticFieldStrength;
44 Hz = Info.Units.Frequency;
45 mV = Info.Units.SensorOutputVoltage;
46
47 % get dataset infos and format strings to place in figures
48 % subtitle string for all figures
49 infoStr = join([Info.SensorManufacturer, ...
50                 Info.Sensor, Info.SensorTechnology, ...
51                 Info.SensorType, "Sensor Characterization Dataset."]);
52 dateStr = join(["Created on", Info.Created, "by", 'Thorben
53                 Schuethe', ...
54                 "and updated on", Info.Edited, "by", Info.Editor +
55                 "."]);
```

56 % load characterization data

```
56 Vcos = Data.SensorOutput.CosinusBridge;
57 Vsin = Data.SensorOutput.SinusBridge;
58 gain = Info.SensorOutput.BridgeGain;
```

59

```
60 % clear dataset all loaded
61 clear Data Info;
```

62

```
63 disp('Info:');
64 disp([infoStr; dateStr]);
```

65

```
66 % reconstruct magnetic stimulus and reduce the view for
67 % example plot by 10
68 %
69 %%%%%%%%%%%%%%
```

```

68 %
69 % number of periods reduced by factor 10
70 reduced = 10;
71 nPeriods = fc / fm / reduced;
72 % number of samples for good looking 40 times nPeriods
73 nSamples = nPeriods * 400;
74 % half number of samples
75 nHalf = round(nSamples / 2);
76 % generate angle base
77 phi = linspace(0, nPeriods * 2 * pi, nSamples);
78 % calculate modulated amplitude, triang returns a column
    vector, transpose
79 Hmag = Hmax * triang(nSamples)';
80 % calculate Hx and Hy stimulus
81 Hx = Hmag .* cos(phi);
82 Hy = Hmag .* sin(phi);
83 % index for rising and falling stimulus
84 idxR = 1:nHalf;
85 idxF = nHalf:nSamples;
86 % find absolute min and max values in bridge outputs for
    uniform colormap
87 A = cat(3, Vcos.Rise, Vcos.Fall, Vcos.All, Vcos.Diff, Vsins.
    Rise, ...
    Vsins.Fall, Vsins.All, Vsins.Diff);
88 Vmax = max(A, [], 'all');
89 Vmin = min(A, [], 'all');
90 clear A;
91
92 %
93 % figure 1 magnetic stimulus
94 %

```

```

95 %
96 fig1 = figure('Name', 'Magnetic Stimulus');
97 tiledlayout(fig1, 2, 2);
98
99 % title and description
100 disp("Title: Magnetic Stimulus Reconstructed H_x-/ H_y-
101 Stimulus" + ...
102 "in Reduced View");
103 disp("Description: Stimulus for characterization in H_x and
104 H_y in " + ...
105 "reduced period view by factor 10");
106 disp(["a) Triangle modulated cosine carrier for H_x
107 stimulus."; ...
108 "b) Triangle modulated sine carrier for H_x stimulus
109 ." ; ...
110 "c) Modulation trajectory for rising stimulus"; ...
111 "d) Modulation trajectory for falling stimulus"]);
112
113 % Hx stimulus
114 nexttile;
115 p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hx(idxR), phi(
116 idxF), Hx(idxF));
117 set(p, {'Color'}, {'k', 'k', 'b', 'r'});
118 legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location
119 ', 'NorthEast');
120 xticks((0:0.25*pi:2*pi) * nPeriods);
121 xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi
122 $', ...
123 '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
124 xlim([0 phi(end)]);
125 ylim([Hmin Hmax]);
126 xlabel('$\phi$ in rad, Periode $\times 10$');
127 ylabel(sprintf('$H_x(\phi)$ in %', kApm));

```

```

121 title(sprintf('a) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm,
122 Hz, fc, Hz));
123
124 % Hy stimulus
125 nexttile;
126 p = plot(phi, Hmag, phi, -Hmag, phi(idxR), Hy(idxR), phi(
127 idxF), Hy(idxF));
128 set(p, {'Color'}, {'k', 'k', 'b', 'r'}');
129 legend([p(1) p(3) p(4)], {'mod', 'rise', 'fall'}, 'Location
130 ', 'NorthEast');
131 xticks((0:0.25*pi:2*pi) * nPeriods);
132 xticklabels({'$0$', '$8\pi$', '$16\pi$', '$24\pi$', '$32\pi$',
133 '$40\pi$', '$48\pi$', '$56\pi$', '$64\pi$'});
134 xlim([0 phi(end)]);
135 ylim([Hmin Hmax]);
136 xlabel('$\phi$ in rad, Periode $\times 10$');
137 ylabel(sprintf('$H_y(\phi)$ in %s', kApm));
138 title(sprintf('b) $f_m = %1.2f$ %s, $f_c = %1.2f$ %s', fm,
139 Hz, fc, Hz));
140
141 % polar for rising modulation
142 nexttile;
143 polarplot(phi(idxR), Hmag(idxR), 'b');
144 p = gca;
145 p.ThetaAxisUnits = 'radians';
146 title('c) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $0 < \phi < 320\pi$');
147
148 % polar for rising modulation
149 nexttile;
150 polarplot(phi(idxF), Hmag(idxF), 'r');
151 p = gca;
152 p.ThetaAxisUnits = 'radians';
153 title('d) $|\vec{H}(\phi)| \cdot e^{j\phi}$, $320 < \phi < 640\pi$');

```

```
150
151 % figure 2 cosinus bridge outputs
152 %
153 %
154 fig2 = figure('Name', 'Cosine and Sine Bridge', 'Position',
155 [0 0 33 30]);
156 tiledlayout(fig2, 2, 2);
157 %
158 % title and description
159 disp("Title: Cosine and Sine Bridge. Measured Bridge
160 Outputs" + ...
161 " of Corresponding H_x-/ H_y-Amplitudes");
162 disp("Description: " + sprintf("H_x, H_y in %s, %d Steps in
163 %.4f %s", ...
164 kApm, Hsteps, Hres, kApm));
165 disp(["a) Cosine Bridge Rising H-Amplitudes"; ...
166 "b) Cosine Bridge Falling H-Amplitudes"; ...
167 "c) Sine Bridge Rising H-Amplitudes"; ...
168 "d) Sine Bridge Falling H-Amplitudes"]);
169 %
170 % cosinus bridge recorded during rising stimulus
171 nexttile;
172 im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Rise);
173 set(gca, 'YDir', 'normal');
174 set(im, 'AlphaData', ~isnan(Vcos.Rise));
175 caxis([Vmin, Vmax]);
176 xlim([Hmin Hmax]);
177 ylim([Hmin Hmax]);
178 axis square xy;
```

```
179 xlabel('$H_x$ in kA/m');
180 ylabel('$H_y$ in kA/m');
181 title('a) $V_{\cos}(H_x, H_y)$');
182 yticks([-20 -10 0 10 20]);
183 xticks([-20 -10 0 10 20]);
184
185 % cosinus bridge recorded during falling stimulus
186 nexttile;
187 im = imagesc([Hmin Hmax], [Hmin Hmax], Vcos.Fall);
188 set(gca, 'YDir', 'normal');
189 set(im, 'AlphaData', ~isnan(Vcos.Fall));
190 caxis([Vmin, Vmax]);
191 xlim([Hmin Hmax]);
192 ylim([Hmin Hmax]);
193 axis square xy;
194 xlabel('$H_x$ in kA/m');
195 ylabel('$H_y$ in kA/m');
196 title('b) $V_{\cos}(H_x, H_y)$');
197 yticks([-20 -10 0 10 20]);
198 xticks([-20 -10 0 10 20]);
199
200 % sinus bridge recorded during rising stimulus
201 nexttile;
202 im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Rise);
203 set(gca, 'YDir', 'normal');
204 set(im, 'AlphaData', ~isnan(Vsin.Rise));
205 caxis([Vmin, Vmax]);
206 xlim([Hmin Hmax]);
207 ylim([Hmin Hmax]);
208 axis square xy;
209 xlabel('$H_x$ in kA/m');
210 ylabel('$H_y$ in kA/m');
211 title('c) $V_{\sin}(H_x, H_y)$');
212 yticks([-20 -10 0 10 20]);
213 xticks([-20 -10 0 10 20]);
214
```

```
215 % sinus bridge recorded during falling stimulus
216 nexttile;
217 im = imagesc([Hmin Hmax], [Hmin Hmax], Vsin.Fall);
218 set(gca, 'YDir', 'normal');
219 set(im, 'AlphaData', ~isnan(Vsin.Fall));
220 caxis([Vmin, Vmax]);
221 xlim([Hmin Hmax]);
222 ylim([Hmin Hmax]);
223 axis square xy;
224 xlabel('$_H_x$ in kA/m');
225 ylabel('$_H_y$ in kA/m');
226 title('d) $V_{\sin}(H_x, H_y)$');
227 yticks([-20 -10 0 10 20]);
228 xticks([-20 -10 0 10 20]);
229
230 % add colorbar and place it overall plots
231 cb = colorbar;
232 cb.Layout.Tile = 'east';
233 cb.Label.String = sprintf(...,
234     '$V(H_x, H_y)$ in %s, Gain $ = %.1f$', mV, gain);
235 cb.Label.Interpreter = 'latex';
236 cb.TickLabelInterpreter = 'latex';
237 cb.Label.FontSize = 24;
238
239 % yesno = input('Save? [y/n]: ', 's');
240 % if strcmp(yesno, 'y')
241 %     % save results of figure 1
242 %     savefig(fig1, fig1Path);
243 %     print(fig1, fig1Path, '-dsvg');
244 %     print(fig1, fig1Path, '-depsc', '-tiff', '-loose');
245 %     print(fig1, fig1Path, '-dpdf', '-loose', '-fillpage')
246 ;
247 %
248 %     % save results of figure 2
249 %     savefig(fig2, fig2Path);
250 %     print(fig2, fig2Path, '-dsvg');
```

```
250 %      print(fig2, fig2Path, '-depsc', '-tiff', '-loose');
251 %      print(fig2, fig2Path, '-dpdf', '-loose', '-fillpage')
252 %
253 %    end
254 %    close(fig1)
255 %    close(fig2)
256 end
```

#### E.4.3.3.5 plotKMZ60CharField

## plotKMZ60CharField

Explore NXP KMZ60 characterization field.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotKMZ60CharField()
```

### Description

**plotKMZ60CharField()** explore characterization field of KMZ60 sensor.

### Examples

```
1 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.

```
1 function plotKMZ60CharField()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
7         %       close all;
8         catch ME
9             rethrow(ME)
10    end
```

```
11 % load needed data from dataset in to local variables for
12 % better handling %%
13 %
14 % get from user which field to investigate and limits for
15 % plateau
16 fields = Info.SensorOutput.CosinusBridge.Determination;
17 nFields = length(fields);
18 fprintf('Choose 1 of %d fields ...\\n', nFields);
19 for i = 1:nFields
20     fprintf('%s\\t:\\t(%d)\\n', fields{i}, i);
21 end
22
23 iField = 1; % input('Choice: ');
24 field = fields{iField};
25 pl = 20; % input('Plateau limit in kA/m: ');
26
27 Vcos = Data.SensorOutput.CosinusBridge.(field);
28 Vsing = Data.SensorOutput.SinusBridge.(field);
29 gain = Info.SensorOutput.BridgeGain;
30 HxScale = Data.MagneticField.hx;
31 HyScale = Data.MagneticField.hy;
32 Hmin = Info.MagneticField.MinAmplitude;
33 Hmax = Info.MagneticField.MaxAmplitude;
34
35 % get unit strings from
36 kApm = Info.Units.MagneticFieldStrength;
37 mV = Info.Units.SensorOutputVoltage;
38
39 % get dataset infos and format strings to place in figures
40 % subtitle string for all figures
41 infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
```

```
42 dateStr = join(["Created on", Info.Created, "by", 'Thorben
43 Sch\"uthe', ...
44 "and updated on", Info.Edited, "by", Info.Editor +
45 "."]);  
46  
47 % clear dataset all loaded  
48 clear Data Info;  
49  
50 % figure save path for different formats  
51 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
52 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
53 % define slices and limits to plot  
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
55 Hslice = [128 154 180 205]; % hit ca. 0, 5, 10, 15 kA/m  
56 Hlims = [-pl pl];  
57 mVpVlims = [-8 8];  
58  
59 % create figure for plots  
60 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
61 fig = figure('Name', 'Char Field', 'OuterPosition', [0 0 35
62 30]);  
63 tiledlayout(fig, 2, 2);  
64  
65 % title and description  
disp('Info:');
```

```
66 disp([infoStr; dateStr]);
67 fprintf('Title: KMZ60 Characterization Field - %s\n', field
68 );
69 disp('Description:');
70 disp(["a) Cosine Bridge Characteristic"; ...
71     "b) Transfer slices for different const. H_y of Vcos
72         ";" ...
73     "c) Sine Bridge Characteristic"; ...
74     "d) Transfer slices for different const. H_x of Vsin
75         "]);
76
77 % set colormap
78 colormap('jet');
79
80 % cosinus bridge
81 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
82
83 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
84
85 nexttile(1);
86 im = imagesc(HxScale, HyScale, Vcos);
87 set(gca, 'YDir', 'normal');
88 set(im, 'AlphaData', ~isnan(Vcos));
89 xticks(-20:10:20);
90 yticks(-20:10:20);
91 axis square xy;
92
93 % plot lines for slice to investigate
94 hold on;
95 for i = Hslice
96     yline(HyScale(i), 'k:', 'LineWidth', 3.5);
97 end
98 hold off;
99
100 xlabel(sprintf('$H_x$ in %s', kApm));
```

```

95 ylabel(sprintf('$H_y$ in %s', kApm));
96 title(sprintf('a) $V_{cos}(H_x,H_y)$, Gain $ = %.1f$', gain
97      ));
98
99 cb = colorbar;
100 cb.Label.String = sprintf('$V_{cos}$ in %s', mV);
101 cb.Label.Interpreter = 'latex';
102 cb.TickLabelInterpreter = 'latex';
103 cb.Label.FontSize = 20;
104
105 % cosinus bridge slices
106 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
107 %
108 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
109
110 nexttile(2);
111 % slices
112 p = plot(HxScale, Vcos(Hslice,:));
113
114 % plateau limits
115 if pl > 0
116     hold on;
117     xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
118     xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
119     hold off;
120 end
121
122 legend(p, {'$H_y \approx 0$ kA/m', ...
123             '$H_y \approx 5$ kA/m', ...
124             '$H_y \approx 10$ kA/m', ...
125             '$H_y \approx 15$ kA/m'}, ...
126             'Location', 'SouthEast');
127 xlabel(sprintf('$H_x$ in %s', kApm));
128 title('b) $V_{cos}(H_x,H_y)$, $H_y = $ const.');
129 ylim(mVpVlims);
130 xlim([Hmin Hmax])

```

```

127
128 % sinus bridge
129 %
130 nexttile(3);
131 im = imagesc(HxScale, HyScale, Vsin);
132 set(gca, 'YDir', 'normal');
133 set(im, 'AlphaData', ~isnan(Vsin));
134 xticks(-20:10:20);
135 yticks(-20:10:20);
136 axis square xy;
137
138 % plot lines for slice to investigate
139 hold on;
140 for i = Hslice
141     xline(HxScale(i), 'k:', 'LineWidth', 3.5);
142 end
143 hold off;
144
145 xlabel(sprintf('$H_x$ in %s', kApm));
146 ylabel(sprintf('$H_y$ in %s', kApm));
147 title(sprintf('c) $V_{sin}(H_x,H_y)$, Gain $ = %.1f$', gain
148 ));
149 cb = colorbar;
150 cb.Label.String = sprintf('$V_{sin}$ in %s', mV);
151 cb.Label.Interpreter = 'latex';
152 cb.TickLabelInterpreter = 'latex';
153 cb.Label.FontSize = 20;
154
155 % sinus bridge slices
156

```

```

156 %
157 nexttile(4);
158 % slices
159 p = plot(HxScale, Vsin(:,Hslice));
160
161 % plateau limits
162 if pl > 0
163     hold on;
164     xline(Hlims(1), 'k-.', 'LineWidth', 2.5);
165     xline(Hlims(2), 'k-.', 'LineWidth', 2.5);
166     hold off;
167 end
168
169 legend(p, {'$H_x \approx 0\$ kA/m', ...
170             '$H_x \approx 5\$ kA/m', ...
171             '$H_x \approx 10\$ kA/m', ...
172             '$H_x \approx 15\$ kA/m'}, ...
173             'Location', 'SouthEast');
174 xlabel(sprintf('$H_y\$ in %s', kApm));
175 title('d) $V_{\sin}(H_x,H_y)\$, $H_x = \$ \text{const.}');
176 ylim(mVpVlims);
177 xlim([Hmin Hmax])
178
179 % save results of figure
180 %
181 % yesno = input('Save? [y/n]: ', 's');
182 % if strcmp(yesno, 'y')
183 %     savefig(fig, fPath);
184 %     print(fig, fPath, '-dsvg');
185 %     print(fig, fPath, '-depsc', '-tiff', '-loose');
186 %     print(fig, fPath, '-dpdf', '-loose', '-fillpage');

```

```
187 %      end  
188 %      close(fig)  
189 end
```

#### E.4.3.3.6 plotKMZ60TransferCurves

## plotKMZ60TransferCurves

Plot NXP KMZ60 characterization field transfer curves.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 plotKMZ60TransferCurves()
```

### Description

**plotKMZ60TransferCurves()** plot characterization field of KMZ 60 sensor.

### Examples

```
1 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.

```
1 function plotKMZ60TransferCurves()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables');
6         load(PathVariables.kmz60DatasetPath, 'Data', 'Info');
7         % close all;
8         catch ME
9             rethrow(ME)
10    end
```

```
11 % load needed data from dataset in to local variables for
12 % better handling %%
13 %
14 % get from user which field to investigate and limits for
15 % plateau
16 fields = Info.SensorOutput.CosinusBridge.Determination;
17 nFields = length(fields);
18 fprintf('Choose 1 of %d fields ...\\n', nFields);
19 for i = 1:nFields
20     fprintf('%s\\t:\\t(%d)\\n', fields{i}, i);
21 end
22
23 iField = 1; % input('Choice: ');
24 field = fields{iField};
25 pl = 20; % input('Plateau limit in kA/m: ');
26
27 Vcos = Data.SensorOutput.CosinusBridge.(field);
28 Vsin = Data.SensorOutput.SinusBridge.(field);
29 gain = Info.SensorOutput.BridgeGain;
30 HxScale = Data.MagneticField.hx;
31 HyScale = Data.MagneticField.hy;
32 Hmin = Info.MagneticField.MinAmplitude;
33 Hmax = Info.MagneticField.MaxAmplitude;
34
35 % get unit strings from
36 kApm = Info.Units.MagneticFieldStrength;
37 mV = Info.Units.SensorOutputVoltage;
38
39 % get dataset infos and format strings to place in figures
40 % subtitle string for all figures
41 infoStr = join([Info.SensorManufacturer, ...
    Info.Sensor, Info.SensorTechnology, ...
    Info.SensorType, "Sensor Characterization Dataset."]);
```

```
42 dateStr = join(["Created on", Info.Created, "by", 'Thorben
43 Sch\"uthe', ...
44 "and updated on", Info.Edited, "by", Info.Editor +
45 "."]); 
46
47 % clear dataset all loaded
48 clear Data Info;
49
50 % figure save path for different formats
51 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
52 %
53 % define slices and limits to plot
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55 Hslice = 128; % hit ca. 0 kA/m
56 Hlims = [-pl pl];
57 mVpVlims = [-8 8];
58
59 % create figure for plots
60 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
61 fig = figure('Name', 'Transfer Curves', 'OuterPosition', [0
62 0 33 30]);
63 tiledlayout(fig, 2, 2);
64
65 disp('Info:');
```

```
66 disp([infoStr; dateStr]);
67 disp('Title:');
68 fprintf('KMZ 60 Transfer Curves: %s\n', field);
69 disp(["a) Cosine Bridge Characteristic"; ...
70 "b) Sine Bridge Characteristic"; ...
71 "c) Transfer Curves for const. H_x = H_y = 0"]);
```

72

```
73 % set colormap
74 colormap('jet');
```

75

```
76 % cosinus bridge
77 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

78 nexttile(1);
79 im = imagesc(HxScale, HyScale, Vcos);
80 set(gca, 'YDir', 'normal');
81 set(im, 'AlphaData', ~isnan(Vcos));
82 xticks(-20:10:20);
83 yticks(-20:10:20);
84 axis square xy;

85

```
86 % plot lines for slice to investigate
87 hold on;
88 yline(HyScale(Hslice), 'k:', 'LineWidth', 3.5);
89 plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth',
90 3.5);
91 hold off;
```

92 xlabel(sprintf('\$H\_x\$ in %s', kApm));
93 ylabel(sprintf('\$H\_y\$ in %s', kApm));
94 title(sprintf('a) \$V\_{\cos}(H\_x, H\_y)\$, Gain \$ = %.1f\$', gain
95 ));

```

96    % sinus bridge
97
98    %
99
100   nexttile(2);
101   im = imagesc(HxScale, HyScale, Vsin);
102   set(gca, 'YDir', 'normal');
103   set(im, 'AlphaData', ~isnan(Vsin));
104   xticks(-20:10:20);
105   yticks(-20:10:20);
106   axis square xy;
107
108   % plot lines for slice to investigate
109   hold on;
110   xline(HxScale(Hslice), 'k:', 'LineWidth', 3.5);
111   plot(pl*cosd(0:360), pl*sind(0:360), 'k-.', 'LineWidth',
112       3.5);
113   hold off;
114
115   xlabel(sprintf('$H_x$ in %s', kApm));
116   ylabel(sprintf('$H_y$ in %s', kApm));
117   title(sprintf('b) $V_{sin}(H_x,H_y)$, Gain $ = %.1f$', gain
118         ));
119
120   % colorbar for both
121
122   cb = colorbar;
123   cb.Label.String = sprintf('$V_{out}$ in %s', mV);
124   cb.TickLabelInterpreter = 'latex';
125   cb.Label.Interpreter = 'latex';

```

```

123 cb.Label.FontSize = 20;
124
125 % cosinus bridge slices
126 %
127 nexttile([1 2]);
128 % slices
129 p = plot(HxScale, Vcos(Hslice,:), HyScale, Vsing(:, Hslice)
130   ');
131 % plateau limits
132 if pl > 0
133   hold on;
134   xline(Hlims(1), 'k-.', 'LineWidth', 3.5);
135   xline(Hlims(2), 'k-.', 'LineWidth', 3.5);
136   hold off;
137 end
138
139 legend(p, {sprintf('$V_{cos}(H_x,H_y)$, $H_y \\\approx 0$ %s
140   ', kApm), ...
141   sprintf('$V_{sin}(H_x,H_y)$, $H_x \\\approx 0$ %s
142   ', kApm) }, ...
143   'Location', 'SouthEast');
144 ylabel(sprintf('$V_{out}$ in %s', mV));
145 xlabel(sprintf('$H$ in %s', kApm));
146 title('c) Cosine and Sine Transfer Curves');
147 ylim(mVpVlims);
148 xlim([Hmin Hmax])
149 %

```

```
150 %     yesno = input('Save? [y/n]: ', 's');
151 %     if strcmp(yesno, 'y')
152 %         savefig(fig, fPath);
153 %         print(fig, fPath, '-dsvg');
154 %         print(fig, fPath, '-depsc', '-tiff', '-loose');
155 %         print(fig, fPath, '-dpdf', '-loose', '-fillpage');
156 %     end
157 %     close(fig)
158 %
159 end
```

#### E.4.3.3.7 plotDipoleMagnet

## plotDipoleMagnet

Plot dipole magnet which approximate a spherical magnet in its far field.

### Contents

- Syntax
- Description
- Examples
- Input Arguments
- Output Arguments
- Requirements
- See Also

### Syntax

```
1 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

```
1 plotDipoleMagnet();
```

## Input Arguments

**None**

## Output Arguments

**None**

## Requirements

- Other m-files: generateDipoleRotationMoments.m, computeDipoleH0Norm.m, computeDipoleHField
- 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.

```
1 function plotDipoleMagnet()
2     try
3         % load dataset path and dataset content into function
4         % workspace
5         load('config.mat', 'PathVariables', 'DipoleOptions');
6         % close all;
7         catch ME
8             rethrow(ME)
9     end
```

```
9
10    % figure save path for different formats
11
12    %
13    figFilename = 'dipole_magnet';
14    figPath = fullfile(PathVariables.saveImagePath,
15                      figFilename);
16
17    % load needed data from dataset in to local variables for
18    % better handling
19
20    %
21    % Radius in mm of magnetic sphere in which the magnetic
22    % dipole is centered.
23    % So it can be seen as z-offset to the sensor array.
24    rsp = DipoleOptions.sphereRadius;
25
26    % H-field magnitude to multiply of generated and relative
27    % normed dipole
28    Hmag = DipoleOptions.H0mag;
29
30    % Distance in zero position of the spherical magnet in
31    % which is imprinted
32    z0 = DipoleOptions.z0;
33
34    % Magnetic moment magnitude attach rotation to the dipole
35    % field
36    m0 = DipoleOptions.M0mag;
```

```
31
32 % clear dataset all loaded
33 clear DipoleOptions;
34
35 % set construction dipole magnet, all length in mm and
36 % areas mm^2
37 %
38 % number of samples for good looking
39 nSamples = 501;
40 % slice in view for quiver, every 25th point
41 slice = 25:25:nSamples-25;
42 % grid edge of meshgrid, square grid
43 xz = 15;
44 % y layer in coordinate system
45 y = 0;
46 % orientat of magnet along z axes
47 pz = pi/2:0.01:3*pi/2;
48 % distances magnet surface to display in plot
49 zd = -rsp:-z0:-xz;
50 xd = zeros(1, length(zd));
51 % scale grid to simulate
52 x = linspace(-xz, xz, nSamples);
53 z = linspace(xz, -xz, nSamples);
54 [X, Z, Y] = meshgrid(x, z, y);
55
56 % compute dipole and fetch to far field to approximate a
57 % sperical magnet
58 %
59 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```

58 %
59 % generate dipole moment for 0
60 m = generateDipoleRotationMoments(m0, 1);
61 % compute H-field norm factor imprieng H magnitude on
62 % dipole, rest position
63 H0norm = computeDipoleH0Norm(Hmag, m, [0; 0 ;-(z0 + rsp)]);
64 % compute dipole H-field for rest position in y = 0 layer
65 H = computeDipoleHField(X, Y, Z, m, H0norm);
66 % calculate magnitudes for each point in the grid
67 Habs = reshape(sqrt(sum(H.^2, 1)), nSamples, nSamples);
68 % split H-field in componets and reshape to meshgrid
69 Hx = reshape(H(1,:), nSamples, nSamples) ./ Habs;
70 Hy = reshape(H(2,:), nSamples, nSamples) ./ Habs;
71 Hz = reshape(H(3,:), nSamples, nSamples) ./ Habs;
72 % exculde value within the spherical magnet, < rsp
73 innerField = X.^2 + Z.^2 <= rsp.^2;
74 Habs(innerField) = NaN;
75 % find relevant magnitudes at anounced distances
76 Hd = interp2(X, Z, Habs, xd, zd, 'nearest', NaN);
77 %
78 % figure dipole magnet
79 %
80 %
81 fig = figure('Name', 'Dipole Magnet');
82 %
83 % plot magnitude as colormap
84 imagesc(x, z, log10(Habs), 'AlphaData', 1);
85 set(gca, 'YDir', 'normal');
86 colormap('jet');

```

```

87     shading flat;
88
89 % set colorbar to log10 scaling of map
90 cb = colorbar;
91 cb.Label.String = '$\log_{10}(|H|)$ in kA/m';
92 cb.TickLabelInterpreter = 'latex';
93 cb.Label.Interpreter = 'latex';
94 cb.Label.FontSize = 24;
95
96 hold on;
97 grid on;
98
99 % plot field lines
100 st = streamslice(X, Z, Hx, Hz, 'arrows', 'cubic');
101 set(st, 'Color', 'k');
102
103
104 % plot magnet with north and south pole
105 rectangle('Position', [-rsp -rsp 2*rsp 2*rsp], ...
106             'Curvature', [1 1], 'LineWidth', 3.5);
107 semicrc = rsp.*[cos(pz); sin(pz)];
108 patch(semicrc(1,:), semicrc(2,:),'r');
109 patch(-semicrc(1,:), -semicrc(2,:),'g');
110 text(-1.25, 0, '\textbf{N}');
111 text(0.5, 0, '\textbf{S}');
112
113 % additional figure text and lines
114 text(-(xz-1), -(xz-1), ...
115         sprintf('$\\mathbf{Y} = %.1f$ \\textbf{mm}', y), 'Color'
116         , 'w');
117
118 % distance scale in -z direction for x =0, distance from
119 % magnet surface
120 line(xd, zd, 'Marker', '_', 'LineStyle', '-', 'LineWidth',
121       3.5, ...
122       'Color', 'w');

```

```
120
121 % place text along marker
122 for i = 2:length(zd)-1
123     markstr = "\textbf{\$\\mathbf{\%d}\$ mm, \$\\mathbf{\%.1f}\$ kA/m\$}";
124     mark = sprintf(markstr, abs(zd(i))-rsp, Hd(i));
125     text(0.5, zd(i), mark, 'Color', 'w');
126 end
127
128 % limits ticks and labels
129 xlim([-xz xz]);
130 ylim([-xz xz]);
131 xticks(-xz:xz);
132 yticks(-xz:xz);
133 labels = string(xticks);
134 labels(1:2:end) = "";
135 xticklabels(labels)
136 yticklabels(labels)
137 xlabel('$X$ in mm');
138 ylabel('$Z$ in mm');
139
140 % axis shape set
141 axis equal;
142 axis tight;
143 grid off;
144
145 % title and description
146 disp('Title: Approximated Spherical Magnet with Dipole Far
147 Field');
148 disp("Description:");
149 fprintf("Sphere whith imprinted H-field magnitude of %.1f
150 kA/m\n", Hmag); ...
151 fprintf("at distance d = %.1f mm with d_z = |z| - r\n", z0)
152 ;
153 fprintf("and sphere radius r = %.1f mm\n", rsp);
```

```
152 % save results of figure
153 %     yesno = input('Save? [y/n]: ', 's');
154 %     if strcmp(yesno, 'y')
155 %         savefig(fig, figPath);
156 %         print(fig, figPath, '-dsvg');
157 %         print(fig, figPath, '-depsc', '-tiff', '-loose');
158 %         print(fig, figPath, '-dpdf', '-loose', '-fillpage');
159 %     end
160 %     close(fig)
161 end
```

#### E.4.3.3.8 plotSimulationDataset

## 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.

### Contents

- Syntax
- Description
- Examples
- Input Argurments
- Output Argurments
- Requirements
- See Also

### Syntax

```
1 plotSimulationDataset()
```

### Description

**plotSimulationDataset()** plot training or test dataset which are loacated 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

```
1 plotSimulationDataset()
```

## Input Argurments

**None**

## Output Argurments

**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.

```
1 function plotSimulationDataset()
2 % scan for datasets and load needed configurations
3 %%%%%%
4 %
5 %%%%%%
6 %
7 try
8     disp('Plot simulation dataset ...');
9     close all;
10    % load path variables
11    load('config.mat', 'PathVariables');
12    % scan for datasets
13    TrainingDatasets = dir(fullfile(PathVariables.
14        trainingDataPath, ...
15        'Training_*.mat'));
16    TestDatasets = dir(fullfile(PathVariables.testDataPath,
17        'Test_*.mat'));
18    allDatasets = [TrainingDatasets; TestDatasets];
19    % check if files available
20    if isempty(allDatasets)
21        error('No training or test datasets found.');
22    end
23    catch ME
24        rethrow(ME)
25    end
26
27    % display available datasets to user, decide which to plot
28    %%%%%%
29    %
30    %%%%%%
31    %
32    %%%%%%
33    %
34    % number of datasets
35    nDatasets = length(allDatasets);
36    fprintf('Found %d datasets:\n', nDatasets)
```

```
28     for i = 1:nDatasets
29         fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
30     end
31 % get numeric user input to indicate which dataset to plot
32 iDataset = input('Type number to choose dataset to plot to:
33     ');
34 %
35 % load dataset and ask user which one and how many angles
36 %
37 try
38     ds = load(fullfile(allDatasets(iDataset).folder, ...
39                 allDatasets(iDataset).name));
40 % check how many angles in dataset and let user decide
41 % how many to
42 % render in plot
43 fprintf('Detect %d angles in dataset ... \n', ...
44         ds.Info.UseOptions.nAngles);
45 nSubAngles = input('How many angles to you wish to plot
46     : ');
47 % nSubAngles = 120;
48 % indices for data to plot, get sample distance for
49 % even distance
50 sampleDistance = length(downsamp(ds.Data.angles,
51                             nSubAngles));
52 % get subset of angles
53 subAngles = downsample(ds.Data.angles, sampleDistance);
54 nSubAngles = length(subAngles); % just ensure
55 % get indices for subset data
56 indices = find(ismember(ds.Data.angles, subAngles));
57 catch ME
58     rethrow(ME)
59 end
```

```
56
57     % create dataset figure for a subset or all angle
58     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
59
60     fig = figure('Name', 'Sensor Array', ...
61                 'NumberTitle', 'off', ...
62                 'WindowSize', 'normal', ...
63                 'MenuBar', 'none', ...
64                 'ToolBar', 'none', ...
65                 'Units', 'centimeters', ...
66                 'OuterPosition', [0 0 30 30], ...
67                 'PaperType', 'a4', ...
68                 'PaperUnits', 'centimeters', ...
69                 'PaperOrientation', 'landscape', ...
70                 'PaperPositionMode', 'auto', ...
71                 'DoubleBuffer', 'on', ...
72                 'RendererMode', 'manual', ...
73                 'Renderer', 'painters');

74     tdl = tiledlayout(fig, 2, 2, ...
75                       'Padding', 'normal', ...
76                       'TileSpacing', 'compact');

77
78
79     title(tdl, 'Sensor Array Simulation', ...
80           'FontWeight', 'normal', ...
81           'FontSize', 18, ...
82           'FontName', 'Times', ...
83           'Interpreter', 'latex');

84
85     subline1 = "Sensor Array (%s) of $%d\\times%d$ sensors, an
86     edge" + ...
87         " length of $%.1f$ mm, a rel. pos. to magnet surface of
88         ";
```

```
87    subline2 = " $(%.1f, %.1f, -(%.1f))$ in mm, a magnet" + ...
88        " tilt of $%.1f^\circ$, a sphere radius of $%.1f$ mm,
89        a imprinted";
90    subline3 = "field strength of $%.1f$ kA/m at $%.1f$ mm" +
91        ...
92        " from sphere surface in z-axis, $%d$ rotation angles
93        with a ";
94    subline4 = "step width of $%.1f^\circ$ and a resolution" +
95        ...
96        " of $%.1f^\circ$. Visualized is a subset of $%d$
97        angles in ";
98    subline5 = "sample distance of $%d$ angles. Based on %s" +
99        ...
100        " characterization reference %s.";
101 sub = [sprintf(subline1, ...
102                 ds.Info.SensorArrayOptions.geometry, ...
103                 ds.Info.SensorArrayOptions.dimension, ...
104                 ds.Info.SensorArrayOptions.dimension, ...
105                 ds.Info.SensorArrayOptions.edge); ...
106     sprintf(subline2, ...
107                 ds.Info.UseOptions.xPos, ...
108                 ds.Info.UseOptions.yPos, ...
109                 ds.Info.UseOptions.zPos, ...
110                 ds.Info.UseOptions.tilt, ...
111                 ds.Info.DipoleOptions.sphereRadius); ...
112     sprintf(subline3, ...
113                 ds.Info.DipoleOptions.H0mag, ...
114                 ds.Info.DipoleOptions.z0, ...
115                 ds.Info.UseOptions.nAngles); ...
116     sprintf(subline4, ...
117                 ds.Data.angleStep, ...
118                 ds.Info.UseOptions.angleRes, ...
119                 nSubAngles)
120     sprintf(subline5, ...
121                 sampleDistance, ...
122                 ds.Info.CharData, ...
```

```
117         ds.Info.UseOptions.BridgeReference) ];
```

```
118
```

```
119     subtitle(tdl, sub, ...
```

```
120         'FontWeight', 'normal', ...
```

```
121         'FontSize', 14, ...
```

```
122         'FontName', 'Times', ...
```

```
123         'Interpreter', 'latex');
```

```
124
```

```
125 % get subset of needed data to plot, only one load
```

```
126 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
127 %
```

```
128 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
129 N = ds.Info.SensorArrayOptions.dimension;
```

```
130 X = ds.Data.X;
```

```
131 Y = ds.Data.Y;
```

```
132 Z = ds.Data.Z;
```

```
133
```

```
134 % calc limits of plot 1
```

```
135 maxX = ds.Info.UseOptions.xPos + ds.Info.SensorArrayOptions
```

```
136 .edge;
```

```
137 maxY = ds.Info.UseOptions.yPos + ds.Info.SensorArrayOptions
```

```
138 .edge;
```

```
139 minX = ds.Info.UseOptions.xPos - ds.Info.SensorArrayOptions
```

```
140 .edge;
```

```
141 minY = ds.Info.UseOptions.yPos - ds.Info.SensorArrayOptions
```

```
142 .edge;
```

```
143
```

```
144 % calculate colormap to identify scatter points
```

```
145 c=zeros(N,N,3);
```

```
146 for i = 1:N
```

```
147     for j = 1:N
```

```
148         c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
```

```
149     end
```

```
150 end
```

```
151 c = squeeze(reshape(c, N^2, 1, 3));
```

```
146
147     % load offset voltage to subtract from cosinus, sinus
148     % voltage
149     Voff = ds.Info.SensorArrayOptions.Voff;
150
151     % plot sensor grid in x and y coordinates and constant z
152     % layer %%%%%%%%%%%%%%
153
154     %% %%%%%%%%%%%%%%
155
156     ax1 = nexttile(1);
157     % plot each cooredinate in loop to create a special shading
158     % constant
159     % reliable to orientation for all matrice
160     hold on;
161     scatter(X(:, ), Y(:, ), [], c, 'filled', 'MarkerEdgeColor', 'k'
162             ,
163             'LineWidth', 0.8);
164
165     % axis shape and ticks
166     axis square xy;
167     axis tight;
168     grid on;
169     xlim([minX maxX]);
170     ylim([minY maxY]);
171
172     % text and labels
173     text(minX+0.2, minY+0.2, ...
174         sprintf('$Z = %.1f$ mm', z(1)), ...
175         'Color', 'k', ...
176         'FontSize', 16, ...
177         'FontName', 'Times', ...
178         'Interpreter', 'latex');
179
180     xlabel('$X$ in mm', ...
181         'FontWeight', 'normal', ...
```

```
176     'FontSize', 12, ...
177     'FontName', 'Times', ...
178     'Interpreter', 'latex');
179
180 ylabel('$Y$ in mm', ...
181     'FontWeight', 'normal', ...
182     'FontSize', 12, ...
183     'FontName', 'Times', ...
184     'Interpreter', 'latex');
185
186 title(sprintf('Sensor Array %d\\times%d', N, N), ...
187     'FontWeight', 'normal', ...
188     'FontSize', 12, ...
189     'FontName', 'Times', ...
190     'Interpreter', 'latex');
191
192 hold off;
193
194 % plot rotation angles in polar view
195 %
196 %%%%%%%%%%%%%%
197 %%%%%%%%%%%%%%
198 %%%%%%%%%%%%%%
199 %%%%%%%%%%%%%%
200 %%%%%%%%%%%%%%
201 %%%%%%%%%%%%%%
202 %%%%%%%%%%%%%%
203 %%%%%%%%%%%%%%
204 %%%%%%%%%%%%%%
205 %%%%%%%%%%%%%%
206 %%%%%%%%%%%%%%
207 %%%%%%%%%%%%%%
208 %%%%%%%%%%%%%%
```

```
196 nexttile(2);
197 % plot all angles grayed out
198 polarscatter(ds.Data.angles/180*pi, ...
199     ones(1, ds.Info.UseOptions.nAngles), ...
200     [], [0.8 0.8 0.8], 'filled');
201
202 % radius ticks and label
203 rticks(1);
204 rticklabels("");
205 hold on;
206
207 % plot subset of angles
208 % polarscatter(subAngles/180*pi, ones(1, nSubAngles), ...
```

```
209      %      'k', 'LineWidth', 0.8);
210      ax2 = gca;
211
212      % axis shape
213      axis tight;
214
215      % text an labels
216      % init first rotation step label
217      tA = text(2/3*pi, 1.5, ...
218          '$\theta$', ...
219          'Color', 'b', ...
220          'FontSize', 16, ...
221          'FontName', 'Times', ...
222          'Interpreter', 'latex');
223
224      title('Rotation around Z-Axis in Degree', ...
225          'FontWeight', 'normal', ...
226          'FontSize', 12, ...
227          'FontName', 'Times', ...
228          'Interpreter', 'latex');
229
230      hold off;
231
232      % Cosinus bridge outputs for rotation step
233      %%%
234      %%%
235      %%%
236      %%%
237      %%%
238      %%%
239      %%%
240      %%%
```

```
241 imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
242             VcosRef);
243 set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
244 set(gca, 'YDir', 'normal')
245
246 % axis shape and ticks
247 axis square xy;
248 axis tight;
249 yticks(xticks);
250 grid on;
251
252 % test and labels
253 xlabel('$H_x$ in kA/m', ...
254     'FontWeight', 'normal', ...
255     'FontSize', 12, ...
256     'FontName', 'Times', ...
257     'Interpreter', 'latex');
258
259 ylabel('$H_y$ in kA/m', ...
260     'FontWeight', 'normal', ...
261     'FontSize', 12, ...
262     'FontName', 'Times', ...
263     'Interpreter', 'latex');
264
265 title('$V_{\cos}(H_x, H_y)$', ...
266     'FontWeight', 'normal', ...
267     'FontSize', 12, ...
268     'FontName', 'Times', ...
269     'Interpreter', 'latex');
270
271 % add colorbar and place it
272 cb1 = colorbar;
273 cb1.Label.String = sprintf(
274     '$V_{\cos}(H_x, H_y)$ in V, $V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
```

```
274     ds.Info.SensorArrayOptions.Vcc, ds.Info.
275         SensorArrayOptions.Voff);
276 cb1.Label.Interpreter = 'latex';
277 cb1.Label.FontSize = 12;
278
279 hold off;
280
281 % Sinus bridge outputs for rotation step
282 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
283 %
284 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
285 ax4 = nexttile(4);
286 hold on;
287
288 % set colormap
289 colormap('gray');
290
291 % plot sinus reference, set NaN values to white color,
292 % orient Y to normal
293 imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
294 VsinRef);
295 set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
296 set(gca, 'YDir', 'normal')
297
298 % axis shape and ticks
299 axis square xy;
300 axis tight;
301 yticks(xticks);
302 grid on;
303
304 % test and labels
305 xlabel('$H_x$ in kA/m', ...
306 'FontSize', 12, ...
307 'FontName', 'Times', ...
```

```

304     'Interpreter', 'latex');
305
306 ylabel('$H_y$ in kA/m', ...
307     'FontWeight', 'normal', ...
308     'FontSize', 12, ...
309     'FontName', 'Times', ...
310     'Interpreter', 'latex');
311
312 title('$V_{sin}(H_x, H_y)$', ...
313     'FontWeight', 'normal', ...
314     'FontSize', 12, ...
315     'FontName', 'Times', ...
316     'Interpreter', 'latex');
317
318 % add colorbar and place it
319 cb2 = colorbar;
320 cb2.Label.String = sprintf(...%
321     '$V_{sin}(H_x, H_y)$ in V, $V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
322     ds.Info.SensorArrayOptions.Vcc, ds.Info.
323         SensorArrayOptions.Voff);
324 cb2.Label.Interpreter = 'latex';
325 cb2.Label.FontSize = 12;
326
327 hold off;
328
329 % zoom axes for scatter on cosinuns reference images
330 %%%
331 nexttile(3);
332 ax5 = axes('Position', [0.07 0.02 0.19 0.19], 'XColor', 'r',...
333     , 'YColor', 'r');
334 hold on;
335 axis square xy;

```

```
334     grid on;
335     hold off;
336
337     % draw everything prepared before start renewing frame wise
338     % and prepare for
339     % recording frames to video file
340     %%%%%%%%%%%%%%
341     %
342     % draw frame
343     drawnow;
344
345     % get file path and change extension
346     [~, fName, ~] = fileparts(ds.Info.filePath);
347     fPath = PathVariables.saveImagePath;
348
349     % string allows simple cat ops
350     VW = VideoWriter(fullfile(fPath, fName + ".avi"), ...
351                     "Uncompressed AVI");
352
353     % scale frame rate on 10 second movies, ensure at least 1
354     % fps
355     fr = floor(nSubAngles / 10) + 1;
356     VW.FrameRate = fr;
357
358
359     % open video file, ready to record frames
360     open(VW)
361
362
363     % loop through subset angle dataset and renew plots
364     %%%%%%%%%%%%%%
365     %
366     %%%%%%%%%%%%%%
367
368     for i = indices
```

```

362 % H load subset
363 Hx = ds.Data.Hx(:,:,i);
364 Hy = ds.Data.Hy(:,:,i);
365 % get min max
366 maxHx = max(Hx, [], 'all');
367 maxHy = max(Hy, [], 'all');
368 minHx = min(Hx, [], 'all');
369 minHy = min(Hy, [], 'all');
370 dHx = abs(maxHx - minHx);
371 dHy = abs(maxHy - minHy);
372
373 % load V subset
374 Vcos = ds.Data.Vcos(:,:,i) - Voff;
375 Vsint = ds.Data.Vsin(:,:,i) - Voff;
376 angle = ds.Data.angles(i);
377
378 % lock plots
379 hold(ax1, 'on');
380 hold(ax2, 'on');
381 hold(ax3, 'on');
382 hold(ax4, 'on');
383 hold(ax5, 'on');
384
385 % update plot 1
386 qH = quiver(ax1, X, Y, Hx, Hy, 0.5, 'b');
387 qV = quiver(ax1, X, Y, Vcos, Vsint, 0.5, 'r');
388 legend([qH qV], {'$quiver(H_x,H_y)$', ...
389 '$quiver(V_{\cos}-V_{\text{off}},V_{\sin}-V_{\text{off}})$'}, ...
390 'FontWeight', 'normal', ...
391 'FontSize', 9, ...
392 'FontName', 'Times', ...
393 'Interpreter', 'latex', ...
394 'Location', 'NorthEast');
395
396 % update plot 2
397 tA.String = sprintf('%.1f^\circ', angle);

```

```
398 pA = polarscatter(ax2, angle/180*pi, 1, 'b', 'filled',
400 % ...
401 % update plot 3 and 4
402 sC = scatter(ax3, Hx(:), Hy(:), 5, c, 'filled', ...
403 % 'MarkerEdgeColor', 'k', ...
404 % 'LineWidth', 0.8);
405 sS = scatter(ax4, Hx(:), Hy(:), 5, c, 'filled', ...
406 % 'MarkerEdgeColor', 'k', ...
407 % 'LineWidth', 0.8);
408
409 % calc position of scatter area frame and reframe
410 pos = [minHx - 0.3 * dHx, minHy - 0.3 * dHy, 1.6 * dHx,
411 1.6 * dHy];
412 rtc = rectangle(ax3, 'Position', pos, 'LineWidth', 1,
413 % ...
414 % 'EdgeColor', 'r');
415 rts = rectangle(ax4, 'Position', pos, 'LineWidth', 1,
416 % ...
417 % 'EdgeColor', 'r');
418
419 % update plot 5 (zoom)
420 sZ = scatter(ax5, Hx(:), Hy(:), [], c, 'filled', ...
421 % 'MarkerEdgeColor', 'k', ...
422 % 'LineWidth', 0.8);
423 xlim(ax5, [pos(1) maxHx + 0.3 * dHx])
424 ylim(ax5, [pos(2) maxHy + 0.3 * dHy])
425
426 % release plots
427 hold(ax1, 'off');
428 hold(ax2, 'off');
429 hold(ax3, 'off');
```

```
430      % draw frame
431      drawnow;
432
433      % record frame to file
434      frame = getframe(fig);
435      writeVideo(VW, frame);
436
437      % delete part of plots to renew for current angle,
438      % delete but last
439      if i ~= indices(end)
440          delete(qH);
441          delete(qV);
442          delete(pA);
443          delete(rtC);
444          delete(rts);
445          delete(sC);
446          delete(ss);
447          delete(sz);
448      end
449      % close video file
450      close(VW)
451      close(fig)
452 end
```

#### E.4.3.3.9 **plotSingleSimulationAngle**

### **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.

### **Contents**

- Syntax
- Description
- Examples
- Input Argurments
- Output Argurments
- Requirements
- See Also

### **Syntax**

```
1 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

```
1 plotSingleSimulationAngle()
```

## Input Argurments

**None**

## Output Argurments

**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.

```
1 function plotSingleSimulationAngle()
2 % scan for datasets and load needed configurations
3 %%%%%%
4 %
5 %%%%%%
6 %
7 try
8     disp('Plot single simulation angle ...');
9     close all;
10    % load path variables
11    load('config.mat', 'PathVariables');
12    % scan for datasets
13    TrainingDatasets = dir(fullfile(PathVariables.
14        trainingDataPath, ...
15        'Training_*.mat'));
16    TestDatasets = dir(fullfile(PathVariables.testDataPath,
17        'Test_*.mat'));
18    allDatasets = [TrainingDatasets; TestDatasets];
19    % check if files available
20    if isempty(allDatasets)
21        error('No training or test datasets found.');
22    end
23    catch ME
24        rethrow(ME)
25    end
26
27    % display available datasets to user, decide which to plot
28    %%%%%%
29    %
30    %%%%%%
31    %
32    % number of datasets
33    nDatasets = length(allDatasets);
34    fprintf('Found %d datasets:\n', nDatasets)
```

```
28     for i = 1:nDatasets
29         fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
30     end
31 % get numeric user input to indicate which dataset to plot
32 iDataset = input('Type number to choose dataset to plot to:
33     ');
34 %
35 % load dataset and ask user which one and how many angles
36 %
37 try
38     ds = load(fullfile(allDatasets(iDataset).folder, ...
39                 allDatasets(iDataset).name));
40 % check how many angles in dataset and let user decide
41 % how many to
42 % render in polt
43     fprintf('Detect %d angles ([1:%d]) in dataset ... \n',
44             ...
45             ds.Info.UseOptions.nAngles, ds.Info.UseOptions.
46             nAngles);
47     fprintf('Resolution\t:\t%.1f\n', ds.Info.UseOptions.
48             angleRes);
49     fprintf('Step width\t:\t%.1f\n', ds.Data.angleStep);
50     fprintf('Start angle\t:\t%.1f\n', ds.Data.angles(1))
51     idx = input('Which angle do you wish to plot (enter
52                 index): ');
53     angle = interp1(ds.Data.angles, idx, 'nearest');
54 catch ME
55     rethrow(ME)
56 end
57 %
58 % figure save path for different formats
59 %%%%%%
```

```

54 %
55 fPath = PathVariables.saveImagesPath;
56
57 % create dataset figure for a subset or all angle
58 %
59
60 fig = figure('Name', 'Sensor Array', ...
61   'NumberTitle', 'off', ...
62   'WindowSize', 'normal', ...
63   'Position', [4381 15 1244 983], ...
64   'Units', 'pixels', ...
65   'WindowState', 'maximized');
66
67 tdl = tiledlayout(fig, 2, 2, ...
68   'Padding', 'normal', ...
69   'TileSpacing', 'compact');
70
71 disp('Sensor Array Simulation');
72
73 subline1 = "Sensor Array (%s) of %dx%d sensors, " + ...
74   "an edge length of %.1f mm, a rel. pos. to magnet
75   surface of";
76 subline2 = " (%.1f, %.1f, -(%.1f)) in mm, a magnet tilt" +
77   ...
78   " of %.1f, a sphere radius of %.1f mm, a imprinted";
79 subline3 = "field strength of %.1f kA/m at %.1f mm from" +
80   ...
81   " sphere surface in z-axis, %d rotation angles with a
82   ";
83 subline4 = "step width of %.1f and a resolution of" + ...
84   " %.1f. Visualized is rotatation angle %d (%.1f)$.";
```

```

81    subline5 = "Based on %s characterization reference %s.";
82    sub = [sprintf(subline1, ...
83                  ds.Info.SensorArrayOptions.geometry, ...
84                  ds.Info.SensorArrayOptions.dimension, ...
85                  ds.Info.SensorArrayOptions.dimension, ...
86                  ds.Info.SensorArrayOptions.edge); ...
87    sprintf(subline2, ...
88            ds.Info.UseOptions.xPos, ...
89            ds.Info.UseOptions.yPos, ...
90            ds.Info.UseOptions.zPos, ...
91            ds.Info.UseOptions.tilt, ...
92            ds.Info.DipoleOptions.sphereRadius); ...
93    sprintf(subline3, ...
94            ds.Info.DipoleOptions.H0mag, ...
95            ds.Info.DipoleOptions.z0, ...
96            ds.Info.UseOptions.nAngles); ...
97    sprintf(subline4, ...
98            ds.Data.angleStep, ...
99            ds.Info.UseOptions.angleRes, ...
100           idx, angle)
101   sprintf(subline5, ...
102           ds.Info.CharData, ...
103           ds.Info.UseOptions.BridgeReference)];
104
105 disp(sub);
106
107 % get subset of needed data to plot, only one load
108 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
109 N = ds.Info.SensorArrayOptions.dimension;
110 X = ds.Data.X;
111 Y = ds.Data.Y;
112 Z = ds.Data.Z;
113

```

```
114 % calc limits of plot 1
115 maxX = ds.Info.UseOptions.xPos + ds.Info.SensorArrayOptions
116     .edge;
117 maxY = ds.Info.UseOptions.yPos + ds.Info.SensorArrayOptions
118     .edge;
119 minX = ds.Info.UseOptions.xPos - ds.Info.SensorArrayOptions
120     .edge;
121 minY = ds.Info.UseOptions.yPos - ds.Info.SensorArrayOptions
122     .edge;
123
124 % calculate colormap to identify scatter points
125 c=zeros(N,N,3);
126 for i = 1:N
127     for j = 1:N
128         c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
129     end
130 end
131 c = squeeze(reshape(c, N^2, 1, 3));
132
133 % load offset voltage to subtract from cosinus, sinus
134 % voltage
135 Voff = ds.Info.SensorArrayOptions.Voff;
136
137 % plot sensor grid in x and y coordinates and constant z
138 % layer %%%%%%%%%%%%%%
139 %
140 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
141
142 ax1 = nexttile(1);
143 % plot each cooredinate in loop to create a special shading
144     constant
145 % reliable to orientation for all matrice
146 hold on;
147 scatter(X(:,1), Y(:,1), [], c, 'filled', 'MarkerEdgeColor', 'k'
148     ,
149     'LineWidth', 0.8);
```

```
140
141 % axis shape and ticks
142 axis square xy;
143 axis tight;
144 grid on;
145 xlim([minX maxX]);
146 ylim([minY maxY]);
147
148 % text and labels
149 text(minX+0.2, minY+0.2, ...
150     sprintf('$Z = %.1f$ mm', z(1)), ...
151     'Color', 'k', ...
152     'FontSize', 20, ...
153     'FontName', 'Times', ...
154     'Interpreter', 'latex');
155
156 xlabel('$X$ in mm');
157
158 ylabel('$Y$ in mm');
159
160 title(sprintf('a) Sensor-Array $%d\\times%d$', N, N));
161
162 hold off;
163
164 % plot rotation angles in polar view
165 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
166
167 nexttile(2);
168 % plot all angles grayed out
169 polarscatter(ds.Data.angles/180*pi, ones(1, ds.Info.
170     UseOptions.nAngles), ...
171     [], [0.8 0.8 0.8], 'filled');
172
173 % radius ticks and label
```

```
172     rticks(1);
173     rticklabels("");
174     hold on;
175
176     % plot subset of angles
177     % polarscatter(subAngles/180*pi, ones(1, nSubAngles), ...
178     %    'k', 'LineWidth', 0.8);
179     ax2 = gca;
180
181     % axis shape
182     axis tight;
183
184     % text an labels
185     % init first rotation step label
186     tA = text(2/3*pi, 1.5, ...
187             '$\theta$', ...
188             'Color', 'b', ...
189             'FontSize', 20, ...
190             'FontName', 'Times', ...
191             'Interpreter', 'latex');
192
193     title('b) Rotation Angle');
194
195     hold off;
196
197     % Cosinus bridge outputs for rotation step
198     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
199
200     ax3 = nexttile(3);
201     hold on;
202
203     % set colormap
204     colormap('gray');
```

```

205 % plot cosinus reference, set NaN values to white color,
206 % orient Y to normal
207 imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
208 VcosRef);
209 set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
210 set(gca, 'YDir', 'normal')
211
212 % axis shape and ticks
213 axis square xy;
214 axis tight;
215 yticks(xticks);
216 grid on;
217
218 % test and labels
219 xlabel('$H_x$ in kA/m');
220
221 ylabel('$H_y$ in kA/m');
222
223 title('c) $V_{\cos}(H_x, H_y)$ in V');
224
225 % add colorbar and place it
226 cb1 = colorbar;
227 cb1.Label.String = sprintf(...,
228 '$V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
229 ds.Info.SensorArrayOptions.Vcc, ds.Info.
230 SensorArrayOptions.Voff);
231 cb1.TickLabelInterpreter = 'latex';
232 cb1.Label.Interpreter = 'latex';
233 cb1.Label.FontSize = 20;
234
235 hold off;
236
237 % Sinus bridge outputs for rotation step
238 %%%%%%%%%%%%%%

```

```

235 %
236 ax4 = nexttile(4);
237 hold on;
238
239 % set colormap
240 colormap('gray');
241
242 % plot sinus reference, set NaN values to white color,
243 % orient Y to normal
244 imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
245 VsinRef);
246 set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
247 set(gca, 'YDir', 'normal')
248
249 % axis shape and ticks
250 axis square xy;
251 axis tight;
252 yticks(xticks);
253 grid on;
254
255 % test and labels
256 xlabel('$H_x$ in kA/m');
257
258 ylabel('$H_y$ in kA/m');
259
260 title('d) $V_{\sin}(H_x, H_y)$ in V');
261
262 % add colorbar and place it
263 cb2 = colorbar;
264 cb2.Label.String = sprintf...
265     '$V_{cc} = %1.1f$ V, $V_{off} = %1.2f$ V', ...
    ds.Info.SensorArrayOptions.Vcc, ds.Info.
        SensorArrayOptions.Voff);
cb2.TickLabelInterpreter = 'latex';

```

```
266 cb2.Label.Interpreter = 'latex';
267 cb2.Label.FontSize = 20;
268
269 hold off;
270
271 % zoom axes for scatter on cosinuns reference images
272 %
273 nexttile(3);
274 ax5 = axes('Position', [0.15 0.115 0.12 0.12], ...
275 'XColor', 'r', 'YColor', 'r');
276 xticklabels(ax5, []);
277 yticklabels(ax5, []);
278 hold on;
279 axis square xy;
280 grid on;
281 hold off;
282
283 ax6 = axes('Position', [0.581 0.115 0.12 0.12], ...
284 'XColor', 'r', 'YColor', 'r');
285 xticklabels(ax6, []);
286 yticklabels(ax6, []);
287 hold on;
288 axis square xy;
289 grid on;
290 hold off;
291
292 % plot angle into plots
293 %
294 % H load subset
295 Hx = ds.Data.Hx(:,:,idx);
```

```
296 Hy = ds.Data.Hy(:,:,idx);
297 % get min max
298 maxHx = max(Hx, [], 'all');
299 maxHy = max(Hy, [], 'all');
300 minHx = min(Hx, [], 'all');
301 minHy = min(Hy, [], 'all');
302 dHx = abs(maxHx - minHx);
303 dHy = abs(maxHy - minHy);
304
305 % load V subset
306 Vcos = ds.Data.Vcos(:,:,idx) - Voff;
307 Vsini = ds.Data.Vsin(:,:,idx) - Voff;
308 angle = ds.Data.angles(idx);
309
310 % lock plots
311 hold(ax1, 'on');
312 hold(ax2, 'on');
313 hold(ax3, 'on');
314 hold(ax4, 'on');
315 hold(ax5, 'on');
316 hold(ax6, 'on');
317
318 % update plot 1
319 qH = quiver(ax1, X, Y, Hx, Hy, 0.7, 'b');
320 qV = quiver(ax1, X, Y, Vcos, Vsini, 0.7, 'r');
321 legend([qH qV], {'$quiver(H_x,H_y)$', ...
322 '$quiver(V_{\cos}-V_{\text{off}},V_{\sin}-V_{\text{off}})$'}, ...
323 'FontSize', 14, ...
324 'Location', 'NorthEast');
325
326 % update plot 2
327 tA.String = sprintf('$%.1f^\circ$', angle);
328 polarscatter(ax2, angle/180*pi, 1, 'b', 'filled', ...
329 'MarkerEdgeColor', 'k', 'LineWidth', 0.8);
330
331 % update plot 3 and 4
```

```
332 scatter(ax3, Hx(:), Hy(:), 5, c, 'filled', 'MarkerEdgeColor
333     ', 'k', ...
334     'LineWidth', 0.8);
335 scatter(ax4, Hx(:), Hy(:), 5, c, 'filled', 'MarkerEdgeColor
336     ', 'k', ...
337     'LineWidth', 0.8);

338 % calc position of scatter area frame and reframe
339 pos = [minHx - 0.3 * dHx, minHy - 0.3 * dHy, 1.6 * dHx, 1.6
340         * dHy];
341 rectangle(ax3, 'Position', pos, 'LineWidth', 1.5, 'EdgeColor', 'r');
342 rectangle(ax4, 'Position', pos, 'LineWidth', 1.5, 'EdgeColor', 'r');

343 % update plot 5 (zoom)
344 scatter(ax5, Hx(:), Hy(:), [], c, 'filled', 'MarkerEdgeColor', 'k', ...
345     'LineWidth', 0.8);
346 xlim(ax5, [pos(1) maxHx + 0.3 * dHx])
347 ylim(ax5, [pos(2) maxHy + 0.3 * dHy])

348 % update plot 6 (zoom)
349 scatter(ax6, Hx(:), Hy(:), [], c, 'filled', 'MarkerEdgeColor', 'k', ...
350     'LineWidth', 0.8);
351 xlim(ax6, [pos(1) maxHx + 0.3 * dHx])
352 ylim(ax6, [pos(2) maxHy + 0.3 * dHy])

353 % release plots
354 hold(ax1, 'off');
355 hold(ax2, 'off');
356 hold(ax3, 'off');
357 hold(ax4, 'off');
358 hold(ax5, 'off');
359 hold(ax6, 'off');
```

```
361
362     % save figure to file
363
364     % get file path to save figure with angle index
365     [~, fName, ~] = fileparts(ds.Info.filePath);
366
367     % save to various formats
368     % yesno = input('Save? [y/n]: ', 's');
369     % if strcmp(yesno, 'y')
370     %     fLabel = input('Enter file label: ', 's');
371     %     fName = fName + sprintf("_AnglePlot_%d_", idx) +
372     %             fLabel;
373     %     savefig(fig, fullfile(fPath, fName));
374     %     print(fig, fullfile(fPath, fName), '-dsvg');
375     %     print(fig, fullfile(fPath, fName), '-depsc', '-tiff',
376     %           '-loose');
377     %     print(fig, fullfile(fPath, fName), '-dpdf', '-loose',
378     %           '-fillpage');
379
380     % end
381     % close(fig);
382
383 end
```

#### E.4.3.3.10 **plotSimulationSubset**

## **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.

## **Contents**

- Syntax
- Description
- Examples
- Input Argurments
- Output Argurments
- Requirements
- See Also

## **Syntax**

```
1 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

```
1 plotSimulationSubset()
```

## Input Argurments

**None**

## Output Argurments

**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.

```
1 function plotSimulationSubset()
2 % scan for datasets and load needed configurations
3 %%%%%%
4 %
5 %%%%%%
6 %
7 try
8     disp('Plot simulation dataset ...');
9     close all;
10    % load path variables
11    load('config.mat', 'PathVariables');
12    % scan for datasets
13    TrainingDatasets = dir(fullfile(PathVariables.
14        trainingDataPath, ...
15        'Training_*.mat'));
16    TestDatasets = dir(fullfile(PathVariables.testDataPath,
17        'Test_*.mat'));
18    allDatasets = [TrainingDatasets; TestDatasets];
19    % check if files available
20    if isempty(allDatasets)
21        error('No training or test datasets found.');
22    end
23    catch ME
24        rethrow(ME)
25    end
26
27    % display available datasets to user, decide which to plot
28    %%%%%%
29    %
30    %%%%%%
31    %
32    % number of datasets
33    nDatasets = length(allDatasets);
34    fprintf('Found %d datasets:\n', nDatasets)
```

```
28     for i = 1:nDatasets
29         fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
30     end
31     % get numeric user input to indicate which dataset to plot
32     iDataset = 3;%input('Type number to choose dataset to plot
33     % to: ');
34
35     % load dataset and ask user which one and how many angles
36     %%%%%%%%%%%%%%
37
38     try
39         ds = load(fullfile(allDatasets(iDataset).folder, ...
40                     allDatasets(iDataset).name));
41         % check how many angles in dataset and let user decide
42         % how many to
43         % render in polt
44         fprintf('Detect %d x %d sensors in dataset ... \n', ...
45                 ds.Info.SensorArrayOptions.dimension, ...
46                 ds.Info.SensorArrayOptions.dimension);
47         xIdx = input("Enter row indices in []: ");
48         yIdx = input("Enter col indices in []: ");
49         if length(xIdx) ~= length(yIdx)
50             error('Indices must have the same length!')
51         end
52         % fprintf('Detect %d angles in dataset ... \n', ...
53         %         ds.Info.UseOptions.nAngles);
54         nSubAngles = input('How many angles to you wish to
55         % plot: ');
56         % indices for data to plot, get sample distance for
57         % even distance
58         % sampleDistance = length(downsampling(ds.Data.angles,
59         % nSubAngles));
60         % get subset of angles
```

```

55 %         subAngles = downsample(ds.Data.angles, sampleDistance
56 % );
57 %         nSubAngles = length(subAngles); % just ensure
58 %         % get indices for subset data
59 %         angleIdx = find(ismember(ds.Data.angles, subAngles));
60 %     catch ME
61 %         rethrow(ME)
62 %     end
63 %
64 % figure save path for different formats
65 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
66 %
67 % create dataset figure for a subset or all angle
68 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
69 fig = figure('Name', 'Sensor Array', ...
70 'NumberTitle', 'off', ...
71 'WindowStyle', 'normal', ...
72 'WindowState', 'maximized');
73
74 tdl = tiledlayout(fig, 4, 6, ...
75 'Padding', 'compact', ...
76 'TileSpacing', 'compact');
77
78
79 disp('Sensor Array Simulation');
80
81 subline1 = "Sensor Array (%s) of %dx%d sensors," + ...
82 " an edge length of %.1f mm, a rel. pos. to magnet
      surface of";

```

```

83    subline2 = " (%.1f, %.1f, -(%.1f)) in mm, a magnet tilt" +
84        ...
85        " of %.1f, a sphere radius of %.1f mm, a imprinted";
86    subline3 = "field strength of %.1f kA/m at %.1f mm from" +
87        ...
88        " sphere surface in z-axis, %d rotation angles with a
89        ";
90    subline4 = "step width of %.1f and a resolution" + ...
91        " of %.1f. Visualized is a subset.";
92    subline5 = "Based on %s characterization reference %s.";
93    sub = [sprintf(subline1, ...
94                ds.Info.SensorArrayOptions.geometry, ...
95                ds.Info.SensorArrayOptions.dimension, ...
96                ds.Info.SensorArrayOptions.dimension, ...
97                ds.Info.SensorArrayOptions.edge); ...
98    sprintf(subline2, ...
99                ds.Info.UseOptions.xPos, ...
100               ds.Info.UseOptions.yPos, ...
101               ds.Info.UseOptions.zPos, ...
102               ds.Info.UseOptions.tilt, ...
103               ds.Info.DipoleOptions.sphereRadius); ...
104    sprintf(subline3, ...
105               ds.Info.DipoleOptions.H0mag, ...
106               ds.Info.DipoleOptions.z0, ...
107               ds.Info.UseOptions.nAngles); ...
108    sprintf(subline4, ...
109               ds.Data.angleStep, ...
110               ds.Info.UseOptions.angleRes)
111    sprintf(subline5, ...
112               ds.Info.CharData, ...
113               ds.Info.UseOptions.BridgeReference)];
114
115    disp(sub);
116
117    %% get subset of needed data to plot, only one load
118    %%%%%%%%

```

```

115 %
116 N = ds.Info.SensorArrayOptions.dimension;
117 X = ds.Data.X;
118 Y = ds.Data.Y;
119 Z = ds.Data.Z;
120
121 % calc limits of plot 1
122 a= ds.Info.SensorArrayOptions.edge;
123 maxX = ds.Info.UseOptions.xPos + a * 0.66;
124 maxY = ds.Info.UseOptions.yPos + a * 0.66;
125 minX = ds.Info.UseOptions.xPos - a * 0.66;
126 minY = ds.Info.UseOptions.yPos - a * 0.66;
127 dp = a / (ds.Info.SensorArrayOptions.dimension - 1);
128 x1 = ds.Info.UseOptions.xPos - a/2;
129 x2 = ds.Info.UseOptions.xPos + a/2;
130 y1 = ds.Info.UseOptions.yPos - a/2;
131 y2 = ds.Info.UseOptions.yPos + a/2;
132
133 % calculate colormap to identify scatter points
134 c=zeros(N,N,3);
135 for i = 1:N
136     for j = 1:N
137         c(i,j,:) = [(2*N+1-2*i), (2*N+1-2*j), (i+j)]/2/N;
138     end
139 end
140 c = squeeze(reshape(c, N^2, 1, 3));
141 % reshape RGB for picking single sensors
142 R = reshape(c(:,1), N, N);
143 G = reshape(c(:,2), N, N);
144 B = reshape(c(:,3), N, N);
145
146 % load offset voltage to subtract from cosinus, sinus
147 % voltage
148 Voff = ds.Info.SensorArrayOptions.Voff;

```

```
148 Vcc = ds.Info.SensorArrayOptions.Vcc;
149
150 % plot sensor grid in x and y coordinates and constant z
151 % layer %%%%%%%%%%%%%%
152 %
153 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
154
155 ax1 = nexttile(6,[2 1]);
156 % plot each cooredinate in loop to create a special shading
157 % constant
158 % reliable to orientation for all matrice
159 hold on;
160
161 scatter(X(:, Y(:, 48, [0.8 0.8 0.8], 'filled', ...
162 'MarkerEdgeColor', 'k', 'LineWidth', 0.8);
163
164 for k = 1:length(xIdx)
165     i = xIdx(k); j = yIdx(k);
166     scatter(X(i, j), Y(i, j), 96, [R(i, j), G(i, j), B(i, j)], 'filled', ...
167         'MarkerEdgeColor', 'k', 'LineWidth', 0.8);
168 end
169
170 % axis shape and ticks
171 axis square xy;
172 axis tight;
173 grid on;
174 xlim([minX maxX]);
175 ylim([minY maxY]);
176 xticks(x1:dp:x2);
177 xticklabels(1:ds.Info.SensorArrayOptions.dimension);
178 yticks(y1:dp:y2);
179 yticklabels(ds.Info.SensorArrayOptions.dimension:-1:1);
180
181 xlabel('$j$');
```

```
179 ylabel('$i$');
180
181 title(sprintf('c) Sensor Array $%d\\times%d$', N, N));
182 hold off;
183
184
185
186 % Cosinus bridge outputs for rotation step
187 %%%%%%%%%%%%%%
188 ax3 = nexttile(1, [2 2]);
189 hold on;
190
191 % set colormap
192 colormap('gray');
193
194 % plot cosinus reference, set NaN values to white color,
195 % orient Y to normal
196 imC = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
197 VcosRef);
198 set(imC, 'AlphaData', ~isnan(ds.Data.VcosRef));
199 set(gca, 'YDir', 'normal')
200
201 % axis shape and ticks
202 axis square xy;
203 axis tight;
204 yticks(xticks);
205 grid on;
206
207 % test and labels
208 xlabel('$H_x$ in kA/m');
209 ylabel('$H_y$ in kA/m');
```

```
210 title('a) $V_{\cos}(H_x, H_y)$');
211
212 % add colorbar and place it
213 cb1 = colorbar;
214 cb1.Label.String = 'in V';
215 cb1.TickLabelInterpreter = 'latex';
216 cb1.Label.Interpreter = 'latex';
217 cb1.Label.FontSize = 20;
218 hold off;
219
220 % Sinus bridge outputs for rotation step
221 %
222 %%%%%%%%%%%%%%
223 %
224 ax4 = nexttile(13, [2 2]);
225 hold on;
226
227 % set colormap
228 colormap('gray');
229
230 % plot sinus reference, set NaN values to white color,
231 % orient Y to normal
232 imS = imagesc(ds.Data.HxScale, ds.Data.HyScale, ds.Data.
233 VsinRef);
234 set(imS, 'AlphaData', ~isnan(ds.Data.VsinRef));
235 set(gca, 'YDir', 'normal')
236
237 % axis shape and ticks
238 axis square xy;
239 axis tight;
240 yticks(xticks);
241 grid on;
242
243 % test and labels
244 xlabel('$H_x$ in kA/m');
```

```

241
242     ylabel('$H_y$ in kA/m');
243
244     title('d) $V_{\sin}(H_x, H_y)$');
245
246     % add colorbar and place it
247     cb2 = colorbar;
248     cb2.Label.String = 'in V';
249     cb2.TickLabelInterpreter = 'latex';
250     cb2.Label.Interpreter = 'latex';
251     cb2.Label.FontSize = 20;
252
253     hold off;
254
255     % plot Vcos Vsin over angles
256
257     % axes limits
258     xlims = [0 360];
259     ylims = [min(cat(...
260         3, ds.Data.VsinRef, ds.Data.VcosRef), [], 'all') - 0.1*...
261         Vcc, ...
262         max(cat(3, ds.Data.VsinRef, ds.Data.VcosRef), [], 'all',...
263             ) + 0.1*Vcc];
264
265     % Vcos
266     ax5 = nexttile(3, [2 3]);
267     yline(Voff, 'k-.', 'LineWidth', 2.5);
268     xlim(xlims);
269     ylim(ylimits);
270     grid on;
271
272     xlabel('$\alpha$ in $\circ$');

```

```

272 %ylabel('in V');
273
274 title(sprintf(...  

275     "b) $V_{cos}(\alpha) f." + ...  

276     " $V_{cc} = %.1f$ V, $V_{off} = %.2f$ V", Vcc, Voff));
277
278 % Vsin
279 ax6 = nexttile(15, [2 3]);
280 yline(Voff, 'k-.', 'LineWidth', 2.5);
281 xlim(xlimits);
282 ylim(ylimits);
283 grid on;
284
285 xlabel('$\alpha$ in $\circ$');
286
287 %ylabel('in V');
288
289 title(sprintf("e) $V_{sin}(\alpha) f." + ...  

290     " $V_{cc} = %.1f$ V, $V_{off} = %.2f$ V", Vcc, Voff));
291
292 % loop through subset of dataset and renew plots  

293 %%%
294 % lock plots
295 hold(ax3, 'on');
296 hold(ax4, 'on');
297 hold(ax5, 'on');
298 hold(ax6, 'on');
299
300 % loop over indices
301 for k = 1:length(xIdx)
302     i = xIdx(k); j = yIdx(k);
303     % H load subset
304     Hx = squeeze(ds.Data.Hx(i, j, :));

```

```

305     Hy = squeeze(ds.Data.Hy(i,j,:));
306     % get min max
307
308     % load V subset
309     Vcos = squeeze(ds.Data.Vcos(i,j,:));
310     Vsincos = squeeze(ds.Data.Vsin(i,j,:));
311
312     % update plot 3, 4, 5 and 6
313     scatter(ax3, Hx, Hy, 5, [R(i,j), G(i,j), B(i,j)] , 'filled');
314     scatter(ax4, Hx, Hy, 5, [R(i,j), G(i,j), B(i,j)] , 'filled');
315     scatter(ax5, ds.Data.angles, Vcos, 8, [R(i,j), G(i,j),
316         B(i,j)] , ...
317         'filled');
318     scatter(ax6, ds.Data.angles, Vsincos, 8, [R(i,j), G(i,j),
319         B(i,j)] , ...
320         'filled');
321
322     % release plots
323     hold(ax3, 'off');
324     hold(ax4, 'off');
325     hold(ax5, 'off');
326     hold(ax6, 'off');
327
328     % save figure to file
329     %%%
330
331     % get file path to save figure with angle index
332     [~, fName, ~] = fileparts(ds.Info.filePath);
333     %
334     % save to various formats
335     yesno = input('Save? [y/n]: ', 's');

```

```
334 %     if strcmp(yesno, 'y')
335 %         fLabel = input('Enter file label: ', 's');
336 %         fName = fName + "_SubsetPlot_" + fLabel;
337 %         savefig(fig, fullfile(fPath, fName));
338 %         print(fig, fullfile(fPath, fName), '-dsvg');
339 %         print(fig, fullfile(fPath, fName), '-depsc', '-tiff',
340 %               '-loose');
341 %         print(fig, fullfile(fPath, fName), '-dpdf', '-loose',
342 %               '-fillpage');
343 %     end
344 %     close(fig);
345 end
```

#### **E.4.3.3.11 plotSimulationCosSinStats**

### **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.

### **Contents**

- Syntax
- Description
- Examples
- Input Argurments
- Output Argurments
- Requirements
- See Also

### **Syntax**

```
1 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

```
1 plotSimulationCosSinStats()
```

## Input Argurments

**None**

## Output Argurments

**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.

```
1 function plotSimulationCosSinStats()
2 % scan for datasets and load needed configurations
3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 %
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 %
7 %
8 %
9 %
10 %
11 %
12 %
13 %
14 %
15 %
16 %
17 %
18 %
19 %
20 %
21 %
22 %
23 %
24 %
25 %
26 %
27 %
```

```
1 function plotSimulationCosSinStats()
2 % scan for datasets and load needed configurations
3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 %
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 %
7 %
8 %
9 %
10 TrainingDatasets = dir(fullfile(PathVariables.
11     trainingDataPath, ...
12         'Training_*.mat'));
13 TestDatasets = dir(fullfile(PathVariables.testDataPath,
14     'Test_*.mat'));
15 allDatasets = [TrainingDatasets; TestDatasets];
16 % check if files available
17 if isempty(allDatasets)
18     error('No training or test datasets found.');
19 end
20 catch ME
21     rethrow(ME)
22 end
23 %
24 %
25 %
26 nDatasets = length(allDatasets);
27 fprintf('Found %d datasets:\n', nDatasets)
```

```
28     for i = 1:nDatasets
29         fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
30     end
31     % get numeric user input to indicate which dataset to plot
32     iDataset = input('Type number to choose dataset to plot to:
33     ');
34
35     %% load dataset and ask user which one and how many angles
36     %%%%%%%%%%%%%%
37
38     try
39         ds = load(fullfile(allDatasets(iDataset).folder, ...
40                     allDatasets(iDataset).name));
41         % check how many angles in dataset and let user decide
42         % how many to
43         % render in plot
44         fprintf('Detect %d angles in dataset ... \n', ...
45                 ds.Info.UseOptions.nAngles);
46         nSubAngles = input('How many angles to you wish to plot
47         : ');
48         % nSubAngles = 120;
49         % indices for data to plot, get sample distance for
50         % even distance
51         sampleDistance = length(ds.Data.angles,
52                                nSubAngles));
53         % get subset of angles
54         subAngles = downsample(ds.Data.angles, sampleDistance);
55         nSubAngles = length(subAngles); % just ensure
56         % get indices for subset data
57         indices = find(ismember(ds.Data.angles, subAngles));
58     catch ME
59         rethrow(ME)
60     end
```

```
56 % figure save path for different formats  
57 %  
58 fPath = fullfile(PathVariables.saveImagePath);  
59  
60 % create dataset figure for a subset or all angle  
61 %  
62 fig = figure('Name', 'Sensor Array', ...  
63 'NumberTitle', 'off', ...  
64 'WindowSize', 'normal', ...  
65 'MenuBar', 'none', ...  
66 'ToolBar', 'none', ...  
67 'Units', 'centimeters', ...  
68 'OuterPosition', [0 0 37 29], ...  
69 'PaperType', 'a4', ...  
70 'PaperUnits', 'centimeters', ...  
71 'PaperOrientation', 'landscape', ...  
72 'PaperPositionMode', 'auto', ...  
73 'DoubleBuffer', 'on', ...  
74 'RendererMode', 'manual', ...  
75 'Renderer', 'painters');  
76  
77 tdl = tiledlayout(fig, 2, 1, ...  
78 'Padding', 'compact', ...  
79 'TileSpacing', 'compact');  
80  
81  
82 title(tdl, 'Sensor Array Simulation', ...  
83 'FontWeight', 'normal', ...  
84 'FontSize', 18, ...  
85 'FontName', 'Times', ...
```

```
86     'Interpreter', 'latex');
87
88 subline1 = "Sensor Array (%s) of $%d\\times%d$ sensors, " +
89             ...
90             "an edge length of $%.1f$ mm, a rel. pos. to magnet
91             surface of";
92 subline2 = " $(%.1f, %.1f, -(%.1f))\$ in mm, a magnet tilt"
93             +
94             " of $%.1f^\\circ$, a sphere radius of $%.1f$ mm, a
95             imprinted";
96 subline3 = "field strength of $%.1f\$ kA/m at $%.1f\$ mm from
97             " +
98             " sphere surface in z-axis, $%d$ rotation angles with a
99             ";
100
101 subline4 = "step width of $%.1f^\\circ$ and a resolution of
102             " +
103             " $%.1f^\\circ$. Visualized is a subset of $%d$ angles
104             in ";
105
106 subline5 = "sample distance of $%d$ angles. Based on %s" +
107             ...
108             " characterization reference %s.";
```

109 sub = [sprintf(subline1, ...
110 ds.Info.SensorArrayOptions.geometry, ...
111 ds.Info.SensorArrayOptions.dimension, ...
112 ds.Info.SensorArrayOptions.dimension, ...
113 ds.Info.SensorArrayOptions.edge); ...
114
115 sprintf(subline2, ...
116 ds.Info.UseOptions.xPos, ...
117 ds.Info.UseOptions.yPos, ...
118 ds.Info.UseOptions.zPos, ...
119 ds.Info.UseOptions.tilt, ...
120 ds.Info.DipoleOptions.sphereRadius); ...
121
122 sprintf(subline3, ...
123 ds.Info.DipoleOptions.H0mag, ...
124 ds.Info.DipoleOptions.z0, ...
125 ds.Info.UseOptions.nAngles); ...]

```
113         sprintf(subline4, ...
114             ds.Data.angleStep, ...
115             ds.Info.UseOptions.angleRes, ...
116             nSubAngles)
117         sprintf(subline5, ...
118             sampleDistance, ...
119             ds.Info.CharData, ...
120             ds.Info.UseOptions.BridgeReference) ];
121
122     subtitle(tdl, sub, ...
123         'FontWeight', 'normal', ...
124         'FontSize', 14, ...
125         'FontName', 'Times', ...
126         'Interpreter', 'latex');
127
128 % get subset of needed data to plot, only one load
129 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
130 %
131 M = ds.Info.SensorArrayOptions.dimension^2;
132 % N = ds.Info.UseOptions.nAngles;
133 res = ds.Info.UseOptions.angleRes;
134 %angles = ds.Data.angles;
135 anglesIP = 0:res:360-res;
136
137 % load V subset and reshape for easier computing statistics
138 Vcos = squeeze(reshape(ds.Data.Vcos(:,:,indices), 1, M,
139                         nSubAngles));
140 Vsin = squeeze(reshape(ds.Data.Vsin(:,:,indices), 1, M,
141                         nSubAngles));
142
143 % load offset voltage to subtract from cosinus, sinus
144 % voltage
145 Voff = ds.Info.SensorArrayOptions.Voff;
146 Vcc = ds.Info.SensorArrayOptions.Vcc;
```

```
143
144     % compute statistics of Vcos Vsin
145     %
146     % interpolate with makima makes best results, ensure to
147     % kill nans for
148     % fill otherwise fill strokes, use linstyle none for fill
149     % without frame
150     interpM = 'makima';
151
152     VcosMean = mean(Vcos, 1);
153     VcosMeanIP = interp1(subAngles, VcosMean, anglesIP, interpM
154     );
155
156     VcosStd = std(Vcos, 1, 1);
157     VcosVar = var(Vcos, 1, 1); % std^2
158
159     % meanvariation coefficient in percent
160     VcosMVCP = mean(VcosStd ./ VcosMean) * 100;
161
162     VcosUpper1 = VcosMean + VcosStd;
163     VcosUpper2 = VcosMean + VcosVar;
164     VcosLower1 = VcosMean - VcosStd;
165     VcosLower2 = VcosMean - VcosVar;
166
167     VcosUpper1IP = interp1(subAngles, VcosUpper1, anglesIP,
168     interpM);
169     VcosUpper1IP = fillmissing(VcosUpper1IP, 'previous');
170
171     VcosLower1IP = interp1(subAngles, VcosLower1, anglesIP,
172     interpM);
173     VcosLower1IP = fillmissing(VcosLower1IP, 'previous');
174
175     VcosUpper2IP = interp1(subAngles, VcosUpper2, anglesIP,
176     interpM);
```

```
170 VcosUpper2IP = fillmissing(VcosUpper2IP, 'previous');
171
172 VcosLower2IP = interp1(subAngles, VcosLower2, anglesIP,
173                         interpM);
174 VcosLower2IP = fillmissing(VcosLower2IP, 'previous');
175
176 VsinMean = mean(Vsin, 1);
177 VsinMeanIP = interp1(subAngles, VsinMean, anglesIP, interpM
178                       );
179
180 VsinStd = std(Vsin, 1, 1);
181 VsinVar = var(Vsin, 1, 1); % std^2
182
183 % meanvariation coefficient in percent
184 VsinMVCP = mean(VsinStd ./ VsinMean) * 100;
185
186 VsinUpper1 = VsinMean + VsinStd;
187 VsinUpper2 = VsinMean + VsinVar;
188 VsinLower1 = VsinMean - VsinStd;
189 VsinLower2 = VsinMean - VsinVar;
190
191 VsinUpper1IP = interp1(subAngles, VsinUpper1, anglesIP,
192                         interpM);
193 VsinUpper1IP = fillmissing(VsinUpper1IP, 'previous');
194
195 VsinLower1IP = interp1(subAngles, VsinLower1, anglesIP,
196                         interpM);
197 VsinLower1IP = fillmissing(VsinLower1IP, 'previous');
198
199 VsinUpper2IP = interp1(subAngles, VsinUpper2, anglesIP,
200                         interpM);
201 VsinUpper2IP = fillmissing(VsinUpper2IP, 'previous');
202
203 VsinLower2IP = interp1(subAngles, VsinLower2, anglesIP,
204                         interpM);
205 VsinLower2IP = fillmissing(VsinLower2IP, 'previous');
```

```

200
201 % plot Vcos Vsin over angles
202 %
203
204 % Vcos
205 nexttile;
206 hold on;
207
208 fillStdX = [anglesIP, fliplr(anglesIP)];
209 fillStdY = [VcosLower1IP, fliplr(VcosUpper1IP)];
210 fill(fillStdX, fillStdY, [0.95 0.95 0.95], 'LineStyle', 'none');
211
212 fillVarX = [anglesIP, fliplr(anglesIP)];
213 fillVarY = [VcosLower2IP, fliplr(VcosUpper2IP)];
214 fill(fillVarX, fillVarY, [0.7 0.7 0.7], 'LineStyle', 'none');
215
216 yline(Voff, 'k--');
217 scatter(subAngles, VcosUpper1, [], 'r*');
218 plot(anglesIP, VcosUpper1IP, 'r-.');
219 scatter(subAngles, VcosMean, [], 'm*');
220 plot(anglesIP, VcosMeanIP, 'm-.');
221 scatter(subAngles, VcosLower1, [], 'b*');
222 plot(anglesIP, VcosLower1IP, 'b-.');
223
224
225 hold off;
226 xlim([-res 360-res]);
227 %ylim(ylimits);
228 grid on;
229
230 xlabel('$\theta$ in Degree', ...

```

```
231     'FontWeight', 'normal', ...
232     'FontSize', 12, ...
233     'FontName', 'Times', ...
234     'Interpreter', 'latex');

235
236 ylabel('$V\{cos}(\theta)$ in V', ...
237     'FontWeight', 'normal', ...
238     'FontSize', 12, ...
239     'FontName', 'Times', ...
240     'Interpreter', 'latex');

241
242 title(sprintf(...
243     "Compare $V_{cos}(\theta)$ for each Array Member $V_{...
244         cc} = %.1f\$" + ...
245     "%V, $V_{off} = %.2f\$ V, $\bar{\sigma}_{\mu} = %.2f\$ ...
246         perc.", ...
247     Vcc, Voff, VcosMVCP), ...
248     'FontWeight', 'normal', ...
249     'FontSize', 12, ...
250     'FontName', 'Times', ...
251     'Interpreter', 'latex');

252
253 % Vsin
254 nexttile;
255 hold on;

256 fillStdX = [anglesIP, fliplr(anglesIP)];
257 fillStdY = [VsinLower1IP, fliplr(VsinUpper1IP)];
258 l1 = fill(fillStdX, fillStdY, [0.95 0.95 0.95], 'LineStyle',
259     , 'none');

260 fillVarX = [anglesIP, fliplr(anglesIP)];
261 fillVarY = [VsinLower2IP, fliplr(VsinUpper2IP)];
262 l2 = fill(fillVarX, fillVarY, [0.7 0.7 0.7], 'LineStyle',
263     , 'none');
```

```
263     13 = yline(Voff, 'k--');
264     14 = scatter(subAngles, VsinUpper1, [], 'r*');
265     15 = plot(anglesIP, VsinUpper1IP, 'r-.');
266     16 = scatter(subAngles, VsinMean, [], 'm*');
267     17 = plot(anglesIP, VsinMeanIP, 'm-.');
268     18 = scatter(subAngles, VsinLower1, [], 'b*');
269     19 = plot(anglesIP, VsinLower1IP, 'b-.');
270
271 hold off;
272 xlim([-res 360-res]);
273 grid on;
274
275 xlabel('$\theta$ in Degree', ...
276     'FontWeight', 'normal', ...
277     'FontSize', 12, ...
278     'FontName', 'Times', ...
279     'Interpreter', 'latex');
280
281 ylabel('V{sin}(\theta) in V', ...
282     'FontWeight', 'normal', ...
283     'FontSize', 12, ...
284     'FontName', 'Times', ...
285     'Interpreter', 'latex');
286 title(sprintf(...
287     "Compare $V_{sin}(\theta)$ for each Array Member $V_{
288         cc} = %.1f $" + ...
289     " V, $V_{off} = %.2f$ V, $\bar{\sigma}_{\mu} = %.2f$ "
290         perc.", ...
291     Vcc, Voff, VsinMVCP), ...
292     'FontWeight', 'normal', ...
293     'FontSize', 12, ...
294     'FontName', 'Times', ...
295     'Interpreter', 'latex');
```

```

295 % plot legend
%
%
296 %
%
297 l = [11 12 13 14 15 16 17 18 19];
298 L = legend(l, {'$2\sigma$', ...
299             '$2\sigma^2$', ...
300             '$V_{off}$', ...
301             '$U_{lim} = \mu + \sigma$', ...
302             sprintf('$%s(U_{lim})$', interpM), ...
303             '$\mu(V)$', ...
304             sprintf('$%s(\mu)$', interpM), ...
305             '$L_{lim} = \mu - \sigma$', ...
306             sprintf('$%s(L_{lim})$', interpM)}, ...
307             'FontWeight', 'normal', ...
308             'FontSize', 12, ...
309             'FontName', 'Times', ...
310             'Interpreter', 'latex');
311 L.Layout.Tile = 'east';
312
313 % save figure to file
%
%
314 %
%
315 % get file path to save figure with angle index
316 [~, fName, ~] = fileparts(ds.Info.filePath);
317
318 % save to various formats
319 yesno = input('Save? [y/n]: ', 's');
320 if strcmp(yesno, 'y')
321     fLabel = input('Enter file label: ', 's');
322     fName = fName + "_StatsPlot_" + fLabel;
323     savefig(fig, fullfile(fPath, fName));

```

```
324     print(fig, fullfile(fPath, fName), '-dsvg');
325     print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '
326         -loose');
327     print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '
328         -fillpage');
329 end
330 close(fig);
331 end
```

#### **E.4.3.3.12 plotSimulationDatasetCircle**

### **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.

### **Contents**

- Syntax
- Description
- Examples
- Input Argurments
- Output Argurments
- Requirements
- See Also

### **Syntax**

```
1 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

```
1 plotSimulationDatasetCircle()
```

## Input Argurments

**None**

## Output Argurments

**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.

```

1 function plotSimulationDatasetCircle()
2 % scan for datasets and load needed configurations
3 %%%%%%
4 %
5 %%%%%%
6 %
7 try
8     disp('Plot simulation dataset ...');
9     close all;
10    % load path variables
11    load('config.mat', 'PathVariables');
12    % scan for datasets
13    TrainingDatasets = dir(fullfile(PathVariables.
14        trainingDataPath, ...
15        'Training_*.mat'));
16    TestDatasets = dir(fullfile(PathVariables.testDataPath,
17        'Test_*.mat'));
18    allDatasets = [TrainingDatasets; TestDatasets];
19    % check if files available
20    if isempty(allDatasets)
21        error('No training or test datasets found.');
22    end
23    catch ME
24        rethrow(ME)
25    end
26
27    % display available datasets to user, decide which to plot
28    %%%%%%
29    %
30    %%%%%%
31    %
32    % number of datasets
33    nDatasets = length(allDatasets);
34    fprintf('Found %d datasets:\n', nDatasets)

```

```
28     for i = 1:nDatasets
29         fprintf('%s\t:\t%d\n', allDatasets(i).name, i)
30     end
31 % get numeric user input to indicate which dataset to plot
32 iDataset = input('Type number to choose dataset to plot to:
33     ');
34
35 % load dataset and ask user which one and how many angles
36 %%%%%%%%%%%%%%
37 %
38 %%%%%%%%%%%%%%
39 %
40 %%%%%%%%%%%%%%
41 %
42 % figure save path for different formats
43 %%%%%%%%%%%%%%
44 %
45 fPath = PathVariables.saveImagePath;
46
47 % create dataset figure for a subset or all angle
48 %%%%%%%%%%%%%%
49 %
50 %%%%%%%%%%%%%%
51 %
52 %
53 fig = figure('Name', 'Sensor Array', ...
    'NumberTitle', 'off', ...
    'WindowStyle', 'normal', ...
    'MenuBar', 'none', ...
    'ToolBar', 'none', ...
```

```
54     'Units', 'centimeters', ...
55     'OuterPosition', [0 0 30 30], ...
56     'PaperType', 'a4', ...
57     'PaperUnits', 'centimeters', ...
58     'PaperOrientation', 'landscape', ...
59     'PaperPositionMode', 'auto', ...
60     'DoubleBuffer', 'on', ...
61     'RendererMode', 'manual', ...
62     'Renderer', 'painters');

63
64 tdl = tiledlayout(fig, 2, 2, ...
65     'Padding', 'compact', ...
66     'TileSpacing', 'compact');

67
68 title(tdl, 'Sensor Array Simulation', ...
69     'FontWeight', 'normal', ...
70     'FontSize', 18, ...
71     'FontName', 'Times', ...
72     'Interpreter', 'latex');

73
74 subline1 = "Sensor Array (%s) of %d\times%d sensors," +
75     ...
76     " an edge length of %.1f mm, a rel. pos. to magnet
77         surface of";
78 subline2 = " (%.1f, %.1f, -(%.1f)) in mm, a magnet tilt"
79     +
80     " of %.1f^\circ, a sphere radius of %.1f mm, a
81         imprinted";
82 subline3 = "field strength of %.1f kA/m at %.1f mm from
83     "
84     " + ...
85     " sphere surface in z-axis, %d rotation angles with a
86         ";
87 subline4 = "step width of %.1f^\circ and a resolution of
88     "
89     " + ...
90     " %.1f^\circ. Visualized are circular path of each
91         array position ";
```

```

82    subline5 = "Based on %s characterization reference %s.";
83    sub = [sprintf(subline1, ...
84                  ds.Info.SensorArrayOptions.geometry, ...
85                  ds.Info.SensorArrayOptions.dimension, ...
86                  ds.Info.SensorArrayOptions.dimension, ...
87                  ds.Info.SensorArrayOptions.edge); ...
88    sprintf(subline2, ...
89                  ds.Info.UseOptions.xPos, ...
90                  ds.Info.UseOptions.yPos, ...
91                  ds.Info.UseOptions.zPos, ...
92                  ds.Info.UseOptions.tilt, ...
93                  ds.Info.DipoleOptions.sphereRadius); ...
94    sprintf(subline3, ...
95                  ds.Info.DipoleOptions.H0mag, ...
96                  ds.Info.DipoleOptions.z0, ...
97                  ds.Info.UseOptions.nAngles); ...
98    sprintf(subline4, ...
99                  ds.Data.angleStep, ...
100                 ds.Info.UseOptions.angleRes)
101    sprintf(subline5, ...
102                  ds.Info.CharData, ...
103                  ds.Info.UseOptions.BridgeReference)];
104
105    subtitle(tdl, sub, ...
106              'FontWeight', 'normal', ...
107              'FontSize', 14, ...
108              'FontName', 'Times', ...
109              'Interpreter', 'latex');
110
111    % get subset of needed data to plot, only one load
112    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
113
114    N = ds.Info.SensorArrayOptions.dimension;
115    M = ds.Info.UseOptions.nAngles;

```

```
115 Voff = ds.Info.SensorArrayOptions.Voff;
116 Vcos = ds.Data.Vcos - Voff;
117 Vsin = ds.Data.Vsin - Voff;
118 Hx = ds.Data.Hx;
119 Hy = ds.Data.Hy;
120
121 % calculate norm values to align circles around position
122 % only for x,y
123 % directition for each sensor dot over all angles.
124 Vmag = sqrt(Vcos.^2 + Vsin.^2);
125 Hmag = sqrt(Hx.^2 + Hy.^2);
126 %Hmag = ds.Data.Habs;
127
128 % related to position, multiply scale factor for circle
129 % diameter
130 diameterFactor = 2 * N / ds.Info.SensorArrayOptions.edge;
131 MaxVmagPos = max(Vmag, [], 3) * diameterFactor;
132 MaxHmagPos = max(Hmag, [], 3) * diameterFactor;
133
134 % Overall maxima, scalar, multiply scale factor for circle
135 % diameter
136 MaxVmagOA = max(Vmag, [], 'all') * diameterFactor;
137 MaxHmagOA = max(Hmag, [], 'all') * diameterFactor;
138
139 % norm and scale volatages and filed strengths
140 VcosNorm = Vcos ./ MaxVmagPos;
141 VcosScaled = Vcos / MaxVmagOA;
142 VsinNorm = Vsin ./ MaxVmagPos;
143 VsinScaled = Vsin / MaxVmagOA;
144
145 HxNorm = Hx ./ MaxHmagPos;
146 HxScaled = Hx / MaxHmagOA;
147 HyNorm = Hy ./ MaxHmagPos;
148 HyScaled = Hy / MaxHmagOA;
149
150 % sensor array grid
```

```
148 X = ds.Data.X;
149 Y = ds.Data.Y;
150 Z = ds.Data.Z;
151
152 % calc limits of plot 1
153 maxX = ds.Info.UseOptions.xPos + 0.7 * ds.Info.
    SensorArrayOptions.edge;
154 maxY = ds.Info.UseOptions.yPos + 0.7 * ds.Info.
    SensorArrayOptions.edge;
155 minX = ds.Info.UseOptions.xPos - 0.7 * ds.Info.
    SensorArrayOptions.edge;
156 minY = ds.Info.UseOptions.yPos - 0.7 * ds.Info.
    SensorArrayOptions.edge;
157
158 % plot sensor grid in x and y coordinates
159 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
160 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
161 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
162 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
163 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
164 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
165 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
166 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
167 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
168 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
169 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
170 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
171 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
172 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
173 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
174
```

```

175 % Field strength scaled to overall maxima
176 %
177 nexttile;
178 hold on;
179 for i = 1:N
180     for j = 1:N
181         plot(squeeze(HxScaled(i, j, :)) + X(i, j), ...
182             squeeze(HyScaled(i, j, :)) + Y(i, j), ...
183             'Color', [R(i, j) G(i, j) B(i, j)], ...
184             'LineWidth', 1.5)
185         line([X(i, j), HxScaled(i, j, 1) + X(i, j)], ...
186             [Y(i, j), HyScaled(i, j, 1) + Y(i, j)], ...
187             'Color', 'k', 'LineWidth', 1.5)
188     end
189 end
190
191 % scatter magnet x,y position (0,0,z)
192 scatter(0, 0, 32, 'r', 'filled');
193
194 hold off;
195
196 % axis shape and ticks
197 axis square xy;
198 axis tight;
199 grid on;
200 xlim([minX maxX]);
201 ylim([minY maxY]);
202
203 xlabel('$X$ in mm', ...
204     'FontWeight', 'normal', ...
205     'FontSize', 12, ...
206     'FontName', 'Times', ...
207     'Interpreter', 'latex');
```

```
208
209     ylabel('$Y$ in mm', ...
210         'FontWeight', 'normal', ...
211         'FontSize', 12, ...
212         'FontName', 'Times', ...
213         'Interpreter', 'latex');
214
215     title('$H_x$, $H_y$ Normed to Max overall Positions', ...
216         'FontWeight', 'normal', ...
217         'FontSize', 12, ...
218         'FontName', 'Times', ...
219         'Interpreter', 'latex');
220
221 % Cosinus, sinus voltage scaled to overall maxima
222 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
223
224 nexttile;
225 hold on;
226 for i = 1:N
227     for j = 1:N
228         plot(squeeze(VcosScaled(i, j, :)) + X(i, j), ...
229             squeeze(VsinScaled(i, j, :)) + Y(i, j), ...
230             'Color', [R(i, j) G(i, j) B(i, j)], ...
231             'LineWidth', 1.5)
232         line([X(i, j), VcosScaled(i, j, 1) + X(i, j)], ...
233             [Y(i, j), VsinScaled(i, j, 1) + Y(i, j)], ...
234             'Color', 'k', 'LineWidth', 1.5)
235     end
236 end
237
238 % scatter magnet x,y position (0,0,z)
239 scatter(0, 0, 32, 'r', 'filled');
240
241 hold off;
```

```
241
242     % axis shape and ticks
243     axis square xy;
244     axis tight;
245     grid on;
246     xlim([minX maxX]);
247     ylim([minY maxY]);
248
249     xlabel('$X$ in mm', ...
250         'FontWeight', 'normal', ...
251         'FontSize', 12, ...
252         'FontName', 'Times', ...
253         'Interpreter', 'latex');
254
255     ylabel('$Y$ in mm', ...
256         'FontWeight', 'normal', ...
257         'FontSize', 12, ...
258         'FontName', 'Times', ...
259         'Interpreter', 'latex');
260
261     title('$V_{\cos}$, $V_{\sin}$ Normed to Max overall Positions
262         ', ...
263         'FontWeight', 'normal', ...
264         'FontSize', 12, ...
265         'FontName', 'Times', ...
266         'Interpreter', 'latex');
267
268     %
269     %% Field strength normed each maxima at position
270     %% nexttile;
271     %% hold on;
272     %% for i = 1:N
273     %%     for j = 1:N
```

```
273 plot(squeeze(HxNorm(i, j, :)) + X(i,j), ...
274     squeeze(HyNorm(i, j, :)) + Y(i,j), ...
275     'Color', [R(i,j) G(i,j) B(i,j)], ...
276     'LineWidth', 1.5)
277 line([X(i,j), HxNorm(i,j,1) + X(i,j)], ...
278       [Y(i,j), HyNorm(i,j,1) + Y(i,j)], ...
279       'Color','k','LineWidth',1.5)
280 end
281 end
282
283 % scatter magnet x,y position (0,0,z)
284 scatter(0, 0, 32, 'r', 'filled');
285
286 hold off;
287
288 % axis shape and ticks
289 axis square xy;
290 axis tight;
291 grid on;
292 xlim([minX maxX]);
293 ylim([minY maxY]);
294
295 xlabel('$X$ in mm', ...
296     'FontWeight', 'normal', ...
297     'FontSize', 12, ...
298     'FontName', 'Times', ...
299     'Interpreter', 'latex');
300
301 ylabel('$Y$ in mm', ...
302     'FontWeight', 'normal', ...
303     'FontSize', 12, ...
304     'FontName', 'Times', ...
305     'Interpreter', 'latex');
306
307 title('$H_x$, $H_y$ Normed to Max at each Position', ...
308     'FontWeight', 'normal', ...
```

```
309     'FontSize', 12, ...
310     'FontName', 'Times', ...
311     'Interpreter', 'latex');
312
313 % Cosinus, sinus voltage normed to each maxima at position
314 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
315
316 nexttile;
317 hold on;
318 for i = 1:N
319     for j = 1:N
320         plot(squeeze(VcosNorm(i, j, :)) + X(i, j), ...
321               squeeze(VsinNorm(i, j, :)) + Y(i, j), ...
322               'Color', [R(i, j) G(i, j) B(i, j)], ...
323               'LineWidth', 1.5)
324         line([X(i, j), VcosNorm(i, j, 1) + X(i, j)], ...
325               [Y(i, j), VsinNorm(i, j, 1) + Y(i, j)], ...
326               'Color', 'k', 'LineWidth', 1.5)
327     end
328 end
329
330 % scatter magnet x,y position (0,0,z)
331 scatter(0, 0, 32, 'r', 'filled');
332
333 hold off;
334
335 % axis shape and ticks
336 axis square xy;
337 axis tight;
338 grid on;
339 xlim([minX maxX]);
340 ylim([minY maxY]);
341 xlabel('$x$ in mm', ...
```

```
342     'FontWeight', 'normal', ...
343     'FontSize', 12, ...
344     'FontName', 'Times', ...
345     'Interpreter', 'latex');

346
347 ylabel('$Y$ in mm', ...
348     'FontWeight', 'normal', ...
349     'FontSize', 12, ...
350     'FontName', 'Times', ...
351     'Interpreter', 'latex');

352
353 title('$V_{\cos}$, $V_{\sin}$ Normed to Max at each Positions
354     ', ...
355     'FontWeight', 'normal', ...
356     'FontSize', 12, ...
357     'FontName', 'Times', ...
358     'Interpreter', 'latex');

359 % save figure to file
360 %
361 % get file path to save figure with angle index
362 [~, fName, ~] = fileparts(ds.Info.filePath);
363
364 % save to various formats
365 yesno = input('Save? [y/n]: ', 's');
366 if strcmp(yesno, 'y')
367     fLabel = input('Enter file label: ', 's');
368     fName = fName + "_CirclePlot_" + fLabel;
369     savefig(fig, fullfile(fPath, fName));
370     print(fig, fullfile(fPath, fName), '-dsvg');
371     print(fig, fullfile(fPath, fName), '-depsc', '-tiff', '
372         -loose');
```

```
372     print(fig, fullfile(fPath, fName), '-dpdf', '-loose', '  
373         -fillpage');  
374     end  
375     close(fig);  
376 end
```

## E.5 Datasets

### 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.

### Contents

- TDK TAS2141 Characterization
- NXP KMZ60 Characterization
- Config Mat
- Training and Test Datasets

### 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.

### E.5.1 TDK TAS2141 Characterization

## 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.

## Contents

- See Also
- Magnetic Stimulus
- Cosine Bridge Output
- Sine Bridge Output
- Operating Point
- Dataset Structure

## 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:

$$m(t) = H_{max} \cdot \text{tri}(t) = H_{max} \cdot \text{tri}(2(t - t_0)f_m)$$

$$t_0 = \frac{1}{2f_m}$$

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 field-strength 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 pre-processed 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 built-in save function in a certain way to perform partial loads from the dataset.

```
1 save('data/TDK_TAS2141_Characterization_2020-10-22_18  
-12-16-827.mat', ...  
2     'Info', 'Data', '-v7.3', '-nocompression')
```

Created on October 27. 2020 Tobias Wulf. Copyright Tobias Wulf 2020.

### E.5.2 NXP KMZ60 Characterization

## 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 Schütte 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 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 characterization fields generates two sinoid periods related to current angle position between 0° and 180° or 180° and 360°.

## Contents

- See Also
- Magnetic Stimulus
- Cosinuns Bridge Output
- Sine Bridge Output
- Operating Point
- Dataset Structure

## 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:

$$m(t) = H_{max} \cdot \text{tri}(t) = H_{max} \cdot \text{tri}(2(t - t_0)f_m)$$

$$t_0 = \frac{1}{2f_m}$$

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 field-strength 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 pre-processed 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.

```
1 save('data/NXP_KMZ60_Characterization_2020-12-03_16-53-16-721.
      mat', ...
2      'Info', 'Data', '-v7.3', '-nocompression')
```

Created on December 05. 2020 Tobias Wulf. Copyright Tobias Wulf 2020.

### E.5.3 Config Mat

## 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.

```
1 load('config.mat')
```

## Contents

- Requirements
- See Also

## 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.

#### **E.5.4 Training and Test Datasets**

### **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 matrices 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.

### **Contents**

- Dataset Structure
- See Also

## 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
- VsineRef - 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 configure 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.

## E.6 Unit Tests

### 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.

### Contents

- runTests
- removeFilesFromDirTest
- rotate3DVectorTest
- generateDipoleRotationMomentsTest
- generateSensorArraySquareGridTest
- computeDipoleH0NormTest
- computeDipoleHFieldTest
- tiltRotationTest
- Requirements
- See Also

#### 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.

### E.6.1 runTests

#### runTests

```
1 % clean workspace
2 clearvars;
3
4 % build suite from test files
5 suite = testsuite({'removeFilesFromDirTest', '
    rotate3DVectorTest', ...
    'generateDipoleRotationMomentsTest', ...
    'generateSensorArraySquareGridTest', ...
    'computeDipoleH0NormTest', ...
    'computeDipoleHFieldTest', ...
    'tiltRotationTest'});
6
7 % run tests
8 results = run(suite);
9
10 % show results
11 disp(results)
12 disp(table(results))
13 cd ..
```

## E.6.2 removeFilesFromDirTest

### Contents

- Test 1: delete all files
- Test 2: delete with pattern

```
1 % create test directory with files
2 cd(fileparts(which('removeFilesFromDirTest')));
3 mkdir('testDir');
4 fclose(fopen(fullfile('testDir', 'testFile1.txt'), 'w'));
5 fclose(fopen(fullfile('testDir', 'testFile2.txt'), 'w'));
6 fclose(fopen(fullfile('testDir', 'testFile3.txt'), 'w'));
```

#### Test 1: delete all files

```
1 removeStatus = removeFilesFromDir(fullfile('testDir'));
2 assert(removeStatus == true)
3
4 % create more files
5 fclose(fopen(fullfile('testDir', 'testFile1.txt'), 'w'));
6 fclose(fopen(fullfile('testDir', 'testFile2.txt'), 'w'));
7 fclose(fopen(fullfile('testDir', 'testFile3.txt'), 'w'));
```

#### Test 2: delete with pattern

```
1 removeStatus = removeFilesFromDir(fullfile('testDir'), '*.txt')
;
2 assert(removeStatus == true)
3
4 % clean up
```

```
5 | rmdir('testDir');
```

### E.6.3 rotate3DVectorTest

## Contents

- Test 1: output dimensions
- Test 2: rotate vectors in x-axes
- Test 3: rotate vectors in y-axes
- Test 4: rotate vectors in z-axes

```
1 % create column vectors with simple direction for rotations
  along the axes
2 % without tilts in other achses.
3 x = [-1; 0; 0];
4 y = [0; -1; 0];
5 z = [0; 0; -1];
6
7 % set angle step width in degree to rotate at choosen axes (x,
  y, or z)
8 angle = 90;
```

### Test 1: output dimensions

```
1 rotated = rotate3DVector(x, 0, 0, angle);
2 assert(isequal(size(rotated), [3, 1]))
3 rotated = rotate3DVector([x x x x x x], 0, 0, angle);
4 assert(isequal(size(rotated), [3, 6]))
```

### Test 2: rotate vectors in x-axes

```
1 rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree
2 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
3
4 rotated = rotate3DVector([x y z], angle, 0, 0); % 90 degree
5 assert(isequal(rotated, [-1 0 0; 0 0 1; 0 -1 0]))
6
7 rotated = rotate3DVector([x y z], 2 * angle, 0, 0); % 180
8     degree
9 assert(isequal(rotated, [-1 0 0; 0 1 0; 0 0 1]))
10
11 rotated = rotate3DVector([x y z], 3 * angle, 0, 0); % 270
12     degree
13 assert(isequal(rotated, [-1 0 0; 0 0 -1; 0 1 0]))
14
15 rotated = rotate3DVector([x y z], 4 * angle, 0, 0); % 360
16     degree
17 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

### Test 3: rotate vectors in y-axes

```
1 rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree
2 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
3
4 rotated = rotate3DVector([x y z], 0, angle, 0); % 90 degree
5 assert(isequal(rotated, [0 0 -1; 0 -1 0; 1 0 0]))
6
7 rotated = rotate3DVector([x y z], 0, 2 * angle, 0); % 180
8     degree
9 assert(isequal(rotated, [1 0 0; 0 -1 0; 0 0 1]))
10
11 rotated = rotate3DVector([x y z], 0, 3 * angle, 0); % 270
12     degree
13 assert(isequal(rotated, [0 0 1; 0 -1 0; -1 0 0]))
```

```
12
13 rotated = rotate3DVector([x y z], 0, 4 * angle, 0); % 360
   degree
14 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

#### Test 4: rotate vectors in z-axes

```
1 rotated = rotate3DVector([x y z], 0, 0, 0); % 0 degree
2 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
3
4 rotated = rotate3DVector([x y z], 0, 0, angle); % 90 degree
5 assert(isequal(rotated, [0 1 0; -1 0 0; 0 0 -1]))
6
7 rotated = rotate3DVector([x y z], 0, 0, 2 * angle); % 180
   degree
8 assert(isequal(rotated, [1 0 0; 0 1 0; 0 0 -1]))
9
10 rotated = rotate3DVector([x y z], 0, 0, 3 * angle); % 270
    degree
11 assert(isequal(rotated, [0 -1 0; 1 0 0; 0 0 -1]))
12
13 rotated = rotate3DVector([x y z], 0, 0, 4 * angle); % 360
   degree
14 assert(isequal(rotated, [-1 0 0; 0 -1 0; 0 0 -1]))
```

#### E.6.4 generateDipoleRotationMomentsTest

## Contents

- Test 1: output dimensions
- Test 2: down sampling
- Test 3: phase shift

```
1 % create full scale rotation with 0.5 resolution and no tilt,
2 % return moments
3 % and corresponding angles theta
4 amp = 1e6;
5 tilt = 0;
6 res = 0.5;
7 [MFS, tFS] = generateDipoleRotationMoments(amp, 0, tilt, res);
8
9 % create same rotation but only a subset of angles N = 7
10 % with equal distances to each and another, return additionally
   index which
11 % reference to full scale
12 [M, t, idx] = generateDipoleRotationMoments(amp, 7, tilt, res);
13
14 % create shifted subset, shift by 22 positions in full scale
   theta,
15 % so with 0.5 resolution it is phase shift by 11
16 [MSH, tSH, idxSH] = generateDipoleRotationMoments(amp, 7, tilt,
   res, 22);
```

### Test 1: output dimensions

```
1 assert(isequal(size(MFS), [3 720]))
2 assert(isequal(size(tFS), [1 720]))
3 assert(isequal(size(M), [3 7]))
4 assert(isequal(size(t), [1 7]))
5 assert(isequal(size(idx), [1 7]))
6 assert(isequal(size(MSH), [3 7]))
7 assert(isequal(size(tSH), [1 7]))
8 assert(isequal(size(idxSH), [1 7]))
```

### Test 2: down sampling

```
1 assert(isequal(MFS(:,idx), M))
2 assert(isequal(tFS(idx), t))
3 assert(isequal(MFS(:,idxSH), MSH))
4 assert(isequal(tFS(idxSH), tSH))
```

### Test 3: phase shift

```
1 assert(isequal(tSH(1), 11))
2 assert(isequal(idx, idxSH - 22))
3 assert(isequal(MFS(:,idx + 22), MSH))
4 assert(isequal(tFS(idx + 22), tSH))
```

### E.6.5 generateSensorArraySquareGridTest

## Contents

- Test 1: output dimensions
- Test 2: equal x and y distances
- Test 3: constant z distances
- Test 3: position shift in x and y direction

```
1 % create sensor array infos for size and position
2 % number of sensors at one edge
3 N = 8;
4
5 % sensor array edge length in mm
6 a = 2;
7
8 % relative position of the sensor array to the center of a 3D
% coordinate
9 % system (z inverse)
10 p = [0; 0; 2];
11
12 % z offset, later used as sphere radius of a dipole which is
% placed in the
13 % center of the coordinate system
14 r = 2;
15
16 % generate coordinates grid
17 [X, Y, Z] = generateSensorArraySquareGrid(N, a, p, r);
18
19 % create a shift in same layer
20 p2 = [-2; 3; 2];
21 [X2, Y2, Z2] = generateSensorArraySquareGrid(N, a, p2, r);
```

### Test 1: output dimensions

```
1 assert(isequal(size(X), [N N]))  
2 assert(isequal(size(Y), [N N]))  
3 assert(isequal(size(Z), [N N]))
```

### Test 2: equal x and y distances

```
1 assert(isequal(diff(Y), diff(-X, [], 2)'))
```

### Test 3: constant z distances

```
1 assert(all(Z == -(p(3) + r), 'all'))
```

### Test 3: position shif in x and y direction

```
1 assert(isequal(X + p2(1), X2))  
2 assert(isequal(Y + p2(2), Y2))  
3 assert(isequal(Z, Z2))
```

### E.6.6 computeDipoleH0NormTest

## Contents

- Test 1: positive scalar factor

```
1 % create a dipole with constant sphere radius in rest position
  and relative
2 % to sensor array with position x=0, y=0, z=0
3 % sphere radius 2mm
4 r = 2;
5 % distance in which the field strength is imprinted
6 z = 5;
7 % field strength magnitude to imprint in dipole field on sphere
  radius kA/m
8 Hmag = 8.5;
9 % magnetic moment magnitude which rotates the dipole without
  tilt
10 Mmag = 1e6;
11
12 % compute norm factor
13 H0norm = computeDipoleH0Norm(Hmag, [Mmag; 0; 0], [0; 0; -(z + r
  )]);
```

### Test 1: positive scalar factor

```
1 assert(isscalar(H0norm))
2 assert(H0norm > 0)
```

### E.6.7 computeDipoleHFieldTest

## Contents

- Test 1: output dimensions
- Test 2: center of field
- Test 3: magnetization
- Test 4: imprinting
- Test 5: symmetry
- Test 6: units milli kilo

```
1 % compute a single point without norming
2 Hsingle = computeDipoleHField(1, 2, 3, [1; 0; 0], 1);
3
4 % compute a 3D grid of positions n+1 samples for even values
5 % in the grid and to
6 % include (0,0,0), in mm
7 x = linspace(-4, 4, 41);
8 y = linspace(4, -4, 41);
9 z = linspace(4, -4, 41);
10 [X, Y, Z] = meshgrid(x, y, z);
11
12 % magnetic dipole moment to define magnet orientation, no tilt
13 m = generateDipoleRotationMoments(-1e6, 1, 0);
14
15 % norm factor to imprint field strength in certain distance d =
16 % 1,
17 % r = 2 in mm,
18 % 200 kA/m, no tilt
19 r0 = rotate3DVector([0; 0; -3], 0, 0, 0);
20 H0norm = computeDipoleH0Norm(200, m, r0);
```

```
21 % allocate memory for field components in x,y,z
22 Hx = zeros(41, 41, 41);
23 Hy = zeros(41, 41, 41);
24 Hz = zeros(41, 41, 41);
25
26 % compute without norming for each z layer and reshape results
27 % into layer
27 for i=1:41
28     Hgrid = computeDipoleHField(X(:,:,:i),Y(:,:,:i),Z(:,:,:i),m,
29         H0norm);
30     Hx(:,:,:,i) = reshape(Hgrid(1,:),41,41);
31     Hy(:,:,:,i) = reshape(Hgrid(2,:),41,41);
32     Hz(:,:,:,i) = reshape(Hgrid(3,:),41,41);
33 end
34
35 % calculate magnitude in each point for better view the results
36 Habs = sqrt(Hx.^2+Hy.^2+Hz.^2);
37
38 % define a index to view only every 4th point for not
39 % overcrowded plot
40 idx = 1:4:41;
41
42 % downsample and norm
43 Xds = X(idx,idx,idx);
44 Yds = Y(idx,idx,idx);
45 Zds = Z(idx,idx,idx);
46 Hxds = Hx(idx,idx,idx) ./ Habs(idx,idx,idx);
47 Hyds = Hy(idx,idx,idx) ./ Habs(idx,idx,idx);
48 Hzds = Hz(idx,idx,idx) ./ Habs(idx,idx,idx);
49
50 % show results for test, comment out for regular unittest run,
51 % run suite
52 quiver3(Xds, Yds, Zds, Hxds, Hyds, Hzds);
53 xlabel('$X$ in mm', ...
54     'FontSize', 16, ...
```

```
53     'FontName', 'Times', ...
54     'Interpreter', 'latex');
55 ylabel('$Y$ in mm', ...
56     'FontWeight', 'normal', ...
57     'FontSize', 16, ...
58     'FontName', 'Times', ...
59     'Interpreter', 'latex');
60 zlabel('$Z$ in mm', ...
61     'FontWeight', 'normal', ...
62     'FontSize', 16, ...
63     'FontName', 'Times', ...
64     'Interpreter', 'latex');
65 title('Dipole H-Field - Equation Test', ...
66     'FontWeight', 'normal', ...
67     'FontSize', 18, ...
68     'FontName', 'Times', ...
69     'Interpreter', 'latex');
70 subtitle('$X$-, $Y$-, $Z$-Meshgrid from $-4 \ldots 4$ mm', ...
71     'FontWeight', 'normal', ...
72     'FontSize', 14, ...
73     'FontName', 'Times', ...
74     'Interpreter', 'latex');
75 axis equal;
76
77 % pattern for logical indexing the center or opposite
78 p0 = false(1, 41);
79 p0(21) = true;
80 pN0 = true(41, 41, 41);
81 pN0(21,21, 21) = false;
82
83 % pattern for symmetry investigation
84 plu = [true(1, 20), false, false(1, 20)];
85 prl = [false(1, 20), false, true(1, 20)];
86
87 % compare values to check if fits in unit pairs of m and A/m
88 % and mm and kA/m
```

```

89 r0Apm = rotate3DVector([0; 0; -3e-3], 0, 0, 0);
90 H0normApm = computeDipoleH0Norm(200e3, m, r0Apm);
91 Xm = X * 1e-3;
92 Ym = Y * 1e-3;
93 Zm = Z * 1e-3;
94 HxApm = zeros(41, 41, 41);
95 HyApm = zeros(41, 41, 41);
96 HzApm = zeros(41, 41, 41);
97 for i=1:41
98     HApm = computeDipoleHField(Xm(:,:,i), Ym(:,:,i), Zm(:,:,i), m,
99                                 H0normApm);
100    HxApm(:,:,:,i) = reshape(HApm(1,:,:), 41, 41);
101    HyApm(:,:,:,i) = reshape(HApm(2,:,:), 41, 41);
102    HzApm(:,:,:,i) = reshape(HApm(3,:,:), 41, 41);
103 end
104 HabsApm = sqrt(HxApm.^2+HyApm.^2+HzApm.^2);

```

**Test 1: output dimensions**

```

1 assert(isequal(size(Hsingle), [3, 1]))
2 assert(isequal(size(Hgrid), [3, 1681]))

```

**Test 2: center of field**

```

1 assert(X(p0,p0,p0) == 0)
2 assert(Y(p0,p0,p0) == 0)
3 assert(Z(p0,p0,p0) == 0)
4 assert(isnan(Hx(p0,p0,p0)))
5 assert(isnan(Hy(p0,p0,p0)))
6 assert(isnan(Hz(p0,p0,p0)))
7 assert(all(Hx(~p0,p0,p0) ~= 0, 'all'))
8 assert(all(Hx(p0,~p0,p0) ~= 0, 'all'))
9 assert(all(Hy(~p0,p0,p0) == 0, 'all'))
10 assert(all(Hy(p0,~p0,p0) == 0, 'all'))

```

```
11 | assert(all(Hz(~p0,~p0,p0) == 0, 'all'))
```

### Test 3: magnetization

```
1 assert(all(Hx(~p0,p0,~p0) ~= 0, 'all'))
2 assert(all(Hx(p0,~p0,~p0) ~= 0, 'all'))
3 assert(all(Hx(p0,p0,~p0) ~= 0, 'all'))
4 assert(all(Hy(~p0,p0,~p0) == 0, 'all'))
5 assert(all(Hy(p0,~p0,~p0) == 0, 'all'))
6 assert(all(Hy(p0,p0,~p0) == 0, 'all'))
7 assert(all(Hz(~p0,p0,~p0) == 0, 'all'))
8 assert(all(Hz(p0,~p0,~p0) ~= 0, 'all'))
9 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

```
1 assert(round(abs(Hx(p0,p0,6)),6) == 200)
2 assert(round(abs(Hx(p0,p0,36)),6) == 200)
```

### Test 5: symmetry

```
1 assert(all((Hx(plu,:,:)-flip(Hx(prl,:,:),1))==0, 'all'))
2 assert(all((Hx(:,plu,:)-flip(Hx(:,prl,:),2))==0, 'all'))
3 assert(all((Hy(plu,:,:)+flip(Hy(prl,:,:),1))==0, 'all'))
4 assert(all((Hy(:,plu,:)+flip(Hy(:,prl,:),2))==0, 'all'))
5 assert(all((Hz(:, :, ~p0)+flip(Hz(:, :, ~p0), 2))==0, 'all'))
```

### Test 6: units milli kilo

```
1 assert(all(round(HxApm(pN0) * 1e-3, 6) == round(Hx(pN0), 6),
2           'all'))
3 assert(all(round(HyApm(pN0) * 1e-3, 6) == round(Hy(pN0), 6),
4           'all'))
5 assert(all(round(HzApm(pN0) * 1e-3, 6) == round(Hz(pN0), 6),
6           'all'))
7 assert(all(round(HabsApm(pN0) * 1e-3, 6) == round(Habs(pN0), 6),
8           'all'))
```

### E.6.8 tiltRotationTest

#### Contents

- Test 1: rotation without tilt
- Test 2: rotation with tilt

```
1 % clean
2 clearvars;
3
4 % relevant tilt in y axes
5 tilt = 0.5:0.5:90;
6
7 % magnetic dipole moment to define magnet orientation, no tilt
8 % rotate angles theta 0, 90, 180, 270
9 [mNoTilt, thetaNoTilt] = generateDipoleRotationMoments(-1e6, 4,
    0);
10
11 % Habs for magnetization from north to south from -x to x
12 HabsMust = [400 400 200 200 200 200];
13
14 % norm factor to imprint field strength in certain distance d =
    1,
15 % r = 2 in mm,
16 % 200 kA/m, no tilt
17 r0NoTilt = [0; 0; -3];
18 H0normNoTilt = computeDipoleH0Norm(200, mNoTilt(:,1), r0NoTilt)
    ;
19
20 % axes with no tilt, rest position
21 AxesNoTilt = [-3, 3, 0, 0, 0, 0;
    0, 0, -3 3, 0, 0;
    0, 0, 0, 0, -3, 3];
```

```
24
25 % calc fields along coordinate cross and magnitudes
26 HNoTilt = zeros(3, 6, 4);
27 for i = 1:4
28     % rotate axes same wise to check pole values
29     RotateNoTiltAxes = rotate3DVector(AxesNoTilt, 0, 0,
30         thetaNoTilt(i));
31     HNoTilt(:, :, i) = computeDipoleHField(RotateNoTiltAxes(1, :),
32         ...
33         RotateNoTiltAxes(2, :), RotateNoTiltAxes(3, :), ...
34         mNoTilt(:, i), H0normNoTilt);
35 end
36
37 % habs must be show imprinted field strength and double of it
38 % at poles
39 HabsNoTilt = sqrt(sum(HNoTilt.^2, 1));
40
41 % calc fields along tilt coordinate cross and magnitudes
42 HTilt = zeros(3, 6, 4, length(tilt));
43 for j = 1:length(tilt)
44     % magnetic dipole moment to define magnet orientation, with
45     % tilt
46     % rotate angles theta 0, 90, 180, 270
47     [mTilt, thetaTilt] = generateDipoleRotationMoments(-1e6, 4,
48             tilt(j));
49
50     % norm factor to imprint field strength in certain distance
51     d = 1,
52     % r = 2 in mm,
53     % 200 kA/m, no tilt
54     r0Tilt = rotate3DVector(r0NoTilt, 0, tilt(j), 0);
55     H0normTilt = computeDipoleH0Norm(200, mTilt(:, 1), r0Tilt);
56
57     % axes with tilt, rest position
58     AxesTilt = rotate3DVector(AxesNoTilt, 0, tilt(j), 0);
```

```
54 for i = 1:4
55     % rotate axes same wise to check pole values
56     RotateTiltAxes = rotate3DVector(AxesTilt, 0, 0,
57         thetaTilt(i));
58     HTilt(:, :, i, j) = computeDipoleHField(RotateTiltAxes
59         (1, :), ...
60         RotateTiltAxes(2, :), RotateTiltAxes(3, :), ...
61         mTilt(:, i), H0normTilt);
62 end
63 % habs must be show imprinted field strength and double of it
64 % at poles
65 HabsTilt = sqrt(sum(HTilt.^2, 1));
```

### Test 1: rotation without tilt

```
1 assert(all(round(HabsNoTilt, 6) == round(HabsMust, 6), 'all'))
```

### Test 2: rotation with tilt

```
1 assert(all(round(HabsTilt, 6) == round(HabsMust, 6), 'all'))
```

**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