DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR

Konzeption und Umsetzung von skalierbaren Algorithmen zur modellbasierten Antennenkalibrierung für mehrkanalige SAR Sensoren auf verteilten HPC Systemen

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informationstechnik

der Dualen Hochschule Baden-Württemberg Mannheim

von

Felix Weinmann

Abgabe am 14. September 2021

Bearbeitungszeitraum:	14.06.2021 – 13.09.2021
Matrikelnummer, Kurs:	2891832, TINF18-IT1
Abteilung:	Abteilung SAR-Technologie, Institut für Hochfrequenztechnik und Radarsysteme
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Betreuer der Ausbildungsfirma:	M. Sc. Marc Jäger
Gutachter der Dualen Hochschule:	Jürgen Schultheis

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

Thema

Konzeption und Umsetzung von skalierbaren Algorithmen zur modellbasierten Antennenkalibrierung für mehrkanalige SAR Sensoren auf verteilten HPC Systemen

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Oberpfaffenhofen, den 3. September 2021

Kurzfassung

Mehr Aufnahmekanäle und genauere Daten machen in der Erdbeobachtung mit dem Radar mit synthetischer Apertur präzisere Modelle und besser skalierende Implementierungen von Kalibrierverfahren notwendig. Ein modellbasiertes Kalibrierverfahren wird auf der Basis einer bestehenden Funktionalität zur Rohdatenanalyse in einer verteilten Spark-Umgebung umgesetzt. Diese Auswertung der Analyseergebnisse bestimmt die dreidimensionale Position des Antennenphasenzentrums unter Berücksichtigung der optimalen Kanalkonsistenz, die Antennenblickrichtungswinkel sowie die Basislinien der einzelnen Antennen untereinander. Zum effizienten Aufstellen und Lösen der Modelle wird die Methode der kleinsten Quadrate angewendet und darauf aufbauend zur Verbesserung des Verhaltens mit realen Messdaten eine L1-Norm Minimierung entwickelt. Die Korrektheit der Implementierung der einzelnen Komponenten sowie der Gesamtimplementierung wird verifiziert. Eine vergleichbare Genauigkeit bei einer um eine Größenordnung geringeren Laufzeit als in der bestehenden Referenzimplementierung wird erzielt.

Abstract

More recording channels and more precise data in the observation with synthetic aperture radar requires more precise models and better scaling implementations of calibration procedures. A model based calibration procedure based on existing functionality for analysing the raw data will be implemented in a distributed Spark environment. The evaluation of the analysis results determins the three dimensional phase center position under consideration of the optimal channel consistency, the antenna mount angles and the baseline between the antennas. A least squares solver for efficient model formulation and solving will be developed. Based on this a l1 norm minimisation for improving the behaviour with real data will be implemented. The propriety of the implementation will be verified for each of the components as well as the complete implementation. In comparison to the existing reference implementation a comparable precision with an order of magnitude reduced runtime will be achieved.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Formelgrößenverzeichnis

β	rad	Squint-Winkel
δl	m	Versatz der Phasenzentrumsposition (Hebelarm)
δr	m	relative Differenz in der Antwortposition
δt	m	konstanter Entfernungs-Versatz
Δr	m	absolute Differenz in der Antwortposition
ϕ	m	Signalphase (umgerechnet in Range)
θ	rad	Off-Nadir Winkel
A		Gewichte des Gleichungssystems
b		Ergebnisse des Gleichungssystems
c	m	Konstanter Versatz
f	Hz	Frequenz
f_0	Hz	mittlere Signalfrequenz
g	dB	Verstärkung (Gain)
los	m	Einheitsvektor in Blickrichtung (line of sight)
n		Anzahl der Azimut-Samples
p_{ypr}	rad	Antennenblickrichtung in Gier- (yaw), Nick- (pitch) und Rollachse (roll)
r	m	Abstand des Punktziels zum Sensor
t_{az}	s	Azimut-Zeit
W		Gewicht zur iterativen L1-Normierung
x		Unbekannte des Gleichungssystems

Abkürzungsverzeichnis

CARAMBA	**C**luster **A**rchtecture for **R**adar **A**nalysis, **M**ulti-processing and **B**eamforming **A**pplications
DGPS	**D**ifferential **G**lobal **P**ositioning **S**ystem
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
F-SAR	**F**lugzeug-**SAR** Sensor
IDL	**I**nteractive **D**ata **L**anguage
RDD	Verteilter Datensatz (**R**esilient **D**istributed **D**ataset)
SAR	Radar mit synthetischer Apertur (**S**ynthetic **A**perture **R**adar)
SSE	Summe der quadrierten Fehler (**S**um of **S**quared **E**rrors)

1 Einleitung

Mit dem Radar mit synthetischer Apertur (SAR) können großflächige 2D- und 3D-Messaufnahmen getätigt werden [1]. Im SAR ist es zum Zweck der Fernerkundung von Bedeutung, dass sich das beobachtete Ziel relativ zur Radarantenne bewegt. Durch die seitliche Blickrichtung entlang der Flugbahn kann anhand des Antwortzeitpunktes eines Ziels die Distanz des Ziels - die sogenannte Range - vom Radargerät bestimmt werden. Das regelmäßige Wiederholen des Sendeimpulses sorgt dafür, dass ein Punktziel mit einer veränderten Distanz mehrfach eine Antwort sendet, wodurch auch entlang der Flugrichtung - im Azimut - ein Ziel fokussiert werden kann. Um wissenschaftlich verwendbare Ergebnisse erzielen zu können, ist die externe Kalibrierung der Antennenmodellparameter notwendig, da bei der Verarbeitung der Rohdaten zum fertigen Bild möglichst alle systematischen Einflüsse des Sensors, also auch der Antenne, kompensiert werden sollen. Kleinste Abweichungen im physikalischen Aufbau des Radarsystems führen im Endergebnis zu größeren Abweichungen. Zunehmend durch bessere Auflösungen und mehr Aufnahmekanäle bedingte größere Datenmengen bremsen bisherige Implementierungen von externen Kalibrierverfahren aus. Die in dieser Arbeit implementierte Rohdatenkalibrierung [2] nutzt dabei sogenannte Winkelreflektoren, welche in SAR-Bildern eine besonders starke Rückantwort liefern und dadurch leicht in den Daten zu erkennen sind. Die derzeitige Implementierung der externen Rohdatenkalibrierung kann nur über einzelne Empfangskanäle parallelisieren, nicht jedoch über die einzelnen Corner-Reflektoren bzw. Punktziele. Da es sich dabei um eine statische Parallelisierung handelt, bei der jedem Prozess ein Kanal zugeordnet ist, können die zur Verfügung stehenden Rechenressourcen nicht optimal genutzt werden. CARAMBA, das auf PySpark[3] aufbauende Framework der Abteilung SAR-Technologie, bietet neben SAR-spezifischer Funktionalität Routinen zur Parallelisierung von Operationen. Dabei wird auf Spark aufgebaut,

dessen verteilte Datensätze (RDDs) verteilte Rechenoperationen auch über mehrere Rechenknoten hinweg ermöglichen.

In dieser Arbeit befindliche Quellcodestücke sind in der Programmiersprache Python 3 geschrieben.

2 Aufgabenstellung

1. Implementierung und Verifikation der Hebelarmkorrektur mithilfe der Methode der kleinsten Quadrate
   - 1.1 Implementierung
   - 1.2 Verifikation
2. Implementierung und Verifikation der optimalen Kanalkonsistenz
   - 2.1 Implementierung
   - 2.2 Performanceoptimierung
   - 2.3 Verifikation
3. Implementierung und Verifizierung einer L1-Norm Regularisierung
   - 3.1 Implementierung
   - 3.2 Verifikation
4. Implementierung und Verifikation einer Antennenblickrichtungskorrektur
   - 4.1 Implementierung
   - 4.2 Verifikation
5. Implementierung und Verifikation der Antennenbasislinienkorrektur
   - 5.1 Implementierung
   - 5.2 Performanceoptimierung

   5.3 Verifikation

6. Vergleich der Implementierung mit der existierenden IDL-Implementierung

   6.1 Erzielen einer vergleichbaren Genauigkeit

   6.2 Erzielen einer um eine Größenordnung geringeren Laufzeit

3 Methoden und Verfahren

Entwicklungsserver: 36-Core x86-64 CPU, 376 GiB RAM

- Kubuntu 18.04
- CARAMBA 606b2a47
- CPython 3.9
- Spark 3.0.1
- pyspark 3.0.1
- numpy 1.21.1
- scipy 1.7.0
- matplotlib 3.4.2

Prozessierungscluster: 6 x86-64 Server

- Ubuntu 20.04.2
- CARAMBA 606b2a47
- CPython 3.9.5
- Spark 3.0.1
- pyspark 3.0.1
- numpy 1.21.1
- scipy 1.7.0
- matplotlib 3.4.2

Um die angestrebte starke Performanceoptimierung zu erzielen ist es notwendig, mithilfe von Profiling langsame Codesegmente zu bestimmen und zu beseitigen. Unter den Profilern gibt es zwei Arten, welche unterschiedliche Einblicke liefern. Maschinennahe Profiler wie Intel VTune[4] liefern Informationen, wie viel Zeit einzelne Subroutinen und Assembler-Instruktionen benötigen, während interpreternahe Profiler wie cProfile[5] besser darin sind, die Zeitdauer von Operationen in interpretierten Sprachen zu messen. Ergebnisse interpreternaher Profiler unterliegen dabei immer gewissen Schwankungen durch externe Einflüsse wie z.B. der Scheduler des Betriebssystems, während maschinennahe Profiler unabhängig von solchen Ereignissen arbeiten. Da es bei der Optimierung der Implementierung in dieser Arbeit vorrangig darum geht, die Bottlenecks aufzulösen und nicht Ziel ist, eine Bibliothek in maschinennaher Sprache zu entwickeln oder zu verbessern, wurde der interpreternahe Profiler cProfile gewählt.

Eine PySpark-Umgebung ist besonders schwierig zu profilen, da dabei der Python-Hauptprozess je nach Situation einen Java-Controller-Prozess startet oder sich mit diesem verbindet. Dieser Controller startet dann selbst auf der gleichen oder anderen Maschinen wieder einen Python-Prozess, welcher dann die Python-Prozesse zur eigentlichen Datenverarbeitung startet. Zum Profiling der eigentlichen Datenverarbeitungsschritte bietet PySpark die Möglichkeit an, beim Start der Spark-Umgebung einen Profiler zu übergeben [6]. Dieser Profiler muss dabei lediglich einige Funktionen zur Verfügung stellen und kann mit cProfile verbunden werden, wie auch im BasicProfiler genutzt [7]. Der einzige Nachteil beim Profilen mit dieser Methode ist das fehlende Profiling der Spark-Architektur, wenn z.B. eine aufwändige Neusortierung der Daten notwendig wird. Zur grafischen Darstellung der Ergebnisse wurde eine modifizierte Fassung des Anzeigewerkzeuges gprof2dot[8] verwendet, welche neben den prozentualen Zeiten auch absolute Zeiten ausgibt.

4 Implementierung und Verifikation der modellbasierten Antennenkalibrierung

Die modellbasierte Antennenkalibrierung basierend auf Range-komprimierten Rohdaten ist ein mehrstufiges Verfahren zur Korrektur von regelmäßig in Flugzeug-SAR-Systemen auftretenden Abweichungen [2]. Durch den regelmäßigen Ein- und Ausbau des Radargerätes am Flugzeug weichen Eigenschaften von den erwarteten Werten ab. Diese Arbeit beinhaltet die Korrektur der Position des Phasenzentrums, welche durch die Hebelarme von der physischen Position der Antenne abweicht, sowie der Antennenblickrichtung. Vor den Kalibrierroutinen findet hierbei stets die Analyse statt, welche die aufgenommenen Rohdaten mit den erwarteten Modellen vergleicht und dabei Werte wie die Abstandsabweichung und die Antennengewinnabweichung über den Azimutverlauf bestimmt. Mit diesen Ergebnissen arbeiten die verschiedenen Kalibrierroutinen zur Bestimmung der Hebelarmunterschiede des Phasenzentrums sowie der Antennenblickrichtung. Nach der Anwendung einer Kalibrierroutine wird die Analyse und Kalibrierroutine mit den ermittelten Änderungen erneut durchgeführt, um die Konvergenz des Modells feststellen zu können sowie noch kleinere Verbesserungen am Modell vornehmen zu können. Da sich die Hebelarmkorrektur mit optimaler Kanalkonsistenz und die Antennenblickrichtungskorrektur kaum gegenseitig beeinflussen, können die beiden Kalibrierroutinen parallel mit den gleichen Analysedaten ausgeführt werden. Die Basislinienkorrektur kann erst nach dem Abschluss der Hebelarmbestimmung durchgeführt werden, da diese vorkalibrierte Hebelarme benötigt und die Hebelarmeunterschiede der einzelnen Antenen zueinander minimiert.

4.1 Implementierung und Verifikation der Hebelarmkorrektur mithilfe der Methode der kleinsten Quadrate

Als Vorbereitung auf diese Bachelorarbeit wurde bereits ein linearer Gleichungssystemlöser zur Hebelarmbestimmung implementiert [9]. Dieser Löser ist jedoch nicht generisch auch für andere Kalibrierungsschritte verwendbar und die Ergebnisse wurden nicht gegenüber einer anderen Implementierung des Kalibrierverfahrens verifiziert. Aus der Rohdatenanalyse [9] liegen die Line of Sight *los* und der gemessene relative Versatz δr in m vor, welche von der Azimutzeit t_{az} abhängig sind. Ziel ist die Berechnung der Differenz in der Phasenzentrumsposition δl, dem Hebelarmversatz. Diese kann mit dem aufnahmeabhängigen unbekannten Laufzeit δt bestimmt werden:

$$\delta r(t_{az}) = \delta l \cdot los(t_{az}) + \delta t$$

Durch die potentielle Verwendung verschiedener Antennen für das Aussenden (transmit) und den Empfang (receive) erweitert sich die Gleichung wie folgt:

$$\delta r(t_{az}) = \frac{1}{2}\delta l_t \cdot los_t(t_{az}) + \frac{1}{2}\delta l_r \cdot los_r(t_{az}) + \delta t$$

Mithilfe der sog. Ordinary Least Square-Methode kann dieses überbestimmte Gleichungssystem gelöst werden [10][11]:

$$b = A \cdot x$$

$$x = (A^T A)^{-1} \cdot A^T b$$

Dabei entsprechen A, x und b im Fall der Hebelarmkorrektur:

$$A = \begin{pmatrix} los_t & los_r & 1 \\ \vdots & \vdots & \vdots \end{pmatrix}$$

$$b = \delta r(t_{az})$$

$$x = \begin{pmatrix} \delta l_t \\ \delta l_r \\ \delta t \end{pmatrix}$$

4.1.1 Implementierung

Um alle Daten für die Hebelarmkorrektur einzubeziehen, müssen alle Antennen zeitgleich in das Gleichungssystem eingefügt werden, da es Kanäle mit unterschiedlicher Sende- und Empfangsantenne gibt. Bestehende Implementierungen eines Lösers mit der Methode der kleinsten Quadrate wie `numpy.linalg.lstsq`[12] benötigen jeweils die volle A Matrix, welche jedoch bei allen Kanälen zusammen übermäßig groß wird. Zudem sollte der Aufbau des Gleichungssystem inkrementell erfolgen, um die verteilte Spark-Architektur ausnutzen zu können. Daher wird ein eigener Löser auf Basis der Methode der kleinsten Quadrate implementiert.

Die A^TA sowie A^Tb Matrix lässt sich aus einzelnen Blöcken zusammensetzen. Durch die additive Eigenschaft der Matrixmultiplikation lassen sich mehrere Teilmatrizen unter Berücksichtigung der entsprechenden Indizes aufeinanderaddieren. Dies ermöglicht das Aufstellen von (Teil-)Gleichungssystemen ohne bereits alle Variablen oder Werte zu kennen und minimiert die nötigen Rechenoperationen, da nicht besetzte Felder nicht mitberechnet werden müssen. Das `LSEquationSystem` beinhaltet daher nur zwei `dicts`, wobei `a` die mit Spalten- und Zeilenschlüssel indizierten Subblöcke der A^TA Matrix beinhaltet und `b` die mit Zeilenschlüssel indizierten Subblöcke des A^Tb Vektors enthält. Daraus ergibt sich die in Quellcode 4.1 gezeigte Funktion zum Hinzufügen neuer Gleichungen mit den Faktoren `egs` und dazugehörigen Messergebnissen `rhs`.

```
1 def add_equations(self, egs: List[Tuple[Any, np.ndarray]], rhs:
      np.ndarray):
2     for cg1_id, cg1_coef in egs:
3         for cg2_id, cg2_coef in egs:
4             _add_to_dict(self.a, (cg1_id, cg2_id), cg1_coef.T @
                  cg2_coef)
```

```
5           _add_to_dict(self.b, cg1_id, cg1_coef.T @ rhs)
```

Quellcode 4.1: Methode add_equations zum Hinzufügen neuer Gleichungen

Die Hilfsfunktion `_add_to_dict` ist in Quellcode 4.7 und Quellcode 4.8 beschrieben. Dieser Aufbau ermöglicht es einfach, mehrere `LSEquationSystem` durch Addition und Ergänzung der einzelnen Subblöcke zusammenzuführen. Um das Gleichungssystem zu lösen, werden vor der Invertierung lediglich die Matrizen A^TA und A^Tb zusammengesetzt, wie Quellcode 4.2 zeigt.

```
1  def solve(self) -> Dict[Any, np.ndarray]:
2      size = 0
3      cg_offset = {}
4      for cg_id in self.b.keys():
5          cg_offset[cg_id] = size
6          size += self.b[cg_id].size
7
8      a = np.zeros((size, size), dtype=np.float)
9      for (cg1_id, cg2_id), cg_value in self.a.items():
10         a[cg_offset[cg1_id]:, cg_offset[cg2_id]:][:cg_value.shape
               [0], :cg_value.shape[1]] = cg_value
11
12     b = np.zeros((size, ), dtype=np.float)
13     for cg_id, cg_value in self.b.items():
14         b[cg_offset[cg_id]:cg_offset[cg_id]+cg_value.shape[0]] =
               cg_value
15
16     x = np.linalg.inv(a) @ b
17
18     return {cg_id: x[i:i+self.b[cg_id].size] for cg_id, i in
           cg_offset.items()}
```

Quellcode 4.2: Methode solve zum Lösen des Gleichungssytems

Um die Hebelarmkorrektur zu berechnen, muss das Gleichungssystems mit den einzelnen Punktzielantworten konstruiert werden. Quellcode 4.3 zeigt die Funktion zur Erstellung eines Gleichungssystems für eine Punktzielantwort.

```
1 def phase_center_correction_ls(response):
2     system = LSEquationSystem()
3     system.add_equations([
4         (response.transmit_id, response.los.T / 2),
5         (response.receive_id, response.los.T / 2),
6         (response.chan.ch, np.ones((response.los.shape[1], 1)))
7     ], response.dr)
```

Quellcode 4.3: Aufstellen des Gleichungssystems zur Hebelarmkorrektur

Die einzelnen Punktzielantworten liegen als Ergebnis der vorausgehenden Analyse in Form eines resilienten verteilten Datensatzes (RDD) vor. Dieses RDD `responses` kann dabei als verteiltes `dict` mit einem Schlüssel bestehend aus dem Kanal und Punktziel und dem Wert der Punktzielantwort betrachtet werden. Mit der RDD-Methode `mapValues` können die Gleichungssysteme für die Punktzielantworten erstellt und mit `reduce` diese zusammengeführt werden [13]. Dazu wird eine Hilfsfunktion `merge` benötigt, welche zwei (Teil-)Gleichungssysteme aufaddiert und damit zusammenführt.

```
1 responses.mapValues(phase_center_correction_ls).reduce(lambda x,
      y: (None, LSEquationSystem.merge(x[1], y[1])))[1].solve()
```

Quellcode 4.4: Nutzung von Spark zur Zusammenführung und Lösung des Gleichungssystems zur Hebelarmkorrektur

4.1.2 Verifikation

Die Verifikation der Implementierung erfolgt mithilfe eines simulierten Datensatzes, welcher mit einer perfekten Kalibrierung - d.h. ohne Einfühung von Abweichungen - erstellt wurde. Bei der Analyse des Datensatzes kann eine fehlerhafte Antennenkalibrierung geladen werden. Die Kalibrierroutinen haben das Ziel der Minimierung der eingeführten Fehler. Eine direkte Verifikation mit Referenzwerten ist nur teils möglich, da das Gleichungssystem zur Korrektur des Hebelarms schlecht konditioniert ist und mehrere gleichwertige Lösungen existieren können. Kleine numerische Unterschiede, welche zwischen den Implementierungen zu erwarten sind, können zu Unterschieden

führen. Daher werden in dieser Arbeit Diagramme mit den gemessenen Werten und daraus ermittelten Änderungen verwendet. Lässt die ermittelte Lösung die gemessenen Werte gegen 0 konvergieren, kann von einer korrekten Implementierung ausgegangen werden.

Im Vergleich zur IDL-Implementierung fiel auf, dass die in der Analyse geschätzte Position von der erwarteten Position abweicht. In Abbildung 4.1 ist die in der IDL-Implementierung und der CARAMBA-Implementierung (CRM) ermittelte Abweichung des gemessenen Rangeabstands eines Punktziels dargestellt. Die Linien stellen den gemessenen Entfernungsversatz als Funktion der Azimutposition dar. Der tatsächliche Versatz beträgt 0 m, wie auch von der IDL-Implementierung im Gegensatz zur CARAMBA-Implementierung erkannt. Zur Bestimmung von δr wer-

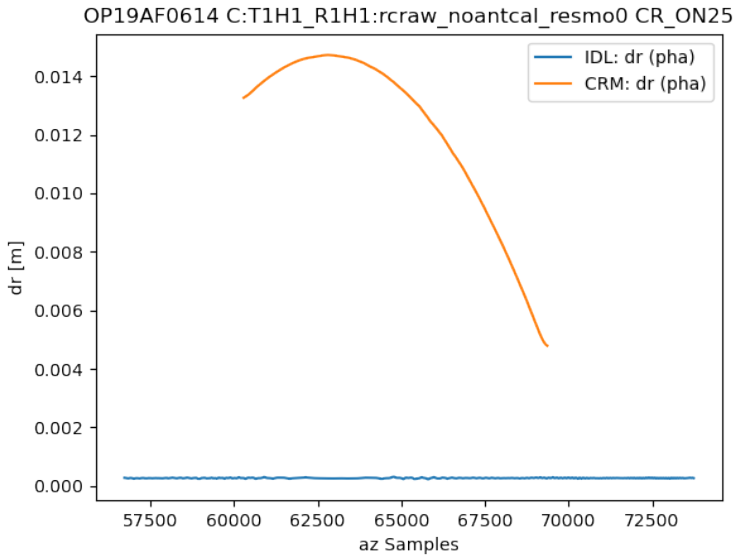

Abbildung 4.1: dr-Abweichung der CARAMBA-Punktzielanalyse

den die rangekomprimierten Daten um die erwartete Position mit Subpixelpräzision verschoben, um anschließend die Abweichung von der erwarteten Position bestimmen zu können. Die Korrektheit der Funktion zur Bestimmung der Maximumsposition sowie zur Verschiebung der Datensätze wurde mithilfe mehrere Tests validiert. Der Vergleich der erwarteten Zielposition der IDL- und CARAMBA-Implementierung zeigte eine konstante Abweichung im Bereich von 0.5 bis 1 mm. Durch weitergehende

Analyse zeigte sich, dass der konstante Versatz durch die Verwendung eines anderen Wertes für die Lichtgeschwindigkeit zustande kam. Der fehlerhafte Phasenverlauf kam durch einen Fehler bei der Korrektur der Antenneneigenschaften bei der Punktzielanalyse zustande. Nach Behebung der Fehler wurden die erwarteten δr-Werte erzielt.

Um die Ergebnisse der Implementierung zu verifizieren, wurde zunächst mit einem Datensatz mit einem erwarteten Hebelarmversatz von 0.00 m verglichen. Dies spiegelt sich in einem gemessenen Rangeabstand von 0.00 m wider. Um das Modell zu verifizieren, wird anschließend unter Berücksichtigung der bestimmten Korrektur die Analyse und Hebelarmkorrektur ein zweites Mal durchgeführt. Dabei ist zu erwarten, dass das gemessene δr des zweiten Durchlaufs der Differenz zwischen dem δr des ersten Durchlaufs und der mit Modell geschätzten Position entspricht und das Modell nicht weiter verbessert werden kann, da die Methode der kleinsten Quadrate mit einer Iteration das bestmögliche Modell bestimmen kann. In den folgenden Diagrammen werden die ermittelten δr-Werte für alle Punktziele eines Kanals im Azimut dargestellt. Dabei werden die einzelnen Punktziele durch die Verwendung unterschiedlicher Farben separiert. Mit gepunkteten Linien wird das mit dem bestimmten Hebelarmversatz berechnete Modell dargestellt. Bei der Überprüfung fiel auf, dass die im 2. Durchlauf erwarteten δr-Werte (in Abbildung 4.2 zu sehen) von den im 2. Durchlauf gemessenen δr-Werten abweichen (in Abbildung 4.3 zu sehen). Insbesondere die Änderung der Anordnung der Punktziele mit steigendem Off-Nadir-Winkel, also bei steigender Distanz vom Radarsensor, deutet hierbei auf einen Fehler hin. Durch die Abweichung ergibt sich zudem ein neues, vermeintlich besseres Modell. Bei der Betrachtung der r und los Werte des ersten und zweiten Durchlaufs zeigten sich die folgenden korrekten Abhängigkeiten, $_n$ kennzeichnet hierbei, dass dieser los Wert nicht normiert ist:

$$los_{2,n} + \delta l = los_{1,n}$$

$$r_2 + los_1 \cdot \delta l = r_1$$

Aufgrund dieser Beobachtung kann ein Fehler bei der Anwendung der δl-Werte

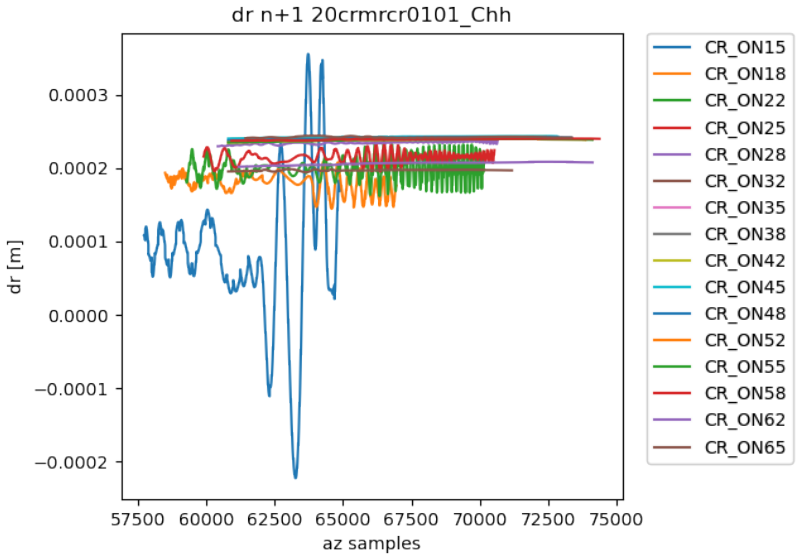

Abbildung 4.2: Durch Abzug des berechneten Modells von den gemessenen dr-Werten bestimmte; für den zweiten Durchlauf erwartete δr-Werte

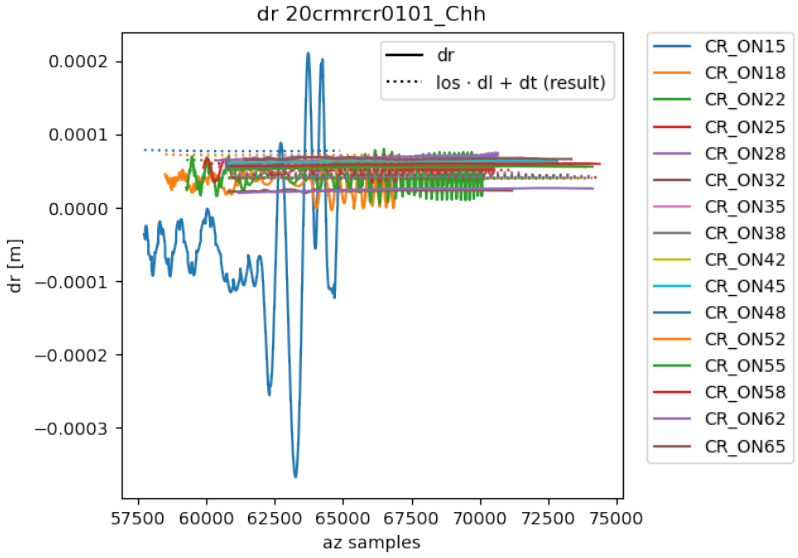

Abbildung 4.3: Im zweiten Durchlauf bestimmte δr-Werte

ausgeschlossen werden. Der Fehler muss daher direkt bei der Bestimmung des δr-Wertes auftreten. Zur Bestimmung des relativen Fehlers über den Azimut-Verlauf hinweg wird die Phase herangezogen. Da sich die Phase über die erneute Analyse konstant zeigt, ist diese nicht das Problem. Der absolute Versatz wird mit der Maximumsposition bestimmt. Dabei wird im Frequenzraum mithilfe einer Phasenrampe die Position des Maximums mit einer Subpixelpräzision bestimmt. Um die absolute Signalstärke zu berücksichtigen, wird die Punktzielantwort zuvor im Zeitraum mit der komplex konjugierten Punktzielantwort multipliziert. Dabei fiel auf, dass das Signal mehr als 50% der Bandbreite groß ist, wodurch die Multiplikation mit dem komplex Konjugierten einen Fehler verursachen würde. Zur Verringerung der Signalbandbreite wurde daher eine Funktion ergänzt, welche das Signal im Frequenzraum mit Nullen erweitert. Wird diese Funktion so aufgerufen, dass die Signallänge verdoppelt wird, ist damit garantiert, dass die Multiplikation mit dem komplex Konjugiertem valide ist.

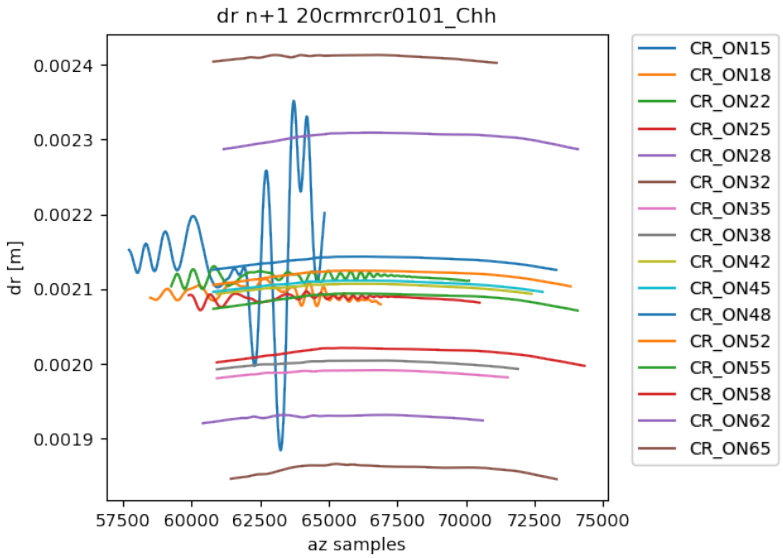

Abbildung 4.4: Für den zweiten Durchlauf erwartete dr-Werte

Mit dieser Korrektur stimmt die Erwartung wie in Abbildung 4.4 zu sehen mit den gemessenen Werten wie in Abbildung 4.5 zu sehen überein. Durch die in Abbildung 4.5 angewandten modellierten Hebelarmkorrekturen wird die perfekte Kalibrierung nicht signifikant verschlechtert, womit nachgewiesen ist, dass die Implementierung gute

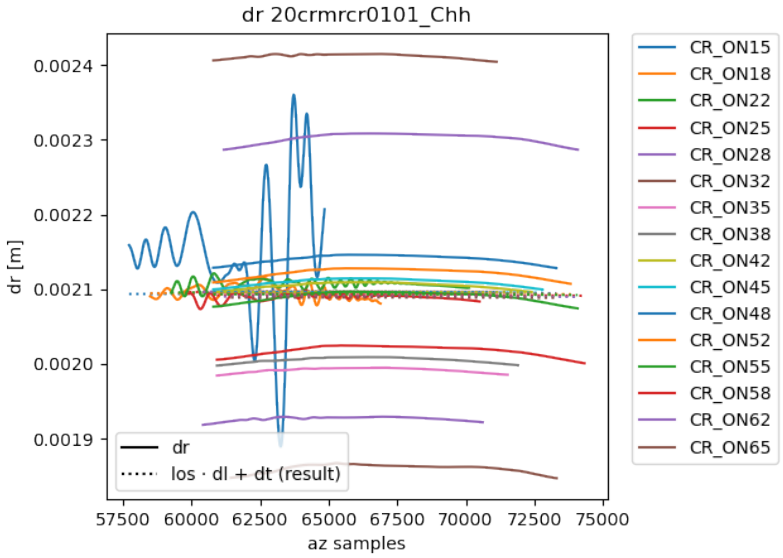

Abbildung 4.5: Im zweiten Durchlauf bestimmte dr-Werte

Hebelarmkorrekturen nicht verschlechtert.

Nach der Verifikation der Implementierung für einen Versatz von 0.00 m wurde auf die simulierten Daten ein bekannter Versatz angewandt. Durch Verwendung dieser Daten soll nachgewiesen werden, dass die Implementierung den Versatz durch die Bestimmung korrekter Hebelarmkorrekturen kompensieren kann und die Rangedifferenzen somit gegen 0 konvergieren. Dabei fiel auf, dass trotz eines größeren erwarteten Versatzes sich alle δr Vektoren in Bereichen nahe 0.00 m Versatz befinden. Ein Vergleich mit den zu erwarteten Werten, wie in Abbildung 4.6 zu sehen, bestätigte die vermutete Differenz. Durch Betrachtung der Rohdaten mit der modellierten Rangeposition jeweils ohne Kalibrierungsfehler (Abbildung 4.7) und mit Kalibrierungsfehler (Abbildung 4.8) fällt auf, dass sich das Modell nicht ändert. Daraus lässt sich schließen, dass im Modell der Kalibrierungsfehler kompensiert wird. Die Betrachtung des zugrundeliegenden Navigationsdaten zeigte, dass für die simulierten Daten dort fehlerhafterweise bereits die Line-Of-Sight korrigiert wurde.

Nach Korrektur der fehlerhaften Kompensation wurde die Hebelarmkorrektur auf dem Datensatz mit künstlichem Versatz erneut durchgeführt. In Abbildung 4.9 ist

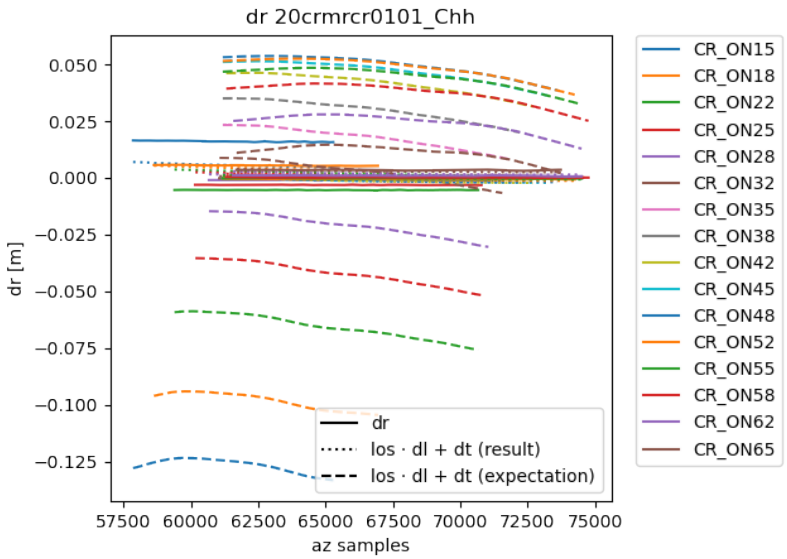

Abbildung 4.6: dr-Abweichung der Analyse von erwarteten Werten

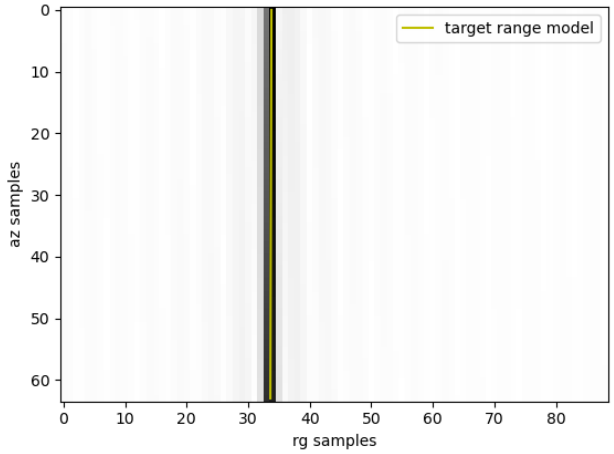

Abbildung 4.7: Rohdaten mit erwarteter Rangeposition ohne Kalibrierungsfehler

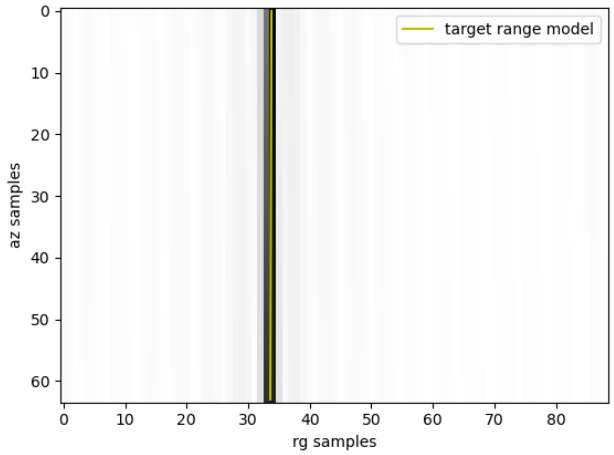

Abbildung 4.8: Rohdaten mit erwarteter Rangeposition mit Kalibrierungsfehler

der im ersten Durchlauf bestimmte δr-Versatz sowie das Modell des Versatzes mit den bestimmtem Hebelarmversatz (gepunktet) und der tatsächlich eingefügte Versatz (gestrichelt) dargestellt. Dabei ist zu sehen, dass das Modell den Hebelarmversatz gut minimiert. Dies spiegelt sich auch in Abbildung 4.10 nach erneuter Durchführung der Analyse wieder. Die Minimierung des Fehlers und Konvergenz des Modells zeigt hiermit auf, dass die Implementierung fehlerhafte Hebelarme erfolgreich korrigieren kann.

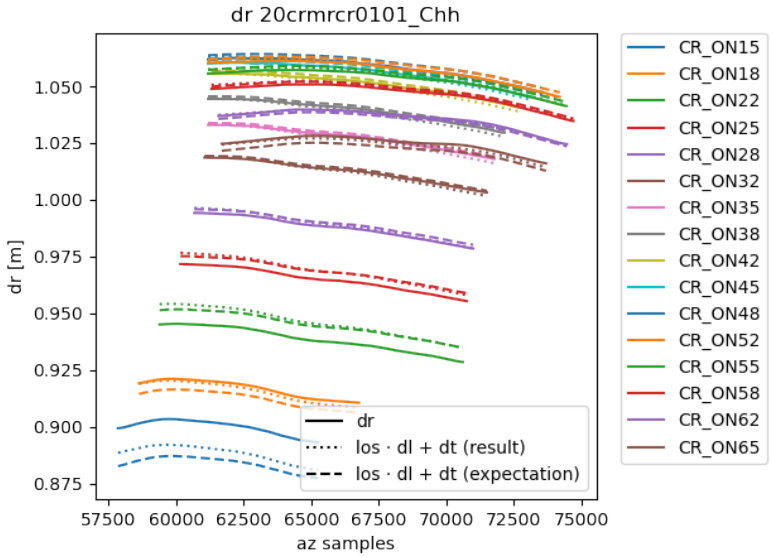

Abbildung 4.9: Ermittelte dr-Werte bei bekanntem Hebelarmversatz

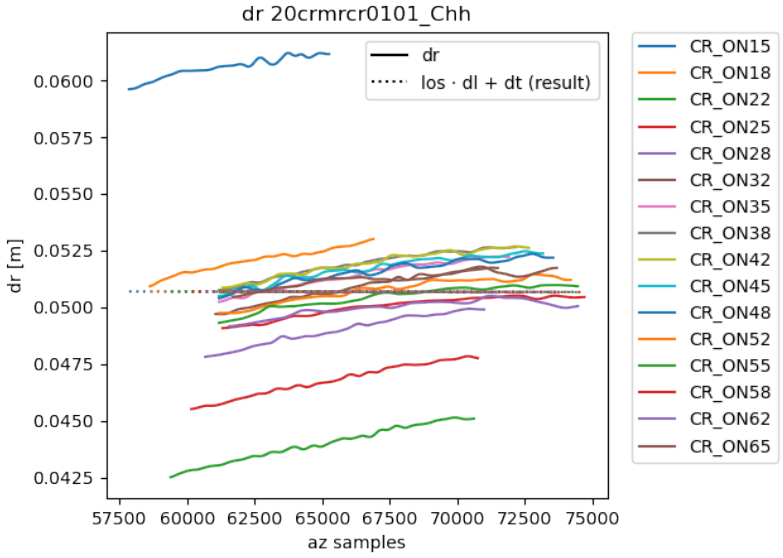

Abbildung 4.10: Ermittelte dr-Werte bei bekanntem Hebelarmversatz nach Korrekturanwendung

4.2 Implementierung und Verifikation der optimalen Kanalkonsistenz

Die gemessenen δr-Werte sind in realen Daten auch für perfekt kalibrierte Daten nicht exakt 0. Restfehler in den mit DGPS-Messungen bestimmten Sensor- und Reflektorpositionen verursachen Restfehler in der Größenordnung von 1 cm. Diese Restfehler sind jedoch für simultan aufgezeichnete Kanäle stets gleich. Daher müssen die δr Werte bei erfolgter Sensorkalibrerung für alle Kanäle des gleichen Überflugs (Passes) für jeweils ein Punktziel identisch sein. Diese Randbedingung kann in Form einer Gleichung in das Gleichungssystem zur Bestimmung des Hebelarmversatzes integriert werden. 1 und 2 markieren hierbei zwei unterschiedliche Kanäle:

$$\delta r_1 - \delta r_2 = (\frac{1}{2}\delta l_1 \cdot los_{t1} + \frac{1}{2}\delta l_1 \cdot los_{r1} + \delta t_1) - (\frac{1}{2}\delta l_2 \cdot los_{t2} + \frac{1}{2}\delta l_2 \cdot los_{r2} + \delta t_2)$$

Da die Aufnahmezeitpunkte nicht exakt zeitgleich sind und die einzelnen Kanäle separat voneinander vorverarbeitet wurden, ist es nötig, die δr sowie los Vektoren eines der beiden Kanäle mithilfe von Timestamps auf den anderen Kanal zu interpolieren.

4.2.1 Implementierung

Für die Durchführung der optimalen Kanalkonsistenz ist es zunächst nötig, die Punktzielantworten nach Pass und Punktziel zu gruppieren. Mit der Methode `groupBy` der Spark RDDs kann die Gruppierung erzielt werden [13], wie in Quellcode 4.5 zu sehen.

```
1 grouped_responses = responses.groupBy(
2     lambda ct_response: (ct_response[1].chan.pass_id(),
          ct_response[1].target)
3 ).mapValues(
4     lambda x: [y[1] for y in x]
5 ).persist(spk-spark.StorageLevel.MEMORY_AND_DISK)
```

Quellcode 4.5: Gruppierung der Punktzielantworten nach Pass und Punktziel

Um alle Kanalpaare mit gleichem Pass und Punktziel zu erhalten, ist es nötig, auf diesen Untermengen ein kartesisches Produkt durchzuführen. Theoretisch könnte auch Spark dazu genutzt werden, das kartesische Produkt zu bilden und dann die entsprechenden Gleichungen aufzustellen. Aufgrund von Befürchtungen, dass Spark deutlich langsamer ist als die sequentielle Abarbeitung der einzelnen Pass- und Punktzielpaare, wurde dieser Schritt nicht mit Spark parallelisiert.

```
1  def canal_consistency_ls(responses):
2      system = LSEquationSystem()
3      for r1 in responses:
4          for r2 in responses:
5              r2los = np.asarray([interpol_uspline1d(r2.t, l, r1.t)
                   for l in r2.los])
6              r2dr = interpol_uspline1d(r2.t, r2.dr, r1.t)
7              system.add_equations([
8                  (r1.transmit_id, r1.los.T / 2),
9                  (r1.receive_id, r1.los.T / 2),
10                 (r1.chan.ch, np.ones((r1.los.shape[1], 1))),
11                 (r2.transmit_id, -r2los.T / 2),
12                 (r2.receive_id, -r2los.T / 2),
13                 (r2.chan.ch, -np.ones((r1.los.shape[1], 1)))
14             ], r1.dr - r2dr)
15     return system
```

Quellcode 4.6: Aufstellen der Gleichungen für die optimale Kanalkonsistenz

Die Zusammenführung der Gleichungssysteme funktioniert analog zur Zusammenführung des Hauptgleichungssystems zur Bestimmung der Hebelarmkorrektur. Zur Lösung müssen nur beide Gleichungssysteme zusammengeführt werden.

4.2.2 Performanceoptimierung

Trotz Aufteilung der Punktzielantworten nach Pässen und Punktzielen ergibt sich durch das kartesische Produkt eine quadratische Laufzeitkomplexität, welche die Zeitdauer der Operation verlangsamt. Abbildung 4.11 zeigt das Laufzeitprofil der

unoptimerten Implementierung und die für Python-Funktionen teils hohen 8 stelligen Aufrufzahlen einzelner Funktionen. Hierbei zeigt sich, dass 42.18% der Laufzeit für

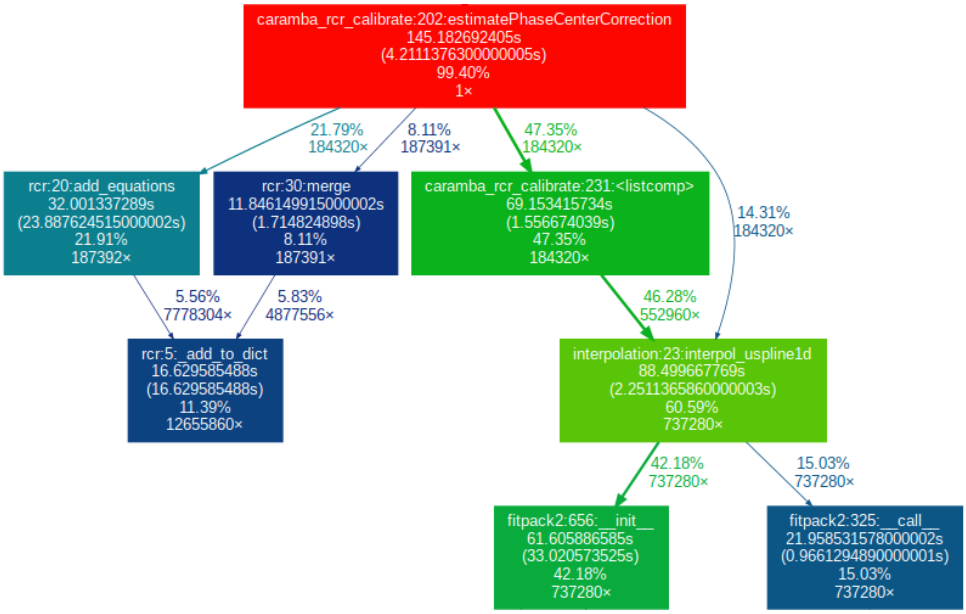

Abbildung 4.11: Unoptimiertes Laufzeitprofil der Kanalkonsistenz

die Initialisierung der Interpolationsobjekte benötigt wird. Für diese Interpolation wird ausschließlich der Zeit- sowie Datenvektor einer Punktzielantwort benötigt, die Interpolation mit dem Zeitvektor der anderen Antwort erfolgt erst in der `__call__` Methode. Daher können die Interpolationsobjekte bereits vor Bildung des kartesischen Produktes gebildet werden, welches die Anzahl an Initialisierungen und damit auch die Laufzeit reduziert. Dies eliminiert die exzessive Laufzeit der Initialisierung auf einen vernachlässigbaren Anteil. Die Funktion `_add_to_dict` nimmt aufgrund der hohen Aufrufmenge mit 11.39% ebenfalls einen hohen Laufzeitanteil ein. Eine Verringerung der Aufrufmenge konnte dadurch erreicht werden, dass beim Akkumulieren des Gleichungssystems keine neuen Objekte erstellt, sondern bestehende Objekte erweitert werden. Diese Funktion ist wie in Quellcode 4.7 gezeigt darüber gelöst, dass im Fall, dass der Schlüssel noch keinen Wert enthält, eine Ausnahme geworfen wird. Da in der überwiegenden Menge (12655860 bzw. 7808244 Aufrufe bei 228 verschiedenen Schlüsseln) der Aufrufe der Schlüssel bereits vorhanden ist, sollte dieser Ansatz schneller als ein Vergleich sein [14].

```
1 def _add_to_dict(d: Dict, key, value):
2     try:
3         d[key] += value
4     except KeyError:
5         d[key] = value
```

Quellcode 4.7: _add_to_dict in Ausnahmearchitektur

Durch einen Test mit Bedingungsarchitektur wie in Quellcode 4.8 beschrieben zeigte sich, dass die Funktion mit Bedingung im konkreten Anwendungsfall ca. 50% schneller als die Funktion mit Ausnahmebehandlung ist.

```
1 def _add_to_dict(d: Dict, key, value):
2     d[key] = d[key] + value if key in d else value
```

Quellcode 4.8: _add_to_dict in Bedingungsarchitektur

Nach [15] ist auch das Auflösen einer Membervariable zu vermeiden, jedoch zeigte eine Optimierung dahingehend keine messbare Laufzeitverbesserung. Abbildung 4.12 zeigt das Laufzeitprofil nach Anwendung der genannten Optimierungsmaßnahmen.

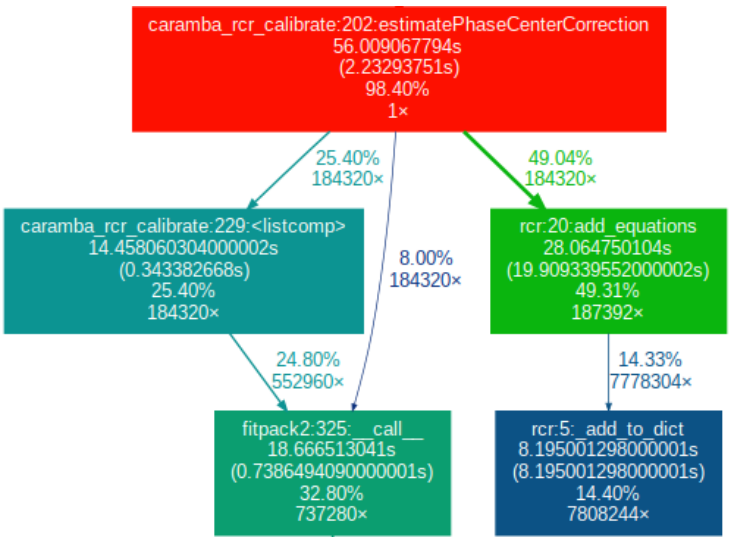

Abbildung 4.12: Optimiertes Laufzeitprofil der Kanalkonsistenz

4.2.3 Verifikation

Zur Verfikation der optimalen Kanalkonsistenz wurde zunächst eine Überprüfung des Ergebnisses mit 0.00 m Hebelarmversatz durchgeführt. Dabei zeigte sich eine größere Abweichung, wie in Abbildung 4.13 zu sehen. In einer Debugging-Session zeigte sich,

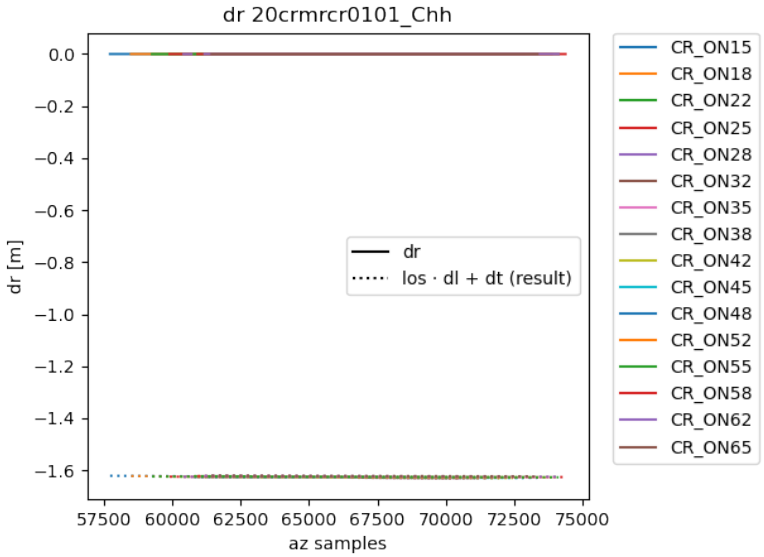

Abbildung 4.13: Abweichung mit aktiver optimaler Kanalkonsistenz

dass die interpolierten δr-Werte im Gegensatz zu den nicht interpolierten δr-Werten teils in der Größenordnung von Metern statt im Millimeter- bzw. Submillimeterbereich sind. Diese Werte waren außerhalb des Bildausschnittes, wodurch eine Extrapolation durchgeführt wurde, welche zu den falschen Werten führte. Durch das Einführen einer Gültigkeitsmaske mit Maskierung der Werte außerhalb des Ausschnittes konnte der Fehler behoben werden, wie Abbildung 4.14 zeigt. Ein Durchlauf mit Hebelarmversatz erzeugte die zu erwartenden Werte, wie in Abbildung 4.15 zu sehen ist, da das erstellte Hebelarmmodell mit den gepunkteten Linien die gemessenen Rangeunterschiede sowie den angewandten Hebelarmversatz (gestrichelte Linien) gut annähert. Somit wird der Hebelarmversatz mit Einführung der Randbedingung der optimalen Kanalkonsistenz weiterhin korrekt minimiert.

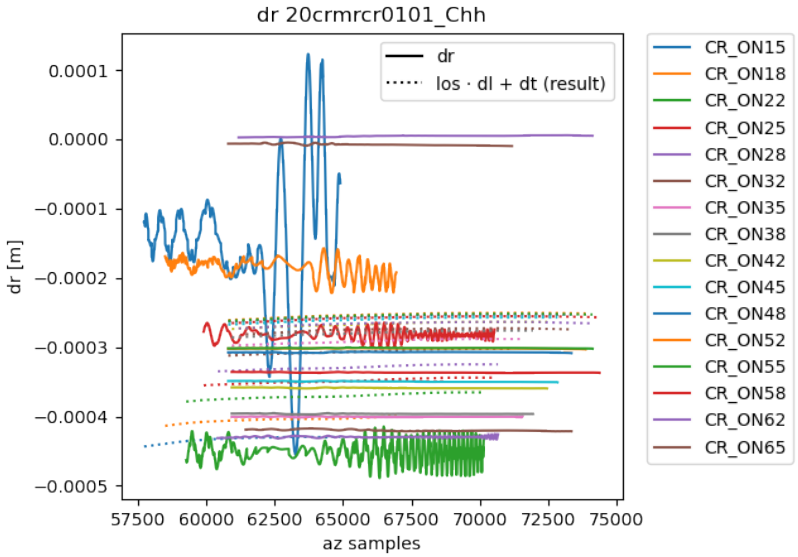

Abbildung 4.14: Hebelarmkorrektur mit optimaler Kanalkonsistenz ohne Hebelarmversatz

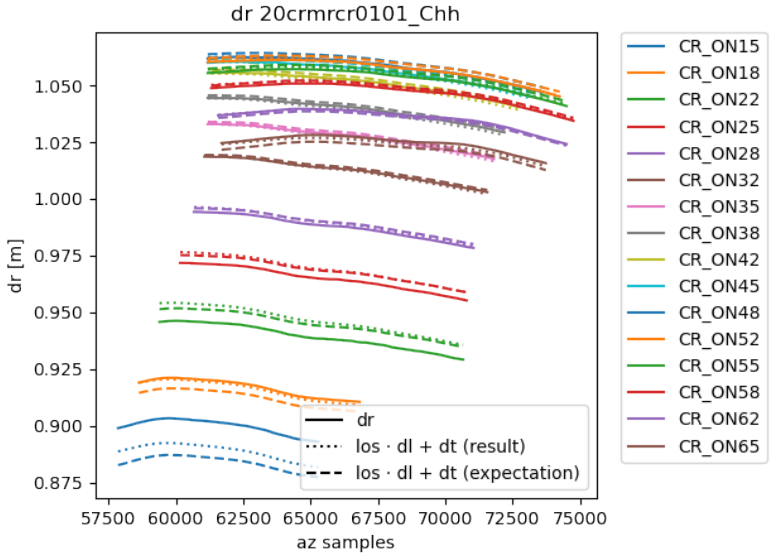

Abbildung 4.15: Hebelarmkorrektur mit optimaler Kanalkonsistenz und Hebelarmversatz

4.3 Implementierung und Verifizierung einer L1-Norm Regularisierung

Die in Abschnitt 4.1 implementierte Methode der kleinsten Quadrate minimiert den quadratischen Fehler des Gleichungssystems. Reale Messdaten entsprechen üblicherweise nicht durchgehend dem optimalen Modell, so können z.B. Interferenzen oder fehlausgerichtete Reflektoren zu Ausreißern in den Daten führen. Daher ist es ein robusterer Ansatz, den linearen Fehler, die L1-Norm, des Gleichungssystems zu minimieren. Da die L1-Norm nur eine nichtlineare Lösung hat, wird diese mithilfe mehrerer Iterationen eines Lösers mit der Methode der kleinsten Quadrate angenähert [16]. Dazu werden die Gleichungen $b = Ax$ auf Basis des Fehlers der letzten Iteration n mit dem Gewicht W gewichtet:

$$W^{(n)} = \frac{1}{\sqrt{\mid \sum_i (A_i \cdot x_i^{(n)}) - b \mid + 1 \cdot 10^{-7}}}$$

$$A^{(n+1)} = A \cdot W^{(n)}$$

$$b^{(n+1)} = b \cdot W^{(n)}$$

i stellt dabei die einzelnen Spalten der Gleichungen dar.

4.3.1 Implementierung

Die Implementierung der iterativen L1-Norm Minimierung wurde in Form einer Erweiterung des in Abschnitt 4.1 implementierten `LSEquationSystem` umgesetzt. Dabei soll weiterhin die Aufstellung großer Matrizen vermieden werden und lediglich bereits vorverarbeitete Matrizen gespeichert werden. Daraus ergibt sich die Notwendigkeit, nach dem Lösen einer Iteration des Gleichungssystems das Gleichungssystem unter der Berücksichtigung der Lösung der vorigen Iteration komplett neu aufzustellen. Um das Aufstellen des Gleichungssystems unabhängig von der Iteration zu gestalten, wird die `add_equations`-Methode um die Möglichkeit zur L1-Normierung erweitert, wie in Quellcode 4.9 beschrieben.

```
def add_equations(self, egs: List[Tuple[Any, np.ndarray]], rhs,
                  sol: Dict[Any, np.ndarray] = None, w=1, eps=1e-8):
    if sol:
        l1_w = w / np.sqrt(np.abs(np.sum([cg_coef.dot(sol[cg_id])
                for cg_id, cg_coef in egs], axis=0) - rhs) + eps)
        egs = [(cg_id, l1_w[:, np.newaxis] * cg_coef) for cg_id,
               cg_coef in egs]
        rhs = rhs * l1_w
    # ...
```

Quellcode 4.9: L1-Norm Erweiterung von add_equations

Dabei wird bei Angabe einer Lösung des Gleichungssystems L1-Normierung durchgeführt. Damit ist der Codeaufruf im Code zum Aufstellen des Gleichungssystems für alle Iterationen gleich, da in der ersten Iteration ohne vorliegende Lösung lediglich `None` statt einer tatsächlichen Lösung weitergereicht wird.

Für die Durchführung der iterativen L1-Norm Minimierung wurde die Funktion `solve_l1_norm` geschrieben, welche in Quellcode 4.10 dargestellt ist. An diese Funktion muss lediglich eine Funktion, welche eine Lösung nimmt und damit ein Gleichungssystem aufbaut, übergeben werden.

```
def solve_l1_norm(system_generator: Callable, iterations=5):
    sol = None
    for _ in range(iterations):
        sol = system_generator(sol).solve()
    return sol
```

Quellcode 4.10: solve_l1_norm zur iterativen Durchführung von L1-Norm Regularisierung

4.3.2 Verifikation

Die L1-Norm Minimierung soll dafür sorgen, dass der Einfluss von Ausreißern auf das Modell reduziert wird. Da die Auswirkungen der L1-Norm Minimierung auf

dem simulierten Datensatz kaum erkennbar ist, wird für diesen Abschnitt eine Geradenmenge verwendet. Hierbei werden alle Geraden durch

$$y = m \cdot x + t \cdot 1$$

dargestellt. Die drei Geraden f1, f2 und f3 sind hierbei alle sehr ähnlich zueinander, wobei die Gerade f4 einen starken Ausreißer darstellt. Da es sich bei der Implementierung um ein numerisches Lösungsverfahren handelt, werden für die x-Werte 0 bis 99 jeweils die y-Werte berechnet und in das Gleichungssystem eingetragen. In Abbildung 4.16 ist die Annäherung mit der Methode der kleinsten Quadrate, welche der Ausgangspunkt für die iterative L1-Norm Minimierung ist, zu sehen. Nach der

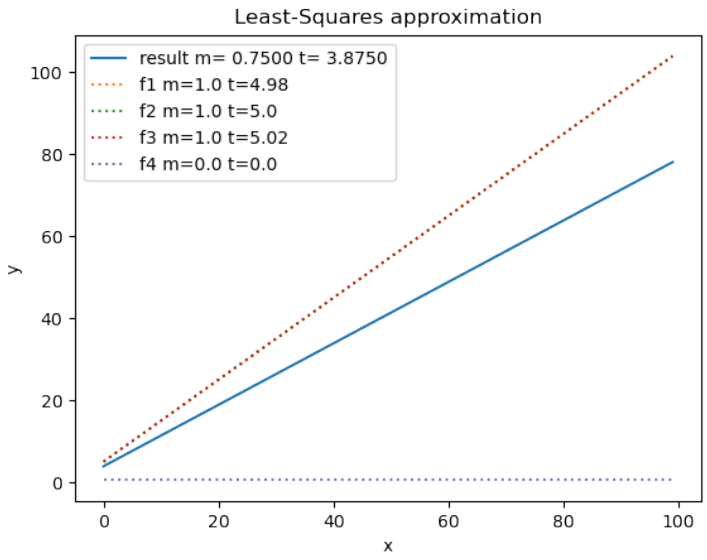

Abbildung 4.16: Annäherung mit Methode der kleinsten Quadrate

ersten Iteration der L1-Normierung, was in Abbildung 4.17 dargestellt ist, wird der Einfluss des Ausreißers bereits reduziert. Nach 10 Iterationen (Abbildung 4.18) ist der Einfluss des Ausreißers aus dem Modell nahezu vollständig aus dem Modell eliminiert worden. Damit ist gezeigt worden, dass diese Implementierung der iterativen L1-Norm Minimierung den Einfluss von Ausreißern aus dem erstellten Modell erfolgreich entfernt.

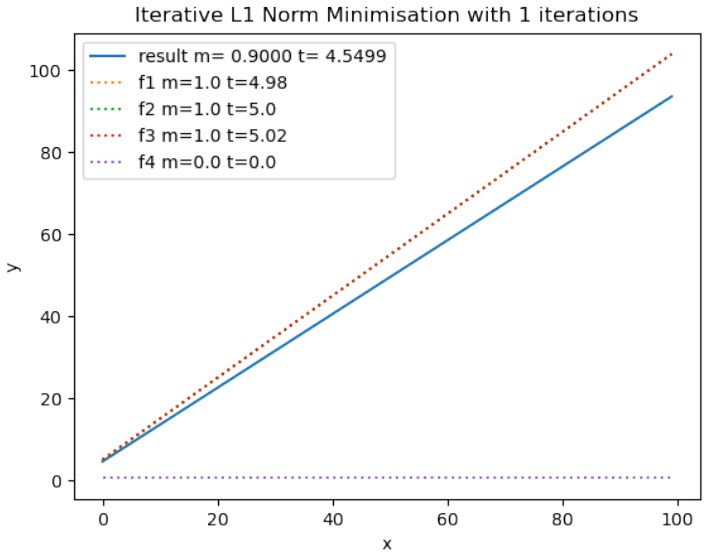

Abbildung 4.17: L1-Norm Minimierung nach einer Iteration

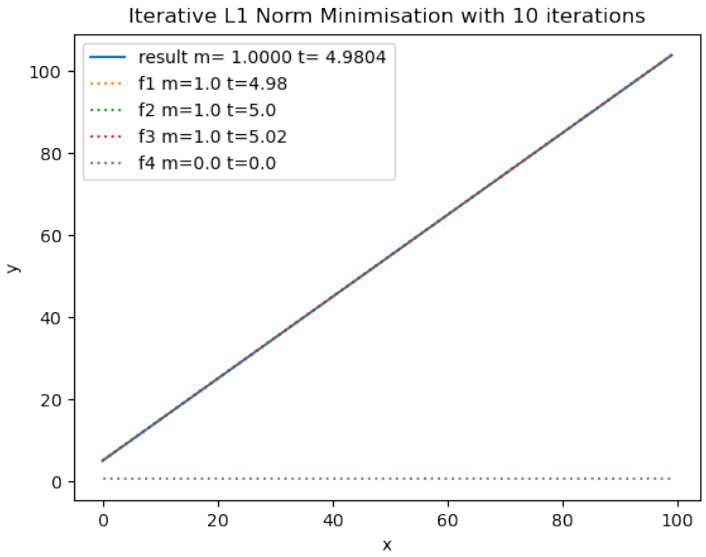

Abbildung 4.18: L1-Norm Minimierung nach 10 Iterationen

4.4 Implementierung und Verifizierung einer Antennenblickrichtungskorrektur

Neben der Hebelarmkorrektur ist auch die Antennenblickrichtungskorrektur Teil der SAR-Rohdatenkalibrierung. Durch Ungenauigkeiten beim Einbau der Antennen sind Abweichungen in der Größenordnung von 0.1° im Gier-, Nick- und Rollwinkel p_{ypr} zu erwarten. Dies verändert die empfangene Signalstärke sowie den Signalgewinn $g(t_{az}, p_{ypr})$ (gain). Das gegebene Antennenmodell A gibt die erwartete Signalstärke für einen Off-Nadir-Winkel θ, Squint-Winkel β und Frequenz f zurück. Für die Frequenz kann für die Blickrichtungkorrektur die mittlere Signalfrequenz f_0 angenommen werden. Der Off-Nadir und Squintwinkel für das betrachtete Punktziel können jeweils aus der Line-of-Sight $los(t_{az})$ und den Rotationswinkeln p_{ypr} berechnet werden. Daraus ergibt sich folgendes nichtlineares Modell [2]:

$$g(t_{az}, p_{ypr}) = A(\theta(los(t_{az}), p_{ypr}), \beta(los(t_{az}), p_{ypr}), f_0)$$

Da die Größenordnung des Blickrichtungsfehlers bekannt ist, kann das nichtlineare Modell mithilfe der Annäherung der partiellen Ableitung durch Differenzierung linearisiert werden.

$$p_{yp\Delta r} = \begin{pmatrix} p_y \\ p_p \\ p_r + \Delta r \end{pmatrix}$$

$$\frac{\partial g(t, p_{yp\Delta r})}{\partial \Delta r} \approx \frac{g(t, p_{yp\Delta r}) - g(t, p_{ypr})}{\Delta r}$$

Mit dem Vorliegen der Differenz kann daraus ein lineares Gleichungssystem aufgestellt werden, um mit dem gemessenen Gewinnunterschied $\Delta g(t)$ die Abweichung im Gierwinkel δy, Nickwinkel δp und Rollwinkel δr mit einem konstanten Versatz δg bestimmen zu können:

$$\Delta g(t) = \delta r \frac{\partial g(t, p_{yp\Delta r})}{\partial \Delta r} + \delta p \frac{\partial g(t, p_{y\Delta pr})}{\partial \Delta p} + \delta y \frac{\partial g(t, p_{\Delta ypr})}{\partial \Delta y} + \delta g$$

4.4.1 Implementierung

Der Antennengewinnverlauf über Azimut liegt bereits aus der Rohdatenanalyse vor. Für das Aufstellen des Gleichungssystems zur Bestimmung des Antennenblickrichtungsversatzes wird daher nur zusätzlich der Gradient benötigt. Die Berechnung des Gradienten wird dabei mit der Analyse durchgeführt, weil die zur Berechnung nötigen Parameter bereits vorliegen. Das Antennenmodell liegt in CARAMBA nicht als Funktion, sondern als numerisches Array vor, welches jeweils für den passenden Off-Nadir-Winkel, Squintwinkel und passende Frequenz interpoliert wird. Für die Berechnung des Gradienten wird nur das Antennenmodell für die mittlere Frequenz benötigt. Da dies die nötige Datenmenge stark reduziert, wird in der Implementierung auch nur die mittlere Frequenz eingelesen. Daher muss ausschließlich zweidimensional interpoliert werden. Da die in CARAMBA existierende Funktion zur Interpolation des Antennendiagrammes nur mit dreidimensionalen Antennenmodellen umgehen kann, musste eine äquivalente Funktion zu Interpolation zweidimensionaler Antennendiagramme implementiert werden. Da wir den Antennengewinn messen, ist es wichtig, auch das jeweilige Antennendiagramm vor der Interpolation in Gewinn umzurechnen.

Aufgrund keiner Notwendigkeit zur Gruppierung verschiedener Punktzielantworten zur Aufstellung des Gleichungssystems können die Gleichungen wie in Quellcode 4.11 dargestellt aufgestellt werden.

```
1 def antenna_mount_angle_ls(response):
2     system = LSEquationSystem()
3     system.add_equations([
4         (response.transmit_id, response.dmount_angle[0].T),
5         (response.receive_id, response.dmount_angle[1].T),
6         (response.chan.ch, np.ones((response.dmount_angle[0].
              shape[1], 1)))
7     ], response.dgain)
8     return system
```

Quellcode 4.11: Aufstellen der Gleichungen zur Bestimmung der Antennenblickrichtung

4.4.2 Verifikation

Zur Verifikation der Antennenblickrichtungskorrektur wird die Abweichung des Antennengewinns vom Antennenmodell vor und nach dem Anwenden der Korrektur verglichen. Dabei stellt eine konstante Abweichung von 0 dB eine perfekte Kalibrierung dar. Ein für alle Punktziele eines Kanals konstanter Versatz ist auch akzeptabel, solange keine Abweichungen im Azimutverlauf auftreten. In Abbildung 4.19 ist die Gewinn-Abweichung über den Azimutverlauf für alle Punktziele eines Kanals dargestellt. Hierbei wurde keine Antennenblickrichtungsabweichung auf den Datensatz angewandt. Der erwartete konstante Wert von 0 dB wird mit der Ausnahme des Punktziels CR_ON15 durchgehend erkannt. Die Abweichung von CR_ON15 kommt dabei durch eine Interferenz aufgrund der physischen Nähe zu CR_ON18 in den Daten zustande. In Abbildung 4.20 ist die Antennengewinnabweichung auf demselben

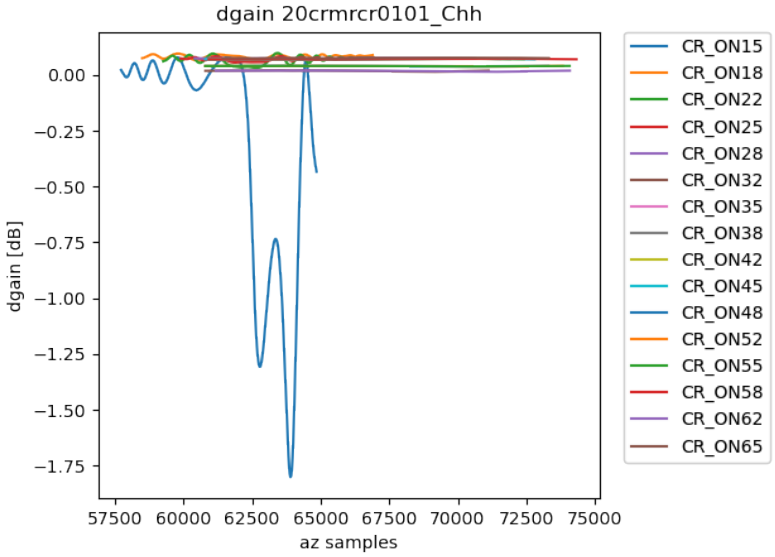

Abbildung 4.19: Vor Antennenwinkelkorrektur ohne erwartetem Versatz

Datensatz nach einmaliger Durchführung der Antennenblickrichtungskorrektur zu sehen. Dabei wurde die perfekte Kalibrierung der Daten durch Anwendung der Kalibrierroutine nicht signifikant verschlechtert, wie an den weiterhin konstanten 0 dB Werten ersichtlich ist. Damit ist nachgewiesen, dass die Implementierung gut

kalibrierte Antennenblickrichtungswerte nicht verschlechtert. In Abbildung 4.21 ist

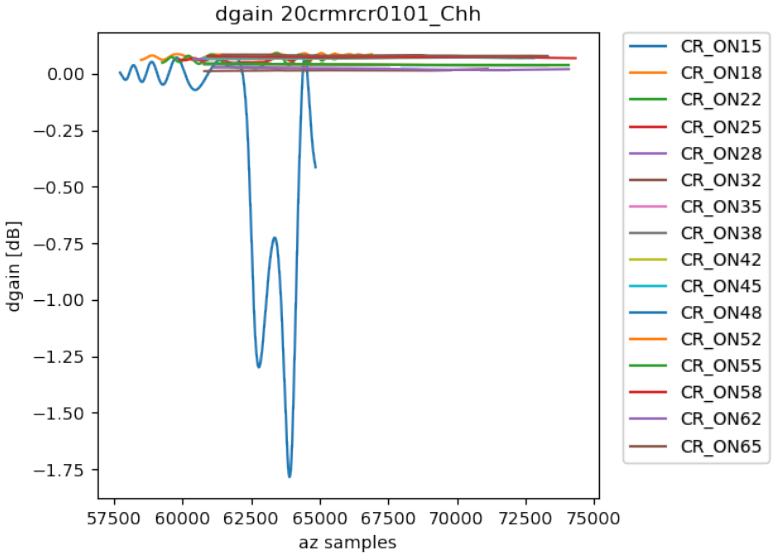

Abbildung 4.20: Nach Antennenwinkelkorrektur ohne erwartetem Versatz

die in der Analyse bestimmte Gewinnabweichung mit Abweichung im Antennenblickrichtungsmodell zu sehen. Dabei erzeugt der Versatz die sichtbaren Azimuttrends, welche von der Kalibrierroutine korrigiert werden sollen. In Abbildung 4.22 ist die Antennengewinnabweichung nach Anwendung der in der Antennenblickrichtungskorrektur bestimmten Korrekturwerte auf die in Abbildung 4.21 dargestellten Daten zu sehen. Der vorherige Kalibrierfehler ist dabei stark minimiert worden mit nahezu konstanten 0 dB Werten. Das Ergebnis ist durch die lineare Approximierung des Antennenmodells nicht perfekt, jedoch sind die Abweichung innerhalb von 0.1 dB. Im Bereich dieser Größenordnung werden Fehler im Antennendiagramm durch Nichtberücksichtigung des Einflusses des Flugzeugs, an welchem die Antenne montiert ist, signifikanter. Auf Basis dieser Ergebnisse kann die Implementierung der Antennenbasislinienkorrektur als korrekt betrachtet werden.

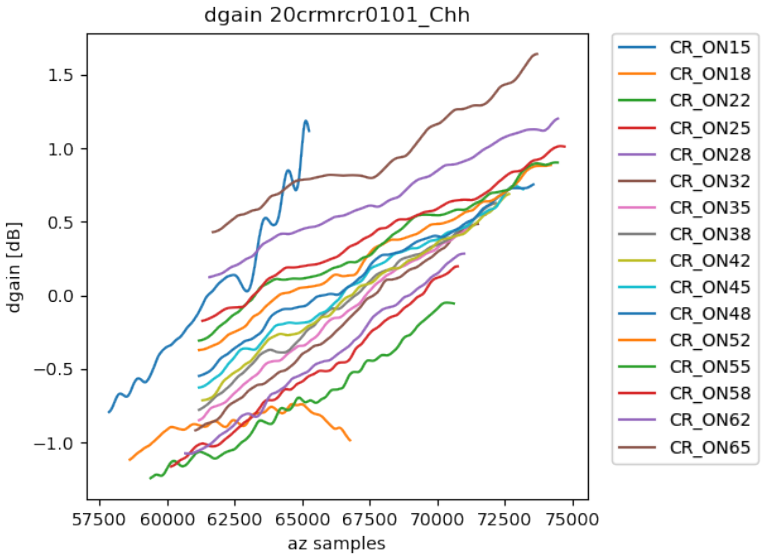

Abbildung 4.21: Vor Antennenwinkelkorrektur mit erwartetem Versatz

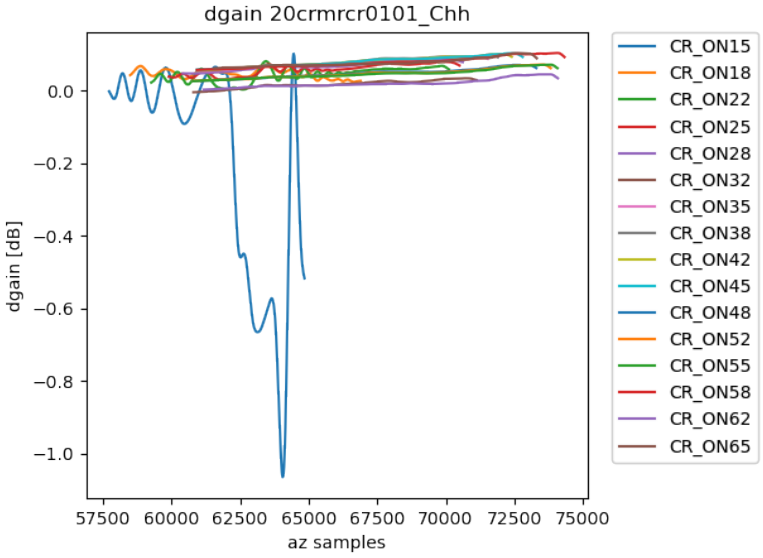

Abbildung 4.22: Nach Antennenwinkelkorrektur mit erwartetem Versatz

4.5 Implementierung und Verifikation einer Antennenbasislinienkorrektur

Die Hebelarmkorrektur, welche durch die in Abschnitt 4.1 und Abschnitt 4.2 Verfahren durchgeführt wird, ist durch die Genauigkeit der Bestimmung der Position des Antwortmaximums beschränkt. Für interferometrische Anwendungsfälle wird jedoch eine deutlich höhere Genauigkeit der relativen Phasenzentrumsposition benötigt. Diese kann durch die interferometrische Analyse der Restphasen erzielt werden, da beispielsweise der F-SAR Radarsensors des DLR die Phase mit einer Präzision von 0.1° bestimmen kann. Daraus ergibt sich eine potentielle Genauigkeit im C-Band (0.05 m Wellenlänge) von

$$r = \frac{0.05m}{720^\circ} \cdot 0.1^\circ < 1 \cdot 10^{-5}m.$$

Dabei wird die abgewickelte Phase eines Punktziels von zwei verschiedenen Kanälen desselben Passes und Frequenzbandes voneinander abgezogen. Beim Abwickeln der Phase muss darauf geachtet werden, dass der konstante Versatz erhalten bleibt. Dies kann durch Anwendung des gewickelten Phasenversatzes an der Stelle der stärksten Rückstrahlung erzielt werden. Die Phasendifferenzen können nicht direkt in ein Gleichungssystem zur Bestimmung des relativen Hebelarmversatzes eingesetzt werden, da über den Verlauf von mehreren Punktzielen mit steigendem Off-Nadir-Winkel eine Phasenwicklung auftreten kann. Daher muss über den Off-Nadir-Verlauf eine weitere Phasenabwicklung unter Berücksichtigung des konstanten Versatzes für Differenzen der gleichen Kanalkombination erfolgen.

So vorbereitete Phasen ϕ können dann nach Umrechnung des Winkels in einen Range-Abstand in Metern zur Aufstellung eines Gleichungssystems mit Gleichungen der folgenden Form verwendet werden:

$$\phi_1 - \phi_2 = (\frac{1}{2}\delta l_1 \cdot los_{t1} + \frac{1}{2}\delta l_1 \cdot los_{r1}) - (\frac{1}{2}\delta l_2 \cdot los_{t2} + \frac{1}{2}\delta l_2 \cdot los_{r2}) + \delta t_{1,2}$$

4.5.1 Implementierung

In der Analyse, welche in Abschnitt 4.1 bereits näher beschrieben wurde, wird der Phasenversatz berechnet und abgewickelt. Dabei wird jedoch bislang der konstante Versatz der Phase sowie die Phase selbst nicht erhalten, sondern direkt in den Abstandsunterschied gewandelt. Um die Phase mit konstant bleibendem Versatz an der Stelle der stärksten Rückstrahlung zu bestimmen, muss die *unwrap*-Funktionalität[17] von numpy in einer eigenen Methode gekapselt werden. Diese Funktion wird in Quellcode 4.12 dargestellt.

```
1 def constant_offset_unwrap(array: np.ndarray, ref_id: int = 0) ->
      np.ndarray:
2     result = np.unwrap(array)
3     return result + array[ref_id] - result[ref_id]
```

Quellcode 4.12: Phasenabwicklung mit konstantem Versatz

Damit nicht sowohl der Vektor, welcher die Phase enthält, als auch der Vektor des Abstandsunterschieds als Ergebnis der Analyse zwischengespeichert und an Spark übergeben werden müssen, wird nur die Phase sowie der zur Umrechnung in den Abstandsunterschied benötigte Faktor und Versatz (jeweils Skalare) berechnet.

Hauptproblem der Implementierung zur Berechnung der Basislinienkorrektur sind die notwendigen verschiedenen Gruppierungen der Punktzielantworten zur Aufstellung des Gleichungssystems. Da alle Verarbeitungsschritte die Gruppierung nach Überflug und Frequenzband gemein haben, wird diese Gruppierung in Spark durchgeführt, die jeweiligen Untergruppierungen werden dann sequentiell durchgeführt.

Für die Berechnung der Phasenunterschiede ist eine weitergehende Gruppierung nach den einzelnen Punktzielen nötig. Für jedes Punktziel kann dann die Differenz für jede Antwortkombination berechnet werden. Dabei muss eine der beiden Phasen auf die Phase des anderen interpoliert werden. Da die Phasen der einzelnen Punktzielantworten mit zunehmenden Off-Nadir Winkel der Punktziele gewickelt sein können, müssen die Phasendifferenzen nach Kanalkombination gruppiert mit konstantem Versatz entwickelt werden. Um Interpolationsprobleme und eine 2d-Phasenabwicklung

zu vermeiden werden dabei lediglich die Antworten an der Stelle der stärksten Rückstrahlung abgewickelt und der resultiernde Versatz auf die Phasendifferenzen aufaddiert. Die entwickelten Phasendifferenzen können in das Gleichungssystem eingefügt werden, dabei ist zu berücksichtigen, dass das Gleichungssystem zur Berechnung der Hebelarmunterschiede durch die Basislinien Werte in Metern enthält und daher der Phasenunterschied mit dem Faktor zur Umrechnung der Phase verrechnet werden muss. Diese Schritte sind zur Verdeutlichung in Quellcode 4.13 skizziert, dort wurden jedoch die Interpolation und die Maskierung der extrapolierten Wertebereiche sowie das Generieren der Differenz-Plots ausgelassen.

```
1  def baseline_correction_ls(responses: List[RawResidualResponse]):
2      system = LSEquationSystem()
3
4      target_responses = defaultdict(list)
5      for r in responses:
6          target_responses[r.target].append(r)
7
8      grouped_pha_diffs = defaultdict(list)
9      for responses in target_responses.values():
10         for r1, r2 in itertools.product(responses, repeat=2):
11             paired_responses[(r1.chan, r2.chan)].append((r1, r2,
                   r1.uwr_pha - r2.uwr_pha))
12
13     for (chan1, chan2), pha_diffs in grouped_pha_diffs.items():
14         # Sort targets with ascending range distance (off-nadir)
15         pha_diffs.sort(key=lambda r1r2dp: r1r2dp[0].r_min)
16         pha_az_min = [pha_diff[r1.az_min] for r1, r2, pha_diff in
               pha_diffs]
17         pha_offset = constant_offset_unwrap(pha_az_min, ref_id=
               len(pha_az_min)//2) - pha_az_min
18
19         for (r1, r2, pha_diff), p_off in zip(pha_diffs,
               pha_offset):
20             system.add_equations([
21                 (r1.transmit_id, r1.los.T / 2),
22                 (r1.receive_id, r1.los.T / 2),
23                 (r2.transmit_id, -r2.los.T / 2),
```

```
24                  (r2.receive_id, -r2.los.T / 2),
25                  ((r1.chan, r2.chan), np.ones((len(pha_diff), 1)))
26              ], (pha_diff + p_off) * r1.dr_factor)
27      return system
```

Quellcode 4.13: Aufstellen des Gleichungssystems zur Berechnung der Basislinienkorrektur

Zur Lösung des entstehenden Gleichungssystems ist es noch zusätzlich notwendig, für jede Antennenkombination innerhalb der Frequenzbänder für jede Koordinatenachse eine Gleichung einzufügen, in welcher diese Achse mit 1 und alle anderen mit 0 gewichtet werden, um eine singuläre Matrix zu vermeiden.

4.5.2 Performanceoptimierung

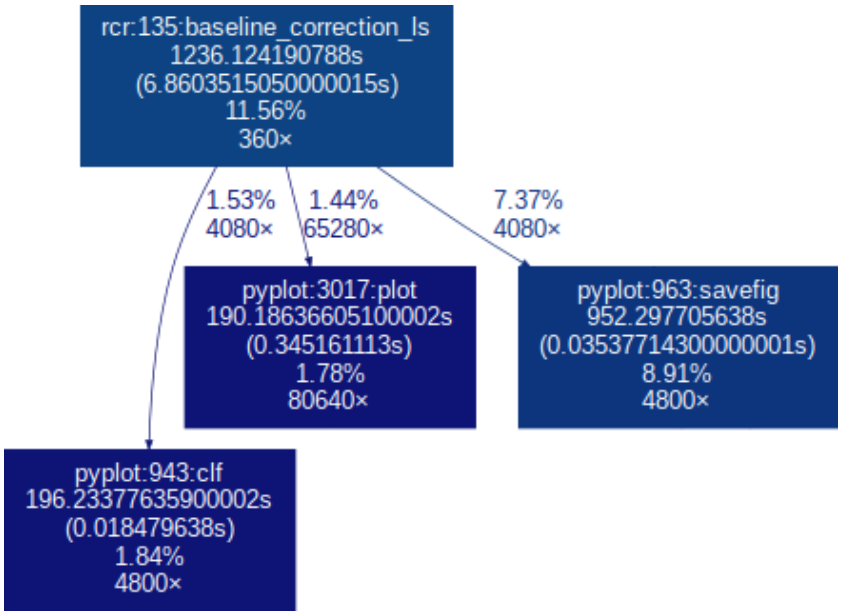

Abbildung 4.23: Unoptimierte Laufzeit von baseline_correction_ls

Bei der Betrachtung der Laufzeit der Basislinienkorrektur, welche in Abbildung 4.23 zu sehen ist, fällt auf, dass von den 1236s Gesamtlaufzeit 92% der Zeit der Funktion `baseline_correction_ls` in den plotbezogenen Funktionen `matplotlib.pyplot.plot`,

`matplotlib.pyplot.savefig` und `matplotlib.pyplot.clf` verbracht wird. Das Gleichungssystem wird im Rahmen der L1-Norm Minimierung mehrfach aufgestellt und somit auch die Plots mehrfach erstellt bzw. überschrieben. Da sich durch die L1-Norm Minimierung der Inhalt der Plots nicht ändert, kann durch eine einfache Abfrage auf die Vorexistenz der Bilddatei die Notwendigkeit zum Erstellen des Plots entschieden werden. Der dadurch erzielte Performancegewinn ist in Abbildung 4.24 mit dem Rückgang auf 292s Gesamtlaufzeit zu sehen.

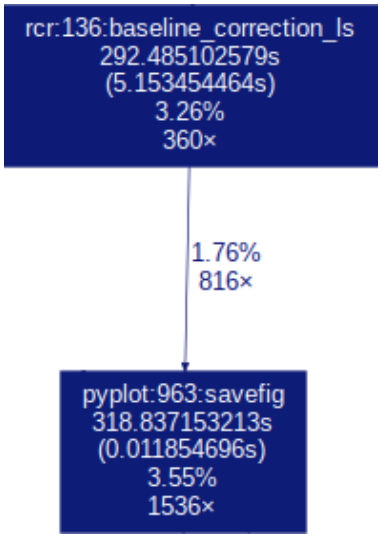

Abbildung 4.24: Optimierte Laufzeit von baseline_correction_ls

4.5.3 Verifikation

Zur Bestimmung der Korrektheit der Implementierung der Basislinienkorrektur werden die gemessenen Unterschiede in der Basislinie zweier Antennen dargestellt. Wenn die Basislinien nicht korrekt kalibriert sind, sind Änderungen im gemessenen Range-Abstand über den Squint- und Off-Nadir-Winkel zu erwarten. Vergleichbar mit den Darstellungen aus Abschnitt 4.1 stellen die einzelnen Linien einzelne Punktzielantworten dar, wobei auf der y-Achse stattdessen die reine Phasendifferenz der beiden Kanäle angegeben wird. In Abbildung 4.25 ist der gemessene Basislinienunterschied

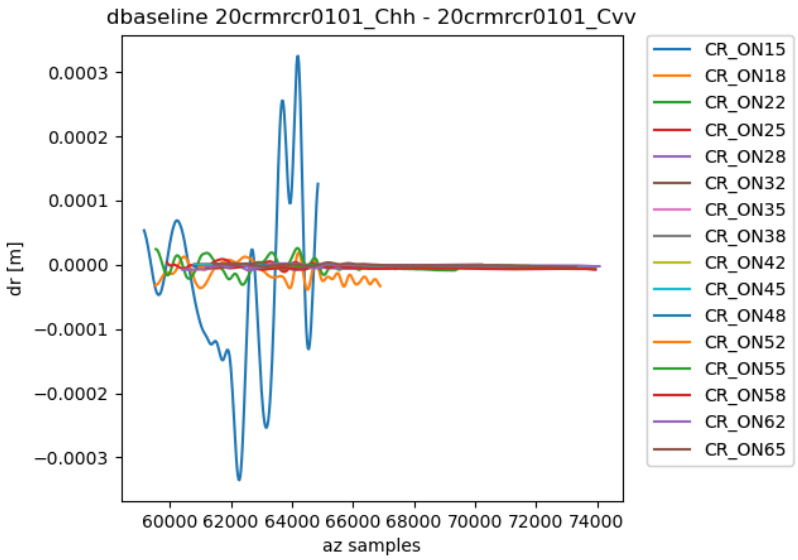

Abbildung 4.25: Vor Basislinienkorrektur ohne erwartetem Versatz

zwischen dem Chh und Cvv-Kanal zu sehen. Da kein Hebelarmversatz auf die eingehenden Daten angewendet wurde, wird ein durchgehender Versatz von 0 erwartet. Dieser wird auch in Abbildung 4.25 erzielt. Das Punktziel CR_ON15 ist hierbei in den Rohdaten sehr nah am Reflektor CR_ON18, wodurch sich Interferenzen ergeben. Nach Anwendung der errechneten Basislinienkalibrierung wird, wie in Abbildung 4.26 zu sehen, die Kalibrierung nicht verfälscht. Damit steht fest, dass eine perfekte Kalibrierung nicht verschlechtert wird. In Abbildung 4.27 ist der Basislinienunterschied mit versetzten Hebelarmen zu sehen. Zu beachten ist hierbei, dass die Differenz erst nach der Hebelarmkorrektur mit optimaler Kanalkonsistenz berechnet wurde, da der Basislinienversatz keine größeren Sprünge enthalten darf, da keine Phasensprünge mit dem Mehrfachen von 2π korrekt abgewickelt werden können. Die Basislinien sind hierbei nicht korrekt, was sich am mit steigendem Off-Nadir Winkel ändernden Distanzunterschied zeigt. Nach der Basislinienkorrektur ergeben sich die Versatzwerte aus Abbildung 4.28. Der sich über den steigenden Off-Nadir Winkel ändernde Unterschied wurde erfolgreich kompensiert. Der minimale konstante Versatz, welcher erhalten geblieben ist, ist akzeptabel, da dieser anderweitig kompensiert wird. Daher kann die Implementierung der Basislinienkorrektur als korrekt betrachtet werden.

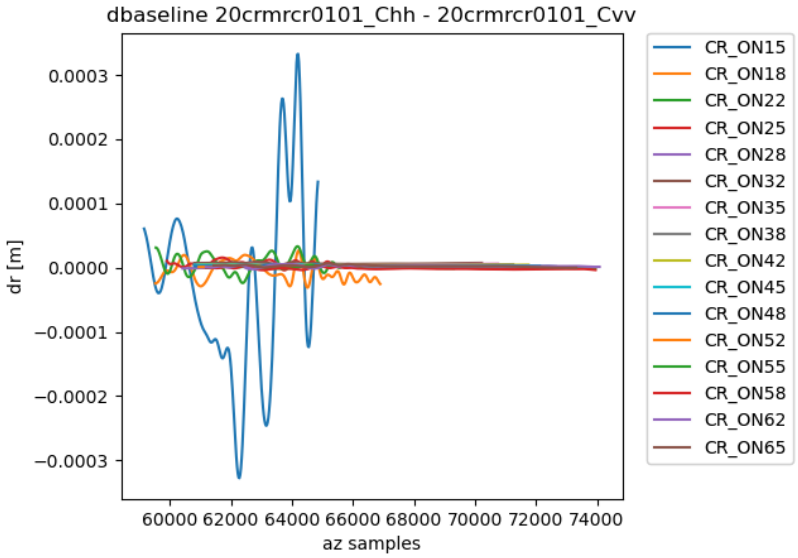

Abbildung 4.26: Nach Basislinienkorrektur ohne erwartetem Versatz

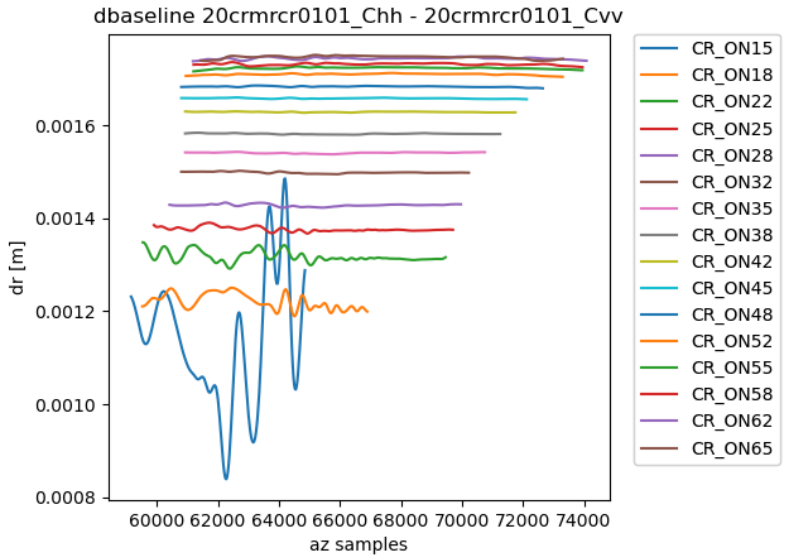

Abbildung 4.27: Vor Basislinienkorrektur mit erwartetem Versatz

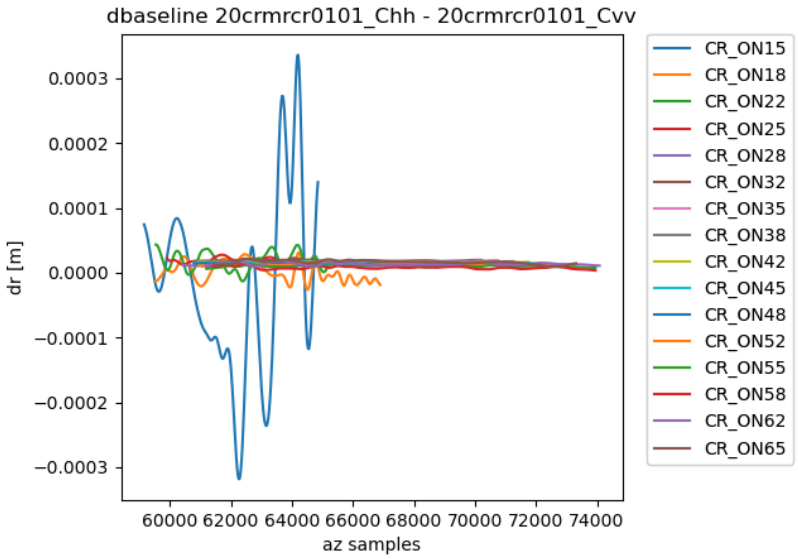

Abbildung 4.28: Nach Basislinienkorrektur mit erwartetem Versatz

4.6 Verifikation der Gesamtimplementierung im Vergleich mit der IDL-Implementierung

Zur Bewertung der Laufzeit und Genauigkeit der neuen CARAMBA-Implementierung wird diese mit der bereits existierenden IDL-Implementierung verglichen.

4.6.1 Erzielen einer vergleichbaren Genauigkeit

Der Vergleich der Genauigkeit der IDL- und CARAMBA-Implementierung findet mit mehreren Methoden statt. Die für simulierte Daten ermittelten Werte für die Hebelarm- und Antennenblickrichtungskorrektur werden mit den tatsächlichen Werten verglichen und die jeweilige Summe der quadrierten Fehler (SSE) bestimmt. Je kleiner die Summe der quadrierten Abweichung ist, desto besser ist die Annäherung an die tatsächlichen Werte. Auch die Auswirkungen dieser Abweichungen auf Abstandsunterschied, Antennengewinnverlauf und Basislinien werden verglichen.

Die absolute Abweichung der bestimmten Hebelarmwerte für die Chh-Antenne ist in Abbildung 4.29 zu sehen. Dabei hat die CARAMBA-Implementierung weit geringere Abweichungen als die IDL-Implementierung. Das zeigt sich auch in der Summe der quadrierten Abweichungen über alle kalibrierten Antennen und Koordinatenachsen hinweg. Die CARAMBA-Implementierung hat hierbei eine SSE von 0.11, während die IDL-Implementierung eine Abweichung von 17.24 hat.

Die absolute Abweichung der bestimmten Antennenblickrichtung für die Chh-Antenne ist in Abbildung 4.30 zu sehen. Hierbei erzielt die CARAMBA-Implementierung eine leicht bessere Genauigkeit gegenüber der IDL-Implementierung. Dies spiegelt sich auch in der SSE von CARAMBA 0.0013 und IDL 0.0073 wieder.

Die Abweichungen in den Hebelarmen führen zu den in Abbildung 4.31 sichtbaren Range-Unterschieden. Aufgrund der Stärke des Range-Fehlers der IDL-Implementierung muss hierbei von einem Fehler in der Implementierung ausgegangen werden. Zur besseren Sichtbarkeit des Range-Fehlers durch die CARAMBA-Implementierung wurde dieser nochmals alleine in Abbildung 4.32 dargestellt. Ein Range-Unterschied

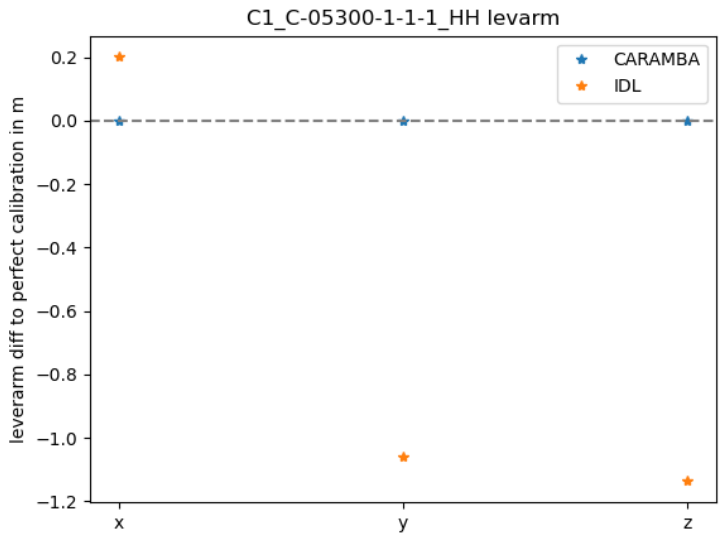

Abbildung 4.29: Abweichung der ermittelten Hebelarme der CARAMBA- und IDL-Implementierungen

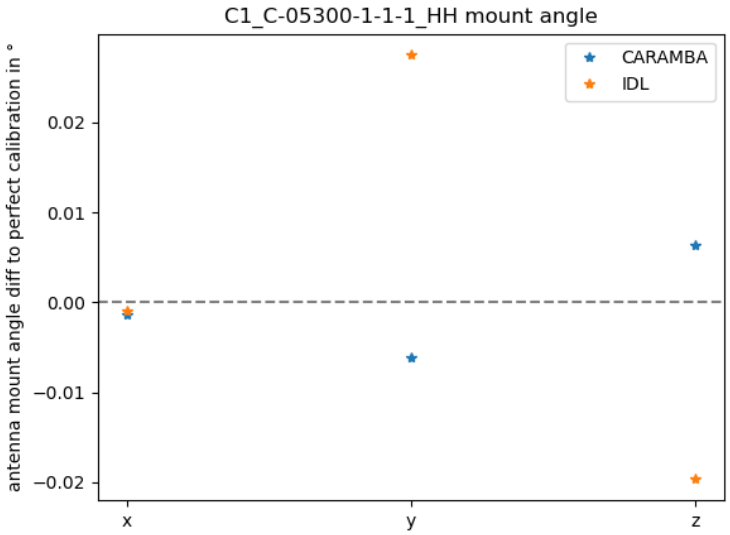

Abbildung 4.30: Abweichung der ermittelten Antennenblickrichtungen der CARAMBA- und IDL-Implementierungen

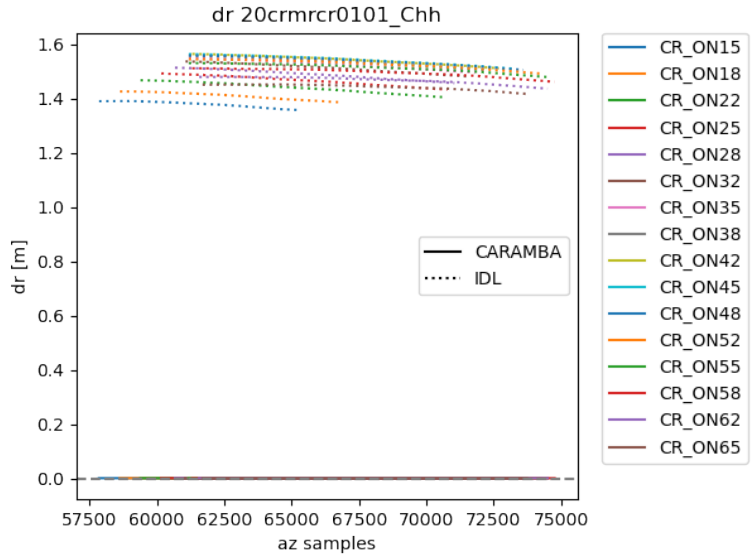

Abbildung 4.31: dr-Abweichungen durch CARAMBA- und IDL-Implementierung

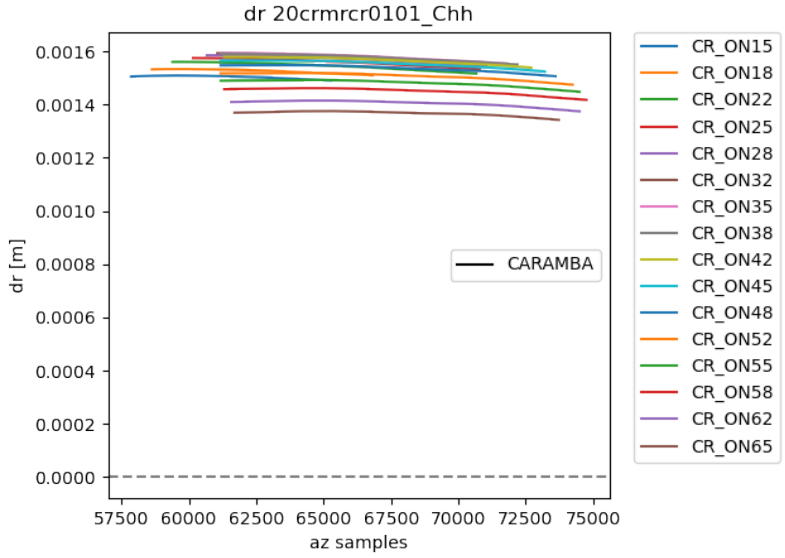

Abbildung 4.32: dr-Abweichungen durch CARAMBA-Implementierung

kleiner als 0.005 m wird als insignifikant gewertet, da in dieser Größenordnung nichtdeterministische Fehler in den Navigationsdaten vorhanden sind. Die durch die CARAMBA-Implementierung verursachten Range-Unterschiede sind im generellen kleiner als die 0.005 m, jedoch gibt es im X-Band, dem einzigen Band mit zwei physisch getrennten Antennen, bei aktivierter nachträglicher Basislinienkorrektur mit 0.03 m eine größere Abweichung. Ohne aktivierte Basislinienkorrektur sind die Werte des X-Bands im akzeptablen Bereich.

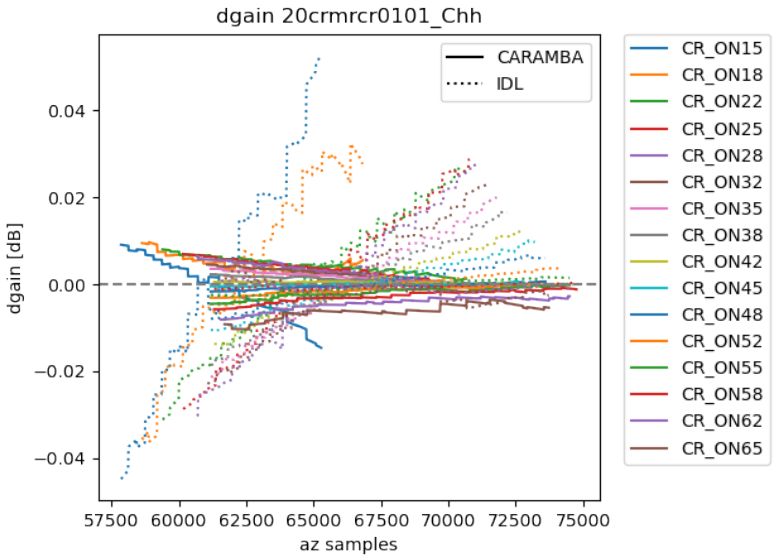

Abbildung 4.33: Antennengewinnabweichungen durch CARAMBA- und IDL-Implementierung

In Abbildung 4.33 sind die durch die Antennenblickrichtungsunterschiede verursachten Gewinnabweichungen zu sehen. Hierbei hat die CARAMBA-Implementierung eine geringere Gewinnabweichung als die IDL-Implementierung. Auch bei der SSE über alle Azimut-Werte aller Punktziele in allen Kanälen schneidet CARAMBA mit 596 besser als IDL mit 7173 ab. Ein Gewinnunterschied kleiner als 0.1 dB wird als insignifkant betrachtet, da Systemfehler wie beispielsweise durch den Einfluss des Flugzeugs auf das Antennendiagramm dann deutliche Auswirkungen haben. Die durch die CARAMBA-Implementierung verursachte Antennengewinnabweichung bleibt überwiegend innerhalb dieses Bereiches, einzelne Punktzielantworten weisen

in einigen Kanälen in einigen Azimutbereichen leicht stärkere Abweichungen auf.

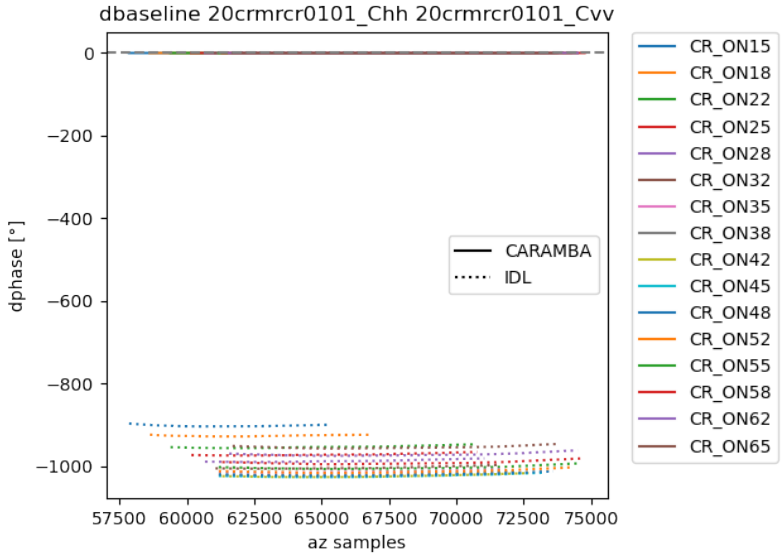

Abbildung 4.34: Basislinien-Abweichungen durch CARAMBA- und IDL-Implementierung

In Abbildung 4.34 sind die durch unterschiedliche Hebelarme verursachten Basislinienunterschiede zu sehen. Hierbei scheint der Fehler, welcher die Hebelarme der IDL-Implementierung beeinflusst auch die Basislinien zu versetzen. Lediglich in manchen Kanalkombinationen im X-Band erzielt die IDL-Implementierung etwas bessere Ergebnisse. Zur besseren Sichtbarkeit der in der CARAMBA-Implementierung auftretenden Basislinienfehler sind diese nochmals getrennt in Abbildung 4.35 dargestellt. Bei den Basislinien wird ein Phasenunterschied kleiner als 1° als unerheblich betrachtet. Die von der CARAMBA-Implementierung ermittelten Werte weichen im C-Band weniger als 1° ab, im L-Band treten leicht größere Abweichungen in der Größenordnung von 3° auf. Im X-Band treten signifikante Fehler von bis zu 400° auf, hierbei ist eine weiterführende Untersuchung nach der Ursache dieser Abweichungen nötig.

Hiermit ist gezeigt worden, dass die CARAMBA-Implementierung überwiegend genauere Ergebnisse als die IDL-Implementierung bestimmt. Der Großteil der Restfehler ist in Größenordnungen, in welchen äußere Einflüsse größere Abweichungen

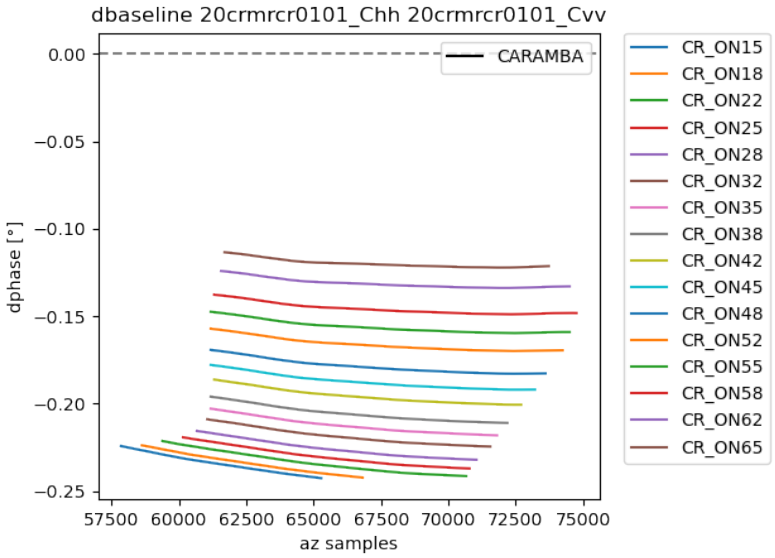

Abbildung 4.35: Basislinien-Abweichungen durch CARAMBA-Implementierung

verursachen. Einzig in der Basislinienkalibrierung mit mehreren physischen Antennen treten größere Abweichungen auf, wobei die Ursache dieser noch untersucht werden sollte.

4.6.2 Erzielen einer um eine Größenordnung geringeren Laufzeit

Im Gegensatz zum Rest der Arbeit werden die Implementierungen in diesem Abschnitt nicht mit einem simulierten Datensatz, sondern mit echten Messdaten verglichen. Dies ermöglicht die Berücksichtigung des zeitlichen Aufwandes der Kompression der Rohdaten in Range. Zudem laufen damit beide Implementierungen unter möglichst realistischen Anwendungsfällen. Der Vergleich der Implementierungen findet auf der gleichen Hardware und dem gleichen Datensatz statt. Der Referenzdurchlauf der IDL-Implementierung benötigte zur Kalibrierung des echten Datensatzes 27 Stunden und 20 Minuten in Echtzeit.

Zur Abschätzung der Gesamtlaufzeit und des Einflusses des Einlesens von echten Messdaten wurde zunächst ein Durchlauf mit ausschließlich Daten im C-Band

durchgeführt (etwa 11% des Gesamtdatensatzes), welcher nach 11 Minuten und 40 Sekunden beendet war. Zur gedanklichen Einordnung der Zahlenverhältnisse wurden im folgenden die Zeitanteile in eine approximierte Menge von beschäftigten Prozessen umgerechnet. Bei der Analyse des Laufzeitprofils fiel auf, dass von 28 von Spark gestarteten Arbeitsprozessen durchschnittlich nur 9 Daten verarbeiten. Von diesen 9 Prozessen sind 5 mit dem Einlesen von Antennendiagrammen beschäftigt, 1.5 Prozesse jeweils mit dem Komprimieren der Rohdaten und dem Rotieren der Rohdaten in Range mit Subpixelpräzision. Das Einlesen der Antennendiagramme wurde durch Anpassen der Datei mit der Methode aus [9], sodass ein direktes Memorymappen möglich wird, stark optimiert. Zur schlechten Parallelisierung und Auslastung der Arbeitsprozesse wurde die Hypothese aufgestellt, dass der Datensatz schlecht partitioniert ist. Ein Testlauf mit manueller Anpassung der Partitionierung zeigte, dass eine Verbesserung der Partitionierung kaum Laufzeitgewinn und bessere Parallelisierung bringt. Eine nähere Analyse des Profiling-Ergebnisses zeigte, dass der Hauptprozess nahezu ausschließlich (98% der Laufzeit) auf Daten aus der verteilten Spark Umgebung wartet. Knapp 20% der Laufzeit des Hauptprozesses wird dabei in der CARAMBA-Funktion `broadcast_kvrdd` verbracht, welche einen verteilten Datensatz (RDD) zu Broadcast-Variablen umwandelt, um diese in einer Blockverarbeitung besser nutzen zu können. Dazu wird zur Vermeidung des zeitgleichen Ladens aller Datensätze mit `RDD.toLocalIterator` über das RDD iteriert. Eine Betrachtung der Arbeitsprozesse mit einem Task-Manager zeigte auf, dass während dieses Verarbeitungsschrittes nur jeweils ein Arbeitsprozess beschäftigt ist. Ein Austausch des Iterators durch ein `RDD.collectAsMap` verringert die Laufzeit von `broadcast_kvrdd` durch die bessere Parallelisierung um mehr als 90%. Der größere Speicherbedarf durch das zeitgleiche Laden aller Datensätze wurde nach Testläufen als akzeptabel bewertet, da mit dieser Funktion ausschließlich Metadaten parallelisiert werden und durch die Umwandlung in Broadcast-Variablen ebenso alle Datensätze im Arbeitsspeicher gehalten werden.

Ein mit den oben genannten Verbesserungen durchgeführter Referenzdurchlauf mit dem gesamten Datensatz erzielte eine Gesamtlaufzeit von 16 Minuten und 41 Sekunden in Echtzeit. Damit kann die angestrebte Laufzeitverbesserung von einer Größenordnung gegenüber der IDL-Implementierung als erreicht betrachtet werden.

5 Ergebnis

Die modellbasierte Antennenkalibrierung auf SAR-Rohdatenbasis konnte erfolgreich implementiert und verifiziert werden. Die Hebelarmkorrektur mithilfe der Methode der kleinsten Quadrate konnte dabei implementiert und die Ergebnisse verifiziert werden. Die Nebenbedingung der optimalen Kanalkonsistenz konnte erfolgreich implementiert und verifiziert werden sowie dabei aufgefallene Performaceengpässe entfernt werden. Die Verbesserung des Gleichungssystemlösers durch Einführung einer L1-Norm Minimierung konnte ebenfalls implementiert und verifiziert werden. Das Verfahren zur Korrektur der Antennenblickrichtung konnte erfolgreich implementiert und verifiziert werden. Die Antennenbasislinienkorrektur der Hebelarme konnte erfolgreich implementiert und verifiziert werden sowie Performanceengpässe beseitigt werden. Im Vergleich der Gesamtimplementierung mit der existierenden IDL-Implementierung wurde eine bessere Genauigkeit sowie eine um eine Magnitude bessere Laufzeit erzielt.

1. Implementierung und Verifikation der Hebelarmkorrektur mithilfe der Methode der kleinsten Quadrate: erfolgreich abgeschlossen

   1.1 Implementierung: erfolgreich abgeschlossen

   1.2 Verifikation: erfolgreich abgeschlossen

2. Implementierung und Verifikation der optimalen Kanalkonsistenz: erfolgreich abgeschlossen

   2.1 Implementierung: erfolgreich abgeschlossen

   2.2 Performanceoptimierung: erfolgreich abgeschlossen

2.3 Verifikation: erfolgreich abgeschlossen

3. Implementierung und Verifizierung einer L1-Norm Regularisierung: erfolgreich abgeschlossen

   3.1 Implementierung: erfolgreich abgeschlossen

   3.2 Verifikation: erfolgreich abgeschlossen

4. Implementierung und Verifikation einer Antennenblickrichtungskorrektur: erfolgreich abgeschlossen

   4.1 Implementierung: erfolgreich abgeschlossen

   4.2 Verifikation: erfolgreich abgeschlossen

5. Implementierung und Verifikation der Antennenbasislinienkorrektur: erfolgreich abgeschlossen

   5.1 Implementierung: erfolgreich abgeschlossen

   5.2 Performanceoptimierung: erfolgreich abgeschlossen

   5.3 Verifikation: erfolgreich abgeschlossen

6. Vergleich der Implementierung mit der existierenden IDL-Implementierung: erfolgreich abgeschlossen

   6.1 Erzielen einer vergleichbaren Genauigkeit: erfolgreich abgeschlossen

   6.2 Erzielen einer um eine Größenordnung geringeren Laufzeit: erfolgreich abgeschlossen

6 Ausblick

Mit der in dieser Arbeit erstellten modellbasierten Antennenkalibrierung können in Zukunft die Antennen deutlich schneller und genauer kalibriert werden. Einige Verbesserungen sind jedoch noch möglich. Die Implementierung zusätzlicher Kalibrierschritte und die Berücksichtigung weiterer Effekte wie z.B. eine Troposphärenkorrektur sind vorstellbar. Dies würde die Präzision der Kalibrierergebnisse zusätzlich verbessern.

Ansonsten sind auch noch zahlreiche kleine Performanceoptimierungen möglich. CARAMBA lädt in der Initialisierung zahlreiche Pythonmodule, wovon einige für die jeweils laufende Prozessierung nicht nötig sind. Dies verlangsamt die Initialisierung und führt zu einem höheren Arbeitsspeicherbedarf während der Laufzeit. Dies ist alleine noch kein Problem, durch die vielfache Initialisierung der Arbeitsprozesse innerhalb der Spark-Umgebung steigert dies jedoch die Laufzeit und den benötigten Arbeitsspeicher stark. Auf dem Prozessierungscluster konnte dabei ein besonders starker Laufzeiteinfluss der Initialisierung mit knapp 14% der Gesamtlaufzeit festgestellt werden. In der Analyse werden derzeit bei mehrfacher Ausführung der Analyse keine (Zwischen-)Ergebnisse voriger Durchläufe bis auf ggf. geänderte Antennenkalibrierungen verwendet. Die geänderten Antennenkalibrierungen verändern jedoch nicht die rangekomprimierten Rohdaten. Daher wäre es vorstellbar, diese zwischenzuspeichern und in der nächsten Iteration wiederzuverwenden. Dies könnte die derzeiten 30% der Gesamtlaufzeit, welche für Rohdatenkompression benötigt wird, deutlich reduzieren. Mögliche Probleme dabei wären der deutlich größere Arbeitsspeicherbedarf für die Zwischenspeicherung der Rohdaten sowie potentiell durch geänderte Antennenparameter verursachte stark geänderte Rohdatenausschnitte, da zurzeit nur die Rohdatenbereiche an der erwarteten Punktzielantwortposition eingelesen werden. Ein weiterer Optimierungskandidat ist die `fftroll`-Funktion, welche die ein-

gelesenen Rohdaten so mit Subpixelpräzision in Range verschiebt, dass die erwartete Maximumsposition der Antwort auf einem festen Index liegt. Diese benötigt auf dem Prozessierungscluster etwa 38% der Gesamtlaufzeit. Dabei wird die Mehrheit der Laufzeit nicht für externe Funktionsaufrufe wie `fft` und `ifft` benötigt, sondern in der Funktion selbst, was auf die dort durchgeführten Multiplikationen zeigt. Hierbei sind weitere Untersuchungen notwendig, warum diese `numpy`-Multiplikationen so viel Rechenzeit benötigen. In den Kalibrierroutinen wird an mehreren Stellen ein kartesisches Produkt gebildet. Dabei wird das vollständige kartesische Produkt nicht benötigt, da es reicht, wenn eine Gleichung in eine Richtung in das Glechungssystem eingetragen wird. Hierbei besteht die Möglichkeit, eine Funktion zu schreiben, welche dies in optimierter Form durchführt.

Literaturverzeichnis

[1] A. Moreira u. a. „A tutorial on synthetic aperture radar“. In: *IEEE Geoscience and Remote Sensing Magazine* 1.1 (03/2013), S. 6–43. DOI: 10.1109/MGRS.2013.2248301.

[2] M. Jäger, R. Scheiber und A. Reigber. „Robust, Model-Based External Calibration of Multi-Channel Airborne SAR Sensors Using Range Compressed Raw Data“. In: *Remote Sensing* 11.22 (11/2019), S. 2674. DOI: 10.3390/rs11222674.

[3] Apache Software Foundation. *PySpark Documentation*. URL: http://spark.apache.org/docs/latest/api/python/ (besucht am 30. 08. 2021).

[4] Intel Corporation. *Intel VTune Profiler*. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.9mi3hk (besucht am 30. 08. 2021).

[5] Python Software Foundation. *The Python Profilers*. URL: https://docs.python.org/3/library/profile.html (besucht am 30. 08. 2021).

[6] Apache Software Foundation. *pyspark.SparkContext*. URL: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html (besucht am 16. 06. 2021).

[7] Apache Software Foundation. *Source code for pyspark.profiler*. URL: https://spark.apache.org/docs/2.3.0/api/python/_modules/pyspark/profiler.html (besucht am 16. 06. 2021).

[8] José Fonseca. *About gprof2dot*. URL: https://github.com/jrfonseca/gprof2dot (besucht am 30. 08. 2021).

[9] Felix Weinmann. *Implementierung einer SAR-Rohdatenanalyse in PySpark*. 01. 02. 2021.

[10] Ryan P. Adams. *Ordinary Least Squares Linear Regression*. URL: https://www.cs.princeton.edu/courses/archive/fall18/cos324/files/linear-regression.pdf (besucht am 21. 06. 2021).

[11] Gilbert Strang. *Introduction to Linear Algebra, Fifth Edition*. Wellesley: Wellesley-Cambridge Press, 2016.

[12] NumPy community. *numpy.linalg.lstsq*. 2021. URL: https://numpy.org/doc/stable/reference/generated/numpy.linalg.lstsq.html (besucht am 27. 07. 2021).

[13] Apache Software Foundation. *RDD Programming Guide*. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html (besucht am 28. 07. 2021).

[14] Tim Pietzcker. *Using try vs if in python*. URL: https://stackoverflow.com/questions/1835756/using-try-vs-if-in-python (besucht am 09. 07. 2021).

[15] Python Software Foundation. *PerformanceTips*. URL: https://wiki.python.org/moin/PythonSpeed/PerformanceTips (besucht am 09. 07. 2021).

[16] Marc Jäger. *Robust, Model-Based External Calibration of Multi-Channel Airborne SAR Sensors Using Range Compressed Raw Data*. 10. 11. 2020.

[17] NumPy community. *numpy.unwrap*. 2021. URL: https://numpy.org/doc/stable/reference/generated/numpy.unwrap.html (besucht am 30. 08. 2021).