KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Data fusion

Tensor factorizations by complex optimization

Laurent Sorber

Dissertation presented in partial
fulfilment of the requirements for the
degree of Doctor in Engineering

May 2014

Data fusion

Tensor factorizations by complex optimization

Laurent SORBER

Examination committee:
Prof. dr. ir. Paul Van Houtte, chair
Prof. dr. ir. Lieven De Lathauwer, supervisor
Prof. dr. ir. Marc Van Barel, co-supervisor
Prof. dr. ir. Johan Suykens
Prof. dr. ir. Sabine Van Huffel
Prof. dr. ir. Pierre-Antoine Absil
(Université Catholique de Louvain)
Prof. dr. Daniel Kressner
(Ecole Polytechnique Fédérale de Lausanne)
Prof. dr. ir. Lek-Heng Lim
(University of Chicago)

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

May 2014

© 2014 KU Leuven – Faculty of Engineering Science
Uitgegeven in eigen beheer, Laurent Sorber, Kasteelpark Arenberg 10, BE-3001 Leuven (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without written permission from the publisher.

ISBN 978-94-6018-826-8

D/2014/7515/50

Voorwoord

Nu het einde van mijn studies bijna in zicht is en er een belangrijk hoofdstuk van mijn leven afgesloten gaat worden, is het een goed moment om eens te reflecteren over hoe ik hier nu eigenlijk geraakt ben. Ik heb er niet lang bij stilgestaan of ik wou doctoreren. Ik voelde mij er enorm tot aangetrokken (zonder volledig te beseffen wat het inhield) en dus was de keuze makkelijk gemaakt. Maar waarom voelde ik me er zo tot aangetrokken? Om die vraag te beantwoorden, moest ik diep in mijn geheugen tasten voor ik het antwoord vond. Wellicht was het meest bepalend moment voor mij toen ik in 1996 achter de schoolbanken in Australië zat en er een van mijn klasgenoten een laptop had meegenomen waarop hij een eigen website ging schrijven. Ik had nog nooit iemand zien programmeren en keek gefascineerd toe terwijl hij code intypte op zijn machine. Ik wist meteen dat ik dat ook wou kunnen. Toen ik thuis kwam vertelde mijn vader me dat er nog vele andere programmeertalen zijn, en ik herinner me dat ik fantaseerde dat ik die later allemaal van binnen en buiten wou kennen. Zo'n vaart is het natuurlijk niet gelopen, maar dat moment zal me wel altijd bijblijven als iets dat mij definieerde. De ijzersterke interesse in al wat met computers te maken heeft was toen geboren, en tegelijk ook mijn diep verlangen naar het veelzijdige—ik wou niet zo maar één taal kennen, ik wou ze allemaal verstaan! Gaandeweg begonnen deze interesses duidelijker vorm te krijgen. Zo leerde ik dat veel talen een gemeenschappelijke basis hebben en dat die basis vaak voldoende is om ook nieuwe talen enigszins te kunnen verstaan. Belangrijker nog was het inzicht dat programmeren eigenlijk slechts een manier is om wiskundige ideeën uit te drukken, zoals een schilder borstel en verf ook enkel als gereedschap gebruikt om zijn verbeelding op doek vast te leggen. Programmeren staat in die zin tot de schilder zijn techniek zoals de wiskunde staat tot zijn verbeelding. Als ik de taal van de wiskunde kon, zou ik op eigen kracht ook veel complexere programma's kunnen schrijven, bedacht ik me. In het middelbaar graviteerde ik daardoor vanzelf naar wetenschappen en wiskunde, en op de universiteit naar wiskundige ingenieurstechnieken. Als ik door mijn doctoraat een betere figuurlijke schilder ben geworden, is dat onmiskenbaar door de steun en liefde die ik van mijn promotoren, collega's, vrienden en familie heb gekregen. Ik wil hier dan ook graag gebruik maken van de gelegenheid om al deze mensen te bedanken.

In de eerste plaats wil ik mijn promotoren prof. Lieven De Lathauwer en prof. Marc Van Barel bedanken. Tijdens mijn studies zag ik dat de lineaire algebra—een tak van de wiskunde die gecentreerd is rond het verwerken van rijen en tabellen van getallen—fundamenteel is aan zowat alle algoritmen in wetenschappelijk rekenen. Lieven heeft mijn ogen geopend en me voor het eerst kennis laten maken met de wereld van de multilineaire algebra, wat een directe veralgemening is naar meerdimensionale tabellen, ofwel tensoren. Hij is een van 's werelds grootste experten in dit domein, en het is dan ook niet voor niets dat hij wel eens de *godfather of tensors* genoemd wordt. Lieven, bedankt voor het vertrouwen dat u keer op keer in mij gesteld hebt, en voor de grote vrijheid die u me gunde in het onderzoek. Het was mijn voorrecht om onder uw begeleiding te mogen doctoreren. Marc was tijdens mijn doctoraat een perfect complement voor Lieven. Zijn expertise in het numerieke gaat ongeëvenaard. Hij heeft mij talloze keren nieuwe inzichten bijgebracht die mij het groter geheel hielpen verstaan. Marc, bedankt voor uw positieve attitude, uw onuitputtelijk doorzettingsvermogen en toewijding aan uw doctoraatsstudenten, en uw goedgemutste manier van aanpakken die me de moeilijkere momenten in perspectief hielp zetten. Het was me een eer om met u te mogen samenwerken.

I also wish to thank the members of the examination committee for their time and commitment, and their valuable suggestions and criticism that have helped to improve this manuscript. Dear prof. Paul Van Houtte, prof. Johan Suykens, prof. Sabine Van Huffel, prof. Pierre-Antoine Absil, prof. Daniel Kressner, and prof. Lek-Heng Lim, thank you.

To my colleagues at the department of computer science: thank you Annelies, Bart, Ben, Dirk A., Dirk N., Hendrik, Jeroen, Koen, Matthias, Maryam, Nele, Nick, Nico A., Nico S., Piers, Roel, Sam, Thomas, and Yaidel for the many interesting discussions we had and for the pleasant dinners. A special thanks to Nick for the lively conversations on tensors and finance, and to Matthias for both offering and solving numerous delightfully compelling brain teasers with me.

To my colleagues at the department of electrical engineering: thank you Dimitri, Ignat, Kim, Mariya, Mikael, and Philippe for sharing my company at the conferences we attended and for the sound advice you gave me in the road to a doctoral degree. I thank Nico, Otto, and Paul for their energy and enthousiasm and wish you the best of luck with your newfound careers.

Ik heb ook het geluk gehad samen te kunnen opgroeien met een geweldige vriendengroep. Bedankt Dries, Jeroen, Maarten, Niels, Steven, Tim en Thomas voor de mooie tijden die we samen hebben doorgebracht, en moge er nog veel bijkomen!

Papa en Mama, bedankt om mij altijd te hebben aangemoedigd in alles wat ik deed, voor jullie interesse in mijn studies en onderzoek, en jullie liefde en steun gedurende dit alles. Papa, ik wil jou ook speciaal bedanken om altijd alles wat ik schreef meermaals na te lezen en zowel inhoudelijke als vormelijke suggesties voor verbetering te geven. Eric, ik ben blij ik jou mijn broer mag noemen en dat we alles samen meegemaakt hebben, zonder jou had ik nooit kunnen worden wie ik nu ben. Bedankt om er altijd voor me te zijn!

Mijn lief Claartje, jij bent het zonnetje in mijn leven. Jouw zuiverheid en onschuld weten altijd een glimlach op mijn gezicht te toveren. Dank je om mij te helpen als ik het moeilijk had, en om een onuitputbare bron van warme liefde voor mij te zijn. Mijn gelukkigste herinneringen zijn met jou, en ik hoop dat ik jou ook altijd zo blij kan maken als jij mij maakt.

Laurent Sorber
Leuven, mei 2014

Abstract

In our information age, the amount of data observed has increased tremendously in volume, velocity and variety. Concealed inside this new natural resource lies a wealth of information waiting to be mined. Arrays with two or more dimensions, known as matrices and tensors, respectively, serve as the computer representation of many types of data. Knowledge can be extracted, or inferred, from such arrays by capturing their underlying patterns and structure with so-called matrix and tensor factorizations that attempt to explain the observations with a small number of variables. The ever increasing volume of data demands new algorithms that can deal with the curse of dimensionality, while the growing variety of data calls for new approaches to integratively analysing several sources of data at once. The goal of this thesis is to develop data fusion as a new paradigm for the joint analysis of one or more data sets by coupled low-rank factorization of dense, sparse or incomplete matrices and tensors, leading to insights which are deeper and more accurate than those resulting from a single source of data. There are many choices to be made in a data fusion model. For each data set, one of several types of tensor decompositions should be chosen with which that data set will be factorized. Tensor decompositions are composed of building blocks called factors, each of which may be imposed to exhibit a certain structure such as nonnegativity or orthogonality. Furthermore, the coupling between the decompositions should be defined by indicating which factors are shared between data sets. Evidently, it is a hopeless task to attempt to design algorithms individually for each of the myriad combinations of tensor decompositions, factor structures and coupling between factorizations that define a data fusion model. In this thesis we present a framework which allows for all of these choices to be made independently and dynamically, we develop accompanying algorithms that fully exploit the structure of the problem on every level and we deliver an efficient software implementation with which data fusion can be brought into practice on big data. Users can choose from a library of tensor decompositions and factor structures or add their own with little effort, enabling the computation of complex data fusion models and classical matrix and tensor factorizations alike.

Korte inhoud

In ons informatietijdperk is de hoeveelheid waargenomen gegevens enorm toegenomen in volume, snelheid en variatie. In deze nieuwe natuurlijke grondstof ligt een schat aan informatie te wachten om te worden ontgonnen. Tabellen met twee of meer dimensies, respectievelijk bekend als matrices en tensoren, dienen als de computer representatie van vele soorten gegevens. Kennis kan gewonnen worden uit dergelijke tabellen door het vastleggen van hun onderliggende patronen en structuur met zogenaamde matrix en tensor factorisaties die de waarnemingen met een zo klein mogelijk aantal variabelen proberen te verklaren. De steeds toenemende hoeveelheid data vereist het ontwikkelen van nieuwe algoritmen die kunnen omgaan met de zogenaamde vloek van de dimensionaliteit, terwijl de groeiende verscheidenheid aan gegevens vraagt om nieuwe benaderingen voor het geïntegreerd analyseren van verschillende gegevensbronnen tegelijk. Het doel van dit proefschrift is om data fusie te ontwikkelen als een nieuw paradigma voor de gezamenlijke analyse van een of meer datasets door gekoppelde lage-rang factorisatie van volle, ijle of onvolledige matrices en tensoren, wat tot inzichten zal leiden die dieper en nauwkeuriger zijn dan die voortvloeiend uit een enkele bron van gegevens. Er zijn veel keuzes te maken in een data fusie model. Voor elke data set moet er een van de verschillende soorten tensor decomposities gekozen worden waarmee die data set gefactoriseerd zal worden. Tensor decomposities bestaan uit bouwstenen genaamd factoren waaraan men, afhankelijk van de toepassing, bepaalde structuren aan kan opleggen zoals niet-negativiteit of orthogonaliteit. Bovendien moet de koppeling tussen de decomposities gedefinieerd door aan te geven welke factoren gedeeld zijn tussen de data sets. Het is kennelijk onbegonnen werk om algoritmen te ontwerpen voor elk van de talloze combinaties van tensor decomposities, factor structuren en koppeling tussen factorisaties die een data fusie model definiëren. In dit proefschrift presenteren we een raamwerk waarin al deze keuzen onafhankelijk en dynamisch gemaakt kunnen worden, ontwikkelen we bijbehorende algoritmen die de structuur van het probleem op elk niveau uitbuiten en leveren we een efficiënte software implementatie waarmee data fusie in de praktijk kan worden gebracht op big data. Gebruikers kunnen kiezen uit een bibliotheek van tensor decomposities en factor structuren of er zelf toevoegen met weinig inspanning, wat de berekening van zowel complexe data fusie modellen als klassieke matrix en tensor factorisaties mogelijk maakt.

List of Abbreviations

ALS
Alternating least squares

AUC
Area under curve

BER
Bit error rate

BFGS
Broyden–Fletcher–Goldfarb–Shanno

BSS
Blind source separation

BTD
Block term decomposition

CCA
Canonical correlation analysis

CDMA
Code division multiple access

CM
Constant modulus

CMD
Collective matrix factorization

CMTF
Coupled matrix and tensor factorization

CPD
Canonical polyadic decomposition

DP
Double precision

DS-CDMA
Direct sequence CDMA

DSL
Domain specific language

EEG
Electroencephalography

EVD
Eigenvalue decomposition

GEP
Generalized eigenvalue problem

GEVD
Generalized EVD

GN
Gauss–Newton

GPS
Global positioning system

ICA
Independent component analysis

ISI
Intersymbol interference

L-BFGS
Limited-memory BFGS

LM
Levenberg–Marquardt

LS
Line search

LS-STM
Least squares support tensor machine

MIMO
Multiple-input multiple-output

MLSVD
Multilinear SVD

NCG
Nonlinear conjugate gradient

NLS
Nonlinear least squares

PCA
Principal component analysis

PEP
Polynomial eigenvalue problem

POI
Point of interest

PS
Plane search

QN
Quasi-Newton

QPSK
Quadrature phase-shift keying

RMSE
Root mean square error

ROC
Receiver operating characteristic

SDF
Structured data fusion

SIMO
Single-input multiple-output

SK
Sinkhorn–Knopp

SLS
Scaled line search

SNR
Signal-to-noise ratio

SPS
Scaled plane search

SVD
Singular value decomposition

VP
Variable precision

List of Symbols

$a, b, \ldots$	Scalars
$\boldsymbol{a}, \boldsymbol{b}, \ldots$	Column vectors
$A, B, \ldots$	Matrices
$\mathcal{A}, \mathcal{B}, \ldots$	Higher-order tensors
$(\boldsymbol{a}, \boldsymbol{b}, \ldots)$	Vertical concatenation of two or more column vectors
$\overset{c}{\cdot}$	Representation in complex vector space $\mathcal{C}$ as $(\cdot, \overline{\cdot})$
$\overset{\mathcal{R}}{\cdot}$	Representation in real vector space $\mathcal{R}$ as $(\mathrm{Re}\{\cdot\}, \mathrm{Im}\{\cdot\})$
$\mathrm{diag}(A, B, \ldots)$	Block diagonal matrix with blocks $A, B, \ldots$
$:=$	Defined as
$\equiv$	Identically equal to
f	Real scalar-valued objective function
F	Vector-valued residual functions
$\mathcal{F}$	Tensor-valued residual functions
$\mathbb{C}$	Set of complex numbers
$\mathbb{C}^{I_1 \times \cdots \times I_N}$	Set of complex-valued tensors of size $I_1 \times \cdots \times I_N$
$\mathbb{R}$	Set of real numbers
$\mathbb{R}^{I_1 \times \cdots \times I_N}$	Set of real-valued tensors of size $I_1 \times \cdots \times I_N$
$\mathbb{I}_n$	Identity matrix of order n
$\mathrm{Im}\{\cdot\}$	Imaginary part

i	Imaginary unit
$\overline{\cdot}$	Complex conjugate
$\cdot^{\mathrm{H}}$	Hermitian conjugate
$\cdot^{-1}$	Matrix inverse
$\cdot^{-\dagger}$	Moore–Penrose pseudoinverse
$\cdot^{(n)}$	The nth element of a sequence
$\cdot^{\mathrm{T}}$	Matrix transpose
$T_{(n)}$	Mode-n matricization of the tensor $\mathcal{T}$
$\lvert \cdot \rvert$	Absolute value or modulus
$\lVert \cdot \rVert$	Euclidian norm
$\lVert \cdot \rVert_{\mathrm{F}}$	Frobenius norm
$\lVert \cdot \rVert_{\mathcal{W}}$	Weighted Frobenius norm $\lVert \mathcal{W} * \cdot \rVert_{\mathrm{F}}$
$1_{m \times n}$	All-one matrix of size $m \times n$
$*$	Hadamard product
$\langle \cdot, \cdot \rangle$	Inner product
$\odot$	Khatri-Rao product
$\otimes$	Kronecker product
$\bullet_n$	Mode-n tensor-matrix product
$\circ$	Outer product
$\mathrm{Re}\{ \cdot \}$	Real part
$\mathrm{vec}(\cdot)$	Column-wise vectorization
$\boldsymbol{z}$	Column vector of complex variables
$0_{m \times n}$	All-zero matrix of size $m \times n$

Contents

Voorwoord i

Abstract v

Korte inhoud vii

Contents xv

List of Figures xxi

List of Tables xxv

1 Introduction 1

1.1 Data everywhere 1

1.2 Inference by factorization 4

1.2.1 Flatland 4

1.2.2 The jump to hyperspace 6

1.3 Research aims and overview 9

2 Unconstrained optimization of real functions in complex variables 17

2.1 Introduction 18

2.2 Notation and preliminaries 19

2.3 Wirtinger calculus and the complex Taylor series expansions . . 19
2.3.1 First-order complex Taylor series expansion 21
2.3.2 Second-order complex Taylor series expansion 22
2.4 Nonlinear optimization 23
2.4.1 The generalized BFGS and L-BFGS methods 23
2.4.2 The generalized nonlinear conjugate gradient method . 29
2.5 Nonlinear least squares 30
2.5.1 The generalized Gauss–Newton and Levenberg–Marquardt methods 30
2.6 Case studies 32
2.6.1 The canonical polyadic decomposition 32
2.6.2 Simulation of nonlinear circuits in the frequency domain 37
2.7 Conclusion 40

3 Optimization-based algorithms for tensor decompositions 43
3.1 Introduction 44
3.2 Notation and preliminaries 45
3.3 Tensor decompositions 47
3.3.1 The canonical polyadic decomposition 47
3.3.2 The rank-$(L_r, L_r, 1)$ block term decomposition 48
3.3.3 The (rank-L_r $\circ$ rank-1) block term decomposition . . . 49
3.4 Algorithms for the general decomposition 50
3.4.1 Unconstrained nonlinear optimization and alternating least squares 50
3.4.2 Nonlinear least squares methods 54
3.4.3 Computational complexity 60
3.5 Numerical experiments 65
3.5.1 The rank-$(L_r, L_r, 1)$ block term decomposition 66

3.5.2 The canonical polyadic decomposition 71
3.6 Conclusion 72

4 Exact line and plane search for tensor optimization 75
4.1 Introduction 76
4.2 Notation and preliminaries 77
4.3 Tensor optimization 78
4.4 Exact line and plane search 82
4.4.1 Scaled line search 86
4.4.2 Scaled plane search 87
4.4.3 Higher-order two-dimensional subspace minimization . . 87
4.5 Solving systems of bivariate and polyanalytic polynomials . . . 88
4.6 Numerical experiments 92
4.7 Conclusion 95

5 Numerical solution of bivariate and polyanalytic polynomial systems 99
5.1 Introduction 100
5.2 Notation and preliminaries 101
5.3 Solving systems of bivariate and polyanalytic polynomials . . . 102
5.3.1 The polynomial eigenvalue problem 102
5.3.2 Balancing the system and its associated pencil 110
5.3.3 Filtering spurious solutions with Newton–Raphson . . . 112
5.4 Numerical experiments 115
5.5 Conclusion 128

6 Structured data fusion 129
6.1 Introduction 130
6.2 Structured data fusion 133

6.2.1 Tensor decompositions 135
6.3 Algorithms for SDF 137
6.3.1 Quasi-Newton 139
6.3.2 Nonlinear least squares 140
6.3.3 Modularizing decompositions and structure 141
6.3.4 Generalizations 148
6.4 Numerical experiments 149
6.4.1 Eigenvalue decomposition as a structured tensor decomposition 149
6.4.2 Recommending movies: the Netflix prize 151
6.4.3 Designing an alloy 152
6.4.4 Predicting user participation in activities 155
6.5 Conclusion 158

7 Conclusion **159**

7.1 Contributions 159
7.2 Future research 163

A Tensorlab user guide **165**

1 Getting started 165
2 Data sets: dense, incomplete and sparse tensors 170
2.1 Representation 170
2.2 Tensor operations 171
3 Canonical polyadic decomposition 175
3.1 Problem and tensor generation 176
3.2 Computing the CPD 177
3.3 Choosing the number of rank-one terms R 178
4 Low multilinear rank approximation 180

4.1 Problem and tensor generation 180
4.2 Computing a LMLRA 181
4.3 Choosing the size of the core tensor 183
5 Block term decomposition 184
5.1 Problem and tensor generation 184
5.2 Computing a BTD 185
6 Structured data fusion 186
6.1 Domain specific language for SDF 187
6.2 Implementing a new factor structure 197
7 Complex optimization 199
7.1 Complex derivatives 200
7.2 Nonlinear least squares 207
7.3 Unconstrained nonlinear optimization 212
8 Global minimization of bivariate functions 214
8.1 Stationary points of polynomials and rational functions 216
8.2 Solving bivariate polynomial systems 219

Bibliography **221**

Curriculum vitae **245**

Publications **247**

List of Figures

2.1 Canonical polyadic decomposition of a third-order tensor. . . . 33

3.1 Rank-$(L_r, L_r, 1)$ block term decomposition of a third-order tensor. 48

3.2 (Rank-$L_r \circ$ rank-1) block term decomposition of a sixth-order tensor. 49

3.3 Sparsity pattern of the error of the Gauss–Newton Hessian approximation. 56

3.4 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = L_2 = L_3 = 3$. 67

3.5 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 4$ and $L_2 = 5$. 68

3.6 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$. 68

3.7 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = L_2 = L_3 = 3$, of noisy tensors. . . . 70

3.8 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 4$ and $L_2 = 5$, of noisy tensors. . . 70

3.9 Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$, of noisy tensors. 71

3.10 Performance profiles for complex rank-4 canonical polyadic decompositions of noisy tensors. 72

4.1 Low multilinear rank approximation of a third-order tensor. . . 80

4.2 Rank-$(L_r, L_r, 1)$ block term decomposition of a third-order tensor. 80

4.3 Coupled data sets of different orders. 81

4.4 The LS-$\mathbb{R}$ objective function of the constant modulus problem along the steepest descent direction. 85

4.5 Mean BER as a function of the SNR of the observed tensor and the condition number of the antenna matrix. 93

4.6 Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor, with $\kappa(A) = 1$. . . 94

4.7 Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor, with $\kappa(A) = 10$. . . 95

4.8 Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor, with $\kappa(A) = 1$ and using quadratically extrapolated search directions. 96

4.9 Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor, with $\kappa(A) = 10$ and using quadratically extrapolated search directions. 96

5.1 Two examples of the low-degree bivariate polynomial systems and their solutions. 119

5.2 Two examples of the moderate-degree bivariate polynomial systems and their solutions. 120

5.3 Two more examples of the moderate-degree bivariate polynomial systems and their solutions. 121

5.4 Performance profiles for low-degree bivariate polynomial systems, where the problem set consists of all polynomial systems. . . . 123

5.5 Performance profiles for moderate-degree bivariate polynomial systems, where the problem set consists of all polynomial systems. 124

5.6 Performance profiles for low-degree bivariate polynomial systems, where the problem set consists of all distinct roots of the systems. 126

5.7 Performance profiles for moderate-degree bivariate polynomial systems, where the problem set consists of all distinct roots of the systems. 127

6.1 Schematic of structured data fusion. 133

6.2 Canonical polyadic decomposition of a third-order tensor. . . . 136
6.3 Low multilinear rank approximation of a third-order tensor. . . 137
6.4 Block term decomposition of a third-order tensor. 137
6.5 Relative accuracy of the colleague matrix's eigenvalues. 150
6.6 The columns of the third factor matrix in a structured CPD of the InsPyro data set. 153
6.7 The tensile strength of an alloy comprising four metals. 154
6.8 ROC curves for the prediction of the 80% missing user-location-activity links. 157
6.9 ROC curve for the prediction of the user-location-activity links of 50 missing users. 157

A.1 Three functions for visualizing third-order tensors in Tensorlab. 174
A.2 Canonical polyadic decomposition of a third-order tensor. . . . 175
A.3 The `rankest` tool for choosing the number of rank-one terms R in a CPD. 179
A.4 Low multilinear rank approximation of a third-order tensor. . . 180
A.5 The `mlrankest` tool for choosing the core tensor size `size_core` in a LMLRA. 183
A.6 Block term decomposition of a third-order tensor. 184
A.8 The six-hump camel back function and its stationary points. . . 217
A.9 A system of bivariate polynomials of total degree 20. 220

List of Tables

3.1 The (rank-$L_r \circ$ rank-1) BTD objective function's scaled conjugate cogradient in the special case of a third-order CPD and a rank-$(L_r, L_r, 1)$ BTD. 52

3.2 Computational complexity of several BTD algorithms in flop/iteration in case of a real CPD of an Nth-order $I_1 \times \cdots \times I_N$ tensor in R rank-one terms. 61

4.1 The type of the objective function each of the search problems correspond to for two choices of the solution field. 85

5.1 Properties of the initial low-degree problem set, containing a total of 50 systems. 117

5.2 Properties of the initial moderate-degree problem set, containing a total of 16 systems. 117

6.1 RMSE on the probe set by the mean μ, Netflix's Cinematch algorithm and the SDF model. 152

Chapter 1

Introduction

> *"The key idea underlying the impending Singularity is that the pace of change of our human-created technology is accelerating and its powers are expanding at exponential pace."*
>
> — Ray Kurzweil

1.1 Data everywhere

Data has become one of the world's natural resources. Its supply is unimaginably vast, and growing ever faster with no end in sight. Experiments, observations and simulations in many areas of science and business are generating massive amounts of the quantity. For example, NASA's Earth observation satellites send four terabytes[1] of data home every day [239]. The Large Hadron Collider—a high-energy particle collider designed to test the Standard Model of particle physics—produces tens of petabytes[2] per year. A Boeing 787 can generate upwards of half a terabyte of data per flight [75]. Multiplied by 100,000 flights per day worldwide [102], flight data easily enters the petabyte range. Self-driving cars analyse almost a gigabyte of sensor data per second [88]. Smart power grids monitor changes in behaviour of suppliers and consumers with phasor measurement units that collect voltage and current measurements at many points in the grid, and smart meters that gather consumption and production

[1] One terabyte equals one million million bytes. The storage capacity of a modern laptop.

[2] One petabyte equals one thousand terabytes. The human brain's storage capacity has been estimated at 2.5 petabytes [181].

of energy in intervals of an hour or less. The deployment of a full-scale phasor system in a continent's power grid alone can generate over four terabytes of data per day [26]. Personalized medicine is on the verge of a breakthrough thanks to recent advances in DNA sequencing. In 2014, a whole human genome can be sequenced in less than a week for $1,000 [97], which is 10,000 times cheaper than a decade ago and 100 times faster than the reduction in cost predicted by Moore's law [59]. The low cost of sequencing will enable analysing genomes on a much broader scale, and will lead to a better understanding of the impact of genetics in disease and new treatments tailored to the individual. The widespread adoption of mobile internet together with information and communications technology services such as cloud computing and social media are also significantly contributing to the data explosion. In 2002, Google's index counted 3 billion web pages [84]. By 2012, they indexed 30 trillion pages and were receiving 100 billion search queries per month [223]. A year later, on their fifteenth birthday, their index counted over 60 trillion web pages [85]. In other words, the observable Internet has more than doubled in size every year for more than a decade since 2002, without even taking into account the tremendous growth of image, audio and video data. The Internet plays a central role in our society. Much human activity now takes place on the Internet and the data it generates is of substantial commercial and scientific value. For example, many companies seek to exploit this data to provide personalized services that adapt to individual preferences and behaviours. In fact, human activity is the predominant source of input for the Internet of today. However, the rapidly growing amount of devices, sensor networks and actuators connected to the Internet—often referred to as the Internet of Things—will drastically change this view. In 2008, the number of devices connected to the Internet exceeded the number of people on Earth. By 2020, this number is expected to grow to between 50 and 75 billion [44,71]. IPv4, the internet protocol with which devices are assigned internet addresses, can only accommodate about 4.3 billion devices concurrently. To deal with the impending exhaustion of this address space, the next version of the protocol, IPv6, supports over one octillion devices—more than the number of atoms in the human body—for each person on Earth. The Internet of Things will far surpass our own capacity to capture and feed data to the Internet, resulting in a world more connected and networked than ever before.

By itself, data is not worth much. Its true value lies in the information that it contains. Data, like other natural resources, can be *mined* for knowledge and understanding of its underlying processes. Data mining and machine learning are two related disciplines which aim to discover patterns within data and make predictions based on data, respectively. Fundamental to both disciplines is the format in which the data is represented. A natural and often used representation for many types of data sets are matrices. A data matrix is a rectangular array

of numbers in which each row corresponds to a sample or observation, and each column corresponds to a number of attributes or properties associated with that sample. In data mining and machine learning, these attributes are referred to as features. For example, each row in a data matrix could correspond to an observed forest fire, while the columns describe the forest fire's properties such as its location, the size of the burned area, the temperature and wind speed. An example of a machine learning task is to predict the size of the burned area of a new forest fire, given the forest fire's other properties and a data matrix of previously observed forest fires.

Graphs are a second type of data set that have a natural representation in matrix form. A graph is a collection of points called nodes, some of which are connected with lines called edges. A matrix that associates its rows and columns with the nodes of a graph and whose elements equal the number of edges between any pair of these nodes is called an adjacency matrix. For example, in the web graph each node is a web page and each edge represents a hyperlink from one page to another. Google uses the adjacency matrix of the web graph to rank websites in their search engine results. In a social graph, nodes are people and edges represent relationships between them. Facebook is currently the world's largest social network and has a social graph of more than a billion active users [72]. Among other tasks, the adjacency matrix associated with a social graph can be used to predict links between nodes. Adjacency matrices are not the only way to represent graphs. The topology of electronic circuits are also conveniently described by graphs: each node refers to a point where two or more circuit elements meet and every edge corresponds to a branch with zero or more circuit elements. To simulate the circuit, Kirchhoff's current and voltage laws are applied to the nodes and edges of the circuit graph, respectively. The resulting equations are then solved numerically by casting the problem as a matrix equation that depends on a different representation of the circuit graph called an incidence matrix.

Relational data consists of entities and relations between these entities. Given two types of entities, relational data is a third type of data set that is commonly described by a matrix. Consider for instance the relation *student-performs-task*, where the two entity types are *student* and *task* and the relation type is *performs*. The rows and columns of a matrix can then be associated with the two respective entity types, while the matrix's entries quantify the relation type. In this case, the entries of the relational matrix describe how well a student performs a certain task. It is often the case that not all entries of the relational matrix are known, in which case a goal could be to predict the unknown entries based on the known entries. Another example is the relation *user-rates-movie*. The relation type *rating* could be an integer number of stars between one and five that a user assigns to a movie. The DVD rental and on-demand streaming

company Netflix has a customer base of over 40 million users and a catalogue of about 100,000 movies and TV shows [220]. While Netflix's relational matrix is massive in size, only relatively few of its entries are known since most users only rate a limited number of titles. To be able to suggest new movies to their customers, Netflix is interested in predicting the unknown ratings in their relational matrix. In 2006, they offered a $1,000,000 prize to the first group to improve on their in-house algorithm for predicting movie ratings by 10%, which was claimed only three years later [154].

A fourth way in which matrices arise is by construction. Partial differential equations (PDEs) are used to model physical phenomena such as heat conduction, fluid flow and electrodynamics. Numerical simulation of PDEs discretizes these equations on a grid, resulting in an important source of structured sparse matrices. In optimization, a fundamental tool is the construction of a second-order Taylor series approximation of an objective function. Instead of minimizing the objective function directly, the computationally more tractable Taylor series is minimized to obtain a step which improves the current solution. Computing the Taylor series' minimizer involves inverting or approximating the inverse of a matrix of second-order derivatives called the Hessian matrix. In systems and control, a prevalent task is to determine a state space description of a linear system based on measurements of the system's output when excited by a certain input. Consider the Hankel matrix obtained by placing the entries of an impulse response, which is the system's output when presented with a brief input signal, on the anti-diagonals of a square matrix. A minimal realization, where minimal refers to the complexity or order of the realization, of the system's state space description can then be obtained by computing a simple low-rank factorization of the resulting Hankel matrix.

1.2 Inference by factorization

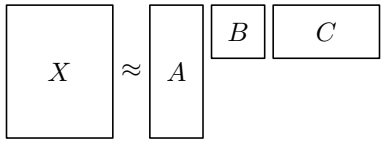
1.2.1 Flatland

Inference, which is the process of extracting knowledge from data, is the common theme to all applications dealing with matrix data. Central to inference is the concept of matrix factorization. Given a matrix data set X, its factorization is the matrix product

$$X = A \cdot B \cdot C, \tag{1.1}$$

where the shape and structure of the factors A, B and C depends on the desired type of factorization. For example, well-known matrix factorizations such as the eigenvalue decomposition (EVD) can be obtained by requiring that B is diagonal

and C is the matrix inverse of A. The so-called QR factorization requires A to have orthonormal columns, B to be the identity matrix and C to be upper triangular. In practice, the factorization need only approximate the data matrix. A generic matrix X has full rank, meaning that the factors A, B and C need to be of full size to approximate the data matrix well. Generic matrices are noisy matrices which don't exhibit any particular patterns. However, data sets rarely consist entirely of noise and often do contain some structure. Under those circumstances, the matrix X is approximated well by a low-rank factorization wherein the factors A, B and C are small compared to X. If the data set X is of size $M \times N$ and R is the rank parameter, the factors A, B and C are of size $M \times R$, $R \times R$ and $R \times N$, respectively. Visually, a low-rank matrix factorization is described by the equation



(1.2)

A fundamental principle in inference is that of Occam's razor[3], which embodies the maxim that the simpler explanation is often the better one. The razor states that a model should be simplified, or shaved, until any further simplification would cut into the model's explanatory power. The smaller the number of variables and the more structure they possess, the better the model satisfies Occam's razor.

For matrix factorizations, one way to reduce model complexity is by minimizing the size of the factors. A second way to decrease the number of degrees of freedom is by imposing structure on the factors. Examples of factor structure include requiring that one factor is the inverse of another factor, constraining the elements of a factor to be nonnegative and imposing an orthogonal, Hankel, Toeplitz or Vandermonde structure on a factor. Returning back to the examples of different types of matrix data, structured low-rank matrix factorizations can be used to cluster forest fires with nonnegative matrix factorization, to rank web pages by computing the dominant eigenvector of the web graph's adjacency matrix, to predict movie ratings, to approximate the inverse of Hessian matrices and to compute a minimal realization of linear dynamical systems.

[3]Attributed to the 14th century English philosopher and monk William of Ockham.

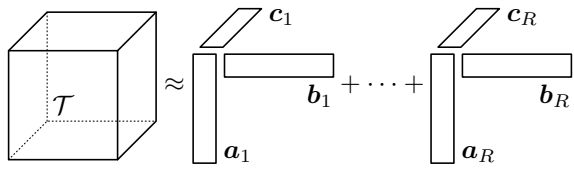
1.2.2 The jump to hyperspace

Matrix theory did not receive much attention until the 1950s, when the growing interest in PDEs prompted the development of the first sparse matrix solvers. Before the rise of matrices, determinants were the main object of study. The term *matrix*, which is Latin for *womb*, itself did not even exist before 1850, when it was coined by James Joseph Sylvester in reference to the array giving rise to a number of determinants called minors [225]. In the 1970s, a set of low-level matrix and vector operations called kernels was identified as underlying many algorithms in linear algebra. Highly optimized implementations of these basic linear algebra subprograms (BLAS) are still widely used today as building blocks for solving higher-level matrix problems. In the 1980s it was recognized that several BLAS routines would benefit from exploiting the computer's memory hierarchy by concentrating computation in matrix blocks instead of element by element. This led to the development of block matrix algorithms which make more efficient use of the computer's different layers of cache memory. Arrays with more than two dimensions, called tensors, were already being studied in the 19th century [183], but did not achieve broad acceptance until the introduction of Einstein's theory of general relativity in 1915. More formally, the different dimensions of a tensor are called its modes and the number of these modes is referred to as the order of the tensor. With the steady increase in computing power, tensors have enjoyed a renewed interest since the 2000s as evidenced by the many books and review articles [35, 91, 112, 117, 121, 200].

Many types of matrix data have a natural extension to a tensor format. For example, a data matrix of forest fires may contain temperature or wind speed information for a whole grid of sensors distributed over the forest. Since a single forest fire can then be described by a matrix of features, a set of forest fires can be stored in a third-order tensor. Graphs such as the web graph and social graphs can evolve over time, and the adjacency matrices for each time step can be stacked into a tensor. Relational data with more than two entity types or more than one relation type are also naturally described by a tensor. For example, fluorescence spectroscopy is a technique to analyse the chemical contents of a solution by exciting it with a beam of light and measuring the output [28, 200]. Among other applications, fluorescence spectroscopy of human blood plasma can be used to detect cancer. In that case, a relational tensor could be based on the entities, or modes, *excitation wavelength*, *emission wavelength* and *person*, and the relation type, or element type, *emitted energy*. Finally, tensors can also be created by construction. For example, in optimization a higher-order Taylor series involves tensors containing third- or higher-order derivatives. Independent component analysis (ICA) is a technique that blindly separates statistically independent sources from a set of measurements by computing and factorizing a so-called fourth-order cumulant tensor [38].

One approach to factorizing tensor data is to first *matricize* it and then apply a matrix factorization. For instance, a third-order tensor of size $50 \times 50 \times 50$ can be viewed as a stack of fifty 50×50 matrices. These matrices could be placed next to each other to form a 50×2500 matrix. However, a low-rank factorization of the resulting matricized tensor would be a far cry from satisfying Occam's razor since the right factor C then holds a large number of variables with respect to the number of elements in the matrix. Although a factorization with many variables can approximate the matrix well, it is unlikely to have exclusively captured the data's underlying process. Instead, the model will fit both the underlying process and any noise or other undesired artefacts present in the data. As a result, any predictions made by the model are corrupted by the noise it has seen in the data. To reduce the danger of overfitting the noise in the data set, a factorization model should have only just enough degrees of freedom to fit the data's essential characteristics. Compared to the factorization of a matricized tensor, a direct factorization of the tensor data offers much more granularity in the number of free variables.

To determine what a low-rank tensor factorization looks like, we first need to answer the question of what tensor rank means. In the matrix case, the rank can be interpreted in several equivalent ways: as the minimal number of rank-one matrices whose sum equals the matrix, and as the dimension of the column- or row-space of the matrix. For tensors, this characterization of rank disjoins into two distinct concepts that engender two archetypal tensor decompositions. On the one hand, the rank of a tensor can be described as the minimal number of rank-one tensors whose sum equals the tensor. The factorization corresponding to this description of rank is called the canonical polyadic decomposition (CPD) [33, 95, 99, 100]. Given a third-order tensor $\mathcal{T}$, its CPD is portrayed by

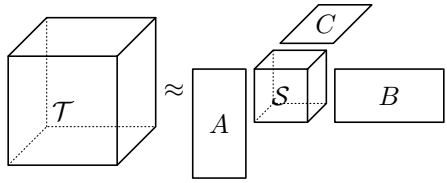


(1.3)

where $\boldsymbol{a}_r$, $\boldsymbol{b}_r$ and $\boldsymbol{c}_r$ are nonzero vectors for $r = 1, \ldots, R$. Similar to the matrix factorization (1.1), the CPD can also be described in terms of its factor matrices $A := \begin{bmatrix} \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_R \end{bmatrix}$, $B := \begin{bmatrix} \boldsymbol{b}_1 & \cdots & \boldsymbol{b}_R \end{bmatrix}$ and $C := \begin{bmatrix} \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_R \end{bmatrix}$.

On the other hand, the rank of a tensor could also be described in terms of its column vectors (mode-one vectors), row vectors (mode-two vectors) and mode-three vectors. In contrast to the matrix case, the spaces spanned by these three sets of vectors are no longer necessarily of the same size. A tensor's multilinear rank is then defined as a tuple of all mode-n ranks, where the mode-n rank is

the dimension of the space spanned by the tensor's mode-n vectors. The tensor factorization that reveals a tensor's multilinear rank is called a low multilinear rank approximation (LMLRA) [54, 231, 232]. The LMLRA is defined in terms of factor matrices that represent the tensor's mode-n vector spaces, and a core tensor that represents the different interactions between those spaces and whose size equals the tensor's multilinear rank. The LMLRA of a third-order tensor $\mathcal{T}$ is depicted by the equation



(1.4)

Next to the CPD and LMLRA, there exist many other low-rank tensor factorizations. De Lathauwer introduced block term decomposition (BTD) [46, 47, 55] as a framework that unifies the CPD and LMLRA. In a BTD, the tensor is approximated by a sum of low multilinear rank terms as in (1.4). The tensor train decomposition [161] and hierarchical Tucker decomposition [86, 92] represent two other examples of recently proposed low-rank tensor factorizations.

Perhaps surprisingly, tensor factorizations are unique under mild conditions. Uniqueness is a desirable property in inference as it is a key facilitator of attaching meaning to factors. A factorization is called unique if its factor matrices are unique up to some trivial indeterminacies such as the permutation and scaling of their columns. In contrast to tensor factorizations, matrix factorizations require constraints on the factors such as orthogonality or nonnegativity together with sparsity to obtain uniqueness. A well-known result on uniqueness of the CPD is due to Kruskal [118, 119, 194] and more relaxed conditions were developed in [67, 68]. De Lathauwer provides sufficient conditions for uniqueness of various types of BTD in [47, 48]. Uniqueness lies at the heart of many blind source separation algorithms. For example, ICA is able to separate a set of measurements into statistically independent sources precisely because the CPD of the fourth-order cumulant tensor derived from the measurements is unique. While the measurement data itself is given in matrix form, the *tensorization* of that data to a cumulant tensor enables ICA to take advantage of the uniqueness properties of the CPD. Tensorization also forms the basis of other applications of source separation such as the blind separation of exponential polynomials by using time series as generators for Hankel matrices [48] and detecting the onset zone in epileptic seizures by applying a wavelet transform to electroencephalography (EEG) measurements to create one wavelet matrix per EEG channel [58]. Another benefit of tensorizing data sets is the more fine-grained control over the number of variables in the factorization, allowing for a much closer shave with Occam's razor.

1.3 Research aims and overview

The explosion of available data has led to a new term to describe the phenomenon: *big data*. A practical definition of big data is data that is too big to comfortably process on a single machine. In the case of matrix data, this implies that at least one of the matrix's modes should be large in comparison to the computer's storage capacity. For tensors, the term big can take on an entirely different perspective. The number of entries in a tensor grows exponentially with its order. For example, storing a 20th-order tensor in which each mode has length 5 requires over three quarters of a petabyte of memory. A combination of factorizations and algorithms that succeed in breaking this curse of dimensionality will induce a transition from matrix- to tensor-based computation. One promising approach of dealing with the curse of dimensionality may be to intentionally observe only a small fraction of a tensor's entries and compute a factorization of the resulting incomplete tensor. Algorithms for factorizing incomplete tensors are not only useful in situations where storage capacity is limited, but also when measuring, aggregating or computing all of a tensor's entries is infeasible. A second definition describes big data as high volume, high velocity and high variety. In short, the 3Vs model [24] refers to the rapidly increasing amount of data, the speed at which it arrives and the wide range of data types and sources, respectively. Any technology hoping to exploit big data to its full extent should address each of these issues. High volume requires algorithms and factorizations that can deal with the curse of dimensionality. High velocity demands online algorithms that can process data as it arrives. High variety calls for the analysis of different types of matrix and tensor data and the integration of data arising from multiple sources in the form of collections of data matrices and tensors.

Tensor factorizations are one approach to integratively analysing a set of matrices that are homogeneous in size. In fact, a tensor factorization can be interpreted as a joint matrix factorization in which the different factorizations are coupled with each other by sharing certain factors. Taking this idea one step further, the information in a set of heterogeneous data sets can be *fused* by factorizing each data set with a tensor decomposition and coupling these factorizations through their factors. An illustrative example of data fusion by coupled tensor factorizations is in the prediction of movie ratings given the relational data sets *user-rates-movie* and *movie-is directed by-director*, both of which are stored as incomplete matrices with some or many missing entries. The former contains ratings between one and five stars given by users (rows) to movies (columns), while the entries of the latter are equal to one when a movie (row) is directed by a certain director (column), and zero otherwise. By computing a low-rank factorization of the *user-rates-movie* matrix as in (1.2), each user and each movie is described by a small number of parameters, or features, that are

stored in the factors A and C, respectively. For instance, each user could hypothetically be represented by three features that indicate that user's interest in the action, romance and comedy genres. Correspondingly, each movie could be represented by three features that indicate the amount of action, romance and comedy content in that movie. The low-rank factorization is then able to predict each rating as the inner product of these two sets of features. In particular, a rating is predicted as the user's interest in action times the movie's amount of action content, plus the user's interest in romance times the movie's amount of romance content, and so on. This works even if a user has not seen a particular movie yet because the user's features are learned from the ratings he or she has given to other movies, while the movie's features are learned from the other users' ratings. Supposing George Lucas, famous for creating the Star Wars saga, is about to release a new movie, then a low-rank factorization of the *user-rates-movie* matrix can not extract the new movie's features because nobody has seen it yet. To solve this problem, we can jointly factorize the *user-rates-movie* matrix with the *movie-is directed by-director* matrix and require both factorizations to share the movie factor C. That way, the features associated with the new movie are estimated from those of the Star Wars saga since the *movie-is directed by-director* matrix implies that the new movie is similar to the saga by virtue of sharing the same director. In effect, fusing the two factorizations has allowed for the transferral of information from one data set to another. Similarly, the features of a new user who has not yet rated any movies can be learned by requiring the low-rank factorization to share the user factor A with a *user-is friends with-user* matrix containing friendships or similarities between users. Data fusion not only enables the transferral of information between data sets, but can also substantially improve the quality of the features and, consequently, predictions made by the low-rank model.

A tough challenge in this definition of data fusion is how to deal with the burden of choice. The complexity of a data fusion model far exceeds that of the comparatively simple low-rank matrix factorization. In the latter, only the rank and the structure of the factors need to be chosen, with the central factor typically being a diagonal matrix. In a data fusion model, a type of tensor factorization should be chosen for each data set. Each factorization may have several parameters associated with it. For example, a LMLRA has as many rank parameters as the order of the tensor that it approximates. Furthermore, the coupling between factorizations in terms of the factors should be defined, along with any structure imposed on the factors. Obviously, it is a hopeless task to attempt to design algorithms for each of the myriad possibilities individually. Instead, we desire a framework which allows for all these choices to be made dynamically, for accompanying algorithms that can fully exploit the structure of the problem on every level, and for an efficient software implementation with which data fusion can be brought into practice on big data.

The goal of this thesis is to develop data fusion as a new paradigm for the joint analysis of one or more data sets by coupled low-rank factorization of dense, sparse or incomplete tensors, leading to insights which are deeper and more accurate than those resulting from a single matrix factorization. The framework should allow a free choice of which decomposition to use for each data set and what kind of structure to impose on the associated factors, enabling users to rapidly iterate through the vast search space of possible data fusion models. Moreover, tensor decompositions and factor structures should be a modular part of the framework so that users will be able to pick from libraries of predefined components, or even add their own if desired. We aim to create a language with which data fusion problems can be modelled in an intuitive yet formalized way, and algorithms that are flexible enough to solve such dynamically generated problems while at the same time being advanced enough to be able to exploit the structure of the tensor decomposition and of the factors. Finally, we seek the development of a software library that implements an efficient set of kernels for operating on tensors, the data fusion modelling language and the accompanying algorithms built on these kernels.

The remainder of the thesis is divided into five chapters that steadily build towards these goals and a final chapter in which we discuss our contributions and offer perspectives for future research. In the following we give a bird's-eye view of its content, starting with the development of a generalized framework for optimization (Chapter 2), followed by structure-exploiting algorithms for tensor decompositions based on that framework (Chapter 3), globalization and acceleration strategies for those algorithms (Chapter 4) and their accompanying solvers (Chapter 5), and finally, a framework for data fusion by coupled tensor factorizations with structured factors (Chapter 6). These chapters are all strongly based on the articles listed in the publications section at the back of the thesis. As such, they are entirely self-contained and can be read independently of each other.

To solve a data fusion problem in which multiple data sets are factorized, the first question that we need to address is how we will go about computing a single tensor factorization. Matrix factorizations such as the singular value decomposition (SVD) are nonlinear nonconvex optimization problems with many stationary points. Remarkably, the global minimizers of these problems can nevertheless be computed. Although iterative methods are unavoidable for reaching the solution, the algebraic algorithms used to compute matrix factorizations are nonetheless referred to as direct methods because the iterative component is both short and cheap. To achieve such a low computational complexity in the iterative phase, the input matrices are preprocessed by introducing a large amount of zeros with matrix transformations such as Givens rotations or Householder reflectors. The result is a sparse matrix whose

structure is easily exploited. Analogously, tensor factorizations are also nonlinear nonconvex optimization problems. However, there is little hope of applying a similar sequence of transformations to obtain direct methods for computing tensor factorizations. The problem here is that, in general, a fixed sequence of matrix transformations does not comprise enough degrees of freedom to create the necessary volume of zeros in a tensor. The nonexistence of such a reduction step is a prime reason that, in contrast to the matrix case, optimization-based algorithms for tensor decompositions are competitive with and often even outperform their algebraic counterparts. The direct methods for tensor factorization that do exist try to reduce the problem to a matrix factorization by making assumptions on the rank or exactness of the decomposition that are often not satisfied in practice. To make matters worse, a best low-rank factorization of a given tensor may not even exist [57]. If the factorization is indeed deemed incomputable, one solution may be to impose nonnegativity or orthogonality constraints on the factors. On a brighter note, the fact that tensor decompositions can be unique seems to have a positive effect on their computation. Experimental evidence suggests that computing tensor factorizations that are known to be unique often results in convergence to the global optimizer.

Meeting multiple objectives and imposing structure or constraints are difficult tasks to achieve in an algebraic framework. We elect to use optimization, which is in comparison much more flexible and generic in nature, as the framework underlying data fusion. However, the departure from algebraic algorithms to those based on optimization also brings its own set of problems. Much like matrices, tensor data may be complex-valued. This rarely poses problems for matrix factorizations as the algebra used to compute them has been built from the ground up to support complex-valued data. In contrast, the foundations of optimization have always focused on finding real-valued solutions. Of course, it could be argued that every complex number is just two real numbers in disguise. In that sense, the traditional view of optimization can be used to compute complex low-rank factorizations as is. However, much of optimization is based on derivatives and how to exploit their structure. Blindly applying standard optimization techniques to complex-valued tensor data is a surefire way to obfuscate any structure present in those derivatives, leading to expensive function evaluations and inefficient software implementations. Problematically, real-valued functions of complex variables do not admit a Taylor series and without a Taylor series, there can be no optimization. In Chapter 2, we develop complex optimization as a convincing generalization of real optimization to functions of complex variables. In particular, complex optimization focuses on solving the nonlinear optimization problem

$$\underset{\boldsymbol{z}}{\text{minimize}} \quad f(\boldsymbol{z}, \overline{\boldsymbol{z}}), \tag{1.5}$$

where $\boldsymbol{z}$ is a vector of complex variables that could, for instance, represent a tensor factorization, $\overline{\boldsymbol{z}}$ is its complex conjugate, and f is the objective function to be minimized. The resulting algorithms are a true generalization of real optimization in the sense that, given a function of real variables and its real-valued gradient, they produce the same iterates as their real optimization counterparts. In a case study, we show that simulation of nonlinear electronic circuits with complex optimization exposes the problem's structure in a way that real optimization can not, and therefore enables a significant reduction in computational complexity and memory cost. An efficient software implementation of these algorithms, as well as complex globalization strategies and numerical complex differentiation schemes are made available online as the Complex Optimization Toolbox for MATLAB [211].

In Chapter 3, we develop algorithms based on complex optimization for computing some of the most prevalent tensor decompositions: the CPD and the rank-$(L_r, L_r, 1)$ BTD [47, 48]. The latter can be thought of as a CPD with structured factors of a third-order tensor. We exploit this fact and introduce a new decomposition called the (rank-L_r $\circ$ rank-1) BTD that subsumes the former two decompositions as special cases. We express the computation of the generalized decomposition as the desire to minimize an objective function that represents a measure of distance between the data and the model. The complex nonlinear optimization problem (1.5) then takes the form

$$\underset{\boldsymbol{z}}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{M}(\boldsymbol{z}) - \mathcal{T}\|_{\text{F}}^2, \tag{1.6}$$

wherein the tensor $\mathcal{T}$ is approximated by the BTD model $\mathcal{M}$, parametrized by the possibly complex variables $\boldsymbol{z}$. We develop matrix-free algorithms that exploit the structure in the model $\mathcal{M}$ and its factors to their full extent, substantially reducing the computational complexity and memory cost. Compared to the workhorse algorithm for tensor decompositions, alternating least squares (ALS), numerical experiments confirm that our matrix-free approach clearly requires less computational effort and is much more robust with respect to the type of initialization and difficulty of the decomposition. Moreover, and in contrast to ALS, the developed algorithms are easy targets for parallelization.

The convergence of algorithms that compute tensor factorizations are known to slow down to a crawl when passing through an area of the search space where the factors are somewhat collinear. Given a current iterate and a descent direction, an exact line search can significantly reduce the time spent in such swamps. In Chapter 4 we introduce a broad class of optimization problems called tensor optimization and show that its exact line and plane search subproblems are tantamount to finding the global minimizer of univariate and bivariate polynomials, respectively. Tensor optimization is a special case of the more general class of polynomial optimization that subsumes, among others, many

types of tensor decompositions and least squares support tensor machines [224, 226]. Furthermore, the framework includes complex line searches and a generalization of line and plane search with which an additional scaling parameter can be determined for almost no additional cost. In the case of a real line search, the subproblem is easy to solve since finding a univariate polynomial's global minimizer can be achieved by computing the roots of its derivative. However, finding the global minimizer of a bivariate polynomial, which requires solving a bivariate polynomial system, is a much more difficult task. In Chapter 5, we present an algorithm to compute all isolated solutions of a bivariate polynomial system based on a single generalized eigenvalue decomposition. Moreover, the algorithm computes both x- and y-coordinates simultaneously and ensures these to be real-valued. Solutions are returned counting multiplicity and their accuracy is maximized by means of several numerical balancing steps and a Newton–Raphson refinement. Numerical experiments show that the method consistently outperforms existing solvers in both the fraction of recovered roots and the time to solution.

In Chapter 6 we present structured data fusion (SDF) as a framework for defining and solving data fusion problems as coupled tensor factorizations with possibly structured factors. In SDF, each data set is represented by a dense, sparse or incomplete tensor and factorized with a tensor decomposition. Factors may be imposed to have any structure which can be constructed as an explicit function of some underlying variables. The SDF optimization problem is

$$\underset{\boldsymbol{z}}{\text{minimize}} \quad \sum_d \frac{\omega_d}{2} \|\mathcal{M}^{(d)}(\mathcal{X}(\boldsymbol{z})) - \mathcal{T}^{(d)}\|^2_{\mathcal{W}^{(d)}}, \tag{1.7}$$

wherein the observed entries of the dth tensor $\mathcal{T}^{(d)}$, indicated by the tensor $\mathcal{W}^{(d)}$, are approximated by the tensor decomposition model $\mathcal{M}^{(d)}$, which is built using the factors $\mathcal{X}(\boldsymbol{z})$, in turn based on the possibly complex variables $\boldsymbol{z}$. By choosing the right decomposition types and factor structures, even classical matrix factorizations such as the EVD, SVD and QR factorization can be computed with SDF. Algorithms are developed with big data in mind, achieving linear complexity per iteration in the total number of known data entries. Moreover, the structure of the problem is exploited at the level of the factorization model and at the level of the factor structures. The algorithms are model and factor structure agnostic, allowing users to add their own factorizations and factor structures to the solver. Moreover, we present a domain specific language (DSL) for modelling SDF problems. We showcase the framework with four diverse applications that are all solved entirely within this DSL, demonstrating its high degree of flexibility and the substantially improved inference that the fusion of multiple data sources offers.

Furthermore, we have developed a concrete implementation of the theoretical frameworks and algorithms established in this thesis in the open-source software package Tensorlab [211]. The package offers a suite of complex optimization algorithms bundled with automatic complex differentiation, an efficient set of kernels for operating on tensors, many algorithms for computing the CPD, LMLRA and BTD, tools to help choose the parameters of those tensor decompositions, algorithms for global minimization of bivariate polynomials and rational functions, a complete implementation of SDF including a DSL for modelling data fusion problems and three types of basic tensor decompositions, a library of 32 factor structures and two types of regularization to choose from, and many other useful features such as tensor visualization and statistics. For a complete overview of Tensorlab's functionality and examples of its usage, see the accompanying user guide in Appendix A.

Chapter 2

Unconstrained optimization of real functions in complex variables

Nonlinear optimization problems in complex variables are frequently encountered in applied mathematics and engineering applications such as control theory, signal processing, and electrical engineering. Optimization of these problems often requires a first- or second-order approximation of the objective function to generate a new step or descent direction. However, such methods cannot be applied to real functions of complex variables because they are necessarily nonanalytic in their argument, i.e., the Taylor series expansion in their argument alone does not exist. To overcome this problem, the objective function is usually redefined as a function of the real and imaginary parts of its complex argument so that standard optimization methods can be applied. However, this approach may needlessly disguise any inherent structure present in the derivatives of such complex problems. Although little known, it is possible to construct an expansion of the objective function in its original complex variables by noting that functions of complex variables can be analytic in their argument and its complex conjugate as a whole. We use these complex Taylor series expansions to generalize existing optimization algorithms for both general nonlinear optimization problems and nonlinear least squares problems. We then apply these methods to two case studies which demonstrate that complex derivatives can lead to greater insight in the structure of the problem, and that this structure can often be exploited to improve computational complexity and storage cost. This chapter is a slightly adapted version of the article [206].

2.1 Introduction

In this chapter we focus on methods to solve unconstrained nonlinear optimization problems of the form

$$\underset{\boldsymbol{z}\in\mathbb{C}^n}{\text{minimize}} \quad f(\boldsymbol{z},\overline{\boldsymbol{z}}), \tag{2.1}$$

where f is a real, smooth function in n complex variables $\boldsymbol{z}$ and their complex conjugates $\overline{\boldsymbol{z}}$. We will also consider unconstrained nonlinear least squares problems of the form

$$\underset{\boldsymbol{z}\in\mathbb{C}^n}{\text{minimize}} \quad \frac{1}{2}\|F(\boldsymbol{z},\overline{\boldsymbol{z}})\|^2, \tag{2.2}$$

where F maps n complex variables $\boldsymbol{z}$ and their complex conjugates $\overline{\boldsymbol{z}}$ to m complex residuals $F(\boldsymbol{z},\overline{\boldsymbol{z}})$. Many nonlinear optimization methods use a first- or second-order approximation of the objective function to generate a new step or a descent direction, where the approximation is either updated or recomputed every iteration. On the other hand, the Cauchy–Riemann conditions assert that real-valued functions f in complex variables $\boldsymbol{z}$ are necessarily nonanalytic in $\boldsymbol{z}$. In other words, there exists no Taylor series in $\boldsymbol{z}$ of f at $\boldsymbol{z}_0$ so that the series converges to $f(\boldsymbol{z})$ in a neighbourhood of $\boldsymbol{z}_0$. A common workaround is to convert the optimization problem to the real domain by regarding f as a function of the real and imaginary parts of $\boldsymbol{z}$. However, by reformulating an optimization problem that is inherently complex to the real domain, it becomes easy to miss important insights about the structure of the problem which might otherwise be exploited. By making the dependence of the objective function on both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ explicit as we have done in (2.1) and (2.2), we will see that there are several ways to expand these functions into a complex Taylor series. The key to their construction lies in the fact that if a function is analytic in the space spanned by $\mathrm{Re}\{\boldsymbol{z}\}$ and $\mathrm{Im}\{\boldsymbol{z}\}$, it is also analytic in the space spanned by $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$. Indeed, there is a simple linear transformation from the real to the complex Taylor series. The resulting expansions allow us to generalize existing real optimization methods to the complex domain, and importantly, depend on complex derivatives that are often described by more elegant expressions than their real counterparts.

This chapter is organized as follows. The next section reviews our notation. In Section 2.3 we give a short overview of Wirtinger calculus, which is the underlying framework for the complex derivatives used in this chapter, and of complex Taylor series expansions. In Section 2.4 we use these expansions to generalize nonlinear optimization methods such as BFGS, limited-memory BFGS, and the nonlinear conjugate gradient algorithm to functions of complex variables. In Section 2.5 we do the same for nonlinear least squares methods such

as Gauss–Newton and Levenberg–Marquardt. In Section 2.6 we demonstrate the potential of these generalized optimization methods with two case studies. The first is the canonical polyadic decomposition, which is a tensor decomposition in rank-one terms. The second is the simulation of nonlinear circuits in the frequency domain. We conclude this chapter in Section 2.7.

2.2 Notation and preliminaries

Vectors are denoted by boldface letters and are lower case, e.g., $\boldsymbol{a}$. Matrices are denoted by capital letters, e.g., A. Higher-order tensors are denoted by Euler script letters, e.g., $\mathcal{A}$. The ith entry of a vector $\boldsymbol{a}$ is denoted by a_i, element (i, j) of a matrix A by a_{ij}, and element (i, j, k) of a third-order tensor $\mathcal{A}$ by a_{ijk}. Indices typically range from one to their capital version, e.g., $i = 1, \ldots, I$. A colon is used to indicate all elements of a mode. Thus $\boldsymbol{a}_{:j}$ corresponds to the jth column of a matrix A, which we also denote more compactly as $\boldsymbol{a}_j$. The nth element in a sequence is denoted by a superscript in parentheses, e.g., $A^{(n)}$ denotes the nth matrix in a sequence. The superscripts $\cdot^{\mathrm{T}}$, $\cdot^{\mathrm{H}}$, $\cdot^{-1}$, and $\cdot^{\dagger}$ are used for the transpose, Hermitian conjugate, matrix inverse and Moore–Penrose pseudoinverse, respectively. The identity matrix of order n is denoted by $\mathbb{I}_n$ and the $m \times n$ all-zero matrix by $0_{m \times n}$. The complex conjugate is denoted by an overline, e.g., $\overline{\boldsymbol{a}}^{\mathrm{T}}$ is equivalent to $\boldsymbol{a}^{\mathrm{H}}$. The two-norm and Frobenius norm are denoted by $\| \cdot \|$ and $\| \cdot \|_{\mathrm{F}}$, respectively. We use parentheses to denote the concatenation of two or more vectors, e.g., $(\boldsymbol{a}, \boldsymbol{b})$ is equivalent to $\begin{bmatrix} \boldsymbol{a}^{\mathrm{T}} & \boldsymbol{b}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}$.

2.3 Wirtinger calculus and the complex Taylor series expansions

The calculus underlying the complex derivatives in this chapter goes back to H. Poincaré and was developed principally by W. Wirtinger in the early 20th century. It is often called, especially in the German literature, the Wirtinger calculus [182]. Brandwood first introduced the notion of a complex gradient in the context of optimization [27], though without making the connection with Wirtinger derivatives. Later, Abatzoglou, Mendel, and Harada generalized Brandwood's idea to construct a second-order Taylor series expansion [2]. Their expansion is defined in the function's complex arguments and their complex conjugates separately, resulting in a sum of four second-order terms and no clear definition of a complex Hessian. The complex Taylor series as presented in this chapter is due to van den Bos [233], who found a compact way of transforming

the real gradient and Hessian into their complex counterparts. Kreutz-Delgado went on to point out that there is actually more than one way to define the complex Hessian [114]. We present the two most intuitive definitions, of which one in particular is useful from an optimization point of view.

In order to facilitate the transition from and to the real and complex numbers, we first define the vector spaces $\mathcal{R}$, $\mathcal{C}$, and $\mathcal{C}^*$ as $\{(\boldsymbol{x},\boldsymbol{y})\,|\boldsymbol{x}\in\mathbb{R}^n,\boldsymbol{y}\in\mathbb{R}^n\}=\mathbb{R}^{2n}$, $\{(\boldsymbol{z},\overline{\boldsymbol{z}})\,|\boldsymbol{z}\in\mathbb{C}^n\}\subset\mathbb{C}^{2n}$, and $\{(\overline{\boldsymbol{z}},\boldsymbol{z})\,|\boldsymbol{z}\in\mathbb{C}^n\}\subset\mathbb{C}^{2n}$, respectively. Elements of $\mathbb{C}^n$ have an equivalent representation in any of these spaces. Let $\boldsymbol{z}\in\mathbb{C}^n$, then we define $\overset{\mathcal{R}}{\boldsymbol{z}}:=(\mathrm{Re}\{\boldsymbol{z}\},\mathrm{Im}\{\boldsymbol{z}\})\in\mathcal{R}$, $\overset{\mathcal{C}}{\boldsymbol{z}}:=(\boldsymbol{z},\overline{\boldsymbol{z}})\in\mathcal{C}$ and $\overset{\mathcal{C}^*}{\boldsymbol{z}}:=(\overline{\boldsymbol{z}},\boldsymbol{z})\equiv\overset{\mathcal{C}}{\overline{\boldsymbol{z}}}\in\mathcal{C}$. Furthermore, the linear map

$$J:=\begin{bmatrix}\mathbb{I}_n & i\mathbb{I}_n\\ \mathbb{I}_n & -i\mathbb{I}_n\end{bmatrix} \tag{2.3}$$

is an isomorphism from $\mathcal{R}$ to $\mathcal{C}$ and its inverse is given by $J^{-1}=\frac{1}{2}J^{\mathrm{H}}$. The swap operator

$$S:=\begin{bmatrix}0 & \mathbb{I}_n\\ \mathbb{I}_n & 0\end{bmatrix} \tag{2.4}$$

is an isomorphism from $\mathcal{C}$ to the dual space $\mathcal{C}^*$. Its inverse is given by $S^{-1}=S^{\mathrm{T}}=S$.

Definition 2.1. *Let $\boldsymbol{z}\in\mathbb{C}^n$ and let $\boldsymbol{x}=\mathrm{Re}\{\boldsymbol{z}\}$ and $\boldsymbol{y}=\mathrm{Im}\{\boldsymbol{z}\}$. The cogradient operator $\frac{\partial}{\partial\boldsymbol{z}}$ and conjugate cogradient operator $\frac{\partial}{\partial\overline{\boldsymbol{z}}}$ are defined as* [27,114,182,233]

$$\frac{\partial}{\partial\boldsymbol{z}}:=\frac{1}{2}\begin{bmatrix}\frac{\partial}{\partial x_1}-\frac{\partial}{\partial y_1}i\\ \vdots\\ \frac{\partial}{\partial x_n}-\frac{\partial}{\partial y_n}i\end{bmatrix}, \tag{2.5a}$$

$$\frac{\partial}{\partial\overline{\boldsymbol{z}}}:=\frac{1}{2}\begin{bmatrix}\frac{\partial}{\partial x_1}+\frac{\partial}{\partial y_1}i\\ \vdots\\ \frac{\partial}{\partial x_n}+\frac{\partial}{\partial y_n}i\end{bmatrix}. \tag{2.5b}$$

The (conjugate) cogradient operator acts as a partial derivative with respect to $\boldsymbol{z}$ ($\overline{\boldsymbol{z}}$), treating $\overline{\boldsymbol{z}}$ ($\boldsymbol{z}$) as a constant. To see this, let $z\in\mathbb{C}$ and let $x=\mathrm{Re}\{z\}$ and $y=\mathrm{Im}\{z\}$ so that $x=\frac{1}{2}(z+\overline{z})$ and $y=\frac{i}{2}(\overline{z}-z)$. Then for a function $f:\mathbb{C}\to\mathbb{C}$ we have that $\frac{\partial f}{\partial z}=\frac{\partial f}{\partial x}\frac{\partial x}{\partial z}+\frac{\partial f}{\partial y}\frac{\partial y}{\partial z}$, which is equal to $\frac{1}{2}(\frac{\partial f}{\partial x}-\frac{\partial f}{\partial y}i)$ if $\frac{\partial\overline{z}}{\partial z}$ is set to zero. Note that the Cauchy–Riemann conditions for f to be analytic in z can be expressed compactly using the cogradient as $\frac{\partial f}{\partial\overline{z}}=0$, i.e., f is a function only of z. Analogously, if f is analytic in $\overline{z}$, then $\frac{\partial f}{\partial z}=0$.

Although their definitions often allow the cogradients to be expressed elegantly in terms of $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$, neither contains enough information by itself to express the change in a function with respect to a change in $\boldsymbol{z}$. This motivates the following definition of a complex gradient operator.

Definition 2.2. *Let $\boldsymbol{z} \in \mathbb{C}^n$. We define the complex gradient operator $\frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}$ as*

$$\frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} := \left(\frac{\partial}{\partial \boldsymbol{z}}, \frac{\partial}{\partial \overline{\boldsymbol{z}}} \right). \tag{2.6}$$

The linear map (2.3) also defines a one-to-one correspondence between the real gradient $\frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}}$ and the complex gradient $\frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}$, namely,

$$\frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} = J^{\mathrm{T}} \frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}. \tag{2.7}$$

Similarly, the real Hessian $\frac{\partial^2}{\partial \overset{\mathcal{R}}{\boldsymbol{z}} \partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}}$ can be transformed into several complex Hessians, two of which are

$$\frac{\partial^2}{\partial \overset{\mathcal{R}}{\boldsymbol{z}} \partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}} = \frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} \left(\frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} \right)^{\mathrm{T}} = J^{\mathrm{T}} \frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \left(\frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} J \right) = J^{\mathrm{T}} \frac{\partial^2}{\partial \overset{\mathcal{C}}{\boldsymbol{z}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} J, \tag{2.8a}$$

$$\frac{\partial^2}{\partial \overset{\mathcal{R}}{\boldsymbol{z}} \partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}} = \frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} \left(\frac{\partial}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} \right)^{\mathrm{T}} = J^{\mathrm{H}} \frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \left(\frac{\partial}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} J \right) = J^{\mathrm{H}} \frac{\partial^2}{\partial \overset{\mathcal{C}}{\boldsymbol{z}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} J. \tag{2.8b}$$

2.3.1 First-order complex Taylor series expansion

Consider the first-order real Taylor series expansion of a function $F : \mathcal{R} \to \mathbb{C}^m$,

$$m_{\mathrm{F}}(\Delta \overset{\mathcal{R}}{\boldsymbol{z}}) = F(\overset{\mathcal{R}}{\boldsymbol{z}}) + \frac{\partial F(\overset{\mathcal{R}}{\boldsymbol{z}})}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{R}}{\boldsymbol{z}}. \tag{2.9}$$

Because $\mathcal{R}$ and $\mathcal{C}$ are isomorphic, the function F can also be regarded as a function of $\overset{\mathcal{C}}{\boldsymbol{z}}$. Although it is generally not true that F is analytic in $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ independently if F is analytic in $\overset{\mathcal{R}}{\boldsymbol{z}}$, it does hold that F is analytic in $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ as a whole. Using (2.3) and (2.7), the first-order complex Taylor series expansion of F can be expressed as

$$m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}}) = F(\overset{\mathcal{C}}{\boldsymbol{z}}) + \frac{\partial F(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}. \tag{2.10}$$

The matrix $\frac{\partial F}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} = \begin{bmatrix} \frac{\partial F}{\partial \boldsymbol{z}^{\mathrm{T}}} & \frac{\partial F}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}} \end{bmatrix}$ is obtained by applying the transpose of the complex gradient operator (2.7) componentwise to F. The matrices $\frac{\partial F}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and $\frac{\partial F}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$ are called the Jacobian and conjugate Jacobian, respectively.

2.3.2 Second-order complex Taylor series expansion

Consider the second-order real Taylor series expansion of a function $f : \mathcal{R} \to \mathbb{C}$,

$$m_{\mathrm{F}}(\Delta \overset{\mathcal{R}}{\boldsymbol{z}}) = f(\overset{\mathcal{R}}{\boldsymbol{z}}) + \Delta \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{R}}{\boldsymbol{z}})}{\partial \overset{\mathcal{R}}{\boldsymbol{z}}} + \frac{1}{2} \Delta \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}} \frac{\partial^2 f(\overset{\mathcal{R}}{\boldsymbol{z}})}{\partial \overset{\mathcal{R}}{\boldsymbol{z}} \partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{R}}{\boldsymbol{z}}. \tag{2.11}$$

Because $\mathcal{R}$ and $\mathcal{C}$ are isomorphic, the function f can also be regarded as a function of $\overset{\mathcal{C}}{\boldsymbol{z}}$. Using (2.3), (2.7), and (2.8), the second-order Taylor series expansion of $f(\overset{\mathcal{C}}{\boldsymbol{z}})$ can be expressed in the following two equivalent ways:

$$m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}}) = f(\overset{\mathcal{C}}{\boldsymbol{z}}) + \Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{1}{2} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}} \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}, \tag{2.12a}$$

$$m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}}) = f(\overset{\mathcal{C}}{\boldsymbol{z}}) + \Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{1}{2} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{H}} \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}. \tag{2.12b}$$

Often $f(\overset{\mathcal{R}}{\boldsymbol{z}})$ will have continuous second-order derivatives. Clairaut's theorem [103] states that the order of differentiation does not matter for functions for which this property holds, and hence that the real Hessian $\frac{\partial^2 f}{\partial \overset{\mathcal{R}}{\boldsymbol{z}} \partial \overset{\mathcal{R}}{\boldsymbol{z}}^{\mathrm{T}}}$ is symmetric. It then follows from (2.8) that $\frac{\partial^2 f}{\partial \overset{\mathcal{C}}{\boldsymbol{z}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}}$ is symmetric and $\frac{\partial^2 f}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}}$ is Hermitian for real-valued f.

Note that (2.12b) is not differentiable with respect to $\Delta \overset{\mathcal{C}}{\boldsymbol{z}}$ because of the dependency on $\Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{H}}$. In contrast, (2.12a) is differentiable with respect to $\Delta \overset{\mathcal{C}}{\boldsymbol{z}}$. To overcome this apparent difficulty, we note that the dependency on $\Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{H}}$ can be resolved by substituting $\Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{H}} = \Delta \overset{\mathcal{C}}{\overline{\boldsymbol{z}}}^{\mathrm{T}} = \Delta \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}} S$. The complex gradient of the second-order Taylor expansions (2.12) can now be computed as

$$\frac{\partial m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \Delta \overset{\mathcal{C}}{\boldsymbol{z}}} = \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{C}}{\boldsymbol{z}} \tag{2.13a}$$

and

$$\frac{\partial m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \Delta \overset{\mathcal{C}}{\boldsymbol{z}}} = \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{1}{2} \left(S \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} + \left(\frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \right)^{\mathrm{T}} S \right) \Delta \overset{\mathcal{C}}{\boldsymbol{z}}, \tag{2.13b}$$

respectively.

Let us now assume f maps to the real numbers. This assumption has certain consequences for the structure of the gradient and Hessian in the Taylor series expansions (2.12). First, because of the identity $\frac{\partial \overline{f}}{\partial \boldsymbol{z}} = \overline{\frac{\partial f}{\partial \overline{\boldsymbol{z}}}}$ [153, 182], it follows

that $\frac{\partial f}{\partial \boldsymbol{z}} = \overline{\frac{\partial f}{\partial \overline{\boldsymbol{z}}}}$ for real-valued f. Second, it can also be shown that $\frac{\partial^2 f}{\partial \boldsymbol{z} \partial \overline{\boldsymbol{z}}^{\mathrm{T}}} = \frac{\partial^2 f}{\partial \overline{\boldsymbol{z}} \partial \boldsymbol{z}^{\mathrm{T}}}$ and $\frac{\partial^2 f}{\partial \boldsymbol{z} \partial \boldsymbol{z}^{\mathrm{T}}} = \overline{\frac{\partial^2 f}{\partial \overline{\boldsymbol{z}} \partial \overline{\boldsymbol{z}}^{\mathrm{T}}}}$. Using these properties, we can simplify expression (2.13b) by premultiplying with S, so that the model's conjugate complex gradient is given by

$$S \frac{\partial m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \Delta \overset{\mathcal{C}}{\boldsymbol{z}}} = S \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{1}{2} \left(S S \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} + S \left(\frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \right)^{\mathrm{T}} S \right) \Delta \overset{\mathcal{C}}{\boldsymbol{z}}$$

$$\overline{\frac{\partial m_{\mathrm{F}}(\Delta \overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \Delta \overset{\mathcal{C}}{\boldsymbol{z}}}} = \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}} + \frac{\partial^2 f(\overset{\mathcal{C}}{\boldsymbol{z}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{z}}} \partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}} \Delta \overset{\mathcal{C}}{\boldsymbol{z}}. \tag{2.14}$$

2.4 Nonlinear optimization

First we consider nonlinear optimization problems of the form (2.1). The space spanned by $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ is equal to $\mathcal{C}$, which is equivalent to $\mathcal{R}$ under the linear map (2.3). It is then understood that the function f can also be regarded as a function that maps $\mathcal{C}$ or $\mathcal{R}$ to $\mathbb{R}$, so that both the second-order model (2.11) in $\overset{\mathcal{R}}{\boldsymbol{z}}$ and the two second-order models (2.12) in $\overset{\mathcal{C}}{\boldsymbol{z}}$ are applicable.

2.4.1 The generalized BFGS and L-BFGS methods

In the generalized BFGS method, we use the quadratic model

$$m_k(\overset{\mathcal{C}}{\boldsymbol{p}}) = f(\overset{\mathcal{C}}{\boldsymbol{z}}_k) + \overset{\mathcal{C}}{\boldsymbol{p}}^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} + \frac{1}{2} \overset{\mathcal{C}}{\boldsymbol{p}}^{\mathrm{H}} B_k \overset{\mathcal{C}}{\boldsymbol{p}} \tag{2.15}$$

of the objective function f at the current iterate $\overset{\mathcal{C}}{\boldsymbol{z}}_k$, where B_k is a Hermitian positive definite matrix that is updated every iteration. Since f is real-valued, the minimizer of this convex quadratic model can be obtained by setting the conjugate complex gradient (2.14) equal to zero, and is given by

$$\overset{\mathcal{C}}{\boldsymbol{p}}_k = -B_k^{-1} \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}}. \tag{2.16}$$

In a line search framework, the next iterate is $\boldsymbol{z}_{k+1} = \boldsymbol{z}_k + \alpha_k \boldsymbol{p}_k$, where the real step length α_k is usually chosen to satisfy the (strong) Wolfe conditions [159,245]. A reasonable requirement for the updated Hessian B_{k+1} is that the gradient of the model m_{k+1} matches the gradient of the objective function f in the last two iterates $\overset{\mathcal{C}}{\boldsymbol{z}}_k$ and $\overset{\mathcal{C}}{\boldsymbol{z}}_{k+1}$. The second condition is satisfied automatically by

(2.15). The first condition can be written as

$$\overline{\frac{\partial m_{k+1}(-\alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{p}}}} = \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_{k+1})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}} - \alpha_k B_{k+1} \overset{\mathcal{C}}{\boldsymbol{p}}_k = \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}}.$$

Define $\overset{\mathcal{C}}{\boldsymbol{y}}_k := \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_{k+1})}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}} - \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}}$ and $\overset{\mathcal{C}}{\boldsymbol{s}}_k := \overset{\mathcal{C}}{\boldsymbol{z}}_{k+1} - \overset{\mathcal{C}}{\boldsymbol{z}}_k$, then by rearranging we find that B_{k+1} should satisfy the secant equation

$$B_{k+1} \overset{\mathcal{C}}{\boldsymbol{s}}_k = \overset{\mathcal{C}}{\boldsymbol{y}}_k. \tag{2.17}$$

The BFGS update [29, 76, 82, 190] is chosen so that the inverse of the updated Hessian B_{k+1}^{-1} is, among all symmetric positive definite matrices satisfying the secant equation (2.17), in some sense closest to the current inverse Hessian approximation B_k^{-1}. In Lemma 2.1 and Theorem 2.2, we generalize Schnabel and Dennis' derivation of the BFGS update [187] to Hermitian Hessians. The proof requires the Broyden update [61], which is generalized to the complex space in Theorem 2.1.

Theorem 2.1. *Let $B \in \mathbb{C}^{n\times n}$, $\boldsymbol{y}, \boldsymbol{s} \in \mathbb{C}^n$, and $\boldsymbol{s}$ be nonzero. Then the unique minimizer of*

$$\begin{aligned} &\underset{\hat{B}\in\mathbb{C}^{n\times n}}{\text{minimize}} && \|B - \hat{B}\|_{\mathrm{F}} \\ &\text{subject to} && \hat{B}\boldsymbol{s} = \boldsymbol{y} \end{aligned} \tag{2.18}$$

is given by the Broyden update

$$B^* = B + \frac{(\boldsymbol{y} - B\boldsymbol{s})\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{s}^{\mathrm{H}}\boldsymbol{s}}. \tag{2.19}$$

Proof. To show that B^* is a solution to (2.18), note that $B^*\boldsymbol{s} = \boldsymbol{y}$ and that

$$\|B - B^*\|_{\mathrm{F}} = \left\|(B - \hat{B})\frac{\boldsymbol{s}\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{s}^{\mathrm{H}}\boldsymbol{s}}\right\|_{\mathrm{F}} \leq \|B - \hat{B}\|_{\mathrm{F}}.$$

That B^* is the unique solution follows from the fact that the mapping $f : \mathbb{C}^{n\times n} \to \mathbb{R}$ defined by $f(\hat{B}) = \|B - \hat{B}\|_{\mathrm{F}}$ is strictly convex in $\mathbb{C}^{n\times n}$ and that the set of $\hat{B} \in \mathbb{C}^{n\times n}$ such that $\hat{B}\boldsymbol{s} = \boldsymbol{y}$ is convex. □

Lemma 2.1. *Let $\boldsymbol{y}, \boldsymbol{s} \in \mathbb{C}^n$, $\boldsymbol{s}$ nonzero and let $Q(\boldsymbol{y}, \boldsymbol{s}) := \{B \in \mathbb{C}^{n\times n} | B\boldsymbol{s} = \boldsymbol{y}\}$. Then the generalized quotient $Q(\boldsymbol{y}, \boldsymbol{s})$ contains a Hermitian positive definite matrix if and only if for some nonzero $\boldsymbol{v} \in \mathbb{C}^n$ and nonsingular $J \in \mathbb{C}^{n\times n}$, $\boldsymbol{y} = J\boldsymbol{v}$, and $\boldsymbol{v} = J^{\mathrm{H}}\boldsymbol{s}$.*

Proof. If $\boldsymbol{v}$ and J exist, then $\boldsymbol{y} = J\boldsymbol{v} = JJ^{\mathrm{H}}\boldsymbol{s}$ and JJ^{H} is a Hermitian positive definite matrix in $Q(\boldsymbol{y}, \boldsymbol{s})$. Now suppose B is a Hermitian positive definite matrix which satisfies $B\boldsymbol{s} = \boldsymbol{y}$. Let $B = LL^{\mathrm{H}}$ be the Cholesky factorization of B and set $J = L$ and $\boldsymbol{v} = L^{\mathrm{H}}\boldsymbol{s}$ to complete the proof. □

Theorem 2.2. *Let $L \in \mathbb{C}^{n\times n}$ be nonsingular, $B = LL^{\mathrm{H}}$, $\boldsymbol{y}, \boldsymbol{s} \in \mathbb{C}^n$, and $\boldsymbol{s}$ be nonzero. There is a Hermitian positive definite matrix $\hat{B} \in Q(\boldsymbol{y}, \boldsymbol{s})$ if and only if $\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s} > 0$. If there is such a matrix, then the generalized BFGS update $B^* := \hat{L}\hat{L}^{\mathrm{H}}$ is one such, where*

$$\hat{L} = \left(\mathbb{I}_n \pm \sqrt{\frac{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}}{\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}}}\frac{\boldsymbol{y}\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}} - \frac{B\boldsymbol{s}\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}}\right)L, \tag{2.20}$$

so that

$$B^* = B - \frac{B\boldsymbol{s}\boldsymbol{s}^{\mathrm{H}}B}{\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}} + \frac{\boldsymbol{y}\boldsymbol{y}^{\mathrm{H}}}{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}} \tag{2.21}$$

and

$$B^{*-1} = \left(\mathbb{I}_n - \frac{\boldsymbol{s}\boldsymbol{y}^{\mathrm{H}}}{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}}\right)B^{-1}\left(\mathbb{I}_n - \frac{\boldsymbol{y}\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}}\right) + \frac{\boldsymbol{s}\boldsymbol{s}^{\mathrm{H}}}{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}}. \tag{2.22}$$

Proof. Let $\hat{B}$ be a Hermitian positive definite matrix in $Q(\boldsymbol{y}, \boldsymbol{s})$, then $\boldsymbol{s}^{\mathrm{H}}\boldsymbol{y} = \boldsymbol{s}^{\mathrm{H}}\hat{B}\boldsymbol{s} > 0$ is a necessary condition for the update to exist. The nearest matrix $\hat{L}$ in Frobenius norm to L that satisfies $\hat{L}\boldsymbol{v} = \boldsymbol{y}$ is given by the Broyden update (2.19),

$$\hat{L} = L + \frac{(\boldsymbol{y} - L\boldsymbol{v})\boldsymbol{v}^{\mathrm{H}}}{\boldsymbol{v}^{\mathrm{H}}\boldsymbol{v}}.$$

If we can find a $\boldsymbol{v} \in \mathbb{C}^n$ such that $\boldsymbol{v} = \hat{L}^{\mathrm{H}}\boldsymbol{s}$, then by Lemma 2.1 we know that $\hat{L}\hat{L}^{\mathrm{H}}$ is in $Q(\boldsymbol{y}, \boldsymbol{s})$. The condition

$$\boldsymbol{v} = \hat{L}^{\mathrm{H}}\boldsymbol{s} = L^{\mathrm{H}}\boldsymbol{s} + \frac{(\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s} - \boldsymbol{v}^{\mathrm{H}}L^{\mathrm{H}}\boldsymbol{s})}{\boldsymbol{v}^{\mathrm{H}}\boldsymbol{v}}\boldsymbol{v}$$

implies that $\boldsymbol{v} = \alpha L^{\mathrm{H}}\boldsymbol{s}$ for some scalar α. Plugging back in, we find

$$\alpha = 1 + \frac{\alpha\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s} - |\alpha|^2\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}}{|\alpha|^2\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}}$$

or

$$|\alpha|^2 = \frac{\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s}}{\boldsymbol{s}^{\mathrm{H}}B\boldsymbol{s}},$$

which shows that $\boldsymbol{y}^{\mathrm{H}}\boldsymbol{s} > 0$ is a sufficient condition for the update to exist. Filling $\boldsymbol{v} = \alpha L^{\mathrm{H}}\boldsymbol{s}$ back in the Broyden update results in (2.20). It is then a matter of verifying (2.21)-(2.22) by forming the product $\hat{L}\hat{L}^{\mathrm{H}}$ and applying the Sherman–Morrison formula [191, 192], respectively. □

Theorem 2.2 holds for any $\boldsymbol{s}$ and $\boldsymbol{y}$ in $\mathbb{C}^{2n}$ if all dimensions are scaled appropriately, and hence also for the subset $\mathcal{C} \subset \mathbb{C}^{2n}$. At every step of the generalized BFGS method, the inverse Hessian can then be updated by (2.22), so that

$$B_{k+1}^{-1} = \left(\mathbb{I}_{2n} - \frac{\overset{\mathcal{C}}{\boldsymbol{s}}_k \overset{\mathcal{C}}{\boldsymbol{y}}_k^{\mathrm{H}}}{\overset{\mathcal{C}}{\boldsymbol{y}}_k^{\mathrm{H}} \overset{\mathcal{C}}{\boldsymbol{s}}_k}\right) B_k^{-1} \left(\mathbb{I}_{2n} - \frac{\overset{\mathcal{C}}{\boldsymbol{y}}_k \overset{\mathcal{C}}{\boldsymbol{s}}_k^{\mathrm{H}}}{\overset{\mathcal{C}}{\boldsymbol{y}}_k^{\mathrm{H}} \overset{\mathcal{C}}{\boldsymbol{s}}_k}\right) + \frac{\overset{\mathcal{C}}{\boldsymbol{s}}_k \overset{\mathcal{C}}{\boldsymbol{s}}_k^{\mathrm{H}}}{\overset{\mathcal{C}}{\boldsymbol{y}}_k^{\mathrm{H}} \overset{\mathcal{C}}{\boldsymbol{s}}_k}.$$

If the number of variables n is large, the cost of storing and manipulating the inverse Hessian B_{k+1}^{-1} can become prohibitive. The limited-memory BFGS (L-BFGS) method [78, 136, 158] circumvents this problem by storing the inverse Hessian implicitly as a set of m vector pairs $\{\overset{\mathcal{C}}{\boldsymbol{s}}_i, \overset{\mathcal{C}}{\boldsymbol{y}}_i\}$, $i = k-m, \ldots, k-1$. In fact, it suffices to store $\{\boldsymbol{s}_i, \boldsymbol{y}_i\}$, since the second half of any vector in $\mathcal{C}$ is just the complex conjugate of its first half.

Suppose we have a real function of complex variables $f : \mathcal{C} \to \mathbb{R}$ that we are interested in minimizing for $\boldsymbol{z} \in \mathbb{C}^n$ as well as for $\boldsymbol{x} \in \mathbb{R}^n$. The quasi-Newton step (2.16) depends on the complex gradient. Because the conjugate cogradient is just the complex conjugate of the cogradient for real-valued f, we need only compute one of the cogradients. In this case, we choose the conjugate cogradient $\frac{\partial f}{\partial \overline{\boldsymbol{z}}}$ because it coincides with the steepest descent direction. For optimization of $\boldsymbol{x} \in \mathbb{R}^n$, we need the real gradient $\frac{\partial f(\boldsymbol{x}_k)}{\partial \boldsymbol{x}}$, which can also be expressed as $\frac{\partial f(\boldsymbol{x}_k)}{\partial \boldsymbol{z}} + \frac{\partial f(\boldsymbol{x}_k)}{\partial \overline{\boldsymbol{z}}} = 2\frac{\partial f(\boldsymbol{x}_k)}{\partial \overline{\boldsymbol{z}}}$. Therefore, by constructing complex optimization algorithms in a way that only requires evaluating objective function values $f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)$ and scaled conjugate cogradients $\boldsymbol{g}(\overset{\mathcal{C}}{\boldsymbol{z}}_k) := 2\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overline{\boldsymbol{z}}}$, the function can be minimized over $\mathbb{C}^n$ as well as $\mathbb{R}^n$ without a separate implementation of the real gradient $\frac{\partial f(\boldsymbol{x}_k)}{\partial \boldsymbol{x}}$.

In Algorithm 2.1 a generalized limited-memory BFGS two-loop recursion for computing the quasi-Newton step (2.16) is presented. Its computational and storage cost are equal to that of the limited-memory BFGS method applied in the real domain. However, the generalized method requires the objective function's gradient in a form that may be more intuitive to the user, and can furthermore minimize the same objective function over $\mathbb{R}^n$ without requiring the real gradient to be treated as a separate case.

The quasi-Newton step $\boldsymbol{p}_k^*$ computed by Algorithm 2.1 can be used in either a line search or trust-region framework. In the former, the next iterate is $\boldsymbol{z}_{k+1} = \boldsymbol{z}_k + \alpha_k \boldsymbol{p}_k^*$. The step length α_k is usually chosen to loosely minimize

Input: $\boldsymbol{g}_k := \boldsymbol{g}(\overset{\mathcal{C}}{\boldsymbol{z}}_k) = 2\overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \boldsymbol{z}}} = 2\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overline{\boldsymbol{z}}}$,
$\boldsymbol{s}_i := \boldsymbol{z}_{i+1} - \boldsymbol{z}_i$,
$\boldsymbol{y}_i := \boldsymbol{g}_{i+1} - \boldsymbol{g}_i$ and
$\rho_i := \mathrm{Re}\{\boldsymbol{y}_i^{\mathrm{H}}\boldsymbol{s}_i\}^{-1}$ for $i = k-m, \ldots, k-1$

Output: $\boldsymbol{p}$ which satisfies $\overset{\mathcal{C}}{\boldsymbol{p}} = -B_k^{-1}\overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}}$

$\boldsymbol{p} \leftarrow -\boldsymbol{g}_k$
for $i = k-1, k-2, \ldots, k-m$ **do**
  $\alpha_i \leftarrow \rho_i \mathrm{Re}\{\boldsymbol{s}_i^{\mathrm{H}}\boldsymbol{p}\}$
  $\boldsymbol{p} \leftarrow \boldsymbol{p} - \alpha_i \boldsymbol{y}_i$
end
$\boldsymbol{p} \leftarrow \frac{1}{2}B_{k-m}^{-1}\boldsymbol{p}$ (e.g., $\frac{1}{2}B_{k-m}^{-1} = \frac{\mathrm{Re}\{\boldsymbol{y}_{k-1}^{\mathrm{H}}\boldsymbol{s}_{k-1}\}}{\boldsymbol{y}_{k-1}^{\mathrm{H}}\boldsymbol{y}_{k-1}}\mathbb{I}_n$ [78])
for $i = k-m, k-m+1, \ldots, k-1$ **do**
  $\beta \leftarrow \rho_i \mathrm{Re}\{\boldsymbol{y}_i^{\mathrm{H}}\boldsymbol{p}\}$
  $\boldsymbol{p} \leftarrow \boldsymbol{p} + (\alpha_i - \beta)\boldsymbol{s}_i$
end

Algorithm 2.1: Generalized L-BFGS two-loop recursion.

the one-dimensional optimization problem $\underset{\alpha_k}{\text{minimize}}\, f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)$ so that the Wolfe conditions [159, 245]

$$f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k) \leq f(\overset{\mathcal{C}}{\boldsymbol{z}}_k) + c_1 \alpha_k \boldsymbol{p}_k^{\mathcal{C}\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \tag{2.23a}$$

and

$$\overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \geq c_2 \overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}, \tag{2.23b}$$

where $0 < c_1 < c_2 < 1$, are satisfied. Inequalities (2.23a) and (2.23b) are known as the sufficient decrease and curvature condition, respectively. The former ensures the objective function is sufficiently smaller at the next iterate, and the latter ensures convergence of the gradient to zero. Furthermore, the curvature condition is a sufficient condition for the BFGS update to exist since

$$\overset{\mathcal{C}}{\boldsymbol{s}}_k^{\mathrm{H}} \overset{\mathcal{C}}{\boldsymbol{y}}_k = \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{H}} \left(\overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}} - \overline{\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}}} \right) \geq (c_2 - 1)\alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} > 0.$$

A step length may satisfy the Wolfe conditions without being particularly close to a minimizer of $f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)$. In the strong Wolfe conditions, the curvature

condition is replaced by

$$\left| \overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \right| \leq c_2 \left| \overset{\mathcal{C}}{\boldsymbol{p}}_k^{\mathrm{T}} \frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}} \right|$$

so that points far from stationary points are excluded. Line search algorithms are an integral part of quasi-Newton methods, but can be difficult to implement. There are several good software implementations available in the public domain, such as Moré and Thuente [147] and Hager and Zhang [93], which can be generalized to functions in complex variables with relatively little effort. Like Algorithm 2.1, their implementation can be organized such that all computations are in $\mathbb{C}^n$ instead of $\mathcal{C}$. For instance, the Wolfe conditions (2.23) are equivalent to

$$f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k) \leq f(\overset{\mathcal{C}}{\boldsymbol{z}}_k) + c_1 \alpha_k \mathrm{Re}\{\boldsymbol{p}_k^{\mathrm{H}} \boldsymbol{g}_k\} \tag{2.24a}$$

and

$$\mathrm{Re}\left\{\boldsymbol{p}_k^{\mathrm{H}} \boldsymbol{g}(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \alpha_k \overset{\mathcal{C}}{\boldsymbol{p}}_k)\right\} \geq c_2 \mathrm{Re}\{\boldsymbol{p}_k^{\mathrm{H}} \boldsymbol{g}_k\}. \tag{2.24b}$$

In a trust-region framework, a region around the current iterate $\boldsymbol{z}_k$ is defined in which the model m_k is trusted to be an adequate representation of the objective function. The next iterate $\boldsymbol{z}_{k+1}$ is then chosen to be the approximate minimizer of the model in this region. In effect, the direction and length of the step is chosen simultaneously. The trust-region radius Δ_k is updated every iteration based on the trustworthiness ρ_k of the model, which is defined as the ratio of the actual reduction $f(\overset{\mathcal{C}}{\boldsymbol{z}}_k) - f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \overset{\mathcal{C}}{\boldsymbol{p}}_k)$ and the predicted reduction $m_k(\boldsymbol{0}) - m_k(\overset{\mathcal{C}}{\boldsymbol{p}}_k)$.

There exist several strategies to approximately solve the trust-region subproblem

$$\begin{aligned} &\underset{\boldsymbol{p} \in \mathbb{C}^n}{\text{minimize}} \quad m_k(\overset{\mathcal{C}}{\boldsymbol{p}}) \\ &\text{subject to} \quad \|\boldsymbol{p}\| \leq \Delta_k \end{aligned} \tag{2.25}$$

The quasi-Newton step $\boldsymbol{p}_k^*$ minimizes (2.25) when $\|\boldsymbol{p}_k^*\| \leq \Delta_k$. If the trust-region radius is small compared to the quasi-Newton step, the quadratic term in m_k has little effect and the solution to the trust-region subproblem can be approximated by $\boldsymbol{p}^* = -\Delta_k \frac{\boldsymbol{g}_k}{\|\boldsymbol{g}_k\|}$. The dogleg method [173, 174], double-dogleg method [60], and two-dimensional subspace minimization [32] all attempt to approximately minimize (2.25) by restricting $\boldsymbol{p}$ to (a subset of) the two-dimensional subspace spanned by the steepest descent direction $-\boldsymbol{g}_k$ and the quasi-Newton step $\boldsymbol{p}_k^*$

when the model Hessian B_k is positive definite. Two-dimensional subspace minimization can also be adapted for indefinite B_k, though in that case it requires an estimate of the most negative eigenvalue of this matrix. For a comprehensive treatment of trust-region strategies, see [40].

2.4.2 The generalized nonlinear conjugate gradient method

In the nonlinear conjugate gradient method, search directions $\overset{\mathcal{C}}{\boldsymbol{p}}_k$ are generated by the recurrence relation

$$\overset{\mathcal{C}}{\boldsymbol{p}}_k = -\overset{\mathcal{C}}{\boldsymbol{g}}_k + \beta_k \overset{\mathcal{C}}{\boldsymbol{p}}_{k-1}, \tag{2.26}$$

where $\boldsymbol{g}_k$ is now defined[1] as $\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overline{\boldsymbol{z}}}$, $\boldsymbol{p}_0$ is initialized as $-\boldsymbol{g}_0$, and β_k is the conjugate gradient update parameter. Different choices for the scalar β_k correspond to different conjugate gradient methods. The generalized Hestenes–Stiefel [98] update parameter β_k^{HS} can be derived from a special case of the L-BFGS search direction, where $m = 1$, $B_{k-1}^{-1} = \mathbb{I}_{2n}$, and an exact line search, for which $\boldsymbol{g}_k^{\mathrm{H}} \boldsymbol{p}_{k-1} = 0$ for all k, is assumed. We then obtain $\overset{\mathcal{C}}{\boldsymbol{p}}_k = -B_k^{-1} \overset{\mathcal{C}}{\boldsymbol{g}}_k = -\overset{\mathcal{C}}{\boldsymbol{g}}_k + \beta_k^{\mathrm{HS}} \overset{\mathcal{C}}{\boldsymbol{p}}_{k-1}$, where

$$\beta_k^{\mathrm{HS}} = \frac{\mathrm{Re}\{(\boldsymbol{g}_k - \boldsymbol{g}_{k-1})^{\mathrm{H}} \boldsymbol{g}_k\}}{\mathrm{Re}\{(\boldsymbol{g}_k - \boldsymbol{g}_{k-1})^{\mathrm{H}} \boldsymbol{p}_{k-1}\}}. \tag{2.27a}$$

When $\boldsymbol{g}_k^{\mathrm{H}} \boldsymbol{p}_{k-1} = 0$, (2.27a) reduces to the Polak–Ribière update parameter [171]

$$\beta_k^{\mathrm{PR}} = \frac{\mathrm{Re}\{(\boldsymbol{g}_k - \boldsymbol{g}_{k-1})^{\mathrm{H}} \boldsymbol{g}_k\}}{\boldsymbol{g}_{k-1}^{\mathrm{H}} \boldsymbol{g}_{k-1}}. \tag{2.27b}$$

Further, if f is quadratic, $\boldsymbol{g}_k^{\mathrm{H}} \boldsymbol{g}_{k-1} = 0$, and we find the Fletcher–Reeves update parameter [77]

$$\beta_k^{\mathrm{FR}} = \frac{\boldsymbol{g}_k^{\mathrm{H}} \boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^{\mathrm{H}} \boldsymbol{g}_{k-1}}. \tag{2.27c}$$

[1] In Section 2.4.1 it was argued that $\boldsymbol{g}_k = 2\frac{\partial f(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overline{\boldsymbol{z}}}$ is a more practical choice for a computer implementation. Using the latter definition throughout this section is also possible, although the generated steps would be twice as large since it is then implicitly assumed that $B_{k-1}^{-1} = 2\mathbb{I}_{2n}$. One way to take this extra scaling factor into account is by scaling the initial line search step length appropriately. However, as it is inherent to the conjugate gradient method that the search directions it generates are often poorly scaled, the extra factor two can safely be ignored depending on the strategy chosen for the initial step length [159].

Powell [175] showed that the Fletcher–Reeves method is susceptible to jamming. That is, the algorithm could take many short steps without making significant progress to the minimum. The Hestenes–Stiefel and Polak–Ribière methods, which share the common numerator $\mathrm{Re}\{(\boldsymbol{g}_k - \boldsymbol{g}_{k-1})^{\mathrm{H}}\boldsymbol{g}_k\}$, possess a built-in restart feature that addresses the jamming problem [94]. When the step $\boldsymbol{z}_k - \boldsymbol{z}_{k-1}$ is small, the factor $\boldsymbol{g}_k - \boldsymbol{g}_{k-1}$ tends to zero. Hence, β_k becomes small and the new search direction $\boldsymbol{p}_k$ is essentially the steepest descent direction $-\boldsymbol{g}_k$.

With an exact line search, the above conjugate gradient methods are all globally convergent. Gilbert and Nocedal [79] proved that the modified Polak–Ribière method [176], where $\beta_k^{\mathrm{PR}+} = \max\{\beta_k^{\mathrm{PR}}, 0\}$, is globally convergent even when an inexact line search satisfying the Wolfe conditions is used. Similarly, it can also be shown [94] that the modified Hestenes–Stiefel method, where $\beta_k^{\mathrm{HS}+} = \max\{\beta_k^{\mathrm{HS}}, 0\}$, is also globally convergent when using an inexact line search.

2.5 Nonlinear least squares

Now we consider the special case where the objective is a nonlinear least squares problem of the form (2.2). The space spanned by $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ is equal to $\mathcal{C}$, which is equivalent to $\mathcal{R}$ under the linear map (2.3). It is then understood that the function F can also be regarded as a function that maps $\mathcal{C}$ or $\mathcal{R}$ to $\mathbb{C}^m$, so that both the first-order model (2.9) in $\overset{\mathcal{R}}{\boldsymbol{z}}$ and the first-order model (2.10) in $\overset{\mathcal{C}}{\boldsymbol{z}}$ are applicable.

2.5.1 The generalized Gauss–Newton and Levenberg–Marquardt methods

The generalized Gauss–Newton and Levenberg–Marquardt methods use the first-order model

$$m_k^F(\overset{\mathcal{C}}{\boldsymbol{p}}) = F(\overset{\mathcal{C}}{\boldsymbol{z}}_k) + \frac{\partial F(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}}\overset{\mathcal{C}}{\boldsymbol{p}} \tag{2.28}$$

to approximate F at the current iterate $\overset{\mathcal{C}}{\boldsymbol{z}}_k$. The objective function $f(\overset{\mathcal{C}}{\boldsymbol{z}}) := \frac{1}{2}\|F(\overset{\mathcal{C}}{\boldsymbol{z}})\|^2$ can then be approximated by

$$m_k^f(\overset{\mathcal{C}}{\boldsymbol{p}}) = \frac{1}{2}\|m_k^F(\overset{\mathcal{C}}{\boldsymbol{p}})\|^2 + \frac{\lambda_k}{2}\|\boldsymbol{p}\|^2, \tag{2.29}$$

where λ_k is the Levenberg–Marquardt regularization parameter which influences both the length and direction of the step $\boldsymbol{p}$ that minimizes m_k^f. In the Gauss–Newton method, $\lambda_k = 0$ for all k, and a trust-region framework can instead be used to control the length and direction of the step. Let $\boldsymbol{F}_k = F(\overset{\mathcal{C}}{\boldsymbol{z}}_k)$ and let $J_k = \frac{\partial F(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}}$. By substituting (2.28) in (2.29), we find

$$m_k^f(\overset{\mathcal{C}}{\boldsymbol{p}}) = \frac{1}{2}\|\boldsymbol{F}_k\|^2 + \frac{1}{2}\overset{\mathcal{C}}{\boldsymbol{p}}^{\mathrm{T}} \begin{bmatrix} J_k S \\ \overline{J}_k \end{bmatrix}^{\mathrm{H}} \begin{bmatrix} \boldsymbol{F}_k \\ \overline{\boldsymbol{F}}_k \end{bmatrix} + \frac{1}{2}\overset{\mathcal{C}}{\boldsymbol{p}}^{\mathrm{H}} \left(J_k^{\mathrm{H}} J_k + \frac{\lambda_k}{2}\mathbb{I}_{2n} \right) \overset{\mathcal{C}}{\boldsymbol{p}}. \tag{2.30}$$

Comparing (2.30) to (2.12b) reveals that the complex gradient of f is given by $\frac{1}{2}(SJ_k^{\mathrm{H}}\boldsymbol{F}_k + J_k^{\mathrm{T}}\overline{\boldsymbol{F}}_k)$ and its Hessian is approximated by $J_k^{\mathrm{H}}J_k$. The conjugate complex gradient of m_k^f is given by

$$\begin{aligned} \frac{\partial m_k^f(\overset{\mathcal{C}}{\boldsymbol{p}})}{\partial \overset{\mathcal{C}}{\overline{\boldsymbol{p}}}} &= \frac{1}{2}\left(SJ_k^{\mathrm{T}}\overline{\boldsymbol{F}}_k + J_k^{\mathrm{H}}\boldsymbol{F}_k\right) + \frac{1}{2}\left(J_k^{\mathrm{H}}J_k S + SJ_k^{\mathrm{T}}\overline{J}_k + \lambda_k S\right)\overset{\mathcal{C}}{\boldsymbol{p}} \\ &= \frac{1}{2}\begin{bmatrix} J_k \\ \overline{J}_k S \end{bmatrix}^{\mathrm{H}} \begin{bmatrix} \boldsymbol{F}_k \\ \overline{\boldsymbol{F}}_k \end{bmatrix} + \frac{1}{2}\left(\begin{bmatrix} J_k \\ \overline{J}_k S \end{bmatrix}^{\mathrm{H}} \begin{bmatrix} J_k \\ \overline{J}_k S \end{bmatrix} + \lambda_k \mathbb{I}_{2n} \right) \overset{\mathcal{C}}{\boldsymbol{p}}. \end{aligned}$$

Since f is real-valued, its conjugate complex gradient must be zero when its complex gradient is zero. The minimizer of the convex quadratic model (2.29) is then

$$\overset{\mathcal{C}}{\boldsymbol{p}}_k^* = -\begin{bmatrix} J_k \\ \overline{J}_k S \\ \sqrt{\lambda_k}\mathbb{I}_{2n} \end{bmatrix}^{\dagger} \begin{bmatrix} \boldsymbol{F}_k \\ \overline{\boldsymbol{F}}_k \\ 0_{2n\times 1} \end{bmatrix}. \tag{2.31}$$

Let $A_k = \frac{\partial F(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and let $B_k = \frac{\partial F(\overset{\mathcal{C}}{\boldsymbol{z}}_k)}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$. A more efficient way to compute the minimizer (2.31) is

$$\overset{\mathcal{R}}{\boldsymbol{p}}_k^* = -\left[\begin{array}{cc} \mathrm{Re}\{A_k\} + \mathrm{Re}\{B_k\} & \mathrm{Im}\{B_k\} - \mathrm{Im}\{A_k\} \\ \mathrm{Im}\{A_k\} + \mathrm{Im}\{B_k\} & \mathrm{Re}\{A_k\} - \mathrm{Re}\{B_k\} \\ \hline \multicolumn{2}{c}{\sqrt{\lambda_k}\mathbb{I}_{2n}} \end{array}\right]^{\dagger} \begin{bmatrix} \mathrm{Re}\{\boldsymbol{F}_k\} \\ \mathrm{Im}\{\boldsymbol{F}_k\} \\ 0_{2n\times 1} \end{bmatrix}. \tag{2.32}$$

There are a number of cases in which additional structure in F can be exploited to simplify (2.32) further:

- F is analytic in $\boldsymbol{z} \in \mathbb{C}^n$, then $B_k \equiv 0_{m\times n}$:

$$\overset{\mathcal{R}}{\boldsymbol{p}}_k^* = -\left[\begin{array}{cc} \mathrm{Re}\{A_k\} & -\mathrm{Im}\{A_k\} \\ \mathrm{Im}\{A_k\} & \mathrm{Re}\{A_k\} \\ \hline \multicolumn{2}{c}{\sqrt{\lambda_k}\mathbb{I}_{2n}} \end{array}\right]^{\dagger} \begin{bmatrix} \mathrm{Re}\{\boldsymbol{F}_k\} \\ \mathrm{Im}\{\boldsymbol{F}_k\} \\ 0_{2n\times 1} \end{bmatrix} \text{ or, equivalently,}$$

$$\boldsymbol{p}_k^* = -\begin{bmatrix} A_k \\ \sqrt{\lambda_k}\mathbb{I}_n \end{bmatrix}^{\dagger} \begin{bmatrix} \boldsymbol{F}_k \\ 0_{n\times 1} \end{bmatrix}. \tag{2.33}$$

- F is analytic in $\boldsymbol{x} \in \mathbb{R}^n$, then for $J_k := \frac{\partial F(\boldsymbol{x}_k)}{\partial \boldsymbol{x}^{\mathrm{T}}}$:

$$\boldsymbol{p}_k^* = -\begin{bmatrix} \mathrm{Re}\{J_k\} \\ \mathrm{Im}\{J_k\} \\ \hline \sqrt{\lambda_k}\mathbb{I}_n \end{bmatrix}^{\dagger} \begin{bmatrix} \mathrm{Re}\{\boldsymbol{F}_k\} \\ \mathrm{Im}\{\boldsymbol{F}_k\} \\ 0_{n\times 1} \end{bmatrix}. \tag{2.34}$$

- $F : \mathbb{C}^n \to \mathbb{R}^m$, then $B_k \equiv \overline{A}_k$:

$$\overset{\mathcal{R}}{\boldsymbol{p}}_k^* = -\left[\begin{array}{cc} 2\mathrm{Re}\{A_k\} & -2\mathrm{Im}\{A_k\} \\ \hline \multicolumn{2}{c}{\sqrt{\lambda_k}\mathbb{I}_{2n}} \end{array}\right]^{\dagger} \begin{bmatrix} \boldsymbol{F}_k \\ 0_{2n\times 1} \end{bmatrix}. \tag{2.35}$$

The least squares system (2.32) is equivalent to the system we would have arrived at had we applied the real Gauss–Newton or Levenberg–Marquardt methods, which is the de facto course of action for complex nonlinear least squares problems. However, expression (2.32) offers an alternative way of computing the real Jacobian using the complex Jacobian and conjugate Jacobian, which are often easier to derive and lead to more elegant expressions than is the case for the real Jacobian, as we demonstrate in the second case study. Moreover, we will also show that the real Jacobian matrix-vector product can be expressed in terms of matrix-vector products with the complex Jacobians. In this way, (preconditioned) conjugate gradient type methods that implicitly solve the normal equations can take advantage of any structure present in the complex Jacobians.

2.6 Case studies

2.6.1 The canonical polyadic decomposition

A tensor may be considered as an element of the tensor product of a set of vector spaces, although for this discussion their representation as a multiway array, given a choice of bases, suffices. Before we introduce the canonical polyadic decomposition, we review a few necessary definitions.

Definition 2.3. *The* inner product $\langle \mathcal{T}, \mathcal{U} \rangle$ *of two tensors* $\mathcal{T}, \mathcal{U} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ *is defined as* $\langle \mathcal{T}, \mathcal{U} \rangle = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} \overline{t}_{i_1 \cdots i_N} u_{i_1 \cdots i_N}$.

Definition 2.4. *The* outer product $\mathcal{T} \circ \mathcal{U}$ *of a tensor* $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_P}$ *and a tensor* $\mathcal{U} \in \mathbb{C}^{J_1 \times \cdots \times J_Q}$ *is the tensor defined by* $(\mathcal{T} \circ \mathcal{U})_{i_1 \cdots i_P j_1 \cdots j_Q} = t_{i_1 \cdots i_P} u_{j_1 \cdots j_Q}$.

Definition 2.5. *An Nth order $\mathcal{T}$ is* rank-one *if it is equal to the outer product of N nonzero vectors $\boldsymbol{a}^{(n)} \in \mathbb{C}^{I_n}$, i.e., $\mathcal{T} = \boldsymbol{a}^{(1)} \circ \cdots \circ \boldsymbol{a}^{(N)}$.*

The canonical polyadic decomposition (CPD) [33, 95, 99, 100] approximates an Nth order tensor by a sum of R rank-one tensors (cf. Figure 2.1), i.e.,

$$\mathcal{T} \approx \sum_{r=1}^{R} \boldsymbol{a}_r^{(1)} \circ \cdots \circ \boldsymbol{a}_r^{(N)}. \tag{2.36}$$

The rank of a tensor, denoted $\operatorname{rank}(\mathcal{T})$, is defined as the minimal number of rank-one terms such that (2.36) is exact. Such a decomposition is called a rank decomposition. For any CPD, the order of the different rank-one terms may be arbitrarily permuted, and the components of the rank-one terms may be arbitrarily scaled as long as their outer product remains the same. These properties are referred to as the permutation and scaling indeterminacy, respectively. When a rank decomposition is subject only to these trivial indeterminacies, it is called essentially unique. An interesting property of higher-order tensors is that their rank decompositions are often essentially unique [112], which is not the case for matrices. Next, we introduce a few operators which will allow us to formulate the CPD as an optimization problem.

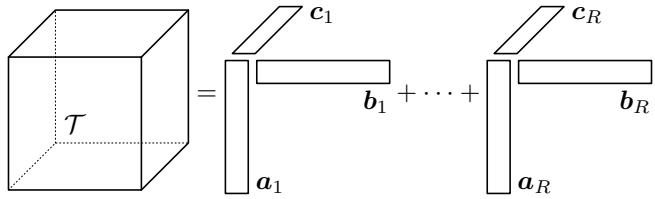


Figure 2.1: Canonical polyadic decomposition of a third-order tensor.

Definition 2.6. *In a* mode-n matricization*,* flattening*, or* unfolding $T_{(n)}$ *of an Nth order tensor $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$, tensor element with indices $(i_1, \ldots, i_N)$ is mapped to matrix element (i_n, j) such that*

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \quad \textit{with} \quad J_k := \prod_{\substack{m=0 \\ m \neq n}}^{k-1} I_m \quad \textit{and} \quad I_0 := 1.$$

In other words, the columns of the mode-n matricization $T_{(n)}$ are the mode-n vectors of $\mathcal{T}$ arranged along the natural order of the remaining modes.

Definition 2.7. *The* Kronecker product *of two matrices* $A \in \mathbb{C}^{I\times J}$ *and* $B \in \mathbb{C}^{K\times L}$ *is defined as*

$$A \otimes B = \begin{bmatrix} a_{11}B & \cdots & a_{1J}B \\ \vdots & \ddots & \vdots \\ a_{I1}B & \cdots & a_{IJ}B \end{bmatrix}.$$

Definition 2.8. *The* Khatri–Rao product [109] *of two matrices* $A \in \mathbb{C}^{I\times K}$ *and* $B \in \mathbb{C}^{J\times K}$ *is defined as* $A \odot B = \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_K \otimes \boldsymbol{b}_K \end{bmatrix}$.

Definition 2.9. *The* Hadamard product *of two matrices* $A \in \mathbb{C}^{I\times J}$ *and* $B \in \mathbb{C}^{I\times J}$ *is the matrix defined by* $(A * B)_{ij} = a_{ij}b_{ij}$.

Let $A^{(n)} = [\boldsymbol{a}_1^{(n)} \ \cdots \ \boldsymbol{a}_R^{(n)}]$ be the factor matrix corresponding to the nth mode. The rank-R CPD of an Nth order tensor $\mathcal{T}$ can be written in terms of its mode-n unfoldings as

$$T_{(n)} \approx A^{(n)}(A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots \odot A^{(1)})^{\mathrm{T}},\ n = 1, \ldots, N. \tag{2.37}$$

The CPD can now be cast as the nonlinear least squares problem

$$\underset{\substack{A^{(1)}\in\mathbb{C}^{I_1\times R}\\ \cdots \\ A^{(N)}\in\mathbb{C}^{I_N\times R}}}{\text{minimize}} \quad f_{\text{CPD}} := \frac{1}{2}\|F_{\text{CPD}}\|_{\text{F}}^2, \tag{2.38}$$

where $F_{\text{CPD}} = A^{(n)}(A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots \odot A^{(1)})^{\mathrm{T}} - T_{(n)}$. The residuals can be arranged in N different ways, depending on which mode-n unfolding is chosen. Let $V^{\{n\}} = A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots \odot A^{(1)}$. The objective function f_{CPD} can be written as the sum $\frac{1}{2}(f_{\text{CPD}}^{(1)} - f_{\text{CPD}}^{(2)} - \overline{f}_{\text{CPD}}^{(2)} + f_{\text{CPD}}^{(3)})$, where

$$f_{\text{CPD}}^{(1)} = \|A^{(n)}{V^{\{n\}}}^{\mathrm{T}}\|_{\text{F}}^2, \tag{2.39a}$$

$$f_{\text{CPD}}^{(2)} = \langle T_{(n)}, A^{(n)}{V^{\{n\}}}^{\mathrm{T}}\rangle \tag{2.39b}$$

and

$$f_{\text{CPD}}^{(3)} = \|T_{(n)}\|_{\text{F}}^2. \tag{2.39c}$$

The partial derivative of $f^{(1)}_{\text{CPD}}$ with respect to the factor matrix $A^{(n)}$ is given by

$$\begin{aligned}\frac{\partial f^{(1)}_{\text{CPD}}}{\partial A^{(n)}} &= \frac{\partial}{\partial A^{(n)}} \|A^{(n)} V^{\{n\}\mathrm{T}}\|_{\mathrm{F}}^2 \\ &= \frac{\partial}{\partial A^{(n)}} \sum_i \left(A^{(n)} \boldsymbol{v}_{i:}^{\{n\}\mathrm{T}}\right)^{\mathrm{H}} \left(A^{(n)} \boldsymbol{v}_{i:}^{\{n\}\mathrm{T}}\right) \\ &= \sum_i \overline{A}^{(n)} \boldsymbol{v}_{i:}^{\{n\}\mathrm{H}} \boldsymbol{v}_{i:}^{\{n\}} \\ &= \overline{A}^{(n)} V^{\{n\}\mathrm{H}} V^{\{n\}}.\end{aligned}$$

Similarly, it is not hard to show that $\frac{\partial f^{(2)}_{\text{CPD}}}{\partial A^{(n)}} = \overline{T}_{(n)} V^{\{n\}}$, $\frac{\partial \overline{f}^{(2)}_{\text{CPD}}}{\partial A^{(n)}} = 0$, and $\frac{\partial f^{(3)}_{\text{CPD}}}{\partial A^{(n)}} = 0$. Let

$$W^{\sigma} = \ast_{\substack{n=1 \\ n \notin \sigma}}^{N} A^{(n)\mathrm{H}} A^{(n)}$$

be the Hadamard product of all matrices $A^{(n)\mathrm{H}} A^{(n)}$, $n \in \{1, \ldots, N\} \backslash \sigma$. Then a corollary of Proposition 2.1 is that $V^{\{n\}\mathrm{H}} V^{\{n\}}$ can be evaluated more efficiently as $W^{\{n\}}$ without explicitly forming any Khatri–Rao products [28, 180].

Proposition 2.1. *Let $A^{(n)} \in \mathbb{C}^{I_n \times J}$, $n = 1, \ldots, N$, let $V = A^{(p_1)} \odot \cdots \odot A^{(p_N)}$ where $\boldsymbol{p}$ is any permutation of $\{1, \ldots, N\}$, and let $W = \ast_{n=1}^{N} A^{(n)\mathrm{H}} A^{(n)}$ be the Hadamard product of all matrices $A^{(n)\mathrm{H}} A^{(n)}$. Then $V^{\mathrm{H}} V = W$.*

Proof.

$$\begin{aligned}(V^{\mathrm{H}} V)_{nm} &= (\boldsymbol{a}_n^{(p_1)} \otimes \cdots \otimes \boldsymbol{a}_n^{(p_N)})^{\mathrm{H}} (\boldsymbol{a}_m^{(p_1)} \otimes \cdots \otimes \boldsymbol{a}_m^{(p_N)}) \\ &= \langle \boldsymbol{a}_n^{(1)} \circ \cdots \circ \boldsymbol{a}_n^{(N)}, \boldsymbol{a}_m^{(1)} \circ \cdots \circ \boldsymbol{a}_m^{(N)} \rangle \\ &= (\boldsymbol{a}_n^{(1)\mathrm{H}} \boldsymbol{a}_m^{(1)}) \cdots (\boldsymbol{a}_n^{(N)\mathrm{H}} \boldsymbol{a}_m^{(N)}) = w_{nm}.\end{aligned}$$

□

Let $\boldsymbol{z} = \big(\mathrm{vec}(A^{(1)}), \ldots, \mathrm{vec}(A^{(N)})\big) \in \mathbb{C}^{R \sum_n^N I_n}$, where $\mathrm{vec}(\cdot)$ denotes the column-wise vectorization of a matrix. The cogradient of f_{CPD} completely

determines the complex gradient of f_{CPD} and is given by

$$\frac{\partial f_{\text{CPD}}}{\partial \boldsymbol{z}} = \frac{1}{2} \begin{bmatrix} \operatorname{vec}\left(\overline{A}^{(1)} W^{\{1\}} - \overline{T}_{(1)} V^{\{1\}}\right) \\ \vdots \\ \operatorname{vec}\left(\overline{A}^{(N)} W^{\{N\}} - \overline{T}_{(N)} V^{\{N\}}\right) \end{bmatrix}. \tag{2.40}$$

As outlined in Section 2.4, it suffices to implement the function f_{CPD} and its associated scaled conjugate cogradient $\boldsymbol{g}_{\text{CPD}} := 2\overline{\frac{\partial f_{\text{CPD}}}{\partial \boldsymbol{z}}}$ to be able to apply any of the generalized gradient based optimization methods discussed in that section. Although we did not explicitly derive an expression for the real gradient, it is not hard to imagine that this derivation and resulting expression would be relatively more complicated compared to that of the complex gradient.

To apply either the generalized Gauss–Newton or Levenberg–Marquardt method, we define J as the Jacobian $\frac{\partial \operatorname{vec}(F_{\text{CPD}})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ of the nonlinear least squares problem (2.38). Since the conjugate Jacobian is identically equal to zero, F_{CPD} is analytic in $\boldsymbol{z}$ and we may use (2.33) to compute the Levenberg–Marquardt step $\boldsymbol{p}_k^*$. However, the number of rows in J is equal to $\prod_n^N I_n$ and is potentially very large compared to the number of columns $R\sum_n^N I_n$. Solving the normal equations $J_k^{\mathrm{H}} J_k \boldsymbol{p}_k = -J_k^{\mathrm{H}} \operatorname{vec}(F_{\text{CPD}}(\boldsymbol{z}_k)) = -\boldsymbol{g}_{\text{CPD}}(\boldsymbol{z}_k)$ is one way to reduce the computational complexity, although at the cost of squaring the condition number of the system. It can be shown that $J^{\mathrm{H}}J$ is an $N \times N$ block matrix, in which the (n, m)th block is the block matrix

$$\left(J^{\mathrm{H}}J\right)_{nm} = \begin{cases} W^{\{n\}} \otimes \mathbb{I}_{I_n} & \text{for } n = m, \\ \begin{bmatrix} w_{1,1}^{\{n,m\}} \boldsymbol{a}_1^{(n)} \boldsymbol{a}_1^{(m)\mathrm{H}} & \cdots & w_{1,R}^{\{n,m\}} \boldsymbol{a}_R^{(n)} \boldsymbol{a}_1^{(m)\mathrm{H}} \\ \vdots & \ddots & \vdots \\ w_{R,1}^{\{n,m\}} \boldsymbol{a}_1^{(n)} \boldsymbol{a}_R^{(m)\mathrm{H}} & \cdots & w_{R,R}^{\{n,m\}} \boldsymbol{a}_R^{(n)} \boldsymbol{a}_R^{(m)\mathrm{H}} \end{bmatrix} & \text{for } n \neq m. \end{cases} \tag{2.41}$$

Inverting $J^{\mathrm{H}}J$ costs $O((R\sum_{n=1}^N I_n)^3)$ flop unless its structure is taken into account. A straightforward way to do so is by solving the normal equations iteratively with a preconditioned conjugate gradient method. Let $Y \in \mathbb{C}^{I_m \times R}$ and let $\boldsymbol{y} = \operatorname{vec}(Y)$, then it can be shown [208] that

$$\left(J^{\mathrm{H}}J\right)_{nm} \boldsymbol{y} = \begin{cases} \operatorname{vec}\left(Y\overline{W}^{\{n\}}\right) & \text{for } n = m, \\ \operatorname{vec}\left(A^{(n)}\left(\overline{W}^{\{n,m\}} * \left(Y^{\mathrm{T}}\overline{A}^{(m)}\right)\right)\right) & \text{for } n \neq m. \end{cases} \tag{2.42}$$

The matrix-vector product $J^{\mathrm{H}}J\boldsymbol{z}$ costs only $O(R^2\sum_{n=1}^N I_n)$ flop using (2.42), which means that inverting $J^H J$ can be done in at most $O(R^3(\sum_{n=1}^N I_n)^2)$ flop.

When $I_n \equiv I$, this is a reduction from $O(N^3R^3I^3)$ flop to $O(N^2R^3I^2)$ flop. In practice an approximate solution can often be obtained in fewer than $R\sum_{n=1}^{N} I_n$ iterations with the help of the block-Jacobi preconditioner P, wherein the nth diagonal block is defined as $W^{\{n\}} \otimes \mathbb{I}_{I_n}$. From (2.42) we see that inverting P requires solving N square Hermitian linear systems of order R, for a total cost of $O(NR^3)$ flop. Numerical experiments comparing the above algorithms and a more detailed treatment of the CPD and its generalizations are presented in [208].

2.6.2 Simulation of nonlinear circuits in the frequency domain

Simulation in the frequency domain avoids many of the problems experienced when trying to use traditional time domain simulators such as SPICE [149,178] to find the steady-state behavior of analog, radio frequency, and microwave circuits. In principle, the periodic response can always be found by integrating the differential algebraic equations that describe the system until the transient responses vanish. However, this approach can be prohibitively expensive whenever the transients are governed by large time constants. The harmonic balance (HB) method [151,222] transforms these differential algebraic equations to a system of nonlinear algebraic equations in the frequency domain by representing the circuit variables by means of a truncated Fourier series of H harmonics. The method computes the steady state solution directly, but is somewhat limited to systems where the latter is approximated well by periodic and quasiperiodic solutions.

Consider a nonlinear circuit of $N + 1$ nodes consisting of voltage-controlled current sources. Kirchoff's current law applied in the frequency domain to all but one of the nodes constitutes the set of nonlinear algebraic equations

$$I(\omega, V(\omega), \overline{V(\omega)}) = 0_{N\times 1}, \tag{2.43}$$

where $I(\omega, V(\omega), \overline{V(\omega)})$ describes the total current through each node as a function of the angular frequency ω and the node voltages $V(\omega) \in \mathbb{C}^N$. One of the nodes is selected as a reference node. The node voltages are then referenced with respect to the reference node. If the circuit were linear, (2.43) would be a parametrized linear system $Y(\omega)V(\omega) + I_s(\omega) = 0$, where $Y(\omega) \in \mathbb{C}^{N\times N}$ is a node admittance matrix and $I_s(\omega) \in \mathbb{C}^N$ represents a vector of current sources. In the HB method, the time domain voltage $v_n(t)$ of the nth node is represented by the complex coefficients $V_h^{(n)}$, $h = 0, \ldots, H-1$, in the frequency domain,

i.e.,

$$v_n(t) \approx V_0^{(n)} + \sum_{h=1}^{H-1} \left(V_h^{(n)} e^{j2\pi f_0 ht} + \overline{V}_h^{(n)} e^{-j2\pi f_0 ht} \right) \tag{2.44a}$$

where

$$V_h^{(n)} = f_0 \int_0^{\frac{1}{f_0}} v_n(t) e^{-j2\pi f_0 ht} \mathrm{d}t. \tag{2.44b}$$

Analogously, the time domain current $i_n(t, v(t))$ through the nth node is a function of all node voltages $v(t) := \{v_1(t), \ldots, v_N(t)\}$ and is represented by the complex coefficients $I_h^{(n)}$, $h = 0, \ldots, H-1$, in the frequency domain.

Let $\boldsymbol{V}^{(n)} = (V_0^{(n)}, \ldots, V_{H-1}^{(n)})$ and let $\boldsymbol{I}^{(n)} = (I_0^{(n)}, \ldots, I_{H-1}^{(n)})$. The steady-state behaviour of a nonlinear circuit can then be approximated as the solution of the nonlinear least squares problem

$$\begin{array}{ll} \underset{\substack{\boldsymbol{V}^{(1)} \in \mathbb{C}^H \\ \cdots \\ \boldsymbol{V}^{(N)} \in \mathbb{C}^H}}{\text{minimize}} & f_{\text{HBM}} := \dfrac{1}{2} \| F_{\text{HBM}} \|^2, \end{array} \tag{2.45}$$

where $F_{\text{HBM}} = (\boldsymbol{I}^{(1)}, \ldots, \boldsymbol{I}^{(N)})$. At this point the optimization problem (2.45) is usually converted to the real domain so that real optimization methods can be applied. However, the derivatives of F_{HBM} with respect to the real and imaginary parts of $V_h^{(n)}$ obscure the structure in the real Jacobian. In comparison, the expressions for the complex (conjugate) Jacobian required by the generalized Gauss–Newton and Levenberg–Marquardt methods are much more elegant. As we will see, their structure is easy to recognize and in this case also easy to exploit. The derivative of the hth harmonic of the current through the nth node $I_h^{(n)}$ with respect to the kth harmonic of the mth node voltage $V_k^{(m)}$ is

$$\begin{aligned} \frac{\partial I_h^{(n)}}{\partial V_k^{(m)}} &= f_0 \int_0^{\frac{1}{f_0}} \frac{\partial i_n(t, v(t))}{\partial v_m} \frac{\partial v_m(t)}{\partial V_k^{(m)}} e^{-j2\pi f_0 ht} \mathrm{d}t \\ &= f_0 \int_0^{\frac{1}{f_0}} \frac{\partial i_n(t, v(t))}{\partial v_m} e^{-j2\pi f_0 (h-k)t} \mathrm{d}t =: G_{h-k}^{(n,m)}, \end{aligned} \tag{2.46a}$$

which is just the $(h-k)$th Fourier series coefficient of the derivative $\frac{\partial i_n(t,v(t))}{\partial v_m}$. Analogously, we have that

$$\frac{\partial I_h^{(n)}}{\partial \overline{V}_k^{(m)}} = f_0 \int_0^{\frac{1}{f_0}} \frac{\partial i_n(t,v(t))}{\partial v_m} e^{-j2\pi f_0 (h+k)t} \mathrm{d}t = G_{h+k}^{(n,m)}. \tag{2.46b}$$

Further, because $\frac{\partial i_n(t,v(t))}{\partial v_m}$ is real, $G_h^{(n,m)} = \overline{G}_{-h}^{(n,m)}$ for any integer h. Let $\boldsymbol{z} = (\boldsymbol{V}^{(1)}, \ldots, \boldsymbol{V}^{(N)})$; then the (n,m)th block of the Jacobian $\frac{\partial F_{\mathrm{HBM}}}{\partial \boldsymbol{z}^{\mathrm{T}}}$ is the Hermitian Toeplitz matrix

$$\left(\frac{\partial F_{\mathrm{HBM}}}{\partial \boldsymbol{z}^{\mathrm{T}}}\right)_{nm} = \frac{\partial \boldsymbol{I}^{(n)}}{\partial \boldsymbol{V}^{(m)\,\mathrm{T}}} = \begin{bmatrix} G_0^{(n,m)} & \overline{G}_1^{(n,m)} & \overline{G}_2^{(n,m)} & \cdots & \overline{G}_{H-1}^{(n,m)} \\ G_1^{(n,m)} & G_0^{(n,m)} & \overline{G}_1^{(n,m)} & \cdots & \overline{G}_{H-2}^{(n,m)} \\ G_2^{(n,m)} & G_1^{(n,m)} & G_0^{(n,m)} & \cdots & \overline{G}_{H-3}^{(n,m)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ G_{H-1}^{(n,m)} & G_{H-2}^{(n,m)} & G_{H-3}^{(n,m)} & \cdots & G_0^{(n,m)} \end{bmatrix}, \tag{2.47a}$$

and the (n,m)th block of the conjugate Jacobian $\frac{\partial F_{\mathrm{HBM}}}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$ is a Hankel matrix preceded by a zero-column, given by

$$\left(\frac{\partial F_{\mathrm{HBM}}}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}\right)_{nm} = \frac{\partial \boldsymbol{I}^{(n)}}{\partial \overline{\boldsymbol{V}}^{(m)\,\mathrm{T}}} = \begin{bmatrix} 0 & G_1^{(n,m)} & G_2^{(n,m)} & \cdots & G_{H-1}^{(n,m)} \\ 0 & G_2^{(n,m)} & G_3^{(n,m)} & \cdots & G_H^{(n,m)} \\ 0 & G_3^{(n,m)} & G_4^{(n,m)} & \cdots & G_{H+1}^{(n,m)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & G_H^{(n,m)} & G_{H+1}^{(n,m)} & \cdots & G_{2H-2}^{(n,m)} \end{bmatrix}. \tag{2.47b}$$

For each such block, the coefficients $G_h^{(n,m)}$, $h = 0, \ldots, 2H-2$, can be approximated in $O(H \log H)$ flop using the fast Fourier transform. Assuming the average degree of the nodes in the graph associated with the circuit is constant with respect to the size of the graph, the cost of computing the (conjugate) Jacobian is $O(NH \log H)$ flop, and can be stored compactly using $O(NH)$ complex numbers. It is well known that the product of a Toeplitz or Hankel matrix of order H with a vector can be done in $O(H \log H)$ operations [17, 106]. This property can be exploited when solving the Gauss–Newton or Levenberg–Marquardt step (2.32) iteratively. Let A_k and B_k be the Jacobian and conjugate

Jacobian, respectively, and let

$$C_k = \begin{bmatrix} \operatorname{Re}\{A_k\} + \operatorname{Re}\{B_k\} & \operatorname{Im}\{B_k\} - \operatorname{Im}\{A_k\} \\ \operatorname{Im}\{A_k\} + \operatorname{Im}\{B_k\} & \operatorname{Re}\{A_k\} - \operatorname{Re}\{B_k\} \\ \hline \multicolumn{2}{c}{\sqrt{\lambda_k}\mathbb{I}_{2n}} \end{bmatrix}$$

be the augmented real Jacobian, then it is not hard to show that

$$C_k \overset{\mathcal{R}}{\boldsymbol{s}} = \begin{bmatrix} \operatorname{Re}\{A_k \boldsymbol{s}\} + \operatorname{Re}\{B_k \overline{\boldsymbol{s}}\} \\ \operatorname{Im}\{A_k \boldsymbol{s}\} + \operatorname{Im}\{B_k \overline{\boldsymbol{s}}\} \\ \sqrt{\lambda_k}\operatorname{Re}\{\boldsymbol{s}\} \\ \sqrt{\lambda_k}\operatorname{Im}\{\boldsymbol{s}\} \end{bmatrix} \tag{2.48a}$$

and

$$C_k^{\mathrm{T}} \begin{bmatrix} \overset{\mathcal{R}}{\boldsymbol{t}} \\ \overset{\mathcal{R}}{\boldsymbol{u}} \end{bmatrix} = \begin{bmatrix} \operatorname{Re}\{A_k^{\mathrm{H}} \boldsymbol{t}\} + \operatorname{Re}\{B_k^{\mathrm{H}} \boldsymbol{t}\} + \sqrt{\lambda_k}\operatorname{Re}\{\boldsymbol{u}\} \\ \operatorname{Im}\{A_k^{\mathrm{H}} \boldsymbol{t}\} - \operatorname{Im}\{B_k^{\mathrm{H}} \boldsymbol{t}\} + \sqrt{\lambda_k}\operatorname{Im}\{\boldsymbol{u}\} \end{bmatrix}. \tag{2.48b}$$

The real matrix-vector products (2.48) depend only on the complex block-Toeplitz and block-Hankel matrix-vector products $A_k \boldsymbol{s}$, $B_k \overline{\boldsymbol{s}}$, $A_k^{\mathrm{H}} \boldsymbol{t}$ and $B_k^{\mathrm{H}} \boldsymbol{t}$, each of which can be computed in $O(NH \log H)$ flop. Iterative methods that implicitly solve the normal equations, such as CGLS or LSQR [98,165], can then be applied to the real system (2.32) using the transformations (2.48). The total cost of computing the Gauss–Newton or Levenberg–Marquardt step is then at most $O(N^2 H^2 \log H)$ flop, with a memory cost of $O(NH)$ complex numbers.

If the Levenberg–Marquardt algorithm were applied in the real domain, the derivation of the first-order derivatives would not only be more involved, it would also be harder to notice and exploit their underlying structure [184]. A naive implementation would require $O(N^2 H^3)$ flop and $O(NH^2)$ memory cells to compute the step. In comparison, relatively many more harmonics can be used to represent the steady-state solution in the generalized nonlinear least squares method, so that even simulation of mildly to highly nonlinear circuits is within reach.

2.7 Conclusion

Functions of complex variables have several complex Taylor series expansions. The complex derivatives underlying these expansions avoid the structural overhead of dealing with derivatives of the real and imaginary parts separately. We used this approach to generalize the L-BFGS and nonlinear conjugate

gradient methods to functions of complex variables. We showed that these methods need only depend on the objective function and its scaled conjugate cogradient. The latter conveniently coincides with the real gradient when optimizing over the real domain. The computational and storage cost of these generalized methods are equal to those of their real counterparts applied on the optimization problem formulated in the real domain. We also generalized Gauss–Newton and Levenberg–Marquardt methods for solving nonlinear least squares problems. We showed that the real Jacobian arising in the equivalent nonlinear least squares problem formulated in the real domain is in fact composed of a combination of two complex Jacobians. In the case studies we demonstrated that the complex gradients and complex Jacobians are not only often easier to derive than their real counterparts, but are also described by more compact expressions. Additionally, we showed that the real Jacobian matrix-vector product can be expressed in terms of complex Jacobian matrix-vector products. This allows the structure of the complex Jacobians to be exploited when applying iterative methods for solving the normal equations associated with the Gauss–Newton or Levenberg–Marquardt step. The algorithms discussed in this chapter, including several line search and trust-region variants and methods for numerical complex differentiation, have been implemented in the complex optimization toolbox (COT) for MATLAB [207].

Chapter 3

Optimization-based algorithms for tensor decompositions

The canonical polyadic and rank-$(L_r, L_r, 1)$ block term decomposition (CPD and BTD, respectively) are two closely related tensor decompositions. The CPD and, recently, BTD are important tools in psychometrics, chemometrics, neuroscience and signal processing. We present a decomposition that generalizes these two and develop algorithms for its computation. Among these algorithms are alternating least squares schemes, several general unconstrained optimization techniques, and matrix-free nonlinear least squares methods. In the latter we exploit the structure of the Jacobian's Gramian to reduce computational and memory cost. Combined with an effective preconditioner, numerical experiments confirm that these methods are among the most efficient and robust currently available for computing the CPD, rank-$(L_r, L_r, 1)$ BTD, and their generalized decomposition. This chapter is a slightly adapted version of the article [208].

3.1 Introduction

Tensor decompositions are important techniques for data mining, dimensionality reduction, pattern recognition, object detection, classification, clustering, and blind source separation [16, 36, 38, 49, 52, 195, 197]. For an introduction to tensor decompositions and their applications we refer to the review chapter and books [112, 117, 200]. The two main tensor generalizations of the singular value decomposition (SVD) are, on one hand, the Tucker decomposition or multilinear SVD (MLSVD) [54, 231, 232] and, on the other hand, the canonical polyadic decomposition (CPD) [33, 95]. In fact, the CPD generalizes any rank-revealing matrix decomposition. The MLSVD and CPD are connected with two different tensor generalizations of the concept of matrix rank. The former is linked with the set of mode-n ranks, which generalize column rank, row rank, etc. The latter generalizes rank in the sense of the minimal number of rank-one terms whose sum is equal to a given tensor. Block term decompositions (BTD) were introduced by De Lathauwer [46, 47, 55] as a framework that unifies the MLSVD and CPD. Of particular interest is the rank-$(L_r, L_r, 1)$ BTD, which has recently proved to be useful in blind source separation [48, 50], and particularly in telecommunication applications [53, 156, 196].

In this chapter, we propose several optimization-based algorithms to compute the CPD and rank-$(L_r, L_r, 1)$ BTD. In fact, we first introduce a more general decomposition called the (rank-$L_r \circ$ rank-1) BTD. Whereas the rank-$(L_r, L_r, 1)$ BTD is a generalization of the third-order CPD, the (rank-$L_r \circ$ rank-1) BTD is a generalization of both the Nth-order CPD and the rank-$(L_r, L_r, 1)$ BTD. Consequently, algorithms designed for the more general decomposition are also applicable to the former decompositions. We develop alternating least squares (ALS) schemes, memory-efficient gradient-based methods such as nonlinear conjugate gradient and limited-memory BFGS using both line search and trust-region frameworks, and nonlinear least squares methods such as Gauss–Newton and Levenberg–Marquardt. For the latter, we derive a matrix-free implementation that exploits the structure inherent to the decomposition, reducing computational complexity significantly compared to the exact method and reducing memory cost to that of ALS. Throughout the chapter, we consider the general case where the decompositions may be complex, although real decompositions are also supported through the choice of the initialization. Our numerical experiments reveal that ALS, despite its popularity, is in many cases not a suitable choice. Nonlinear least squares methods are less sensitive to the type of initialization and are often not only more efficient and less prone to so-called swamps but are also very robust for more difficult decompositions in which there is some degree of collinearity between factors.

The chapter is organized as follows. In Section 3.2 we review our notation and introduce some basic definitions. In Section 3.3 we recall the canonical polyadic decomposition and the rank-$(L_r, L_r, 1)$ block term decomposition and also introduce the (rank-L_r $\circ$ rank-1) block term decomposition. In Section 3.4 we derive the (co)gradient necessary for alternating least squares and gradient-based methods. We then demonstrate that the associated Gramian of the Jacobian may be intuitively expected to approximate the objective function's Hessian relatively well as the residuals decrease, depending on the tensor's order and rank. We also show that this matrix has a very specific structure, which we exploit in matrix-free nonlinear least squares algorithms. We close this section with an overview of each algorithm's computational complexity. In Section 3.5 we evaluate the performance of the proposed algorithms with Monte Carlo simulations of both exact and noisy decompositions. We conclude the chapter in Section 3.6.

3.2 Notation and preliminaries

Formally, a tensor is an element of a tensor product of vector spaces. Here, we refer to a tensor represented as a multidimensional array, given a choice of bases for each of these vector spaces. The order, or the number of modes, of a tensor is the number of indices associated with each element of that tensor. Vectors are denoted by boldface letters and are lower case, e.g., $\boldsymbol{a}$. Matrices are denoted by capital letters, e.g., A. Higher-order tensors are denoted by Euler script letters, e.g., $\mathcal{A}$. An entry of a vector $\boldsymbol{a}$, matrix A, or tensor $\mathcal{A}$ is denoted by a_i, a_{ij}, or $a_{ijk\ldots}$, depending on the number of modes. A colon is used to select all entries of a mode. For instance, $\boldsymbol{a}_{:j}$ corresponds to the jth column of a matrix A. When there is no confusion, we also use $\boldsymbol{a}_j$ to denote the jth column of the matrix A. Mode-n vectors are the generalization of matrix rows and columns to tensors. A mode-n vector is a vector in which all but one of the indices are fixed. For example, the mode-1, mode-2, and mode-3 vectors of a third-order tensor $\mathcal{T}$ are denoted by $\boldsymbol{t}_{:jk}$, $\boldsymbol{t}_{i:k}$ and $\boldsymbol{t}_{ij:}$, respectively. Sequences are denoted by a superscript in parentheses, e.g., $\{A^{(n)}\}_{n=1}^{N}$. The superscripts $\cdot^{\mathrm{T}}$, $\cdot^{\mathrm{H}}$, $\cdot^{-1}$ and $\cdot^{\dagger}$ are used for the transpose, Hermitian conjugate, matrix inverse, and Moore–Penrose pseudoinverse, respectively. The complex conjugate is denoted by an overbar, e.g., $\overline{a}$ is the complex conjugate of the scalar a. We use parentheses to denote the concatenation of two or more vectors, e.g., $(\boldsymbol{a}, \boldsymbol{b})$ is equivalent to $\begin{bmatrix}\boldsymbol{a}^{\mathrm{T}} & \boldsymbol{b}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$. The $n \times n$ identity matrix is denoted by $\mathbb{I}_n$, and the all-zero and all-one $m \times n$ matrices by $0_{m\times n}$ and $1_{m\times n}$, respectively. The Kronecker product of two matrices A and B is denoted by $A \otimes B$.

Definition 3.1. *The* inner product $\langle \mathcal{T}, \mathcal{U} \rangle$ *of two tensors* $\mathcal{T}, \mathcal{U} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ *is defined as* $\langle \mathcal{T}, \mathcal{U} \rangle = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} \bar{t}_{i_1 \cdots i_N} u_{i_1 \cdots i_N}$.

Definition 3.2. *The* Frobenius norm *is denoted by* $\|\mathcal{T}\|_{\mathrm{F}} = \sqrt{\langle \mathcal{T}, \mathcal{T} \rangle}$.

Definition 3.3. *The* outer product $\mathcal{T} \circ \mathcal{U}$ *of a tensor* $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_P}$ *and a tensor* $\mathcal{U} \in \mathbb{C}^{J_1 \times \cdots \times J_Q}$ *is the tensor defined by* $(\mathcal{T} \circ \mathcal{U})_{i_1 \cdots i_P j_1 \cdots j_Q} = t_{i_1 \cdots i_P} u_{j_1 \cdots j_Q}$.

Definition 3.4. *An* N*th-order tensor* $\mathcal{T}$ *is* rank-one *if it is equal to the outer product of* N *nonzero vectors* $\boldsymbol{a}^{(n)} \in \mathbb{C}^{I_n}$, *i.e.,* $\mathcal{T} = \boldsymbol{a}^{(1)} \circ \cdots \circ \boldsymbol{a}^{(N)}$.

Definition 3.5. *In a* vectorization $\mathrm{vec}(A)$ *of a matrix* $A \in \mathbb{C}^{I \times J}$, *matrix element* (i, j) *is mapped to vector element* $(i + (j-1)J)$.

Definition 3.6. *In a* mode-n matricization, flattening, *or* unfolding $T_{(n)}$ *of an* N*th-order tensor* $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$, *tensor element with indices* $(i_1, \ldots, i_N)$ *is mapped to matrix element* (i_n, j) *such that*

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \quad \textit{with} \quad J_k := \prod_{\substack{m=0 \\ m \neq n}}^{k-1} I_m \quad \textit{and} \quad I_0 := 1.$$

In other words, the columns of the mode-n matricization $T_{(n)}$ are the mode-n vectors of $\mathcal{T}$ arranged along the natural order of the remaining modes.

Definition 3.7. *The* mode-n rank *of an* N*th-order tensor* $\mathcal{T}$, *denoted by* $R_n = \mathrm{rank}_n(\mathcal{T})$, *is defined as the column rank of* $T_{(n)}$. *In other words, the mode-n rank is the dimension of the space spanned by the mode-n vectors of* $\mathcal{T}$. *A tensor characterized by its n-ranks* R_n, $n = 1, \ldots, N$, *is said to be of rank-*$(R_1, \ldots, R_N)$.

Definition 3.8. *The* mode-n tensor-matrix product $\mathcal{T} \bullet_n A$ *of a tensor* $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ *with a matrix* $A \in \mathbb{C}^{J \times I_n}$ *is the tensor defined by the tensor unfolding of the matrix* $(\mathcal{T} \bullet_n A)_{(n)} = A \cdot T_{(n)}$.

Definition 3.9. *The* Khatri–Rao product [109] *of two matrices* $A \in \mathbb{C}^{I \times K}$ *and* $B \in \mathbb{C}^{J \times K}$ *is defined as* $A \odot B = \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_K \otimes \boldsymbol{b}_K \end{bmatrix}$.

Definition 3.10. *The* Hadamard product *of two matrices* $A \in \mathbb{C}^{I \times J}$ *and* $B \in \mathbb{C}^{I \times J}$ *is the matrix defined by* $(A * B)_{ij} = a_{ij} b_{ij}$.

3.3 Tensor decompositions

3.3.1 The canonical polyadic decomposition

The canonical polyadic decomposition (CPD) approximates a tensor with a sum of R rank-one tensors. It was introduced by Hitchcock in 1927 [99, 100], and was later referred to as the canonical decomposition (CANDECOMP) [33] and parallel factor decomposition (PARAFAC) [95] in psychometrics and phonetics, respectively. Let $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ and $\boldsymbol{a}_r^{(n)} \in \mathbb{C}^{I_n}$, $\boldsymbol{a}_r^{(n)}$ nonzero, then

$$\mathcal{T} \approx \sum_{r=1}^{R} \boldsymbol{a}_r^{(1)} \circ \cdots \circ \boldsymbol{a}_r^{(N)} \tag{3.1}$$

is a CPD of the tensor $\mathcal{T}$ in R rank-one terms. The rank of the tensor is defined as the smallest R for which (3.1) is exact. Such a decomposition is called a rank decomposition and is, in contrast to the matrix case, often unique. Let $A^{(n)} = [\boldsymbol{a}_1^{(n)} \ \cdots \ \boldsymbol{a}_R^{(n)}]$ be the factor matrix corresponding to the nth mode. The CPD can then be written in matrix form as

$$T_{(n)} \approx A^{(n)} \cdot {V^{\{n\}}}^{\mathrm{T}}, \text{ where } V^{\sigma} := \bigodot_{\substack{n=0 \\ N-n \notin \sigma}}^{N-1} A^{(N-n)} \tag{3.2}$$

for any $1 \leq n \leq N$. Note that σ defines a set of factor matrices to exclude from the string of Khatri–Rao products. For example, $V^{\{n\}} = A^{(N)} \odot \cdots \odot A^{(n+1)} \odot A^{(n-1)} \odot \cdots \odot A^{(1)}$.

To a large extent, the practical importance of the CPD stems from its uniqueness properties. It is clear that one can arbitrarily permute the different rank-one terms. Also, the factors of a single rank-one term may be arbitrarily scaled, as long as their product remains the same. We call a CPD essentially unique when it is subject only to these trivial indeterminacies. The most well-known result on uniqueness is due to Kruskal [118,119] and depends on the concept of k-rank. The k-rank of a matrix A, denoted k_A, is defined as the maximum value k such that any k columns of A are linearly independent [118]. Kruskal's sufficient condition for the unicity of a rank decomposition, generalized to Nth-order tensors by Sidiropoulos and Bro [194], is

$$\sum_{n=1}^{N} k_{A^{(n)}} \geq 2R + (N-1). \tag{3.3}$$

This condition can be relaxed by an order of magnitude if the tensor is long in one mode [45, 104]. More recent relaxations of Kruskal's condition are available in [67, 68].

3.3.2 The rank-$(L_r, L_r, 1)$ block term decomposition

The rank-$(L_r, L_r, 1)$ block term decomposition (BTD) [46, 47, 55] approximates a third-order tensor by a sum of R terms, each of which is an outer product of a rank-L_r matrix and a nonzero vector. Let $\mathcal{T}$ be a third-order tensor, $A_r \in \mathbb{C}^{I_1 \times L_r}$ and $B_r \in \mathbb{C}^{I_2 \times L_r}$ be rank-L_r matrices and $\boldsymbol{c}_r \in \mathbb{C}^{I_3}$, $\boldsymbol{c}_r$ nonzero, then

$$\mathcal{T} \approx \sum_{r=1}^{R} (A_r \cdot B_r^{\mathrm{T}}) \circ \boldsymbol{c}_r \tag{3.4}$$

is a BTD of the tensor $\mathcal{T}$ in R rank-$(L_r, L_r, 1)$ terms (cf. Figure 3.1). In addition to the permutation and scaling indeterminacies inherited from the CPD, the factors A_r may be postmultiplied by any nonsingular matrix $F_r \in \mathbb{C}^{L_r \times L_r}$, provided B_r^{T} is premultiplied by the inverse of F_r. This decomposition is also called essentially unique when it is subject only to these trivial indeterminacies. When the matrices $\begin{bmatrix} A_1 & \cdots & A_R \end{bmatrix}$ and $\begin{bmatrix} B_1 & \cdots & B_R \end{bmatrix}$ are full column rank and the matrix $\begin{bmatrix} \boldsymbol{c}_1 & \cdots & \boldsymbol{c}_R \end{bmatrix}$ does not contain collinear columns, the decomposition is guaranteed to be essentially unique. Furthermore, the decomposition may then be computed with a generalized eigenvalue decomposition (GEVD). These GEVD-type sufficient conditions, along with Kruskal-type conditions, were derived in [47]. A new necessary and sufficient condition was derived in [48]. The CPD and rank-$(L_r, L_r, 1)$ BTD have both been applied in telecommunication applications [53, 156, 195–197]. However, the terms of a rank-$(L_r, L_r, 1)$ BTD possess a more general low-rank structure that, together with its relatively mild conditions for unicity, make it a promising new candidate for blind source separation [48, 50]. The rank-$(L_r, L_r, 1)$ BTD generalizes the CPD

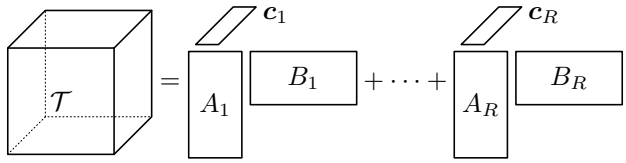


Figure 3.1: Rank-$(L_r, L_r, 1)$ block term decomposition of a third-order tensor.

for third-order tensors. We now introduce a new block term decomposition which generalizes both these decompositions. Designing algorithms for this more general decomposition requires little additional effort and will result in algorithms applicable to all of the aforementioned decompositions.

3.3.3 The (rank-L_r ∘ rank-1) block term decomposition

The (rank-L_r ∘ rank-1) block term decomposition approximates a tensor by a sum of R terms, each of which is an outer product of a rank-L_r tensor and a rank-one tensor. Let N, P and Q be positive integers so that $N = P + Q$ and let $\mathcal{T}$ be an Nth-order tensor, $R' = \sum_{r=1}^{R} L_r$, $A^{(p)} = [\boldsymbol{a}_1^{(p)} \ \cdots \ \boldsymbol{a}_{R'}^{(p)}]$ and $C^{(q)} = [\boldsymbol{c}_1^{(q)} \ \cdots \ \boldsymbol{c}_R^{(q)}]$, then

$$\mathcal{T} \approx \sum_{r=1}^{R} \left[\left(\sum_{r'=1+\sum_{u=1}^{r-1} L_u}^{\sum_{u=1}^{r} L_u} \boldsymbol{a}_{r'}^{(1)} \circ \cdots \circ \boldsymbol{a}_{r'}^{(P)} \right) \circ \left(\boldsymbol{c}_r^{(1)} \circ \cdots \circ \boldsymbol{c}_r^{(Q)} \right) \right] \tag{3.5}$$

is a BTD of the tensor $\mathcal{T}$ in R (rank-L_r ∘ rank-one) terms (cf. Figure 3.2). By setting $L_r \equiv 1$, it is obvious that this decomposition reduces to the CPD. On the other hand, it is equivalent to the rank-$(L_r, L_r, 1)$ BTD if $P = 2$ and $Q = 1$.

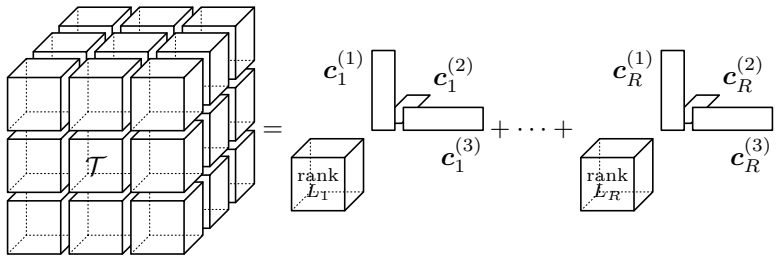


Figure 3.2: (Rank-L_r ∘ rank-1) block term decomposition of a sixth-order tensor.

Another way to interpret this decomposition, is as a structured CPD in which the nth factor matrix is just $A^{(n)}$ when $1 \leq n \leq P$, and defined as

$$A^{(n)} := C^{(n-P)} \cdot E \text{ when } P < n \leq N. \tag{3.6}$$

Here, E is defined as the $R \times R'$ block-diagonal matrix $\operatorname{diag}(\mathbf{1}_{1\times L_1}, \ldots, \mathbf{1}_{1\times L_R})$ in which the rth block on the diagonal is the row vector $\mathbf{1}_{1\times L_r}$. We use this interpretation to formulate (3.5) as the nonlinear least squares problem

$$\underset{\substack{A^{(1)},\ldots,A^{(P)} \\ C^{(1)},\ldots,C^{(Q)}}}{\text{minimize}} \quad f_{\text{BTD}}, \text{ where } f_{\text{BTD}} := \frac{1}{2}\|\mathcal{F}_{\text{BTD}}\|^2, \tag{3.7}$$

and the mode-n unfolding of the residual tensor $\mathcal{F}_{\text{BTD}}$ is defined as

$$(\mathcal{F}_{\text{BTD}})_{(n)} := A^{(n)} \cdot V^{\{n\}^{\mathrm{T}}} - T_{(n)} \text{ for any } 1 \leq n \leq N. \tag{3.8}$$

Although the residual tensor $\mathcal{F}_{\text{BTD}}$ is analytic in its argument, the objective function f_{BTD} is not because of its dependency on the complex conjugate of the argument. This implies that f_{BTD} is not complex differentiable and hence that its Taylor series does not exist everywhere on its domain. In the following section, we derive gradient-based unconstrained optimization methods and nonlinear least squares methods for (3.7), built with generalized algorithms for this class of optimization problems [206]. These methods are based on the observation that if a function of complex variables is analytic in the real and imaginary part of its argument, it is also analytic in its argument and the complex conjugate of its argument as a whole [2, 233].

3.4 Algorithms for the general decomposition

3.4.1 Unconstrained nonlinear optimization and alternating least squares

Solving (3.7) not only allows us to compute the (rank-$L_r \circ$ rank-1) BTD but also the CPD and the rank-$(L_r, L_r, 1)$ BTD. To this end, we first consider gradient-based unconstrained optimization methods that attempt to minimize f_{BTD} directly. Among these methods are the nonlinear conjugate gradient method and quasi-Newton methods such as the (limited-memory) BFGS method [159]. The complex counterparts of these methods [206] are built on the second-order model

$$m_k^f(\overset{c}{\boldsymbol{p}}) := f(\overset{c}{\boldsymbol{z}}_k) + \overset{c}{\boldsymbol{p}}^{\mathrm{T}} \frac{\partial f(\overset{c}{\boldsymbol{z}}_k)}{\partial \overset{c}{\boldsymbol{z}}} + \frac{1}{2} \overset{c}{\boldsymbol{p}}^{\mathrm{H}} B_k \overset{c}{\boldsymbol{p}} \tag{3.9}$$

of the objective function f at the current iterate $\overset{c}{\boldsymbol{z}}_k$, where the superscript $\overset{c}{\cdot}$ denotes the concatenation of its argument with its complex conjugate, i.e., $\overset{c}{\boldsymbol{z}} \equiv (\boldsymbol{z}, \overline{\boldsymbol{z}})$, and B_k is a Hermitian positive definite matrix that is updated at every iteration. The partial derivative in (3.9) is called the complex gradient at $\overset{c}{\boldsymbol{z}}_k$ and is composed of two parts: its top half $\frac{\partial f}{\partial \boldsymbol{z}}$ is the cogradient and its bottom half $\frac{\partial f}{\partial \overline{\boldsymbol{z}}}$ is the conjugate cogradient. For real-valued f, the cogradients are each other's complex conjugates. By definition, these complex derivatives are to be interpreted as partial derivatives with respect to complex variables while treating their complex conjugates as constant. They are also known, especially in the German literature, as Wirtinger derivatives [182]. The minimizer of the convex quadratic model (3.9) can be obtained by setting the model's conjugate complex gradient equal to zero and is given by

$$\overset{c}{\boldsymbol{p}}_k^* = -B_k^{-1} \overline{\frac{\partial f(\overset{c}{\boldsymbol{z}}_k)}{\partial \overset{c}{\boldsymbol{z}}}}. \tag{3.10}$$

Storing and manipulating the full Hessian approximation B_k or its inverse B_k^{-1} can be quite expensive for functions of many variables, as is often the case for tensor decompositions. Many strategies to store the approximation to the Hessian exist. For instance, the nonlinear conjugate gradient method can be interpreted to build B_k as the sum of the identity matrix and a rank-two update. The limited-memory BFGS (L-BFGS) method generalizes this concept to the sum of a (scaled) identity matrix and m rank-two updates. In practice, L-BFGS often performs better than nonlinear conjugate gradient due to its flexibility and is regarded as superior to the latter.

In Theorem 3.1 we derive an expression for f_{BTD}'s cogradient, which enables us to update B_k and compute the search direction (3.10). The following proposition is a useful tool that allows us to initially disregard any structure in $A^{(P+q)}$ by relating the partial derivative with respect to $C^{(q)}$ to that of $A^{(P+q)}$. Proposition 3.2 is a well-known property of the Khatri–Rao product that we will use in the derivation of the cogradient.

Proposition 3.1. *Let* $F^{(q)} = E \otimes \mathbb{I}_{I_{(P+q)}}$*, then we have that* $\operatorname{vec}(C^{(q)} \cdot E)^{\mathrm{T}} = \operatorname{vec}(C^{(q)})^{\mathrm{T}} \cdot F^{(q)}$*. Furthermore, applying the chain rule leads to*

$$\frac{\partial}{\partial C^{(q)}} = \frac{\partial}{\partial A^{(P+q)}} \cdot E^{\mathrm{T}} \quad \textit{and} \tag{3.11a}$$

$$\frac{\partial}{\partial \operatorname{vec}(C^{(q)})^{\mathrm{T}}} = \frac{\partial}{\partial \operatorname{vec}(A^{(P+q)})^{\mathrm{T}}} \cdot {F^{(q)}}^{\mathrm{T}}. \tag{3.11b}$$

Proposition 3.2 (see e.g., [28, 180])**.** *Let* $A^{(n)} \in \mathbb{C}^{I_n \times J}$*,* $n = 1, \ldots, N$*, let* $V = A^{(p_1)} \odot \cdots \odot A^{(p_N)}$ *where* $\boldsymbol{p}$ *is any permutation of* $\{1, \ldots, N\}$ *and let* $W = \circledast_{n=1}^{N} {A^{(n)}}^{\mathrm{H}} A^{(n)}$ *be the Hadamard product of all matrices* ${A^{(n)}}^{\mathrm{H}} A^{(n)}$*. Then* $V^{\mathrm{H}} V = W$*.*

Corollary 3.1. *Define* $W^{\sigma} := \circledast_{\substack{n=1 \\ n \notin \sigma}}^{N} {A^{(n)}}^{\mathrm{H}} A^{(n)}$*, then* $(V^{\sigma})^{\mathrm{H}} V^{\sigma} = W^{\sigma}$*.*

Theorem 3.1. *Let* $\boldsymbol{z}$ *be the vector of unknowns* $(\operatorname{vec}(A^{(1)}), \ldots, \operatorname{vec}(A^{(P)}), \operatorname{vec}(C^{(1)}), \ldots, \operatorname{vec}(C^{(Q)}))$*, then* f_{BTD}*'s complex cogradient is given by*

$$\begin{aligned} \frac{\partial f_{\mathrm{BTD}}}{\partial \boldsymbol{z}} = \Bigg(& \operatorname{vec}\left(\frac{\partial f_{\mathrm{BTD}}}{\partial A^{(1)}}\right), \ldots, \operatorname{vec}\left(\frac{\partial f_{\mathrm{BTD}}}{\partial A^{(P)}}\right), \\ & \operatorname{vec}\left(\frac{\partial f_{\mathrm{BTD}}}{\partial C^{(1)}}\right), \ldots, \operatorname{vec}\left(\frac{\partial f_{\mathrm{BTD}}}{\partial C^{(Q)}}\right)\Bigg), \end{aligned} \tag{3.12}$$

where

$$2\frac{\partial f_{\text{BTD}}}{\partial A^{(p)}} = \overline{A}^{(p)} \cdot W^{\{p\}} - \overline{T}_{(p)} \cdot V^{\{p\}}, \tag{3.13a}$$

$$2\frac{\partial f_{\text{BTD}}}{\partial C^{(q)}} = \overline{C}^{(q)} \cdot E \cdot W^{\{P+q\}} \cdot E^{\mathrm{T}} - \overline{T}_{(P+q)} \cdot V^{\{P+q\}} \cdot E^{\mathrm{T}} \tag{3.13b}$$

for $1 \leq p \leq P$ *and* $1 \leq q \leq Q$*, respectively.*

Proof. Let $f^{(1)}_{\text{BTD}} = \|A^{(n)} \cdot V^{\{n\}^{\mathrm{T}}}\|^2$, $f^{(2)}_{\text{BTD}} = \langle T_{(n)}, A^{(n)} \cdot V^{\{n\}^{\mathrm{T}}} \rangle$, and $f^{(3)}_{\text{BTD}} = \|T_{(n)}\|^2$, so that $f_{\text{BTD}} = \frac{1}{2}(f^{(1)}_{\text{BTD}} - f^{(2)}_{\text{BTD}} - \overline{f}^{(2)}_{\text{BTD}} + f^{(3)}_{\text{BTD}})$. We have that

$$\begin{aligned}
\frac{\partial f^{(1)}_{\text{BTD}}}{\partial A^{(n)}} &= \frac{\partial}{\partial A^{(n)}} \sum_i \left(\overline{\boldsymbol{v}}^{\{n\}}_{i:} \overline{A}^{(n)^{\mathrm{T}}} \right) \left(A^{(n)} \boldsymbol{v}^{\{n\}^{\mathrm{T}}}_{i:} \right) \\
&= \sum_i \overline{A}^{(n)} \boldsymbol{v}^{\{n\}^{\mathrm{H}}}_{i:} \boldsymbol{v}^{\{n\}}_{i:} \\
&= \overline{A}^{(n)} \cdot W^{\{n\}},
\end{aligned}$$

where the last equality follows from Corollary 3.1. Similarly, it can be shown that $\frac{\partial f^{(2)}_{\text{BTD}}}{\partial A^{(n)}} = \overline{T}_{(n)} \cdot V^{\{n\}}$, and trivially that $\frac{\partial \overline{f}^{(2)}_{\text{BTD}}}{\partial A^{(n)}} = \frac{\partial f^{(3)}_{\text{BTD}}}{\partial A^{(n)}} = 0$. Expression (3.13b) follows from (3.13a), (3.6), and Proposition 3.1. □

In Table 3.1 we summarize Theorem 3.1 for the special case of a (complex) third-order CPD and rank-$(L_r, L_r, 1)$ BTD. If the decomposition is real, then it is easy to show that the real gradient is exactly the scaled conjugate cogradient $2\overline{\frac{\partial f_{\text{BTD}}}{\partial \boldsymbol{z}}}$ [206].

Decomposition	**Scaled conjugate cogradient** $2\overline{\frac{\partial f_{\text{BTD}}}{\partial \boldsymbol{z}}}$
third-order CPD	$\begin{bmatrix} \text{vec}(A^{(1)}\overline{[(A^{(2)\mathrm{H}}A^{(2)}) * (A^{(3)\mathrm{H}}A^{(3)})]} - T_{(1)}\overline{(A^{(3)} \odot A^{(2)})}) \\ \text{vec}(A^{(2)}\overline{[(A^{(1)\mathrm{H}}A^{(1)}) * (A^{(3)\mathrm{H}}A^{(3)})]} - T_{(2)}\overline{(A^{(3)} \odot A^{(1)})}) \\ \text{vec}(A^{(3)}\overline{[(A^{(1)\mathrm{H}}A^{(1)}) * (A^{(2)\mathrm{H}}A^{(2)})]} - T_{(3)}\overline{(A^{(2)} \odot A^{(1)})}) \end{bmatrix}$
rank-$(L_r, L_r, 1)$ BTD	$\begin{bmatrix} \text{vec}(A^{(1)}\overline{[(A^{(2)\mathrm{H}}A^{(2)}) * (E^{\mathrm{T}}C^{(1)\mathrm{H}}C^{(1)}E)]} - T_{(1)}\overline{((C^{(1)}E) \odot A^{(2)})}) \\ \text{vec}(A^{(2)}\overline{[(A^{(1)\mathrm{H}}A^{(1)}) * (E^{\mathrm{T}}C^{(1)\mathrm{H}}C^{(1)}E)]} - T_{(2)}\overline{((C^{(1)}E) \odot A^{(1)})}) \\ \text{vec}(C^{(1)}E\overline{[(A^{(1)\mathrm{H}}A^{(1)}) * (A^{(2)\mathrm{H}}A^{(2)})]}E^{\mathrm{T}} - T_{(3)}\overline{(A^{(2)} \odot A^{(1)})}E^{\mathrm{T}}) \end{bmatrix}$

Table 3.1: The (rank-L_r ∘ rank-1) BTD objective function's scaled conjugate cogradient in the special case of a third-order CPD and a rank-$(L_r, L_r, 1)$ BTD.

In a line search framework, the next iterate is $\boldsymbol{z}_{k+1} = \boldsymbol{z}_k + \alpha_k \boldsymbol{p}_k$, where the real step length α_k is usually chosen to satisfy the (strong) Wolfe conditions [159,245]. Line search algorithms are an integral part of quasi-Newton methods but can be difficult to implement. There are several good software implementations available in the public domain, such as Moré and Thuente [147] and Hager and Zhang [93].

Another approach to select the step is using a trust-region framework, where a region around the current iterate $\boldsymbol{z}_k$ is defined in which the model m_k is trusted to be an adequate representation of the objective function. The next iterate $\boldsymbol{z}_{k+1}$ is then chosen to be the approximate minimizer of the model in this region. In effect, the direction and length of the step is chosen simultaneously. The trust-region radius Δ_k is updated every iteration based on the trustworthiness ρ_k of the model, which is defined as the ratio of the actual reduction $f(\overset{\mathcal{C}}{\boldsymbol{z}}_k) - f(\overset{\mathcal{C}}{\boldsymbol{z}}_k + \overset{\mathcal{C}}{\boldsymbol{p}}_k)$ of the objective function and the predicted reduction $m_k^f(\mathbf{0}) - m_k^f(\overset{\mathcal{C}}{\boldsymbol{p}}_k)$. The dogleg method [173, 174], double-dogleg method [60], and two-dimensional subspace minimization [32] all attempt to approximately minimize the trust-region subproblem by restricting the step $\boldsymbol{p}$ to (a subset of) the two-dimensional subspace spanned by the steepest descent direction $-\frac{\partial f}{\partial \overline{\boldsymbol{z}}}$ and the quasi-Newton step (3.10) when the model Hessian B_k is positive definite.

Aside from quasi-Newton methods, the cogradient also gives rise to the simple but effective alternating least squares (ALS) algorithm [33, 55, 95], which is an application of the nonlinear block Gauss–Seidel method. In the latter, the partial gradients (3.13) are alternately set to zero. Each factor matrix $A^{(p)}$ is then computed as the right Moore–Penrose pseudoinverse of the matrix $V^{\{p\}\mathrm{T}}$ applied to the unfolding $T_{(p)}$ and is subsequently used to recompute $V^{\{p\}}$ and $W^{\{p\}}$. This can be done by explicitly solving the associated normal equations as in Algorithm 3.1 or by first decomposing $V^{\{p\}}$ with a QR factorization. The latter is numerically more stable but also more expensive in both memory and floating point operations considering that the normal equations can be computed efficiently using Corollary 3.1. We refer to these two types of updates as fast and accurate, respectively. The difference in accuracy between the two methods often has a negligible effect on convergence speed. Hence, a practical implementation might start by using the fast updates until some convergence criterion is satisfied, after which a few more accurate updates are computed to obtain maximal precision.

Although conceptually simple and often effective, ALS is not guaranteed to converge to a stationary point and is also sensitive to so-called swamps, where convergence is very slow for many iterations. For a given tensor, its best rank-R approximation may not exist. For example, no real $2 \times 2 \times 2$ rank-3 tensor has a best rank-2 approximation [57]. This ill-posedness is a consequence of

while (not converged) **do**
  for $p = 1, \ldots, P$ **do**
    $A^{(p)} \leftarrow T_{(p)} \cdot \overline{V}^{\{p\}} \cdot \overline{W}^{\{p\}^{-1}}$
  end
  for $q = 1, \ldots, Q$ **do**
    $C^{(q)} \leftarrow T_{(P+q)} \cdot \overline{V}^{\{P+q\}} \cdot E^{\mathrm{T}} \cdot (E \cdot \overline{W}^{\{P+q\}} \cdot E^{\mathrm{T}})^{-1}$
  end
end

Algorithm 3.1: Alternating least squares.

the fact that the set of rank-R tensors is not closed, i.e., there exist sequences of rank-R tensors that converge to rank-R^* tensors, where $R < R^*$ [57, 164]. For such sequences, two or more terms grow without bound yet nearly cancel each other out. This behaviour is referred to as degeneracy and has been linked with swamps [120, 144], which are observed when either the best rank-R approximation does not exist (strong degeneracy [115,217,218]) or when the best rank-R approximation does exist, but the path between the initialization and the solution displays degenerate factors for a certain number of iterations (weak degeneracy [144]). Several approaches to mitigate swamp-like behaviour in ALS have been proposed in the literature, among which are (exact) line search [28,95, 156, 179] and iterated Tikhonov regularization [152]. In contrast, optimization-based algorithms coupled with line search or trust-region globalization strategies do guarantee convergence to a stationary point and, depending on the method, spend comparatively fewer iterations in swamps.

3.4.2 Nonlinear least squares methods

Another approach to solve (3.7) is by means of nonlinear least squares methods such as Gauss–Newton or Levenberg–Marquardt. In these methods, the residual tensor $\mathcal{F}$ is approximated by the linear model

$$m_k^{\mathcal{F}}(\overset{\mathbb{C}}{\boldsymbol{p}}) := \operatorname{vec}(\mathcal{F}(\overset{\mathbb{C}}{\boldsymbol{z}}_k)) + \frac{\partial \operatorname{vec}(\mathcal{F}(\overset{\mathbb{C}}{\boldsymbol{z}}_k))}{\partial \overset{\mathbb{C}}{\boldsymbol{z}}^{\mathrm{T}}} \overset{\mathbb{C}}{\boldsymbol{p}}, \tag{3.14}$$

where the partial derivative is called the complex Jacobian at $\overset{\mathbb{C}}{\boldsymbol{z}}_k$ and is a straightforward generalization of the complex gradient [206]. The model $m_k^{\mathcal{F}}$ is then used in the modified quadratic model of the objective function

$$m_k^f(\overset{\mathbb{C}}{\boldsymbol{p}}) := \frac{1}{2}\|m_k^{\mathcal{F}}(\overset{\mathbb{C}}{\boldsymbol{p}})\|^2 + \frac{\lambda_k}{2}\|\boldsymbol{p}\|^2, \tag{3.15}$$

where λ_k is the Levenberg–Marquardt regularization parameter which influences both the length and direction of the step $\boldsymbol{p}$ that minimizes m_k^f. In the Gauss–Newton method, $\lambda_k = 0$ for all k, and a trust-region framework can instead be used to control the length and direction of the step.

By noting that $\mathcal{F}_{\text{BTD}}$ is analytic in its argument, we need only work with half of the complex Jacobian, since the conjugate Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F}_{\text{BTD}})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$ is identically equal to zero. As a consequence, the quadratic model $m_k^{f_{\text{BTD}}}$ is minimized by the Levenberg–Marquardt step

$$\boldsymbol{p}_k^* = -\begin{bmatrix} J_k \\ \sqrt{\lambda_k}\mathbb{I} \end{bmatrix}^{\dagger} \begin{bmatrix} \operatorname{vec}(\mathcal{F}_{\text{BTD}}(\boldsymbol{z}_k)) \\ \mathbf{0} \end{bmatrix}, \tag{3.16}$$

where J_k is the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F}_{\text{BTD}})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ at the kth iterate $\boldsymbol{z}_k$ and the identity matrix and zero vector are of the appropriate dimensions. Due to the scaling indeterminacy of the decomposition, the Jacobian is rank deficient and a certain number of its singular values are equal to zero. In general, each of the R terms contains a Qth-order rank-one tensor and a Pth-order rank-L_r tensor, which contribute $(Q-1)R$ and $(P-1)R'$ degrees of freedom, respectively. Furthermore, there is one more degree of freedom in the scaling between the rank-one and rank-L_r part of each term, bringing the total number of singular values equal to zero to at least $QR + (P-1)R'$. Note that this is just a lower bound and that the actual amount of singular values equal to zero depends on the unicity of the individual terms. For example, when $P = 2$, each term contains a rank-L_r matrix, the factors of which are only unique up to a linear transformation. In the Levenberg–Marquardt algorithm, the Jacobian is allowed to be singular since each of the steps are regularized by a norm constraint on the solution. However, in a Gauss–Newton trust-region algorithm, special care must be taken to invert the Jacobian in a meaningful way. One approach is to compute the Moore–Penrose pseudoinverse $J_k^{\dagger}$. Another is to compute an approximate solution with a truncated conjugate gradient algorithm, where the amount of regularization is controlled by the number of iterations. As we will see, the latter method allows for significant savings in both memory and computational cost.

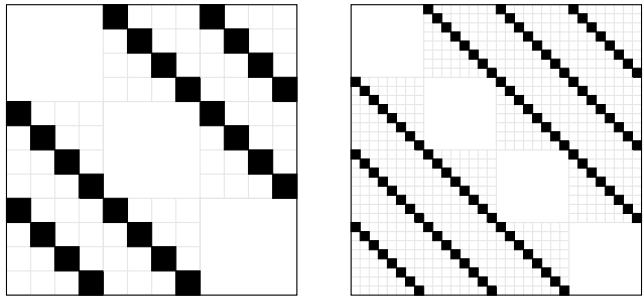
Storing the Jacobian as a dense matrix is impractical as each of its columns requires as much memory as the tensor $\mathcal{T}$ itself. A sparse representation is one way to store the Jacobian more efficiently but is likely of limited use due to a large amount of fill-in appearing when applying the QR factorization to solve (3.16) [162, 229]. At the cost of squaring the condition number of the system, the memory requirement can be reduced by solving the normal equations

$$(J_k^{\mathrm{H}} J_k + \lambda_k \mathbb{I})\boldsymbol{p}_k^* = -J_k^{\mathrm{H}} \operatorname{vec}(\mathcal{F}_{\text{BTD}}(\boldsymbol{z}_k)) = -2\frac{\partial f_{\text{BTD}}}{\partial \overline{\boldsymbol{z}}}(\boldsymbol{z}_k), \tag{3.17}$$

where the last equality follows from the analyticity of $\mathcal{F}_{\text{BTD}}$ [206], associated with (3.16) and looking for an expression for the Gramian $J^{\text{H}}J$ [228]. In the real case, the latter matrix represents an approximation of the objective function's Hessian. The error of the approximation is the sum [159]

$$\sum_{i_1,\ldots,i_N=1}^{I_1,\ldots,I_N} (\mathcal{F}_{\text{BTD}})_{i_1\ldots i_N} \frac{\partial (\mathcal{F}_{\text{BTD}})_{i_1\ldots i_N}}{\partial \boldsymbol{z} \partial \boldsymbol{z}^{\text{T}}}$$

and hence is small when the residuals $\mathcal{F}_{\text{BTD}}$ are small, or when $\mathcal{F}_{\text{BTD}}$ is nearly linear in its argument. Fortunately, the multilinear structure of $\mathcal{F}_{\text{BTD}}$ ensures that the total contribution of the Hessians $\frac{\partial (\mathcal{F}_{\text{BTD}})_{i_1\ldots i_N}}{\partial \boldsymbol{z} \partial \boldsymbol{z}^{\text{T}}}$ is quite sparse, depending on the number of rank-one terms. In fact, one can show that the relative number of nonzero elements in the superimposed Hessian is equal to $\frac{N(N-1)RI^2}{(NRI)^2} = \frac{N-1}{NR}$ for a CPD of an Nth-order tensor of dimensions $I \times \cdots \times I$ in R rank-one terms (cf. Figure 3.3). Moreover, it can be observed that these nonzero elements are often relatively small in comparison to the elements of $J^{\text{H}}J$. It may therefore be expected that, as the residuals $(\mathcal{F}_{\text{BTD}})_{i_1\ldots i_N}$ decrease, the Gramian $J^{\text{H}}J$ rapidly becomes a good approximation to the real Hessian. In Theorem 3.2 we derive the structure of $J^{\text{H}}J$: it is a block matrix consisting of diagonal and low-rank blocks.



(a) $N = 3$, $R = 4$ (b) $N = 4$, $R = 8$

Figure 3.3: Sparsity pattern of the total contribution of the Hessians $\frac{\partial (\mathcal{F}_{\text{BTD}})_{i_1\ldots i_N}}{\partial \boldsymbol{z} \partial \boldsymbol{z}^{\text{T}}}$ in the case of a real CPD of a (hyper)cubical third-order tensor (left) and fourth-order tensor (right).

Theorem 3.2. *Let J be the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and let $F^{(q)}$ be defined as in Proposition 3.1, then the Gramian $J^{\mathrm{H}}J$ is given by*

$$J^{\mathrm{H}}J = \Sigma \cdot \begin{bmatrix} \Pi^{(1,1)} & \dots & \Pi^{(1,N)} \\ \vdots & \ddots & \vdots \\ \Pi^{(N,1)} & \dots & \Pi^{(N,N)} \end{bmatrix} \cdot \Sigma^{\mathrm{T}}, \tag{3.18}$$

where $\Sigma := \operatorname{diag}\left(\mathbb{I}_{I_1 R'}, \dots, \mathbb{I}_{I_P R'}, F^{(1)}, \dots, F^{(Q)}\right)$ *and*

$$\Pi^{(n_1,n_2)} := \left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \operatorname{vec}(A^{(n_1)})^{\mathrm{T}}}\right)^{\mathrm{H}} \left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \operatorname{vec}(A^{(n_2)})^{\mathrm{T}}}\right)$$

$$= \begin{cases} W^{\{n\}} \otimes \mathbb{I}_{I_n} & n = n_1 = n_2 \\ \begin{bmatrix} w_{1,1}^{\{n_1,n_2\}} \boldsymbol{a}_1^{(n_1)} \boldsymbol{a}_1^{(n_2)\mathrm{H}} & \cdots & w_{1,R'}^{\{n_1,n_2\}} \boldsymbol{a}_{R'}^{(n_1)} \boldsymbol{a}_1^{(n_2)\mathrm{H}} \\ \vdots & \ddots & \vdots \\ w_{R',1}^{\{n_1,n_2\}} \boldsymbol{a}_1^{(n_1)} \boldsymbol{a}_{R'}^{(n_2)\mathrm{H}} & \cdots & w_{R',R'}^{\{n_1,n_2\}} \boldsymbol{a}_{R'}^{(n_1)} \boldsymbol{a}_{R'}^{(n_2)\mathrm{H}} \end{bmatrix} & \text{otherwise.} \end{cases} \tag{3.19}$$

Proof. Let $\boldsymbol{e}_i^{(n)}$ be the ith column of the identity matrix $\mathbb{I}_{I_n}$, then

$$\frac{\partial \mathcal{F}_{\mathrm{BTD}}}{\partial a_{i_n r}^{(n)}} = \boldsymbol{a}_r^{(1)} \circ \cdots \circ \boldsymbol{a}_r^{(n-1)} \circ \boldsymbol{e}_{i_n}^{(n)} \circ \boldsymbol{a}_r^{(n+1)} \circ \cdots \circ \boldsymbol{a}_r^{(N)}.$$

Diagonal blocks. For $n = n_1 = n_2$ we have

$$\left\langle \frac{\partial \mathcal{F}_{\mathrm{BTD}}}{\partial a_{ir_1}^{(n)}}, \frac{\partial \mathcal{F}_{\mathrm{BTD}}}{\partial a_{jr_2}^{(n)}} \right\rangle = \begin{cases} w_{r_1 r_2}^{\{n\}} & i = j \\ 0 & \text{otherwise} \end{cases}$$

and so

$$\left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \boldsymbol{a}_{r_1}^{(n)\mathrm{T}}}\right)^{\mathrm{H}} \left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \boldsymbol{a}_{r_2}^{(n)\mathrm{T}}}\right) = w_{r_1 r_2}^{\{n\}} \mathbb{I}_{I_n}.$$

Off-diagonal blocks. For $n_1 \neq n_2$ we have

$$\left\langle \frac{\partial \mathcal{F}_{\mathrm{BTD}}}{\partial a_{i_{n_1} r_1}^{(n_1)}}, \frac{\partial \mathcal{F}_{\mathrm{BTD}}}{\partial a_{i_{n_2} r_2}^{(n_2)}} \right\rangle = w_{r_1 r_2}^{\{n_1,n_2\}} a_{i_{n_1} r_2}^{(n_1)} \overline{a}_{i_{n_2} r_1}^{(n_2)}$$

and so

$$\left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \boldsymbol{a}_{r_1}^{(n_1)\mathrm{T}}}\right)^{\mathrm{H}} \left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \boldsymbol{a}_{r_2}^{(n_2)\mathrm{T}}}\right) = w_{r_1 r_2}^{\{n_1,n_2\}} \boldsymbol{a}_{r_2}^{(n_1)} \boldsymbol{a}_{r_1}^{(n_2)\mathrm{H}}.$$

The $(p, P+q)$th block in $J^{\mathrm{H}}J$ is defined as $\left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \operatorname{vec}(A^{(p)})^{\mathrm{T}}}\right)^{\mathrm{H}} \left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{BTD}})}{\partial \operatorname{vec}(C^{(q)})^{\mathrm{T}}}\right)$ and can be computed from $\Pi^{(p,P+q)}$ using (3.11b) as $\Pi^{(p,P+q)} \cdot F^{(q)\mathrm{T}}$. Similarly, the $(P+q, p)$th and $(P+q_1, P+q_2)$th blocks can be computed as $F^{(q)} \cdot \Pi^{(p,P+q)}$ and $F^{(q_1)} \cdot \Pi^{(P+q_1,P+q_2)} \cdot F^{(q_2)\mathrm{T}}$, respectively. □

Because $\mathcal{F}_{\mathrm{BTD}}$ is analytic in its argument, the expression for $J^{\mathrm{H}}J$ is nearly the same as in the real case. Its structure has been noticed before for the real CPD [162], and several methods of exploiting it have been proposed in the literature, for instance by efficient construction of the matrix [228] or by writing it as a block-diagonal matrix plus a low-rank correction [168]. We propose to exploit its structure by storing the matrix implicitly as the collection of factor matrices $A^{(n)}$ and the factor matrices' Gramians $A^{(n)\mathrm{H}}A^{(n)}$, from which the Hadamard products $W^{\{n\}}$ and $W^{\{n_1,n_2\}}$ can easily be reconstructed. Storing the Jacobian's Gramian in this way is not appropriate for direct solvers, but does admit the use of inexact adaptations of the Levenberg–Marquardt and Gauss–Newton algorithms by computing only matrix-vector products. The latter methods only partially invert the Jacobian's Gramian using a limited number of (preconditioned) conjugate gradient iterations [159]. Besides the significant reduction in memory cost, the following theorem shows that the Gramian's structure also allows for an efficient matrix-vector product. Compared to a dense matrix-vector product, Theorem 3.3 can reduce the computational complexity from $O(N^2R^2I^2)$ to $O(NR^2I)$ flop per matrix-vector product, depending on the implementation.

Theorem 3.3. *Let* $\boldsymbol{p} = (\boldsymbol{b}^{(1)}, \ldots, \boldsymbol{b}^{(P)}, \boldsymbol{d}^{(1)}, \ldots, \boldsymbol{d}^{(Q)})$*, where* $B^{(p)} \in \mathbb{C}^{I_p \times R'}$ *and* $\boldsymbol{b}^{(p)} = \operatorname{vec}(B^{(p)})$*,* $p = 1, \ldots, P$*,* $D^{(q)} \in \mathbb{C}^{I_{P+q} \times R}$ *and* $\boldsymbol{d}^{(q)} = \operatorname{vec}(D^{(q)})$*,* $q = 1, \ldots, Q$*. Furthermore, we define* $B^{(P+q)} := D^{(q)} \cdot E$*, and* $\boldsymbol{b}^{(P+q)} := \operatorname{vec}(B^{(P+q)})$*. The matrix-vector product* $J^H J \cdot \boldsymbol{p}$ *then follows from*

$$\Pi^{(p,p)} \cdot \boldsymbol{b}^{(p)} = \operatorname{vec}(X^{(p)}) \tag{3.20a}$$

$$\Pi^{(p_1,p_2)} \cdot \boldsymbol{b}^{(p_2)} = \operatorname{vec}(Y^{(p_1,p_2)}) \tag{3.20b}$$

$$(F^{(q)} \cdot \Pi^{(P+q,p)}) \cdot \boldsymbol{b}^{(p)} = \operatorname{vec}(Y^{(P+q,p)} \cdot E^{\mathrm{T}}) \tag{3.20c}$$

$$(\Pi^{(p,P+q)} \cdot F^{(q)^{\mathrm{T}}}) \cdot \boldsymbol{d}^{(q)} = \operatorname{vec}(Y^{(p,P+q)}) \tag{3.20d}$$

$$(F^{(q_1)} \cdot \Pi^{(P+q_1,P+q_2)} \cdot F^{(q_2)^{\mathrm{T}}}) \cdot \boldsymbol{d}^{(q_2)} = \operatorname{vec}(Y^{(P+q_1,P+q_2)} \cdot E^{\mathrm{T}}) \tag{3.20e}$$

$$(F^{(q)} \cdot \Pi^{(P+q,P+q)} \cdot F^{(q)^{\mathrm{T}}}) \cdot \boldsymbol{d}^{(q)} = \operatorname{vec}(X^{(P+q)} \cdot E^{\mathrm{T}}) \tag{3.20f}$$

where

$$X^{(n)} := B^{(n)} \cdot \overline{W}^{\{n\}}, \tag{3.21}$$

$$Y^{(n_1,n_2)} := A^{(n_1)} \cdot (\overline{W}^{\{n_1,n_2\}} * (B^{(n_2)^{\mathrm{T}}} \cdot \overline{A}^{(n_2)})) \text{ and} \tag{3.22}$$

$1 \leq p_1 \neq p_2 \leq P$, $1 \leq q_1 \neq q_2 \leq Q$, *and* $1 \leq n, n_1 \neq n_2 \leq N$.

Proof. We have that $\Pi^{(p,p)} \cdot \boldsymbol{b}^{(p)} = (W^{\{p\}} \otimes \mathbb{I}_{I_p}) \cdot \mathrm{vec}(B^{(p)}) = \mathrm{vec}(B^{(p)} \cdot W^{\{p\}\mathrm{T}}) = \mathrm{vec}(B^{(p)} \cdot \overline{W}^{\{p\}})$. A similar structure is present in $\Pi^{(p_1,p_2)} \cdot \boldsymbol{b}^{(p_2)}$. Let $Z = W^{\{p_1,p_2\}} * (A^{(p_2)\mathrm{H}} B^{(p_2)})$, then it is not hard to show that $\Pi^{(p_1,p_2)} \cdot \boldsymbol{b}^{(p_2)} = (Z \otimes \mathbb{I}_{I_{p_1}}) \cdot \mathrm{vec}(A^{(p_1)})$, from which (3.20b) follows. The special cases (3.20c–3.20f) are a direct result of (3.20a–3.20b) and Proposition 3.1. □

The convergence rate of the conjugate gradient algorithm depends on how well the system's eigenvalues are clustered and is consequently also influenced by its condition number. In the preconditioned conjugate gradient (PCG) algorithm, the eigenvalue distribution is improved by solving a system of the form

$$M^{-1} \cdot J^{\mathrm{H}} J \cdot \boldsymbol{p} = M^{-1} \cdot \left(-2 \frac{\partial f_{\mathrm{BTD}}}{\partial \overline{\boldsymbol{z}}} \right),$$

where M is a symmetric positive definite matrix called the preconditioner. The inverse of the preconditioner M^{-1} should be cheap to apply and is often designed so that $M^{-1} \cdot J^{\mathrm{H}} J \approx \mathbb{I}$. The block Jacobi preconditioner

$$M_{\mathrm{BJ}} = \Sigma \cdot \begin{bmatrix} \Pi^{(1,1)} & & \\ & \ddots & \\ & & \Pi^{(N,N)} \end{bmatrix} \cdot \Sigma^{\mathrm{T}} \tag{3.23}$$

is one such example. It is a block-diagonal approximation of (3.18) and can be inverted efficiently using (3.20a) and (3.20f). For instance, the solution of the system $\Pi^{(p,p)} \cdot \mathrm{vec}(X) = \mathrm{vec}(Y)$ can be computed by solving the much smaller system $X = Y \cdot \overline{W}^{\{p\}-1}$. It is interesting to note that if $M_{\mathrm{BJ}}^{-1} \cdot J^{\mathrm{H}} J = \mathbb{I}$, the computed step $\boldsymbol{p}$ amounts to a simultaneous version of the fast ALS updates of Algorithm 3.1. Indeed, we then have that $\boldsymbol{p} = M_{\mathrm{BJ}}^{-1} \cdot \left(-2 \frac{\partial f_{\mathrm{BTD}}}{\partial \overline{\boldsymbol{z}}} \right)$, of which the pth component $B^{(p)}$ is given by

$$B^{(p)} = -2 \frac{\partial f_{\mathrm{BTD}}}{\partial \overline{A^{(p)}}} \cdot \overline{W}^{\{p\}^{-1}} = -A^{(p)} + T_{(p)} \cdot \overline{V}^{\{p\}} \cdot \overline{W}^{\{p\}^{-1}}.$$

The pth factor matrix of the next iterate $A^{(p)} + B^{(p)}$ is hence equal to that obtained by a fast ALS update in which the updated factor matrices are not used

to recompute $V^{\{p\}}$ or $W^{\{p\}}$. This preconditioner adds only $O(\frac{1}{3}NR^3 + 2NR^2I)$ flop per conjugate gradient iteration, which is relatively cheap in comparison to the computation of the scaled conjugate cogradient that acts as the right-hand side of the linear system. In summary, the initial solution computed by PCG with a block Jacobi preconditioner is similar to that of an ALS update, and the effect of the off-diagonal blocks is subsequently taken into account in an iterative manner. In this light, these matrix-free nonlinear least squares methods can be viewed as a refinement of the alternating least squares algorithm with simultaneous updates. The numerical experiments show that this refinement pays off for difficult problems and is still competitive with ALS for simpler problems.

3.4.3 Computational complexity

In Section 3.5 we compare the robustness and efficiency of ALS, quasi-Newton methods (nonlinear conjugate gradient and L-BFGS with Moré–Thuente line search, L-BFGS with dogleg trust-region [173, 174]), and inexact nonlinear least squares methods (inexact Gauss–Newton with CG-Steihaug [219] or dogleg trust-region, inexact Levenberg–Marquardt) on complex block term and canonical polyadic decompositions. The comparison is based on performance profiles [65] with the number of floating point operations (flop), expressed as an equivalent number of objective function evaluations, as a metric for the amount of computational resource required by solver s to solve problem p. Table 3.2 gives an overview of the amount of flop per iteration for the above algorithms, where it is assumed that the inexact nonlinear least squares methods use a block Jacobi preconditioner. For most algorithms, a significant part of the cost in each iteration is to compute the gradient of the objective function. This cost can be reduced by a factor N by either exploiting parallelism or, as recently proposed, by recursive computation [169]. We use Table 3.2 to estimate the total amount of floating point operations given the iteration counts and then divide by the number of floating point operations equivalent to one function evaluation to obtain an equivalent number of function evaluations. For example, alternating least squares requires a number of floating point operations per iteration equivalent to the cost of $N + 1$ function evaluations.

Aside from the substantial reduction in floating point operations that the inexact nonlinear least squares methods offer, they also require significantly less memory in comparison to their exact counterparts. The exact methods need $\mathcal{O}(N^2R^2I^2)$ memory cells to store the Jacobian's Gramian, while the inexact methods only require $\mathcal{O}(NR^2)$ memory cells, the same amount of memory that the alternating least squares algorithm requires. Furthermore, it is to be expected that the Gauss–Newton with dogleg trust-region strategy is more efficient than the

Levenberg–Marquardt method; the former only solves one system involving the Jacobian's Gramian per iteration, while the latter solves such a system at every inner iteration. We also note that nonlinear conjugate gradient [5, 163] and exact Levenberg–Marquardt [96, 162, 168, 228] have been applied to the real CPD before. For an overview and comparison of existing algorithms for the CPD, see [39] and [229], respectively.

Algorithm	**Complexity** (flop/iteration)
ALS	$\mathcal{O}(2(N+1)R\prod I_n)$
L-BFGS-DL	$\mathcal{O}(2(N+\text{it}_{\text{dl}})R\prod I_n + 8mR\sum I_n)$
{CG,L-BFGS}-MT	$\mathcal{O}(\text{it}_{\text{mt}}2(N+1)R\prod I_n + 8mR\sum I_n)$
LM	$\mathcal{O}(2(N+\text{it}_{\text{lm}})R\prod I_n + \text{it}_{\text{lm}}\frac{1}{3}R^3(\sum I_n)^3)$
GN-DL	$\mathcal{O}(2(N+\text{it}_{\text{gn}})R\prod I_n + \frac{8}{3}R^3(\sum I_n)^3)$
LM (inexact)	$\mathcal{O}(2(N+\text{it}_{\text{lm}})R\prod I_n$ $+\text{it}_{\text{lm}}\text{it}_{\text{cg}}(\frac{5}{2}N^2R^2 + \frac{1}{3}NR^3 + 8R^2\sum I_n))$
GN-{CGS,DL} (inexact)	$\mathcal{O}(2(N+\text{it}_{\text{gn}})R\prod I_n$ $+\text{it}_{\text{cg}}(\frac{5}{2}N^2R^2 + \frac{1}{3}NR^3 + 8R^2\sum I_n))$

Table 3.2: Computational complexity of several BTD algorithms in flop/iteration in case of a real CPD of an Nth-order $I_1 \times \cdots \times I_N$ tensor in R rank-one terms. From top to bottom, the algorithms are alternating least squares, limited-memory BFGS with dogleg trust-region, nonlinear conjugate gradient and limited-memory BFGS with Moré–Thuente line search, Levenberg–Marquardt, Gauss–Newton with dogleg trust-region, inexact Levenberg–Marquardt and inexact Gauss–Newton with CG-Steihaug or dogleg trust-region. (Note that for CG-MT, $m \equiv 1$.)

In the following, we derive the computational complexity per iteration of several algorithms, which can often conveniently be expressed in terms of a number of function evaluations. To simplify the obtained expressions, we restrict the discussion to the CPD of a real Nth-order tensor of dimensions $I_1 \times \cdots \times I_N$ in R rank-one terms. In the complex case, a good rule of thumb is to multiply the number of floating point operations by a factor of four. A function evaluation of f_{BTD} (or $\mathcal{F}_{\text{BTD}}$) then costs

$$\begin{array}{rl} \mathcal{O}(2R\prod_{n=1}^{N} I_n) & (\text{Computing } \mathcal{F}_{\text{BTD}}) \\ \underline{\mathcal{O}(2\prod_{n=1}^{N} I_n)} & (\text{Computing } \frac{1}{2}\|\mathcal{F}_{\text{BTD}}\|^2) \\ \mathcal{O}(2R\prod_{n=1}^{N} I_n) & \text{flop.} \end{array}$$

A cogradient evaluation $\frac{\partial f_{\text{BTD}}}{\partial \boldsymbol{z}}$ costs the equivalent of N function evaluations, namely

$$\begin{aligned} \mathcal{O}(\textstyle\sum_{n=1}^{N} \prod_{\substack{m=1 \\ m\neq n}}^{N} I_m) &\quad \text{(Computing all } V^{\{n\}}\text{)} \\ \mathcal{O}(2NR \textstyle\prod_{n=1}^{N} I_n) &\quad \text{(Computing all } \overline{T}_{(n)} \cdot V^{\{n\}}\text{)} \\ \mathcal{O}(\tfrac{1}{2}(N-1)R^2 + R^2 \textstyle\sum_{n=1}^{N} I_n) &\quad \text{(Computing all } W^{\{n\}} \text{ and } \overline{A}^{(n)} \cdot W^{\{n\}}\text{)} \\ \hline \mathcal{O}(2NR \textstyle\prod_{n=1}^{N} I_n) &\quad \text{flop.} \end{aligned}$$

Alternating least squares The cost of a fast ALS iteration is very similar to a gradient evaluation, with the most important difference being that the matrix-matrix product $\overline{A}^{(n)} \cdot W^{\{n\}}$ is replaced by solving a Hermitian linear system of order R. The matrices $W^{\{n\}} = {V^{\{n\}}}^{\text{H}} V^{\{n\}}$ can be factorized with a Cholesky decomposition, in which case solving all N systems costs $\mathcal{O}(\frac{1}{3}NR^3 + 2R^2 \sum_{n=1}^{N} I_n)$ flop, which is often negligible compared to the cost of computing the products $\overline{T}_{(n)} \cdot V^{\{n\}}$. Every iteration, the objective function is also evaluated to check for convergence. The complexity of a fast ALS iteration is hence equal to

$$\begin{aligned} \mathcal{O}(2NR \textstyle\prod_{n=1}^{N} I_n) &\quad \text{(Computing all } T_{(n)} \cdot \overline{V}^{\{n\}}\text{)} \\ \mathcal{O}(\tfrac{1}{3}NR^3 + 2R^2 \textstyle\sum_{n=1}^{N} I_n) &\quad \text{(Solving the linear systems)} \\ \mathcal{O}(2R \textstyle\prod_{n=1}^{N} I_n) &\quad \text{(Function evaluation)} \\ \hline \mathcal{O}(2(N+1)R \textstyle\prod_{n=1}^{N} I_n) &\quad \text{flop/iteration.} \end{aligned}$$

An accurate ALS iteration projects the rows of $T_{(n)}$ onto an orthogonal basis of $V^{\{n\}}$, instead of on $V^{\{n\}}$ itself as in a fast ALS iteration. An orthogonal basis of $V^{\{n\}}$ can be obtained by means of a QR decomposition. These decompositions require a total of $\mathcal{O}(-2R^3 + 2R^2 \sum_{n=1}^{N} \prod_{\substack{m=1 \\ m\neq n}}^{N} I_m)$ flop, which is comparable in cost to a fast ALS iteration for moderate R. Hence the cost of an accurate ALS iteration is roughly double that of a fast ALS iteration.

General unconstrained optimization Limited memory BFGS with a dogleg trust-region strategy (L-BFGS-DL) requires one cogradient evaluation every iteration and one function evaluation every inner iteration. Let it_{dl} be the number of dogleg iterations in each outer iteration, then for a small number of L-BFGS updates m the computational complexity is $\mathcal{O}(2(N + \text{it}_{\text{dl}})R \prod_{n=1}^{N} I_n)$ flop/iteration.

The general optimization methods based on a Moré–Thuente line search, require one function and one gradient evaluation per line search iteration. Let it_{mt} be the number of line search iterations in each outer iteration, then L-BFGS and

nonlinear conjugate gradient with Moré–Thuente line search (L-BFGS-MT and CG-MT) cost $\mathcal{O}(\mathrm{it}_{\mathrm{mt}} 2(N+1)R\prod_{n=1}^{N} I_n)$ flop/iteration for small m.

Nonlinear least squares Let us first consider the exact nonlinear least squares methods. The matrix $J^{\mathrm{H}}J$ requires $\mathcal{O}(\frac{1}{2}R^2(\sum_{n=1}^{N} I_n)^2)$ memory cells. Precomputing the Gramians $A^{(n)^{\mathrm{H}}}A^{(n)}$ costs $\mathcal{O}(R^2\sum_{n=1}^{N} I_n)$ flop. For each diagonal block $\Pi^{(n,n)}$, the Hadamard products $W^{\{n\}}$ must be generated given these precomputed Gramians, which costs $\mathcal{O}(\frac{1}{2}NR^2)$ flop given $W^{\varnothing}$, where $\varnothing$ is the empty set. Each off-diagonal block $\Pi^{(n,m)}$ also requires a matrix $W^{\{n,m\}}$ and a further three multiplications per entry. Taking the symmetry into account, this is a total of $\mathcal{O}(\sum_{n=1}^{N}\sum_{m=n+1}^{N}(\frac{1}{2}R^2 + 3R^2 I_n I_m))$ flop. In the hypercubical case $I = I_1 = \cdots = I_N$, the cost of evaluating $J^{\mathrm{H}}J$ is then

$$\begin{aligned}
\mathcal{O}(NR^2I) &\quad \text{(Computing all } A^{(n)^{\mathrm{H}}}A^{(n)}\text{)}\\
\mathcal{O}(\tfrac{1}{2}NR^2) &\quad \text{(Computing all } W^{\{n\}}\text{)}\\
\mathcal{O}(1) &\quad \text{(Computing all } \Pi^{(n,n)}\text{)}\\
\mathcal{O}(\tfrac{1}{2}\tfrac{N(N-1)}{2}R^2) &\quad \text{(Computing all } W^{\{n,m\}}\text{)}\\
\mathcal{O}(3\tfrac{N(N-1)}{2}R^2I^2) &\quad \text{(Computing all } \Pi^{(n,m)}\text{)}\\
\hline
\mathcal{O}(\tfrac{3}{2}N^2R^2I^2) &\quad \text{flop.}
\end{aligned}$$

In each iteration of the exact Levenberg–Marquardt algorithm, a linear system involving the Gramian $J^{\mathrm{H}}J$ and the conjugate cogradient must be solved repeatedly for different values of the regularization parameter λ until some descent criterion is satisfied, which is checked using function evaluations. If the number of such inner iterations in each outer iteration is denoted by $\mathrm{it}_{\mathrm{lm}}$, the cost of the exact Levenberg–Marquardt algorithm is given by

$$\begin{aligned}
\mathcal{O}(\tfrac{3}{2}N^2R^2I^2) &\quad \text{(Evaluating } J^{\mathrm{H}}J\text{)}\\
\mathcal{O}(2NRI^N) &\quad \text{(Cogradient evaluation)}\\
\mathcal{O}(\mathrm{it}_{\mathrm{lm}}(\tfrac{1}{3}N^3R^3I^3 + 2N^2R^2I^2)) &\quad \text{(Solving the linear system)}\\
\mathcal{O}(\mathrm{it}_{\mathrm{lm}}2RI^N) &\quad \text{(Function evaluation)}\\
\hline
\mathcal{O}(2(N+\mathrm{it}_{\mathrm{lm}})RI^N + \mathrm{it}_{\mathrm{lm}}\tfrac{1}{3}N^3R^3I^3) &\quad \text{flop/iteration.}
\end{aligned}$$

The Gauss–Newton method with dogleg trust-region solves only one linear system per iteration. The trust-region subproblem is then iteratively minimized along a line connecting the resulting Gauss–Newton step (2.31) and the steepest descent direction. These inner iterations each require one function evaluation to check if the model is sufficiently accurate. Unfortunately, the system $J^{\mathrm{H}}J$ is always singular and a filtered solution should be computed. One way of obtaining an accurate filtered solution is by means of the singular value decomposition, which in this case costs $\mathcal{O}(\frac{8}{3}N^3R^3I^3)$ flop. Let $\mathrm{it}_{\mathrm{gn}}$ be the number of inner

iterations in each outer iteration, then the cost of the exact Gauss–Newton algorithm with dogleg trust-region is given by

$$\begin{array}{rl} \mathcal{O}(\frac{3}{2}N^2R^2I^2) & \text{(Evaluating } J^{\mathrm{H}}J\text{)} \\ \mathcal{O}(2NRI^N) & \text{(Cogradient evaluation)} \\ \mathcal{O}(\frac{8}{3}N^3R^3I^3) & \text{(Solving the linear system)} \\ \mathcal{O}(\mathrm{it}_{\mathrm{gn}}2RI^N) & \text{(Function evaluation)} \\ \hline \mathcal{O}(2(N+\mathrm{it}_{\mathrm{gn}})RI^N + \frac{8}{3}N^3R^3I^3) & \text{flop/iteration.} \end{array}$$

In the proposed inexact adaptations of these nonlinear least squares methods, the Levenberg–Marquardt and Gauss–Newton steps are solved iteratively using a preconditioned conjugate gradient algorithm. To compute the matrix-vector product $J^{\mathrm{H}}J \cdot \boldsymbol{p}$, where $\boldsymbol{p}$ is defined as in Theorem 3.3, the matrices ${A^{(n)}}^{\mathrm{T}}\overline{A}^{(n)}$ and ${B^{(n)}}^{\mathrm{T}}\overline{A}^{(n)}$ should first be precomputed (the former need only be computed once). These products cost $\mathcal{O}(R^2\sum_{n=1}^{N} I_n)$ and $\mathcal{O}(2R^2\sum_{n=1}^{N} I_n)$ flop, respectively. The contribution of the diagonal blocks $\Pi^{(n,n)} \cdot \mathrm{vec}(B^{(n)})$ is obtained by first computing $W^{\{n\}}$ from $W^{\varnothing}$ and then the matrix products $B^{(n)} \cdot \overline{W}^{\{n\}}$, which costs a total of $\mathcal{O}(\sum_{n=1}^{N}(\frac{1}{2}R^2+2I_nR^2))$ flop. There are several ways to compute the contribution of the off-diagonal blocks. We propose to compute them row by row, which has the advantage that only one multiplication by $A^{(n)}$ is needed per row. The cost of the contribution of the off-diagonal blocks can then be shown to be $\mathcal{O}(\sum_{n=1}^{N}(\frac{1}{2}R^2 + \frac{1}{2}(N-1)R^2 + (N-1)R^2 + (N-2)R^2 + 2I_nR^2))$ flop. Applying a block-diagonal preconditioner costs a further $\mathcal{O}(\frac{1}{3}NR^3 + 2R^2\sum_{n=1}^{N} I_n)$ flop per conjugate gradient iteration. Let $\mathrm{it}_{\mathrm{cg}}$ be the number of conjugate gradient iterations required to solve the linear system to a prescribed accuracy, then the inexact Levenberg–Marquardt and Gauss–Newton algorithms cost

$$\begin{array}{rl} \mathcal{O}(2NRI^N) & \text{(Cogradient evaluation)} \\ \mathcal{O}(\mathrm{it}_{\mathrm{lm}}\mathrm{it}_{\mathrm{cg}}(\frac{5}{2}N^2R^2 + 8NR^2I + \frac{1}{3}NR^3)) & \text{(Solving the linear system)} \\ \mathcal{O}(\mathrm{it}_{\mathrm{lm}}2RI^N) & \text{(Function evaluation)} \\ \hline \mathcal{O}(2(N+\mathrm{it}_{\mathrm{lm}})RI^N + \mathrm{it}_{\mathrm{lm}}\mathrm{it}_{\mathrm{cg}}(\frac{5}{2}N^2R^2 & \\ \quad +8NR^2I + \frac{1}{3}NR^3)) & \text{flop/iteration} \end{array}$$

and

$$\begin{array}{rl} \mathcal{O}(2NRI^N) & \text{(Cogradient evaluation)} \\ \mathcal{O}(\mathrm{it}_{\mathrm{cg}}(\frac{5}{2}N^2R^2 + 8NR^2I + \frac{1}{3}NR^3)) & \text{(Solving the linear system)} \\ \mathcal{O}(\mathrm{it}_{\mathrm{gn}}2RI^N) & \text{(Function evaluation)} \\ \hline \mathcal{O}(2(N+\mathrm{it}_{\mathrm{gn}})RI^N + \mathrm{it}_{\mathrm{cg}}(\frac{5}{2}N^2R^2 & \\ \quad +8NR^2I + \frac{1}{3}NR^3)) & \text{flop/iteration,} \end{array}$$

respectively.

3.5 Numerical experiments

We compare the relative performance of the algorithms in Table 3.2 using the performance profiles proposed by Dolan and Moré [65]. In a first set of experiments, we are interested in the performance on complex rank-$(L_r, L_r, 1)$ block term decompositions. In the second set of experiments, we look at the performance on complex fourth-order canonical polyadic decompositions. The set of solvers S consists of the algorithms in Table 3.2, excluding the exact nonlinear least squares methods. For each experiment, we generate 50 sets of factor matrices which are used to generate the to-be decomposed tensors. Another 50 sets of factor matrices of the same dimensions are generated to be used as initialization for the algorithms. The set of problems P comprises all 2500 {tensor, initialization} pairs. The amount of work required by solver s to solve problem p is denoted by $t_{p,s}$ and is measured as an equivalent number of objective function evaluations (cf. Section 3.4.3). If solver s did not successfully solve problem p, we set $t_{p,s} = \infty$. The performance profile for solver s,

$$\rho_s(\tau) = \frac{|\{p \in P : t_{p,s} \leq 2^\tau \cdot (\min_{s \in S} t_{p,s})\}|}{|P|},$$

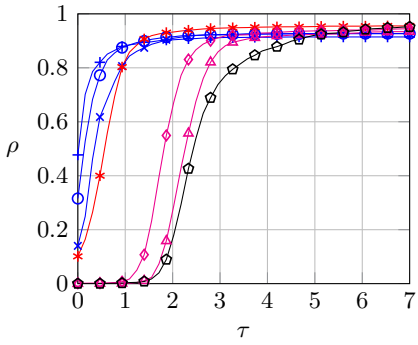
depicts the fraction of problems that solver s was able to solve with a required amount of work within a factor 2^τ of the most efficient solver for that problem. For the inexact nonlinear least squares methods, we use the block Jacobi preconditioner described in §3.4.2 and have opted to truncate the inner CG iterations after a maximum of 10 iterations or when a relative residual of 10^{-4} is obtained. We have experimented with other preconditioners such as Jacobi and symmetric successive over-relaxation (SSOR) but found these to be less efficient than the former. We let L-BFGS use a maximum of $m = 30$ rank-two updates to represent the approximate Hessian. The maximum number of iterations is 5000 for all algorithms except the inexact nonlinear least squares methods, which are allowed a maximum of 500 iterations per problem. The stop criteria are identical for all algorithms and are such that an algorithm stops if either the relative change between two successive iterates is less than a specified tolerance ϵ_z, or if the change in objective function value relative to the norm of the tensor is less than a specified tolerance ϵ_f. Formally, these conditions are described by $\|\boldsymbol{z}_k - \boldsymbol{z}_{k-1}\| \leq \epsilon_z \|\boldsymbol{z}_{k-1}\|$ and $f_{\text{BTD}}(\boldsymbol{z}_{k-1}) - f_{\text{BTD}}(\boldsymbol{z}_k) \leq \epsilon_f^2 \frac{1}{2} \|T\|_{\text{F}}^2$, respectively. All experiments were performed in MATLAB 7.14 (R2012a) on two hexacore Intel Xeon E5645 CPUs with 48 GB RAM. The underlying complex optimization algorithms were validated independently on different optimization problems and then adapted for the (rank-$L_r \circ$ rank-1) BTD.

3.5.1 The rank-$(L_r, L_r, 1)$ block term decomposition

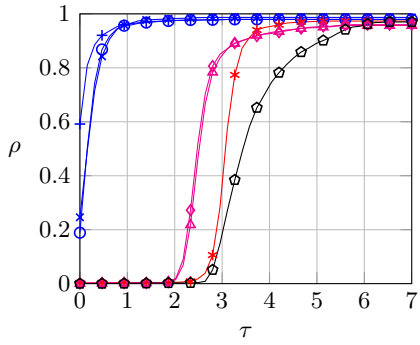
Without noise

For our first set of experiments, we generate 50 sets of factor matrices of which the real and imaginary part of the entries are drawn from either the standard normal distribution or from a uniform distribution in the interval $(0, 1)$. We refer to these two types of factor matrices as normally distributed and uniformly distributed factor matrices, respectively. Each set of factor matrices corresponds to a rank-$(L_r, L_r, 1)$ BTD in three terms, where $L_1 = L_2 = L_3 = 3$. For each set, we generate its associated full $10 \times 11 \times 12$ tensor, which is then scaled such that it has unit norm. Note that these decompositions are generically (essentially) unique since the factor matrices are generically full column rank [47]. For each tensor, another 50 sets of factor matrices of the same dimensions are generated using the same distribution as that of the factor matrices that generate the BTD. A second set of initializations is obtained by orthogonalizing pseudorandomly generated factor matrices with a QR factorization, which is a popular initialization method for ALS. We set $\epsilon_z = \epsilon_f = 10^{-12}$ and say that a decomposition is successful if and only if $\|\mathcal{F}_{\text{BTD}}(\boldsymbol{z}^*_{p,s})\|_{\text{F}} \leq 10^{-8}$, where $\boldsymbol{z}^*_{p,s}$ is the solution computed by solver s for problem p.

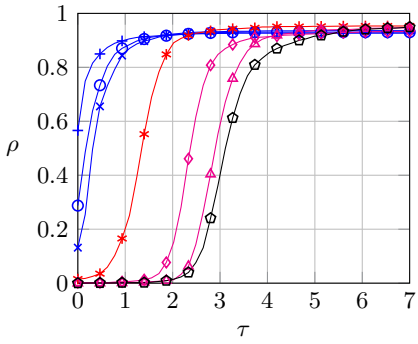
The results of the first four experiments are shown in Figure 3.4. Performance profiles are useful for judging relative difference in performance between algorithms on a set of problems but do not say anything about the difficulty of those problems. For each experiment, we use the median number of function evaluations of the best solver for each problem $t^* := \text{median}_{p \in P}(\min_{s \in S} t_{p,s})$ as a metric for the difficulty of the experiment's set of problems. Comparing Figure 3.4b with 3.4d we see that the quasi-Newton methods can be quite sensitive to the type of initialization. Paired with a uniformly distributed initialization they can require up to 16 times more flop than the most efficient solver, while with an orthogonalized initialization it can be up to 32 times. Second, the fraction of tensors successfully decomposed by the quasi-Newton methods, visible on the far right of each figure, decreases from about 95% to less than 90%. NCG's relatively poor performance is likely due to its limited ability of approximating the objective function's Hessian and possibly also because it is known to generate poorly scaled search directions. Although the relative performance between ALS and the NLS methods seems relatively robust with respect to the type of initialization, using orthogonalized factor matrices as initialization for tensors generated using uniformly distributed factor matrices does increase the overall required amount of work. The NLS methods are the most efficient in each of the experiments, and they excel most for difficult problems such as in the uniformly distributed factor matrices case.



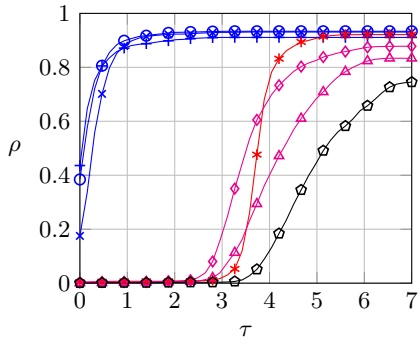
(a) normally distributed factor matrices and initialization ($t^* = 195.03$)



(b) uniformly distributed factor matrices and initialization ($t^* = 647.40$)

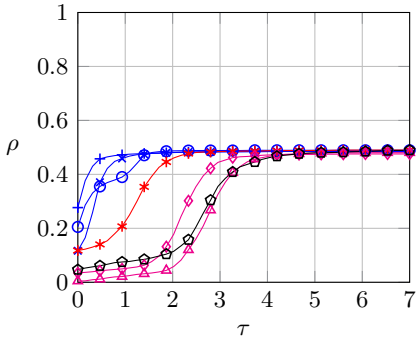


(c) normally distributed factor matrices, orthogonalized initialization ($t^* = 112.66$)

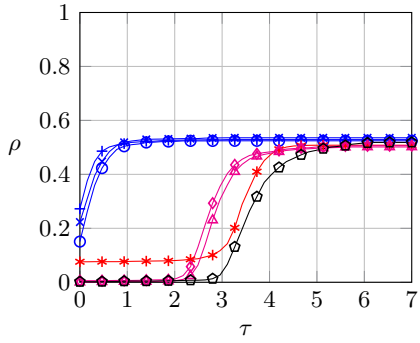


(d) uniformly distributed factor matrices, orthogonalized initialization ($t^* = 466.52$)

Figure 3.4: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = L_2 = L_3 = 3$, of tensors of dimensions $10 \times 11 \times 12$. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✳—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).

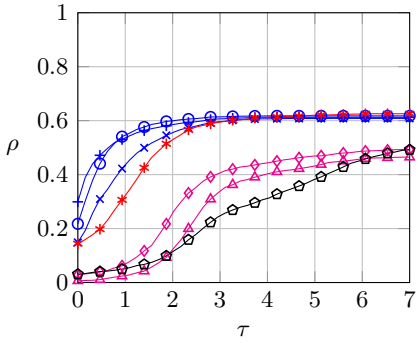


(a) normally distributed factor matrices and initialization ($t^* = 242.72$)

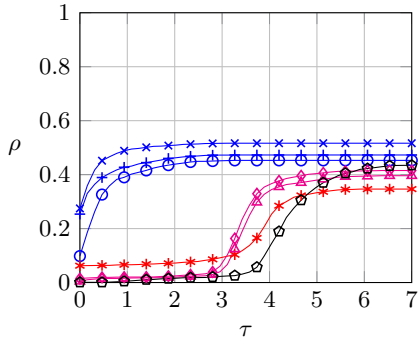


(b) uniformly distributed factor matrices and initialization ($t^* = 830.36$)

Figure 3.5: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 4$ and $L_2 = 5$, of tensors of dimensions $10 \times 11 \times 12$. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✳—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).



(a) normally distributed factor matrices and initialization ($t^* = 465.44$)



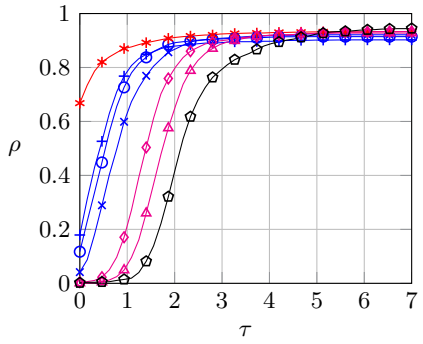
(b) uniformly distributed factor matrices and initialization ($t^* = 983.35$)

Figure 3.6: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$, of tensors of dimensions $10 \times 11 \times 12$. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✳—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).

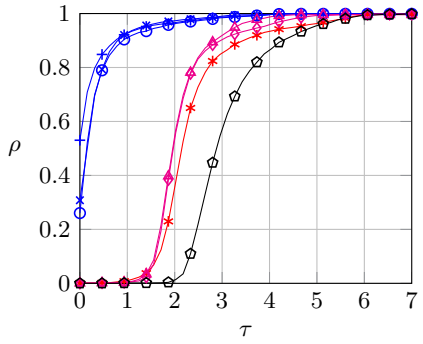
Now we examine what happens when the terms have different ranks. The following experiments are identical in setup to the previous experiments, except we generate the tensors using $L_1 = 4$ and $L_2 = 5$ (cf. Figure 3.5) and $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$ (cf. Figure 3.6). We omit the orthogonalized initialization as it has an adverse effect on the general unconstrained optimization methods. Two terms of different rank can create additional local minima. For example, in Figure 3.5, tensors are generated as the sum of a rank-$(4, 4, 1)$ term and a rank-$(5, 5, 1)$ term. Initially, neither of the two terms are strongly inclined to converge to a specific term of the decomposition and so we may expect that the rank-$(4, 4, 1)$ term of the initialization will start to converge to the rank-$(5, 5, 1)$ term of the decomposition and vice versa for roughly half of the initializations. Such a solution represents a local minimum because there is no way for the two terms to 'cross' each other without increasing the objective function value and explains why the fraction of successful decompositions is much lower than in Figure 3.4.

With noise

In the following experiments, we add complex Gaussian noise to the tensors before decomposing them. A noise term $\mathcal{N}$, of which the real and imaginary part are drawn from the standard normal distribution, is generated for each tensor. Each solver is then to decompose a tensor of the form $\mathcal{T}+\sigma\mathcal{N}$, where the scalar σ is chosen so that the signal-to-noise ratio (SNR) $20\log_{10}(\|\mathcal{T}\|_{\mathrm{F}}\|\mathcal{N}\|_{\mathrm{F}}^{-1})$ is 25 dB. We set $\epsilon_z = \epsilon_f = 10^{-5}$, which is necessary to avoid premature termination due to swamp-like behaviour. We say a decomposition is successful if and only if the approximate tensor $\mathcal{T}_s$ computed by solver s satisfies $20\log_{10}(\|\mathcal{T}\|_{\mathrm{F}}\|\mathcal{T} - \mathcal{T}_s\|_{\mathrm{F}}^{-1}) \geq 25\,\mathrm{dB}$. As in the noiseless case, we examine rank-$(L_r, L_r, 1)$ block term decompositions where $L_1 = L_2 = L_3 = 3$ (cf. Figure 3.7), $L_1 = 4$ and $L_2 = 5$ (cf. Figure 3.8) and $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$ (cf. Figure 3.9). Perhaps surprisingly, the fraction of successful decompositions can be significantly higher in the noisy case. For instance, compare the fraction of successful decompositions in Figure 3.5b, which is close to 50%, with the noisy case in Figure 3.8b, which is 100% for all solvers. A possible explanation for this marked improvement is that a noise term may eliminate many local minima by providing additional directions of descent in which the model can describe the noise term instead of the data. Moreover, we observe this effect for several types of factor matrices so long as the noise term is sufficiently large. Future research might investigate other methods of introducing new directions of descent, perhaps by means of a 'lifting' scheme [9], in which the model is modified instead of the data.

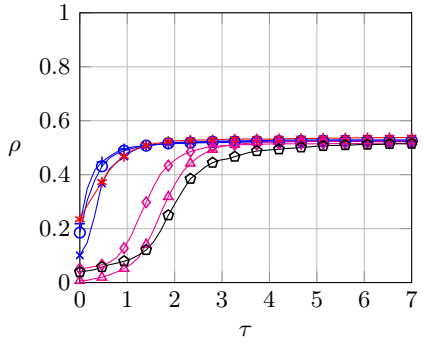


(a) normally distributed factor matrices and initialization ($t^* = 136.00$)

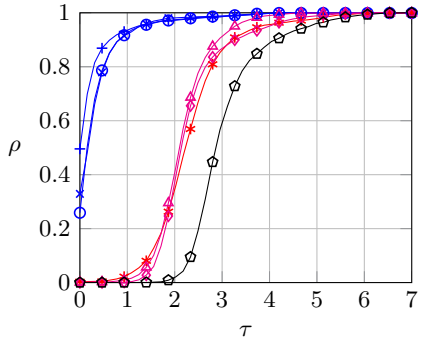


(b) uniformly distributed factor matrices and initialization ($t^* = 351.85$)

Figure 3.7: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = L_2 = L_3 = 3$, of noisy tensors of dimensions $10 \times 11 \times 12$ with an SNR of 25 dB. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✳—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).

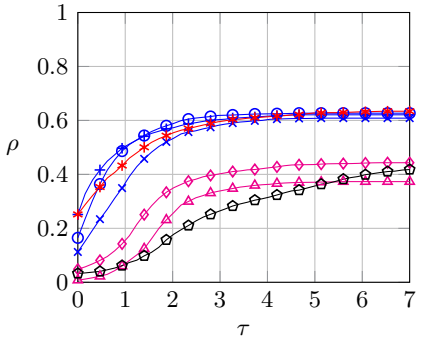


(a) normally distributed factor matrices and initialization ($t^* = 177.41$)

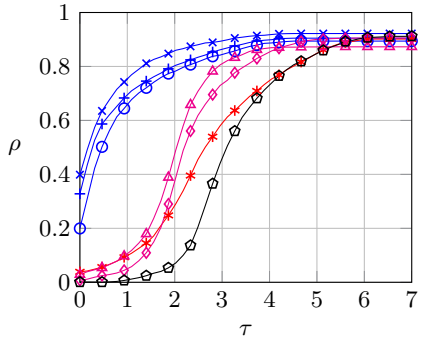


(b) uniformly distributed factor matrices and initialization ($t^* = 424.69$)

Figure 3.8: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 4$ and $L_2 = 5$, of noisy tensors of dimensions $10 \times 11 \times 12$ with an SNR of 25 dB. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✳—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).



(a) normally distributed factor matrices and initialization ($t^* = 352.00$)



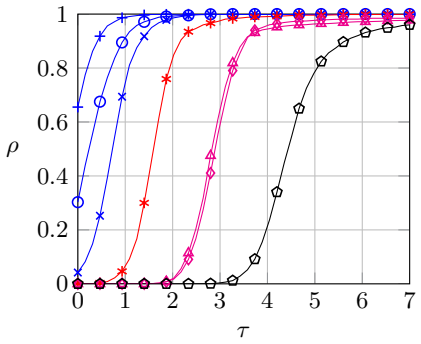
(b) uniformly distributed factor matrices and initialization ($t^* = 649.90$)

Figure 3.9: Performance profiles for complex rank-$(L_r, L_r, 1)$ block term decompositions, where $L_1 = 1$, $L_2 = 3$ and $L_3 = 5$, of noisy tensors of dimensions $10 \times 11 \times 12$ with an SNR of 25 dB. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—∗—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).

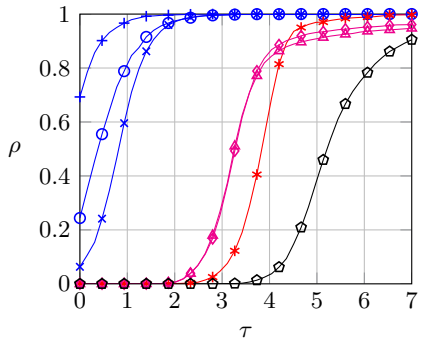
3.5.2 The canonical polyadic decomposition

In the previous experiments, ALS generally seemed to perform quite well in the normally distributed factor matrices case and much less so in the uniformly distributed factor matrices case. ALS is known to perform poorly when the terms of the decomposition are difficult to separate [39], which is the case when they are similar to each other in some sense. One metric for similarity between terms is the cosine of the angle between them. Normally distributed factor matrices tend to generate highly uncorrelated terms, while uniformly distributed factor matrices will naturally generate terms with some degree of collinearity between them. In our last two experiments, we look at the performance on decompositions with mildly to moderately collinear terms. First, we generate 50 sets of normally distributed factor matrices. Each set of factor matrices corresponds to a complex rank-4 CPD of a fourth-order tensor of dimensions $10 \times 10 \times 10 \times 10$. For each set, we then modify the mode-n vectors of the second rank-one term so that the angle with the corresponding mode-n vectors in the first rank-one term is fixed. More specifically, we set $\boldsymbol{a}_2^{(n)} \leftarrow \alpha \boldsymbol{a}_1^{(n)} + \beta \boldsymbol{a}_2^{(n)}$, where the scalars α and β are chosen so that we obtain vectors $\boldsymbol{a}_2^{(n)}$ for which $\|\boldsymbol{a}_2^{(n)}\| = \|\boldsymbol{a}_1^{(n)}\|$ and $\boldsymbol{a}_1^{(n)\mathrm{H}} \boldsymbol{a}_2^{(n)} = \|\boldsymbol{a}_1^{(n)}\| \|\boldsymbol{a}_2^{(n)}\| \gamma$ hold for all $1 \leq n \leq N$. Here, γ is the parameter that determines the cosine of the angle between $\boldsymbol{a}_1^{(n)}$ and

$\boldsymbol{a}_2^{(n)}$. We set γ equal to 0.65 and 0.85 in the first and second experiment, respectively. Effectively, this means cosine of the angle between the first two rank-one terms is $0.65^4 \approx 0.18$ and $0.85^4 \approx 0.52$, respectively. This corresponds to an angle of about 80° and 59°, respectively. For each tensor we generate another 50 sets of factor matrices of the same distribution as initialization for the solvers. We use the same stop criteria and condition for success as in the BTD experiments without noise. The performance profiles for the two experiments are plotted in Figure 3.10. It is clear that ALS is very sensitive to the amount of collinearity between terms, in contrast to the relatively robust performance of the optimization-based methods. Of the optimization-based methods, the inexact NLS algorithms clearly outperform the others. Moreover, they are well suited for parallelization, while ALS is an inherently sequential algorithm.



(a) normally distributed factor matrices and initialization with $\gamma = 0.65$ ($t^* = 69.36$)



(b) normally distributed factor matrices and initialization with $\gamma = 0.85$ ($t^* = 79.81$)

Figure 3.10: Performance profiles for complex rank-4 canonical polyadic decompositions of noisy tensors of dimensions $10 \times 10 \times 10 \times 10$ with some collinearity between two rank-one terms. Legend: GN-DL (—+—), LM (—×—), GN-CGS (—○—), ALS (—✱—), L-BFGS-DL (—△—), L-BFGS-MT (—◇—), and NCG-MT (—⬠—).

3.6 Conclusion

The rank-$(L_r, L_r, 1)$ block term decomposition is an emerging decomposition for signal processing and blind source separation. We introduced the (rank-L_r ∘ rank-1) block term decomposition as a generalization of the former and the canonical polyadic decomposition. We then developed several algorithms

for its computation, among which are alternating least-squares schemes, memory-efficient unconstrained gradient-based optimization methods such as nonlinear conjugate gradient and limited-memory BFGS with line search and trust-region frameworks, and matrix-free adaptations of nonlinear least squares methods such as Levenberg–Marquardt and Gauss–Newton. The resulting algorithms are all applicable to the canonical polyadic decomposition, the rank-$(L_r, L_r, 1)$ block term decomposition, and their generalized decomposition. Due to the multilinear structure of the objective function, the latter may be expected to converge close to quadratically, especially as the residuals decrease. Exploiting the structure of the Jacobian's Gramian has led to a significant decrease in computational complexity, compared to their exact counterparts, and reduced the memory cost to that of alternating least squares. The singularity of the Gramian is inherently handled by the built-in regularization of the conjugate gradient algorithm. Additionally, we increase the quality of the search directions generated by the conjugate gradient algorithm with a block Jacobi preconditioner. Numerical experiments confirm that these improvements make the inexact nonlinear least squares methods among the most efficient currently available. Furthermore, they are also robust with respect to the type of initialization, collinearity between terms, and the number of successful decompositions. The algorithms discussed in this chapter were implemented as part of the MATLAB toolbox Tensorlab [211].

Chapter 4

Exact line and plane search for tensor optimization

Line and plane searches are used as accelerators and globalization strategies in many optimization algorithms. We introduce a class of optimization problems called tensor optimization, which comprises applications ranging from tensor decompositions to least squares support tensor machines. We develop algorithms to efficiently compute the global minimizers of their line and plane search subproblems. Furthermore, we introduce scaled line and plane search, which compute an optimal scaling of the solution simultaneously with the optimal line or plane search step, and show that this scaling can be computed at almost no additional cost. Obtaining the global minimizers of (scaled) line and plane search problems often requires solving a bivariate or polyanalytic polynomial system. We show how to compute the isolated real solutions of bivariate polynomial systems and the isolated complex solutions of polyanalytic polynomial systems using a single generalized eigenvalue decomposition. Finally, we apply block term decompositions to the problem of blind multi-user detection-estimation in DS-CDMA communication to demonstrate that exact line and plane search can significantly reduce computation time of the workhorse tensor decomposition algorithm alternating least squares. This chapter is a slightly adapted version of the technical report [205].

4.1 Introduction

Line and plane search algorithms are an integral component of many optimization algorithms, both for accelerating and guaranteeing convergence to an optimum. In this chapter, we develop several types of exact line and plane search algorithms for a broad class of optimization problems, dubbed *tensor optimization* problems, in which the residual function is a polynomial tensor. In fact, these search algorithms may just as well be applied to the more general class of polynomial optimization problems, in which the objective function and constraints are (possibly polyanalytic) polynomials. However, we restrict our discussion to tensor optimization, which is interesting in and of itself and comprises applications ranging from tensor decompositions [112] to data fusion [7] and least squares support tensor machines [226]. Line and plane searches can often significantly reduce the length of so-called swamps [28,95,156,179], in which the optimization algorithm's convergence almost slows down to a halt.

The line and plane searches we develop are not limited to tasks which can be formulated as tensor optimization problems. In general, any nonlinear least squares problem can benefit from exact line or plane search by using a higher-order Taylor series expansion as an approximation of its residual function, effectively turning the nonlinear least squares problem into a tensor optimization problem.

Furthermore, we propose a variant of these search problems called *scaled* line and plane search which allows for an optimal scaling of the solution while simultaneously minimizing the objective function along a line or on a plane of search directions. Remarkably, we will see that for tensor optimization problems with a homogeneous residual function, the optimal scaling parameter can be obtained for almost no additional cost compared to a regular line or plane search.

Finally, we show that tensor optimization line and plane search subproblems are tantamount to solving a bivariate or polyanalytic polynomial system and present a method to compute the isolated solutions of bivariate and polyanalytic polynomial systems using a single generalized eigenvalue decomposition. In our numerical experiments, we compare a total of six line and plane search algorithms to accelerate complex tensor decomposition algorithms for blind detection-estimation of multi-user DS-CDMA communication using structured rank-$(L_r, L_r, 1)$ block term decompositions [53, 155]. The search algorithms are able to reduce the CPU time of the baseline algorithm, alternating least squares, by between 50% to 75%.

The chapter is organized as follows. In the next section we review our notation and introduce some basic definitions. Section 4.3 introduces tensor optimization

and how it can be used to compute various tensor decompositions. In Section 4.4 we define the tensor optimization (scaled) line and plane search subproblems, explain what type of objective functions they correspond to and show that in the case of a scaled line or plane search, the scaling parameter can be obtained at little additional cost. In Section 4.5 we present a method to compute the isolated real solutions of bivariate polynomial systems and isolated complex solutions of polyanalytic polynomial systems using a single generalized eigenvalue decomposition. In Section 4.6 we evaluate the performance of the proposed search algorithms with Monte Carlo simulations in which we blindly separate and equalize multi-user DS-CDMA signals using structured rank-$(L_r, L_r, 1)$ block term decompositions. We conclude the chapter in Section 4.7.

4.2 Notation and preliminaries

We refer to a tensor represented as a multidimensional array, given a choice of bases for each of these vector spaces. The order, or the number of modes, of a tensor is the number of indices associated with each element of that tensor. Vectors are denoted by boldface letters and are lower case, e.g., $\boldsymbol{a}$. Matrices are usually denoted by capital letters, e.g., A. Higher-order tensors are denoted by Euler script letters, e.g., $\mathcal{A}$. An entry of a vector $\boldsymbol{a}$, matrix A or tensor $\mathcal{A}$ is denoted by a_i, a_{ij} or $a_{ijk\ldots}$, depending on the number of modes. A colon is used to select all entries of a mode. For instance, $\boldsymbol{a}_{:j}$ corresponds to the jth column of a matrix A. When there is no confusion, we also use $\boldsymbol{a}_j$ to denote the jth column of the matrix A. Sequences are denoted by a superscript in parentheses, e.g., $\{A^{(n)}\}_{n=1}^{N}$. The superscripts $\cdot^{\mathrm{T}}$, $\cdot^{\mathrm{H}}$, $\cdot^{-1}$ and $\cdot^{\dagger}$ are used for the transpose, Hermitian conjugate, matrix inverse and Moore–Penrose pseudoinverse, respectively. The identity matrix of order n is denoted by $\mathbb{I}_n$ and the complex conjugate is denoted by an overbar, e.g., $\overline{a}$ is the complex conjugate of the scalar a. We use parentheses to denote the concatenation of two or more vectors, e.g., $(\boldsymbol{a}, \boldsymbol{b})$ is equivalent to $\begin{bmatrix}\boldsymbol{a}^{\mathrm{T}} & \boldsymbol{b}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$.

The vec$(\cdot)$ operator represents the column-wise vectorization of matrices and tensors. Let $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ and $\mathcal{U} \in \mathbb{C}^{J_1 \times \cdots \times J_N}$ be two Nth-order tensors. In a mode-n matricization $T_{(n)} \in \mathbb{C}^{I_n \times I_1 \cdots I_{n-1} I_{n+1} \cdots I_N}$ of the tensor $\mathcal{T}$, tensor element with indices $(i_1, \ldots, i_N)$ is mapped to matrix element (i_n, j) such that

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \text{ with } J_k = \prod_{\substack{m=0 \\ m \neq n}}^{k-1} I_m,$$

wherein we define $I_0 := 1$. The inner product of $\mathcal{T}$ and $\mathcal{U}$ (assuming $I_n \equiv J_n$) is defined as $\langle \mathcal{T}, \mathcal{U} \rangle = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} \overline{t}_{i_1 \cdots i_N} u_{i_1 \cdots i_N}$. The Frobenius norm is then

given by $\|\mathcal{T}\|_{\mathrm{F}} = \sqrt{\langle \mathcal{T}, \mathcal{T} \rangle}$. The outer product $\mathcal{T} \circ \mathcal{U}$ is the tensor defined by $(\mathcal{T} \circ \mathcal{U})_{i_1 \cdots i_P j_1 \cdots j_Q} = t_{i_1 \cdots i_P} u_{j_1 \cdots j_Q}$. The mode-$n$ product $\mathcal{T} \bullet_n A$ of the tensor $\mathcal{T}$ with a matrix $A \in \mathbb{C}^{J \times I_n}$ is the tensor defined by $(\mathcal{T} \bullet_n A)_{(n)} = A \cdot T_{(n)}$.

We will associate a coefficient vector $\boldsymbol{p} \in \mathbb{C}^{d_p+1}$ and coefficient matrices $Q \in \mathbb{C}^{(d_q^{(y)}+1)\times(d_q^{(x)}+1)}$ and $R \in \mathbb{C}^{(d_r^{(\overline{z})}+1)\times(d_r^{(z)}+1)}$ with polynomials $p(z)$, bivariate polynomials $q(x, y)$ and polyanalytic polynomials $r(z, \overline{z})$, defined by

$$p(z) := \begin{bmatrix} 1 & z & \cdots & z^{d_p} \end{bmatrix} \boldsymbol{p},$$

$$q(x, y) := \begin{bmatrix} 1 & y & \cdots & y^{d_q^{(y)}} \end{bmatrix} Q \begin{bmatrix} 1 & x & \cdots & x^{d_q^{(x)}} \end{bmatrix}^{\mathrm{T}} \text{ and}$$

$$r(z, \overline{z}) := \begin{bmatrix} 1 & \overline{z} & \cdots & \overline{z}^{d_r^{(\overline{z})}} \end{bmatrix} R \begin{bmatrix} 1 & z & \cdots & z^{d_r^{(z)}} \end{bmatrix}^{\mathrm{T}},$$

respectively. The term polyanalytic refers to the fact that the function is not analytic in its argument z, but is analytic when viewed as a function of its argument z and its complex conjugate $\overline{z}$ as a whole [206]. Here, $p_{d_p+1} \neq 0$ and $\max_i |q_{i,d_q^{(x)}+1}| \neq 0 \wedge \max_j |q_{d_q^{(y)}+1,j}| \neq 0$, so that d_p is the degree of $p(z)$ and $q_{d_q^{(x)}}$ and $q_{d_q^{(y)}}$ are the coordinate degrees of Q. An analogous condition holds for R, so that $d_r^{(z)}$ and $d_r^{(\overline{z})}$ are its coordinate degrees. The total degree of the polynomial $q(x, y)$ ($r(z, \overline{z})$) is defined as the largest value $i + j$ for which $q_{i+1,j+1} \neq 0$ ($r_{i+1,j+1} \neq 0$).

4.3 Tensor optimization

Tensor optimization is a special case of the more general class of polynomial optimization, in which the objective function and constraints are polynomials. Let $\mathcal{T}$ be a given tensor in $\mathbb{C}^{I_1 \times \cdots \times I_N}$ and let $\mathcal{M}(\boldsymbol{z}) : \mathbb{C}^K \to \mathbb{C}^{I_1 \times \cdots \times I_N}$ be a polynomial tensor-valued function. A polynomial tensor is a tensor whose elements are (possibly polyanalytic) multivariate polynomials. The degree $d_{\mathcal{M}}$ of a polynomial tensor $\mathcal{M}(\boldsymbol{z})$ is the maximum total degree of its elements. If each element's total degree is equal to $d_{\mathcal{M}}$, $\mathcal{M}$ is called homogeneous. Unconstrained tensor optimization problems are nonlinear least squares problems with a polynomial tensor-valued residual function $\mathcal{F}$ of the form

$$\underset{\boldsymbol{z} \in \mathbb{C}^K}{\text{minimize}} \quad \frac{1}{2} \|\mathcal{F}(\boldsymbol{z})\|_{\mathrm{F}}^2, \text{ where } \mathcal{F}(\boldsymbol{z}) := \mathcal{M}(\boldsymbol{z}) - \mathcal{T}. \tag{TO}$$

Among others, this model comprises the canonical polyadic decomposition (CPD) [33, 95], low multilinear rank approximation or Tucker approximation

(LMLRA) [112,232], block term decompositions (BTD) [46,47,55], data fusion as coupled matrix and tensor factorizations (CMTF) [7] and least squares support tensor machines (LS-STM) [226]. Due to the inner product, the objective function in (TO) is polyanalytic in $\boldsymbol{z}$, meaning it is function of both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ and its Taylor series in $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ is convergent for every point on its domain. More specifically, the objective function is a polyanalytic polynomial of total degree $2d_{\mathcal{M}}$ in K complex variables.

For instance, the CPD approximates a tensor by a sum of R rank-one tensors. Let $A^{(n)} \in \mathbb{C}^{I_n \times R}$, $n = 1, \ldots, N$, consist of nonzero columns, then

$$\mathcal{T} \approx \mathcal{M}_{\text{CPD}}(\boldsymbol{z}) := \sum_{r=1}^{R} \boldsymbol{a}_r^{(1)} \circ \cdots \circ \boldsymbol{a}_r^{(N)} \tag{CPD}$$

is a CPD of $\mathcal{T}$ in R rank-one tensors. The model $\mathcal{M}_{\text{CPD}}$ has degree N and its argument $\boldsymbol{z} \in \mathbb{C}^{\sum I_n R}$ concatenates the factor matrices as $(\text{vec}(A^{(1)}), \ldots, \text{vec}(A^{(N)}))$. In principle, the decomposition is only called canonical if it is exact and R is minimal. However, in the approximation sense the term can be interpreted to mean that R is the smallest integer for which the decomposition approximates the tensor sufficiently well.

To a large extent, the practical importance of the CPD stems from the fact that it is unique under relatively mild conditions. A CPD is called (essentially) unique when it is subject only to a permutation and scaling ambiguity. It is clear that one can arbitrarily permute the different rank-one terms. Also, the vectors in a single rank-one term may be arbitrarily scaled, as long as their product remains the same. The most well-known sufficient condition for uniqueness is due to Kruskal [118,119] and more recent results on uniqueness can be found in [67,68]. This striking property has been exploited in a wide range of applications in signal processing, including blind source separation [37,38,49], blind multi-user detection-estimation in DS-CDMA communication [1, 197], multiple-invariance sensor array processing [195], 3D radar clutter modeling and mitigation [157] and multi-dimensional harmonic retrieval [195].

One could also consider isolating the rth rank-one tensor's norm in the real scalar λ_r and constraining the vectors $\boldsymbol{a}_r^{(n)}$ to have unit norm, leading to the modified model

$$\mathcal{M}_{\text{mCPD}} := \sum_{r=1}^{R} \lambda_r \cdot \boldsymbol{a}_r^{(1)} \circ \cdots \circ \boldsymbol{a}_r^{(N)} \tag{mCPD}$$

of degree $N + 1$. This form of the CPD is useful in the context of CMTF, as we will see later. Although the scaling ambiguity is largely resolved in the modified model, it is still subject to a phase ambiguity if the decomposition is complex.

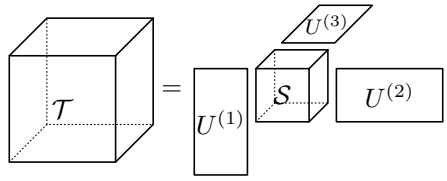


Figure 4.1: Low multilinear rank approximation of a third-order tensor.

In a LMLRA, a tensor is approximated by a tensor of low multilinear rank (cf. Figure 4.1). Let $\mathcal{S} \in \mathbb{C}^{R_1 \times \cdots \times R_N}$ and let $U^{(n)} \in \mathbb{C}^{I_n \times R_n}$, $n = 1, \ldots, N$, then

$$\mathcal{T} \approx \mathcal{M}_{\text{LMLRA}}(\boldsymbol{z}) := \mathcal{S} \bullet_1 U^{(1)} \bullet_2 \cdots \bullet_N U^{(N)} \tag{LMRLA}$$

is a rank-$(R_1, \ldots, R_N)$ LMLRA of $\mathcal{T}$. The model $\mathcal{M}_{\text{LMLRA}}(\boldsymbol{z})$ now has degree $N + 1$ and its argument $\boldsymbol{z} \in \mathbb{C}^{\prod R_n + \sum I_n R_n}$ concatenates the core tensor $\mathcal{S}$ and factor matrices $U^{(n)}$ as $(\text{vec}(\mathcal{S}), \text{vec}(U^{(1)}), \ldots, \text{vec}(U^{(N)}))$. The LMLRA offers a way to generalize subspace problems in signal processing to multi-way data, with applications in dimensionality reduction [56], feature extraction and classification [90, 139, 236] and noise reduction [108].

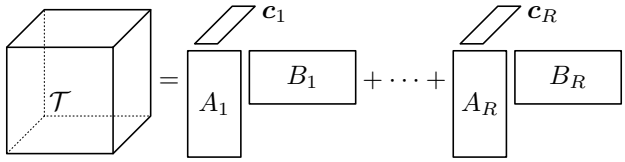


Figure 4.2: Rank-$(L_r, L_r, 1)$ block term decomposition of a third-order tensor.

The block term decompositions framework unifies the CPD and LMLRA by approximating tensors with a sum of low multilinear rank terms. Of particular interest is the rank-$(L_r, L_r, 1)$ BTD, which approximates a tensor by a sum of outer products of rank-L_r matrices and nonzero vectors (cf. Figure 4.2). The rank-$(L_r, L_r, 1)$ BTD combines the mild conditions for uniqueness of the CPD [47, 48] with the more general low-rank structure of a LMLRA to form a promising new candidate for blind source separation [48, 50, 53, 155, 156, 195–197]. Let $A_r \in \mathbb{C}^{I_1 \times L_r}$ and $B_r \in \mathbb{C}^{I_2 \times L_r}$, $r = 1, \ldots, R$, have rank L_r and let $C \in \mathbb{C}^{I_3 \times R}$ consist of nonzero columns, then

$$\mathcal{T} \approx \mathcal{M}_{\text{BTD}}(\boldsymbol{z}) := \sum_{r=1}^{R} (A_r \cdot B_r^{\text{T}}) \circ \boldsymbol{c}_r \tag{BTD}$$

is a rank-$(L_r, L_r, 1)$ BTD of the third-order tensor $\mathcal{T}$. The model $\mathcal{M}_{\text{BTD}}$ has degree three and its argument $\boldsymbol{z} \in \mathbb{C}^{(I_1 + I_2) \sum L_r + I_3 R}$ concatenates the factor matrices as $(\text{vec}(A_1), \ldots, \text{vec}(A_R), \text{vec}(B_1), \ldots, \text{vec}(B_R), \text{vec}(C))$.

The CPD, LMLRA and rank-$(L_r, L_r, 1)$ BTD are all examples of multilinear polynomial tensors. A function is said to be multilinear in its argument $\boldsymbol{z}$ if for all i it is linear in z_i when the remaining variables z_j $(j \neq i)$ are fixed. However, this property can be lost by imposing symmetry. For example, INDSCAL [33] is a special case of the CPD for third-order tensors in which $A^{(1)} \equiv A^{(2)}$. Its model is a polynomial tensor, but is no longer multilinear.

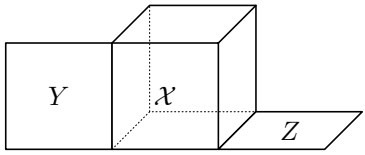


Figure 4.3: Coupled data sets of different orders. The tensor $\mathcal{X}$ has modes restaurant $\times$ meal $\times$ customer and contains ratings assigned by a customer to a meal at a certain restaurant. The matrix Y has modes restaurant $\times$ category, is coupled along the first mode with $\mathcal{X}$ and contains information about the type of restaurant. The matrix Z has modes customer $\times$ customer, is coupled along the third mode with $\mathcal{X}$ and contains social network information between customers.

In CMTF, the objective is to minimize a weighted sum of tensor optimization objective functions $\sum_k \lambda_k \|\mathcal{F}_k(\boldsymbol{z})\|_{\mathrm{F}}^2$, wherein $\mathcal{F}_k$ are coupled polynomial tensor-valued residual functions. Data fusion in the form of CMTF is readily cast as a tensor optimization problem by vectorizing and concatenating the residual functions $\mathcal{F}_k$ into a single polynomial tensor-valued residual function as

$$\mathcal{F}_{\mathrm{CMTF}}(\boldsymbol{z}) := \left(\sqrt{\lambda_1}\, \mathrm{vec}(\mathcal{F}_1(\boldsymbol{z})), \ldots, \sqrt{\lambda_K}\, \mathrm{vec}(\mathcal{F}_K(\boldsymbol{z})) \right).$$

The polynomial tensor-valued function $\mathcal{F}_{\mathrm{CMTF}}$ can then be separated into a model $\mathcal{M}_{\mathrm{CMTF}}(\boldsymbol{z})$ and a constant $\mathcal{T}$ as in (TO). The degree of the model $\mathcal{M}_{\mathrm{CMTF}}$ is then equal to the maximum degree of its constituent models.

Consider for example the coupled data set shown in Figure 4.3 [6]. One way to jointly analyse this data set would be to compute a CMTF consisting of three modified CPD models (mCPD) as follows. If the tensor $\mathcal{X}$ is decomposed using the factor matrices $A^{(1)}$, $A^{(2)}$, $A^{(3)}$ and scaling vector $\boldsymbol{\lambda}$, the matrices Y and Z can be coupled with $\mathcal{X}$ by decomposing them using the modified CPD models $B \cdot \mathrm{diag}(\boldsymbol{\mu}) \cdot A^{(1)\mathrm{T}}$ and $A^{(3)} \cdot \mathrm{diag}(\boldsymbol{\nu}) \cdot C^{\mathrm{T}}$, respectively. The absolute values of the scaling vectors $\boldsymbol{\lambda}$, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$ give an indication of how strongly certain factors are shared between the two matrices and the tensor. Note that although we have treated the three scaling vectors separately from the factor matrices, they may equally well be interpreted as factor matrices of canonical polyadic decompositions of tensors of one order higher than shown in Figure 4.3. More specifically, $\boldsymbol{\lambda}$, $\boldsymbol{\mu}$ and $\boldsymbol{\nu}$ are the factor matrices corresponding to the

singleton dimension following the last nonsingleton dimension of $\mathcal{X}$, Y and Z, respectively.

4.4 Exact line and plane search

In optimization, two-dimensional subspace minimization is a prevalent subproblem in trust-region globalization strategies. Given a vector-valued residual function F, a current iterate $\boldsymbol{z}_k$ and two search directions $\Delta\boldsymbol{z}_1$ and $\Delta\boldsymbol{z}_2$, the objective is to minimize $\|F(\boldsymbol{z})\|$ in the plane spanned by the two search directions, subject to a norm constraint on the step. The corresponding optimization problem

$$\begin{aligned} \underset{\alpha,\,\beta\,\in\,\mathbb{R}}{\text{minimize}} \quad & \frac{1}{2}\left\|F(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\right\|^2 \\ \text{subject to} \quad & \|\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2\|^2 \leq \delta^2 \end{aligned}$$

is often simplified using a linear approximation of F around $\boldsymbol{z}_k$ in order to make the problem tractable. The resulting Gauss–Newton two-dimensional subspace minimization problem [32]

$$\begin{aligned} \underset{\alpha,\,\beta\,\in\,\mathbb{R}}{\text{minimize}} \quad & \frac{1}{2}\left\|F(\boldsymbol{z}_k) + \frac{\partial F(\boldsymbol{z}_k)}{\partial \boldsymbol{z}^T}\cdot(\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\right\|^2 \\ \text{subject to} \quad & \|\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2\|^2 \leq \delta^2 \end{aligned}$$

is equivalent to solving a bivariate polynomial system with polynomials of total degree two. Instead of a linear approximation, one could consider using a second-, third- or higher-order Taylor series approximation of F. In that case, the approximation is a polynomial tensor, and we will see that the two-dimensional subspace minimization problem still leads to a bivariate polynomial system, but where one of the two polynomials now has a total degree higher than two.

From here on, $\mathcal{F}$ is assumed to be a polynomial tensor (approximation of a) residual function. Given a current iterate $\boldsymbol{z}_k$ and two descent directions $\Delta\boldsymbol{z}_1$

and $\Delta \boldsymbol{z}_2$, we refer to the following tensor optimization subproblems

$$\underset{\alpha}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1)\|_{\text{F}}^2 \tag{LS}$$

$$\underset{\alpha,\,\gamma}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\gamma\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1)\|_{\text{F}}^2 \tag{SLS}$$

$$\underset{\alpha,\,\beta}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\|_{\text{F}}^2 \tag{PS}$$

$$\underset{\alpha,\,\beta,\,\gamma}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\gamma\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\|_{\text{F}}^2 \tag{SPS}$$

as a line search (LS), scaled line search (SLS), plane search (PS) and scaled plane search (SPS), respectively. We append $\mathbb{R}$ or $\mathbb{C}$ to the optimization problem's acronym to distinguish over which field the solution is sought, e.g., a real line search is denoted by LS-$\mathbb{R}$, while a complex scaled line search would be denoted by SLS-$\mathbb{C}$.

Line and plane search problems play an important part in algorithms that solve tensor optimization problems. Many optimization-based algorithms rely on either line or plane searches [159] and other popular algorithms for tensor decompositions such as alternating least squares have been combined with line searches to mitigate the effect of so-called swamps [28, 120, 144, 179]. In particular, LS-$\mathbb{R}$ and LS-$\mathbb{C}$ have received some attention recently in the case of a CPD [34, 156, 179].

In signal processing, the recovery of data symbols transmitted through a distorting medium is a fundamental problem. Assuming that zero-mean data symbols s_n are transmitted through a linear time-invariant time-dispersive channel of order M, the channel output measured with an antenna array of length P can be modelled as

$$\boldsymbol{x}_n = \sum_{k=0}^{M} \boldsymbol{h}_k s_{n-k} + \boldsymbol{v}_n, \tag{4.1}$$

where $H = \begin{bmatrix}\boldsymbol{h}_0 & \cdots & \boldsymbol{h}_M\end{bmatrix} \in \mathbb{C}^{P\times(M+1)}$ contains the channel impulse responses for each of the P antennas and $\boldsymbol{v}_n \in \mathbb{C}^{P\times 1}$ is an additive noise term. The single-input multiple-output (SIMO) model (4.1) reduces to the single-input single-output (SISO) model for $P = 1$. The SIMO model can also be obtained by exploiting temporal diversity (e.g., time oversampling) instead of spatial diversity (e.g., an antenna array), and can easily be extended to the multiple-input (MIMO) case. To recover the data symbols s_n, a linear equalizer $\boldsymbol{f} \in \mathbb{C}^{PL}$ with finite impulse response spanning L measurements is used which produces

the output symbols $y_n := \boldsymbol{f}^{\mathrm{H}}\tilde{\boldsymbol{x}}_n$, where $\tilde{\boldsymbol{x}}_n := (\boldsymbol{x}_n, \ldots, \boldsymbol{x}_{n-L+1})$. One of the most widespread blind channel equalization principles is the constant modulus (CM) [80], which looks for the filter $\boldsymbol{f}$ which minimizes a tensor optimization problem (TO) with a residual function $\mathcal{F}_{\mathrm{CM}}$ defined by

$$(\mathcal{F}_{\mathrm{CM}}(\boldsymbol{f}))_n := |y_n|^2 - 1.$$

Furthermore, it is not hard to show that the steepest ascent direction at the kth iterate $\boldsymbol{f}_k$ of the tensor optimization objective function is given by

$$\boldsymbol{g}_k := 2\left(\frac{\partial \operatorname{vec}(\mathcal{F}_{\mathrm{CM}})}{\partial \boldsymbol{f}^{\mathrm{T}}}(\boldsymbol{f}_k)\right)^{\mathrm{H}} \cdot \mathcal{F}_{\mathrm{CM}}(\boldsymbol{f}_k).$$

The CM criterion often suffers from the existence of numerous local minima, usually associated with different equalization delays, in the equalizer parameter space [62]. An exact line search may be useful to help reduce the chances of converging to an undesired local minimum. By means of an example, we choose $M = 1$, $P = 2$ and $L = 2$ and consider the measurements $\boldsymbol{x}_2 := (1+i, -i)$, $\boldsymbol{x}_1 := (1, -1)$ and $\boldsymbol{x}_0 := (1, 1-i)$ and an initial filter $\boldsymbol{f}_0 = (0.2413 + 0.1593i, 0.3496 + 0.0858i, 0.2875 + 0.3305i, 0.5979 + 0.5929i)$. The line search objective function (LS) of the CM tensor optimization problem along the steepest descent direction $-\boldsymbol{g}_0$ at $\boldsymbol{f}_0$ is the polynomial of degree four shown in Figure 4.4. Whereas an approximate line search will likely choose a suboptimal line search parameter α near 0.4, an exact line search would find the global minimizer $\alpha^* \approx 12.6429$. Previous work has shown that the optimal line search parameter can be obtained by computing the roots of a polynomial of degree three [253]. Viewed in the context of tensor optimization, scaled line searches and even plane searches may also be applied efficiently to the problem of constant modulus equalization.

Table 4.1 gives an overview of the type of objective function the different kinds of search problems correspond to for the two most common choices of the solution field. From the table, it is clear that LS-$\mathbb{R}$ and SLS-$\mathbb{R}$ are easy problems to solve since their stationary points are just the roots of the derivative of their polynomial and rational objective functions, respectively. Obtaining the global minimizer of the other four search problems is less straightforward because their objective functions are either bivariate or polyanalytic. Remarkably, the scaling parameter γ in SLS and SPS can be obtained at almost no additional cost compared to LS and PS, assuming the objective function is homogeneous and more expensive to compute than solving the underlying polynomial system.

The following two subsections explain how to arrive at the type of objective function given by Table 4.1. The last subsection describes how a norm constraint on the step can be imposed on PS-$\mathbb{R}$, which generalizes two-dimensional subspace minimization from a linear approximation of the residual function to a

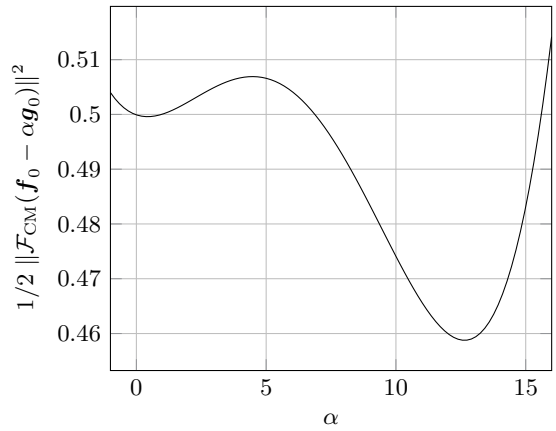


Figure 4.4: The LS-$\mathbb{R}$ objective function of the constant modulus problem along the steepest descent direction $-\boldsymbol{g}_0$ at the iterate $\boldsymbol{f}_0$ defined by the measurements $\boldsymbol{x}_2$, $\boldsymbol{x}_1$ and $\boldsymbol{x}_0$.

Problem \ Field	$\mathbb{R}$	$\mathbb{C}$
LS	degree $2d_{\mathcal{M}}$ polynomial	coordinate degree $d_{\mathcal{M}}$ polyanalytic polynomial
SLS*	degree $2d_{\mathcal{M}}$ rational function	coordinate degree $d_{\mathcal{M}}$ polyanalytic rational function
PS	total degree $2d_{\mathcal{M}}$ bivariate polynomial	—
SPS*	total degree $2d_{\mathcal{M}}$ bivariate rational function	—

*Assuming $\mathcal{M}$ is homogeneous.

Table 4.1: The type of the objective function each of the search problems correspond to for two choices of the solution field.

polynomial tensor approximation such as a higher-order Taylor series expansion. Section 4.5 then describes how to obtain the global minimizers of (S)LS-$\mathbb{C}$ and (S)PS-$\mathbb{R}$ by recasting them as a generalized eigenvalue problem.

4.4.1 Scaled line search

Since LS-$\mathbb{R}$, LS-$\mathbb{C}$ and SLS-$\mathbb{R}$ are special cases of SLS-$\mathbb{C}$, we first focus on SLS-$\mathbb{C}$. Separate $\mathcal{F}$ into a model $\mathcal{M}$ and a constant $\mathcal{T}$ as in (TO) and expand the objective function in (SLS) as

$$\frac{1}{2}(\|\mathcal{T}\|_{\mathrm{F}}^2 + \|\mathcal{M}(\gamma \boldsymbol{z}_k + \alpha \Delta \boldsymbol{z}_1)\|_{\mathrm{F}}^2 - \langle \mathcal{T}, \mathcal{M}(\gamma \boldsymbol{z}_k + \alpha \Delta \boldsymbol{z}_1)\rangle - \langle \mathcal{M}(\gamma \boldsymbol{z}_k + \alpha \Delta \boldsymbol{z}_1), \mathcal{T}\rangle).$$

Under the assumption that $\mathcal{M}$ is homogeneous[1] with degree $d_{\mathcal{M}}$, we introduce a change of variables $c := \gamma^{d_{\mathcal{M}}}$ and $a := \alpha/\gamma$ so the objective function can be written as

$$f(a, \overline{a}, c, \overline{c}) := \frac{1}{2}(\|\mathcal{T}\|_{\mathrm{F}}^2 + |c|^2 q(a, \overline{a}) - cp(a) - \overline{c}\,\overline{p(a)}),$$

where $p(a) := \langle \mathcal{T}, \mathcal{M}(\boldsymbol{z}_k + a\Delta \boldsymbol{z}_1)\rangle$ and $q(a, \overline{a}) := \|\mathcal{M}(\boldsymbol{z}_k + a\Delta \boldsymbol{z}_1)\|_{\mathrm{F}}^2$ are two univariate polynomials. In the case of a line search, $c \equiv \gamma \equiv 1$, $a \equiv \alpha$ and (LS) is equivalent to

$$\underset{\alpha \in \mathbb{R}}{\text{minimize}} \quad f(\alpha) \tag{LS-$\mathbb{R}$}$$

$$\underset{\alpha \in \mathbb{C}}{\text{minimize}} \quad f(\alpha, \overline{\alpha}) \tag{LS-$\mathbb{C}$}$$

for LS-$\mathbb{R}$ and LS-$\mathbb{C}$, respectively. In the case of a scaled line search, notice that f is quadratic in c so that a global optimum (a^*, c^*) must satisfy $c^* = \mathrm{Re}\{p(a^*)\}/q(a^*)$ and $c^* = \overline{p(a^*)}/q(a^*, \overline{a}^*)$ for SLS-$\mathbb{R}$ and SLS-$\mathbb{C}$, respectively. Substituting back in f reveals that a^* is the solution of

$$\underset{a \in \mathbb{R}}{\text{minimize}} \quad -\frac{\mathrm{Re}\{p(a)\}^2}{q(a)} \tag{SLS-$\mathbb{R}$}$$

$$\underset{a \in \mathbb{C}}{\text{minimize}} \quad -\frac{|p(a)|^2}{q(a, \overline{a})} \tag{SLS-$\mathbb{C}$}$$

for SLS-$\mathbb{R}$ and SLS-$\mathbb{C}$, respectively.

[1] The homogeneity requirement, which is satisfied by most tensor decompositions, is only necessary for scaled line and plane search.

4.4.2 Scaled plane search

Analogously to scaled line search, expand the objective function in (SPS) as

$$\begin{aligned}\frac{1}{2}(\|\mathcal{T}\|_{\mathrm{F}}^2 + \|\mathcal{M}(\gamma \boldsymbol{z} + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\|_{\mathrm{F}}^2 \\ - \langle \mathcal{T}, \mathcal{M}(\gamma \boldsymbol{z} + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\rangle \\ - \langle \mathcal{M}(\gamma \boldsymbol{z} + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2), \mathcal{T}\rangle)\end{aligned}$$

and assume that $\mathcal{M}$ is homogeneous with degree $d_{\mathcal{M}}$ to introduce a change of variables $c := \gamma^{d_{\mathcal{M}}}$, $a := \alpha/\gamma$ and $b := \beta/\gamma$. For real α, β and γ, the objective function can then be written as

$$f(a, b, c) := \frac{1}{2}(\|\mathcal{T}\|_{\mathrm{F}}^2 + c^2 q(a, b) - c(p(a, b) + \overline{p}(a, b))),$$

where $p(a, b) := \langle \mathcal{T}, \mathcal{M}(\boldsymbol{z} + a\Delta\boldsymbol{z}_1 + b\Delta\boldsymbol{z}_2)\rangle$ and $q(a, b) := \|\mathcal{M}(\boldsymbol{z} + a\Delta\boldsymbol{z}_1 + b\Delta\boldsymbol{z}_2)\|_{\mathrm{F}}^2$ are two bivariate polynomials. In the case of a plane search, $c \equiv \gamma \equiv 1$, $a \equiv \alpha$, $b \equiv \beta$ and (PS) is equivalent to

$$\underset{\alpha,\,\beta\,\in\,\mathbb{R}}{\text{minimize}} \quad f(\alpha, \beta). \tag{PS-$\mathbb{R}$}$$

In the case of a scaled plane search, notice that f is quadratic in c so that a global optimum (a^*, b^*, c^*) must satisfy $c^* = \mathrm{Re}\{p(a^*, b^*)\}/q(a^*, b^*)$. Substituting back in f reveals that (a^*, b^*) is the solution of

$$\underset{a,\,b\,\in\,\mathbb{R}}{\text{minimize}} \quad -\frac{\mathrm{Re}\{p(a, b)\}^2}{q(a, b)}. \tag{SPS-$\mathbb{R}$}$$

4.4.3 Higher-order two-dimensional subspace minimization

When $\mathcal{F}$ is a polynomial tensor approximation of a nonlinear tensor-valued residual function F, it may be desirable to impose a norm constraint on the plane search step to avoid the next iterate from straying too far from where $\mathcal{F}$ is considered to be an accurate approximation of F. For example, $\mathcal{F}$ could be a higher-order Taylor series expansion of F. Given a trust-region radius $\delta \in \mathbb{R}_+^0$, the higher-order two-dimensional subspace minimization problem is defined as

$$\begin{aligned}\underset{\alpha,\,\beta\,\in\,\mathbb{R}}{\text{minimize}} \quad & \frac{1}{2}\|\mathcal{F}(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\|_{\mathrm{F}}^2 \\ \text{subject to} \quad & \|\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2\|^2 \leq \delta^2\end{aligned}.$$

The case where the norm constraint is not active corresponds to solving PS-$\mathbb{R}$. When the norm constraint is active, it can be incorporated into the objective function using the method of Lagrange multipliers. Write $\mathcal{F}(\boldsymbol{z})$ as $\mathcal{M}(\boldsymbol{z}) - \mathcal{T}$, let $p(\alpha, \beta) := \langle \mathcal{T}, \mathcal{M}(\boldsymbol{z} + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\rangle$, $q(\alpha, \beta) := \|\mathcal{M}(\boldsymbol{z} + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2)\|_{\mathrm{F}}^2$ and $r(\alpha, \beta) := \|\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2\|^2$ and introduce the Lagrange multiplier λ to obtain the Lagrangian

$$\frac{1}{2}(\|\mathcal{T}\|_{\mathrm{F}}^2 + q(\alpha, \beta) - p(\alpha, \beta) - \overline{p}(\alpha, \beta)) + \frac{\lambda}{2}(r(\alpha, \beta) - \delta^2).$$

Let $s(\alpha, \beta) := q(\alpha, \beta) - 2\mathrm{Re}\{p\}(\alpha, \beta)$. Setting the Lagrangian's gradient equal to zero and eliminating λ, it follows that the global minimizer (α^*, β^*) is a solution of the bivariate polynomial system

$$\begin{cases} \frac{\partial r(\alpha,\beta)}{\partial \alpha}\frac{\partial s(\alpha,\beta)}{\partial \beta} - \frac{\partial r(\alpha,\beta)}{\partial \beta}\frac{\partial s(\alpha,\beta)}{\partial \alpha} = 0 \\ r(\alpha, \beta) - \delta^2 = 0 \end{cases} \tag{4.2}$$

in which the polynomials are of total degree $2d_{\mathcal{M}}$ and 2, respectively.

4.5 Solving systems of bivariate and polyanalytic polynomials

The two search problems LS-$\mathbb{R}$ and SLS-$\mathbb{R}$ are easily solved by computing the roots of their objective function's derivative, which is simply a univariate polynomial. On the other hand, the four search problems (S)LS-$\mathbb{C}$ and (S)PS-$\mathbb{R}$ give rise to polyanalytic and bivariate polynomial systems, respectively (cf. Table 4.1). For example, the LS-$\mathbb{C}$ objective function is a polyanalytic polynomial $f(\alpha, \overline{\alpha})$ of coordinate degree $d_{\mathcal{M}}$ and its stationary points are the solutions of the polyanalytic polynomial system

$$\frac{\partial f(\alpha, \overline{\alpha})}{\partial \alpha} = \frac{\partial f(\alpha, \overline{\alpha})}{\partial \overline{\alpha}} = 0,$$

where $\frac{\partial}{\partial \alpha}$ ($\frac{\partial}{\partial \overline{\alpha}}$) is a Wirtinger derivative and acts as partial derivative with respect to α ($\overline{\alpha}$), while treating $\overline{\alpha}$ (α) as constant [206]. The PS-$\mathbb{R}$ objective function is a bivariate polynomial $f(\alpha, \beta)$ of total degree $2d_{\mathcal{M}}$ and its stationary points are the solutions of the bivariate polynomial system

$$\frac{\partial f(\alpha, \beta)}{\partial \alpha} = \frac{\partial f(\alpha, \beta)}{\partial \beta} = 0.$$

Of the many techniques to solve systems of polynomials [30,74,131,146,159,238], we will focus on resultant-based methods [69], which originate from algebraic

geometry and are used to eliminate variables from systems of equations. The Sylvester resultant is perhaps the most well-known resultant formulation for systems of bivariate polynomials. Typically, the technique is used to eliminate one variable, say y, from a bivariate polynomial system $f(x,y) = g(x,y) = 0$, leaving us with a so-called resultant polynomial $r(x)$. If (x^*, y^*) is a root of the system $f(x,y) = g(x,y) = 0$, then x^* is also a root of the resultant $r(x)$.

At least two approaches can be applied to recover the corresponding y-coordinates. The first is to eliminate both x and y separately to obtain two resultants $r(x)$ and $s(y)$ [23, 63, 189]. Let x^* and y^* be roots of $r(x)$ and $s(y)$, respectively. For each such pair (x^*, y^*), the task is then to determine whether or not that pair constitutes a root of the system $f(x,y) = g(x,y) = 0$. The second approach is often more efficient but also less robust. The roots of the resultant $r(x)$ are computed as the solution of an eigenvalue problem, and then these roots are lifted to obtain the corresponding y-coordinates by solving an additional smaller eigen-problem for each x-coordinate [31, 41, 141]. However, the lifting step requires estimating the root's multiplicity. Furthermore, for both approaches, some real roots may have a small imaginary part due to numerical rounding errors. Detecting which roots are real and removing candidate roots which are not roots of the system $f(x,y) = g(x,y) = 0$ are the two most delicate steps in numerical algorithms for solving polynomial systems based on resultants.

However, we shall see that it is possible to compute the real (x^*, y^*) roots of the system $f(x,y) = g(x,y) = 0$ simultaneously as the solution of a single eigen-problem [210]. In fact, the same method can be applied to solve systems of polyanalytic polynomials $f(z,\overline{z}) = g(z,\overline{z}) = 0$. The key is that Sylvester's resultant can also be used to eliminate a complex variable from a polyanalytic polynomial system, as shown by the following lemma.

Lemma 4.1. *Let $f(z,\overline{z})$ and $g(z,\overline{z})$ be two nonconstant polyanalytic polynomials. If $f(z,\overline{z})$ and $g(z,\overline{z})$ have a common zero z^*, then there are nonzero polynomials $s(\overline{z})$ and $t(\overline{z})$ satisfying*

$$f(z^*,\overline{z})s(\overline{z}) + g(z^*,\overline{z})t(\overline{z}) \equiv 0, \tag{4.3}$$

where $d_s < d_g^{(\overline{z})}$ and $d_t < d_f^{(\overline{z})}$.

Proof. Write $f(z,\overline{z})$ and $g(z,\overline{z})$ as

$$f(z,\overline{z}) = \sum_{i=0}^{d_f^{(\overline{z})}} f_i(z)\overline{z}^i \quad \text{and} \quad g(z,\overline{z}) = \sum_{i=0}^{d_g^{(\overline{z})}} g_i(z)\overline{z}^i$$

where $f_i(x)$ ($g_i(x)$) is the polynomial in x associated with the coefficient vector $\boldsymbol{f}_{i+1,:}$ ($\boldsymbol{g}_{i+1,:}$). Since $f(z^*, \overline{z}^*) = g(z^*, \overline{z}^*) = 0$, $f(z^*, \overline{z})$ and $g(z^*, \overline{z})$ will have a common factor $k(\overline{z}) := (\overline{z} - \overline{z}^*)^n$ for some positive integer n. We have that $\frac{f(z^*,\overline{z})}{k(\overline{z})} g(z^*, \overline{z}) = \frac{g(z^*,\overline{z})}{k(\overline{z})} f(z^*, \overline{z})$. Set $s(\overline{z}) := g(z^*, \overline{z})/k(\overline{z})$ and $t(\overline{z}) := -f(z^*, \overline{z})/k(\overline{z})$. □

In other words, if z^* is a common root of $f(z, \overline{z}) = g(z, \overline{z}) = 0$, then once f and g are partially evaluated in z^*, there exist nonzero polynomials $s(\overline{z})$ and $t(\overline{z})$ such that the polynomial $f(z^*, \overline{z})s(\overline{z}) + g(z^*, \overline{z})t(\overline{z})$ is identically zero. Equation (4.3) can be rewritten as a set of linear equations in terms of the coefficient vectors $\boldsymbol{s}$ and $\boldsymbol{t}$ as

$$S^{f,g}(z^*) \cdot \begin{bmatrix} \boldsymbol{s} \\ \boldsymbol{t} \end{bmatrix} = \mathbf{0}, \tag{4.4}$$

where the Sylvester matrix $S^{f,g}(z)$ is a polynomial matrix defined as

$$S^{f,g}(z) := \left[\begin{array}{ccc|ccc} f_0(z) & & & g_0(z) & & \\ \vdots & \ddots & & \vdots & \ddots & \\ f_{d_f^{(\overline{z})}}(z) & & f_0(z) & g_{d_g^{(\overline{z})}}(z) & & g_0(z) \\ & \ddots & \vdots & & \ddots & \vdots \\ & & f_{d_f^{(\overline{z})}}(z) & & & g_{d_g^{(\overline{z})}}(z) \end{array}\right]. \tag{4.5}$$

$d_g^{(\overline{z})}$ columns $\quad$ $d_f^{(\overline{z})}$ columns

The Sylvester matrix is singular when evaluated at a common root z^*. It follows that by Lemma 4.1, the resultant $\mathrm{res}^{f,g}(z) := \det(S^{f,g}(z))$ vanishes when z is a root of the system $f(z, \overline{z}) = g(z, \overline{z}) = 0$. Hence, the roots of a polyanalytic polynomial system can be recovered as (a subset of) the roots of its resultant. Exploiting this fact, both coordinates of the real roots of a bivariate polynomial system $f(x, y) = g(x, y) = 0$ can now be obtained by encoding the variables x and y in the complex variables z and $\overline{z}$ using the transformation

$$(x, y) = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ -i & i \end{bmatrix} \cdot (z, \overline{z}). \tag{4.6}$$

It is well known that a polynomial's roots can be very sensitive to small changes in the polynomial's coefficients [243]. For this reason, explicitly building the resultant $\mathrm{res}^{f,g}(z)$ as the determinant of the polynomial matrix $S^{f,g}(z)$ [188]

and then computing its roots should be avoided. Instead, the roots can be recovered directly by noticing that (4.4) is a polynomial eigenvalue problem (PEP) with matrix pencil

$$S^{(0)} + S^{(1)}z + \cdots + S^{(d)}z^d, \tag{4.7}$$

where $S^{(i)} \in \mathbb{C}^{n\times n}$ for $i = 0, \ldots, d$, $n := d_f^{(\overline{z})} + d_g^{(\overline{z})}$ and $d := \max(d_f^{(z)}, d_g^{(z)})$. The PEP (4.4) can be solved by linearizing it into a generalized eigenvalue problem (GEP) of the form $(A - z^*B) \cdot \boldsymbol{y} = \mathbf{0}$. The matrices A and B are chosen so that $A - z^*B$ has the same spectrum as (4.7). For example, one such linearization is the second companion form [81], defined by

$$A := \begin{bmatrix} -S^{(d-1)} & \mathbb{I}_n & & \\ \vdots & & \ddots & \\ -S^{(1)} & & & \mathbb{I}_n \\ -S^{(0)} & & & \end{bmatrix} \text{ and} \tag{4.8a}$$

$$B := \begin{bmatrix} S^{(d)} & & & \\ & \mathbb{I}_n & & \\ & & \ddots & \\ & & & \mathbb{I}_n \end{bmatrix}. \tag{4.8b}$$

To further improve the accuracy of the computed eigenvalues, the system $f(z, \overline{z}) = g(z, \overline{z}) = 0$ and its associated pencil $A - z^*B$ can be balanced and the eigenvalues iteratively refined and filtered by a complex Newton–Raphson algorithm. For details, we refer the interested reader to the companion article [210].

In summary, the polyanalytic polynomial systems arising in (S)LS-$\mathbb{C}$ can be solved efficiently by building the pencil (4.8) associated with the polyanalytic polynomial system and then computing its eigenvalues. For bivariate polynomial systems such as those arising in (S)PS-$\mathbb{R}$, the bivariate system can first be transformed into a polyanalytic polynomial system using (4.6), after which its roots can be recovered as the real and imaginary parts of the eigenvalues of the associated pencil. Since we are interested in global minimizers of (S)LS-$\mathbb{C}$ or (S)PS-$\mathbb{R}$, we have only to evaluate the corresponding objective function for each of the eigenvalues to see which one corresponds to a global optimizer. Consequently, any spurious roots introduced by the resultant are of no concern, as they will simply evaluate to a poor objective function value.

4.6 Numerical experiments

The following numerical experiments investigate the benefit of combining tensor optimization algorithms with exact line and plane search in the context of blind separation and equalization of convolutive DS-CDMA mixtures received by an antenna array after multipath propagation. Similarly to [155, 156], we assume that the signal of the rth user is subject to intersymbol interference (ISI) over L consecutive symbols and that this signal arrives at the antenna array via P specular paths.

Let I be the CDMA spreading code length and let $\mathcal{H}^{(r)} \in \mathbb{C}^{I\times L\times P}$, $r = 1, \ldots, R$, contain the convolution of the rth user's spreading code with the corresponding channel for each of the P specular paths. Furthermore, let $S^{(r)} \in \mathbb{C}^{J\times L}$, $r = 1, \ldots, R$, be a Toeplitz matrix containing the J transmitted QPSK symbols of the rth user (along with $L-1$ interfering symbols) and let $A^{(r)} \in \mathbb{C}^{K\times P}$, $r = 1, \ldots, R$, contain the response of the K antennas according to the angles of arrival of the P paths. The convolutive model [155] for the observed tensor $\mathcal{Y} \in \mathbb{C}^{I\times J\times K}$ is then a rank-(L, L, P) BTD of the form

$$\mathcal{Y} = \sum_{r=1}^{R}\sum_{p=1}^{P}(H_{::p}^{(r)} \cdot S^{(r)\mathrm{T}}) \circ \boldsymbol{a}_p^{(r)}, \tag{4.9}$$

which can also be interpreted as a structured CPD or rank-$(L, L, 1)$ BTD.

In our experiments, we use pseudorandom spreading codes of length $I = 32$, frames of $J = 64$ QPSK symbols, $K = 8$ antennas, $L = 4$ interfering symbols, $P = 2$ major paths per user and $R = 16$ users. The signal-to-noise ratio (SNR) at the receiver is defined as $10\log_{10}(\|\mathcal{Y}\|_{\mathrm{F}}^2/\|\mathcal{N}\|_{\mathrm{F}}^2)$, where $\mathcal{Y}$ is the complex-valued noise-free tensor of observations and $\mathcal{N}$ is complex-valued zero-mean white Gaussian noise. We measure the bit error rate (BER) and the speedup in terms of iterations and CPU time compared to a baseline algorithm with two Monte Carlo experiments consisting of 500 runs for each SNR level. In the first experiment, we set the condition number of the antenna matrix $A := \begin{bmatrix} A^{(1)} & \cdots & A^{(R)} \end{bmatrix}$ equal to $\kappa(A) = 1$ and in the second, we choose $\kappa(A) = 10$.

For each run, we generate the observations $\mathcal{Y}$ according to (4.9) and generate a noise tensor $\mathcal{N}$. We choose the structured alternating least squares (ALS) algorithm implemented in Tensorlab [211] as the baseline algorithm to decompose the noisy tensor $\mathcal{Y} + \mathcal{N}$ as a structured CPD. The algorithm alternatingly computes least squares updates for the channel tensors $\mathcal{H}^{(r)}$, the users' symbols in $S^{(r)}$ and the antenna matrix A and is in this respect similar to the ALS algorithm described in [55, 155]. We also decompose the tensor using ALS combined with the (S)LS-$\mathbb{R}$, (S)LS-$\mathbb{C}$ and (S)PS-$\mathbb{R}$ search algorithms, which are

applied every four iterations. In our experience, applying the search algorithms once every few iterations give the parent optimization algorithm some breathing room for the iterates to stray away from valley floors in the objective function. This allows for similar or even better gains in the number of iterations, compared to executing the search algorithm every iteration, for a fraction of the cost.

Let $\boldsymbol{z}^{(k)}$ be the kth iterate of the ALS algorithm. For line searches, we choose to search along the line $\Delta\boldsymbol{z}_1 := \boldsymbol{z}_k - \boldsymbol{z}_{k-1}$ and for plane searches we add a second direction $\Delta\boldsymbol{z}_2 := \boldsymbol{z}_{k-1} - \boldsymbol{z}_{k-2}$. We refer to these two search directions as linear search directions. The bivariate systems arising in (scaled) complex line search and real plane search are solved using the generalized eigenvalue method described in the previous section. All aforementioned line and plane search algorithms were implemented as part of Tensorlab. We measure the iteration and CPU time speedup as the mean number of iterations and CPU time required by ALS, relative to that required by ALS with search algorithm, respectively. The algorithms are stopped if the maximum of 1000 iterations has been reached or if the norm of the step relative to the norm of the current iterate is smaller than 10^{-3}. All experiments were performed in MATLAB 8.1 (R2013a) on two hexacore Intel Xeon E5645 CPUs with 48 GB RAM.

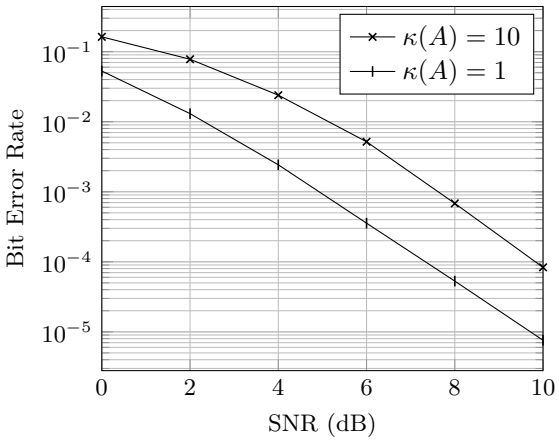
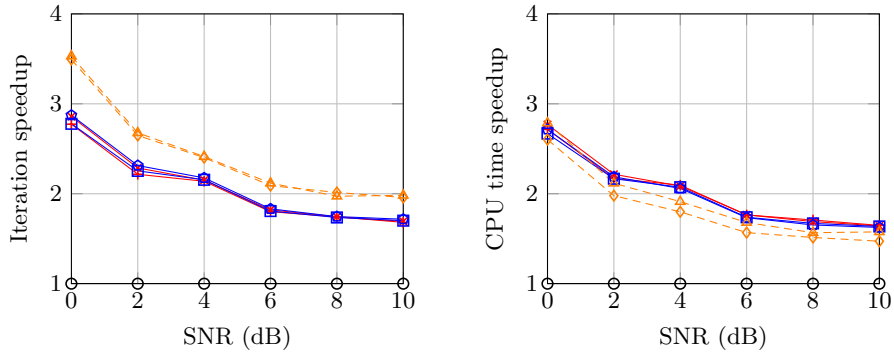


Figure 4.5: Mean BER as a function of the SNR of the observed tensor and the condition number of the antenna matrix A.

Figure 4.5 shows the mean BER of the two Monte Carlo experiments as a function of the SNR of the observed tensor. Increasing the condition number of the antenna matrix has an adverse effect on the BER, and combining ALS with a search algorithm does not deteriorate nor improve the BER.

In Figure 4.6a and 4.6b, we see the mean iteration and CPU time speedup

compared to ALS, respectively, for the case $\kappa(A) = 1$. Both line and plane searches offer a good speedup of the number of iterations and CPU time, relative to the baseline algorithm ALS. Compared to LS-$\mathbb{R}$, LS-$\mathbb{C}$ and PS-$\mathbb{R}$, the scaled searches SLS-$\mathbb{R}$, SLS-$\mathbb{C}$ and SPS-$\mathbb{R}$ offer only a small increase in iteration speedup. For real and complex line search, their scaled counterparts also translate into slightly faster CPU times. However, the improvement is almost negligible, and in the case of real scaled plane search even negative due to the additional overhead of solving a higher degree polynomial system than that of a real plane search. There also does not seem to be much difference in performance between real and complex line searches. These observations suggest that the linear search directions are already well scaled in magnitude and phase and that ALS mainly benefits from increasing the dimension of the search space.



(a) (b)

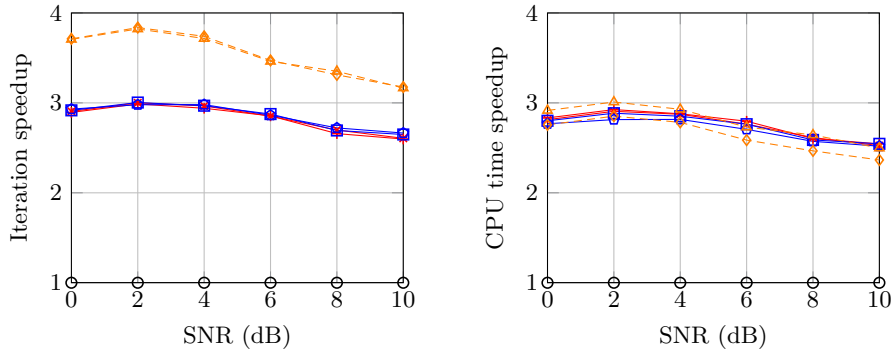
Figure 4.6: Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor. The condition number of the antenna matrix is $\kappa(A) = 1$. Both line and plane searches use linear search directions. Legend: ALS (—○—), ALS+LS-$\mathbb{R}$ (—+—), ALS+SLS-$\mathbb{R}$ (—×—), ALS+LS-$\mathbb{C}$ (—□—), ALS+SLS-$\mathbb{C}$ (—○—), ALS+PS-$\mathbb{R}$ (- -△- -), ALS+SPS-$\mathbb{R}$ (- -◇- -).

Figures 4.7a and 4.7b show the speedups for the case $\kappa(A) = 10$, which is expected to be more difficult than the previous situation as the terms in the decomposition tend to be more similar, and hence more difficult to separate, as the condition number of the antenna matrix increases. The search algorithms are now able to reduce the CPU time by about 60%. The speedup now seems to depend less on the SNR of the observed tensor.

For a final set of experiments, we define the search direction [34]

$$\Delta \boldsymbol{z}_1 := 2\boldsymbol{z}_k - 3\boldsymbol{z}_{k-1} + \boldsymbol{z}_{k-2} \tag{4.10}$$

which is based on a quadratic extrapolation of the iterates $\boldsymbol{z}_k$, assuming they are spaced equally apart. Figures 4.8 and 4.9 show the results of repeating the previous two experiments using the quadratically extrapolated search direction for the line searches. It is immediately clear that line searches are able to obtain speedups close to those of the plane searches by using the improved search direction. Evidently, the quality of the search direction plays an important role in the improvement a line or plane search method can offer.



(a) (b)

Figure 4.7: Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor. The condition number of the antenna matrix is $\kappa(A) = 10$. Both line and plane searches use linear search directions. Legend: ALS (—○—), ALS+LS-$\mathbb{R}$ (—+—), ALS+SLS-$\mathbb{R}$ (—×—), ALS+LS-$\mathbb{C}$ (—□—), ALS+SLS-$\mathbb{C}$ (—⬡—), ALS+PS-$\mathbb{R}$ (- -△- -), ALS+SPS-$\mathbb{R}$ (- -◇- -).

4.7 Conclusion

We introduced tensor optimization as a generic framework for computing tensor decompositions, least squares support tensor machines and jointly analysing multiple data sets. We showed that the global minimizers of their line and plane search subproblems can be obtained by computing the roots of univariate polynomials or bivariate polynomial systems. In nonlinear least squares problems, a linear approximation of the residual function is typically used to solve the underlying two-dimensional subspace minimization problem. We

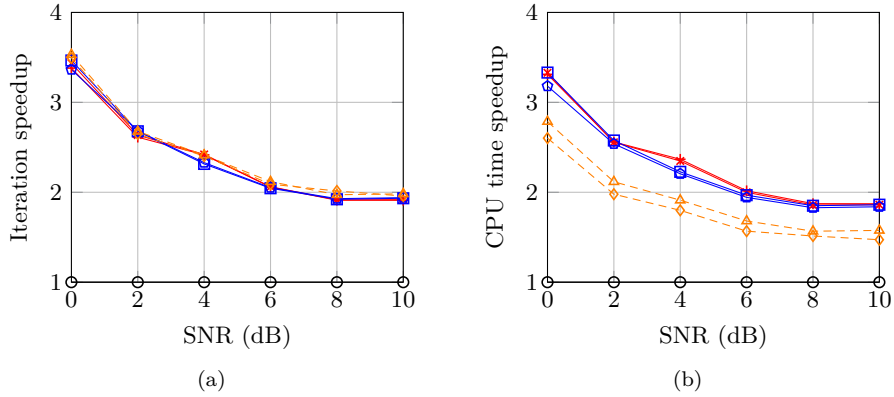


Figure 4.8: Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor. The condition number of the antenna matrix is $\kappa(A) = 1$. Line searches use the quadratically extrapolated direction (4.10), while plane searches use linear search directions. Legend: ALS (—○—), ALS+LS-$\mathbb{R}$ (—+—), ALS+SLS-$\mathbb{R}$ (—×—), ALS+LS-$\mathbb{C}$ (—□—), ALS+SLS-$\mathbb{C}$ (—⬡—), ALS+PS-$\mathbb{R}$ (- -△- -), ALS+SPS-$\mathbb{R}$ (- -◇- -).

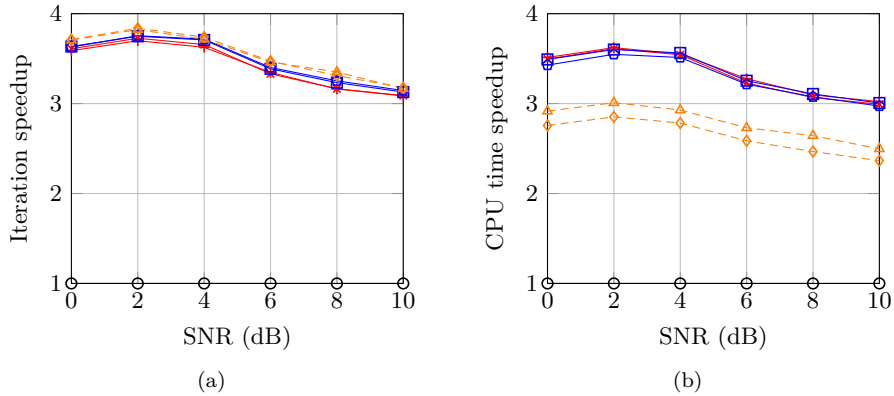


Figure 4.9: Mean iteration and CPU time speedup compared to ALS as a function of the SNR of the observed tensor. The condition number of the antenna matrix is $\kappa(A) = 10$. Line searches use the quadratically extrapolated direction (4.10), while plane searches use linear search directions. Legend: ALS (—○—), ALS+LS-$\mathbb{R}$ (—+—), ALS+SLS-$\mathbb{R}$ (—×—), ALS+LS-$\mathbb{C}$ (—□—), ALS+SLS-$\mathbb{C}$ (—⬡—), ALS+PS-$\mathbb{R}$ (- -△- -), ALS+SPS-$\mathbb{R}$ (- -◇- -).

showed that one may instead use a higher-order Taylor series expansion of the residual function and efficiently obtain its global minimizers in the same way as for tensor optimization problems. Furthermore, we introduced scaled line and plane searches which compute an additional scaling parameter for little additional cost. Finally, we showed how to obtain the real solutions of a bivariate polynomial system and the complex solutions of a polyanalytic polynomial system by solving a single generalized eigenvalue problem. In the numerical experiments, we applied a structured ALS algorithm in combination with (S)LS-$\mathbb{R}$, (S)LS-$\mathbb{C}$ and (S)PS-$\mathbb{R}$ to blind multi-user detection-estimation in DS-CDMA communication, in which the exact line and plane searches were able to speed up the computational time by up to a factor of 3. All algorithms were implemented in the MATLAB toolbox Tensorlab [211] and are available online.

Chapter 5

Numerical solution of bivariate and polyanalytic polynomial systems

Finding the real solutions of a bivariate polynomial system is a central problem in robotics, computer modeling and graphics, computational geometry and numerical optimization. We propose an efficient and numerically robust algorithm for solving bivariate and polyanalytic polynomial systems using a single generalized eigenvalue decomposition. In contrast to existing eigen-based solvers, the proposed algorithm does not depend on Gröbner bases or normal sets, nor does it require computing eigenvectors or solving additional eigenproblems to recover the solution. The method transforms bivariate systems into polyanalytic systems and then uses resultants in a novel way to project the variables onto the real plane associated with the two variables. Solutions are returned counting multiplicity and their accuracy is maximized by means of numerical balancing and Newton–Raphson refinement. Numerical experiments show that the proposed algorithm consistently recovers a higher percentage of solutions and is at the same time significantly faster and more accurate than competing double precision solvers. This chapter is a slightly adapted version of the article [210].

5.1 Introduction

Computing the roots of a set of polynomial equations is a well studied problem for which a wide range of techniques have been developed. These can roughly be categorized into numeric, symbolic and hybrid methods. Numeric techniques based on Newton's method [159] are useful for finding solutions locally. However, there is no guarantee of finding all solutions and multiple roots may cause convergence issues. Methods based on interval arithmetic [146] are robust, but can suffer from slow convergence and require bounds on the solutions of interest. In the context of optimization, there also exist various convex relaxation methods for global minimization of polynomials such as a semidefinite programming relaxation [122, 167, 193], but these only return a lower bound on the objective function in general. Gröbner bases [30, 74] and resultants [69] are symbolic methods that originate from algebraic geometry. They are both used to eliminate variables, essentially reducing the problem to finding the roots of univariate polynomials. However, Gröbner bases are expensive to compute and are often numerically ill-conditioned. On the other hand, resultants can introduce spurious solutions and have historically not led to numerically robust algorithms either. Homotopy continuation methods [131, 238] are hybrid techniques that combine homotopy methods with numerical continuation methods. The idea is to start from a system with a known solution set, and then to numerically track the solutions as the system is gradually transformed into the target system. Although homotopy continuation has been vastly improved in recent years, issues such as path crossing can decrease their robustness [148]. Adaptive multiprecision has proven to be an effective means of dealing with these problems [20]. For a recent overview on homotopy methods and algebraic geometry see the survey [240] and book [204].

In this chapter, we propose an efficient and numerically robust algorithm to compute the isolated real solutions of bivariate polynomial systems. Instead of using resultants to project the solution onto the complex plane associated with one of the two variables, we apply them in a novel way to project solutions onto the real plane associated with both variables. The key to this projection is to first transform the bivariate polynomial system into a so-called polyanalytic polynomial system. The projected system is then reduced to a polynomial eigenvalue problem, which can in turn be linearized into a generalized eigenvalue problem. As such, our method can be seen as a hybrid method, although symbolic manipulations are absent. In contrast to existing eigen-based solvers [13, 31, 41, 105, 123, 140, 141, 145, 221], the proposed algorithm does not depend on Gröbner bases or normal sets, nor does it require computing eigenvectors or solving additional eigenproblems to recover the solution. Moreover, the method returns solutions counting multiplicity and is also applicable to polyanalytic

polynomial systems. We aim to maximize accuracy with numerical balancing and Newton–Raphson refinement, and present a robust method to filter out spurious solutions introduced by the resultant. In our numerical experiments, we compare performance with Bertini [21], Chebfun [230], PHCpack [237], the RealSolving C library [185, 186], the RegularChains package [12] and Maple's BivariatePolynomial method on a range of bivariate systems of varying difficulty. The resulting performance profiles show that the proposed algorithm consistently recovers a higher percentage of solutions and is at the same time significantly faster and more accurate.

The chapter is organized as follows. In the next section we review our notation and introduce some basic definitions. In Section 5.3 we examine a classical approach to solving bivariate polynomial systems and propose an algorithm for solving both bivariate and polyanalytic polynomial systems with a single generalized eigenvalue decomposition. Furthermore, we also present several balancing steps and a robust strategy for removing spurious roots. In Section 5.4 we evaluate the performance of the proposed algorithm on a sizable set of low- to moderate-degree bivariate polynomial systems using performance profiles [65]. We conclude the chapter in Section 5.5.

5.2 Notation and preliminaries

Vectors are denoted by boldface letters and are lower case, e.g., $\boldsymbol{a}$. Matrices are usually denoted by capital letters, e.g., A. Higher-order tensors are denoted by Euler script letters, e.g., $\mathcal{A}$. An entry of a vector $\boldsymbol{a}$, matrix A or tensor $\mathcal{A}$ is denoted by a_i, a_{ij} or $a_{ijk\ldots}$, depending on the number of modes. A colon is used to select all entries of a mode. For instance, $\boldsymbol{a}_{:j}$ corresponds to the jth column of a matrix A. When there is no confusion, we also use $\boldsymbol{a}_j$ to denote the jth column of the matrix A. Sequences are denoted by a superscript in parentheses, e.g., $\{A^{(n)}\}_{n=1}^{N}$. The superscripts $\cdot^{\mathrm{T}}$, $\cdot^{\mathrm{H}}$, $\cdot^{-1}$ and $\cdot^{\dagger}$ are used for the transpose, Hermitian conjugate, matrix inverse and Moore–Penrose pseudoinverse, respectively. The complex conjugate is denoted by an overbar, e.g., $\overline{a}$ is the complex conjugate of the scalar a. We use parentheses to denote the concatenation of two or more vectors, e.g., $(\boldsymbol{a}, \boldsymbol{b})$ is equivalent to $\begin{bmatrix}\boldsymbol{a}^{\mathrm{T}} & \boldsymbol{b}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$. The $n \times n$ identity matrix is denoted by $\mathbb{I}_n$, and the all-zero and all-one $m \times n$ matrices by $0_{m\times n}$ and $1_{m\times n}$, respectively.

Let $\mathcal{B}$ be a basis for the ring of polynomials $\mathbb{C}[x]$ with basis polynomials $B_n(x)$ of degree n. For example, if $\mathcal{B}$ is the monomial basis or Chebyshev basis, the corresponding basis polynomials $B_n(x)$ are x^n and $T_n(x) := \cos(n \arccos(x))$, respectively. Given a choice of basis, we associate a coefficient vector $\boldsymbol{p} \in \mathbb{C}^{d_p+1}$,

$p_{d_p+1} \neq 0$, with polynomials

$$p(x) := \begin{bmatrix} B_0(x) & B_1(x) & \cdots & B_{d_p}(x) \end{bmatrix} \cdot \boldsymbol{p}$$

of degree d_p. Analogously, we associate a coefficient matrix $P \in \mathbb{C}^{(d_p^{(y)}+1)\times(d_p^{(x)}+1)}$, $\max_i |p_{i,d_p^{(x)}+1}| \neq 0 \wedge \max_j |p_{d_p^{(y)}+1,j}| \neq 0$, with bivariate polynomials

$$p(x,y) := \begin{bmatrix} B_0(y) & B_1(y) & \cdots & B_{d_p^{(y)}}(y) \end{bmatrix} \cdot P \cdot \begin{bmatrix} B_0(x) & B_1(x) & \cdots & B_{d_p^{(x)}}(x) \end{bmatrix}^{\mathrm{T}}$$

of coordinate degree $(d_p^{(x)}, d_p^{(y)})$. The total degree d_p of the polynomial $p(x,y)$ is defined as the largest value $i+j$ for which $p_{i+1,j+1} \neq 0$. We represent bivariate polynomials in product bases, i.e., as a linear combination of products $B_i(y)B_j(x)$.

The choice of basis can depend on the application and in which region the roots of interest lie. For example, the product monomial basis is orthogonal with respect to the inner product over the two unit circles in the complex planes associated with x and y, while the product Chebyshev basis is orthogonal on the unit square associated with x and y.

Substituting x and y by z and its complex conjugate $\overline{z}$ produces so-called *polyanalytic* polynomials $p(z,\overline{z})$. The term polyanalytic refers to the fact that the function is not analytic in z, but is analytic in z and $\overline{z}$ as a whole. Furthermore, it is not hard to show that if $p(z,\overline{z})$ is real-valued for all $z \in \mathbb{C}$, then P is Hermitian.

5.3 Solving systems of bivariate and polyanalytic polynomials

5.3.1 The polynomial eigenvalue problem

In robotics [256], computer modeling and graphics [83, 107], computational geometry [18] and numerical optimization [32,205], a common task is to compute the isolated real roots of bivariate polynomial systems

$$p(x,y) = q(x,y) = 0 \tag{5.1}$$

or the isolated complex roots of polyanalytic polynomial systems

$$r(z,\overline{z}) = s(z,\overline{z}) = 0, \tag{5.2}$$

where the associated coefficient matrices P, Q, R and S may be real- or complex-valued. For example, such systems appear as line and plane search subproblems in tensor optimization [205], and more generally also in polynomial optimization. Unconstrained tensor optimization problems are of the form

$$\underset{\boldsymbol{z} \in \mathbb{C}^n}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{M}(\boldsymbol{z}) - \mathcal{T}\|_{\text{F}}^2, \tag{5.3}$$

where $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ is a given tensor and $\mathcal{M}$ is usually a multilinear function of the variables $\boldsymbol{z}$. A function is said to be multilinear in its argument $\boldsymbol{z}$ if for all i it is linear in z_i when the remaining variables z_j $(j \neq i)$ are fixed. Among other applications, (5.3) can be used to compute tensor decompositions such as the canonical polyadic decomposition [33, 95] and low multilinear rank approximation [112, 232]. Given a current iterate $\boldsymbol{z}_k$ and two descent directions $\Delta\boldsymbol{z}_1$ and $\Delta\boldsymbol{z}_2$,

$$\underset{\alpha \in \mathbb{C}}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{M}(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1) - \mathcal{T}\|_{\text{F}}^2 \tag{5.4a}$$

$$\underset{\alpha,\beta \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{M}(\boldsymbol{z}_k + \alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2) - \mathcal{T}\|_{\text{F}}^2 \tag{5.4b}$$

are referred to as complex line search and real plane search subproblems, respectively. Let $f_{\text{LS}}(\alpha, \overline{\alpha})$ and $f_{\text{PS}}(\alpha, \beta)$ be the objective functions of (5.4a) and (5.4b), respectively, then the global line and plane search minimizers are solutions of the bivariate and polyanalytic polynomial systems

$$\frac{\partial f_{\text{LS}}}{\partial \alpha} = \frac{\partial f_{\text{LS}}}{\partial \overline{\alpha}} = 0 \qquad \text{and} \qquad \frac{\partial f_{\text{PS}}}{\partial \alpha} = \frac{\partial f_{\text{PS}}}{\partial \beta} = 0,$$

where $\frac{\partial}{\partial \alpha}$ $(\frac{\partial}{\partial \overline{\alpha}})$ is a Wirtinger derivative and acts as partial derivative with respect to α $(\overline{\alpha})$, while treating $\overline{\alpha}$ (α) as constant [206].

Two-dimensional subspace minimization [32] is similar to the plane search (5.4b), where $\mathcal{M}$ is a linear approximation of a nonlinear residual function F and a norm constraint on the step size such as $\|\alpha\Delta\boldsymbol{z}_1 + \beta\Delta\boldsymbol{z}_2\| \leq \delta$ is added. The stationary points of the resulting optimization problem are the solutions of a bivariate polynomial system in which the polynomials have total degree 2. In [205], it is shown that two-dimensional subspace minimization can be generalized to higher-order Taylor series expansions of the residual function F and that the resulting optimization problem is a bivariate polynomial system with polynomials of total degree 2 and $2d$, where d is the order of the Taylor series approximation.

A widely used technique to solve bivariate polynomial systems (5.1) is to eliminate one of the two variables by reducing the problem to finding the roots

of a so-called resultant polynomial. Let $V_{\mathbb{C}}$ denote the set $\{(x,y) \in \mathbb{C}^2 \,|\, p(x,y) = q(x,y) = 0\}$. If p and q have no common factor, then by Bézout's theorem $V_{\mathbb{C}}$ is zero-dimensional and contains at most $d_p d_q$ elements. The resultant of p and q is an algebraic description of the projections of the set $V_{\mathbb{C}}$ onto the complex plane associated with x or y, depending on which variable is eliminated. In other words, the resultant is a polynomial whose roots include the set

$$V_{\mathbb{C}}^{(x)} := \{x \in \mathbb{C} \,|\, \exists y \in \mathbb{C} : p(x,y) = q(x,y) = 0\} \text{ or} \tag{5.5a}$$

$$V_{\mathbb{C}}^{(y)} := \{y \in \mathbb{C} \,|\, \exists x \in \mathbb{C} : p(x,y) = q(x,y) = 0\}. \tag{5.5b}$$

From here on, we assume the variable y is eliminated so that the resultant is a polynomial in x. Of course, the role of x and y may be reversed in the subsequent discussion.

Resultants have many formulations, but are typically represented by (ratios of) determinants. The two most common resultants are those of Sylvester and Bézout, the former of which is a corollary of the following lemma.

Lemma 5.1. *Let $p(x,y)$ and $q(x,y)$ be two nonconstant bivariate polynomials. If $p(x,y)$ and $q(x,y)$ have a common zero $(x^*, y^*) \in \mathbb{C}^2$, then there are nonzero polynomials $u(y)$ and $v(y)$ satisfying*

$$p(x^*, y)u(y) + q(x^*, y)v(y) \equiv 0, \tag{5.6}$$

where $d_u < d_q^{(y)}$ and $d_v < d_p^{(y)}$.

Proof. Write $p(x,y)$ and $q(x,y)$ as

$$p(x,y) = \sum_{i=0}^{d_p^{(y)}} p_i(x) B_i(y) \quad \text{and} \quad q(x,y) = \sum_{i=0}^{d_g^{(y)}} q_i(x) B_i(y),$$

where $p_i(x)$ ($q_i(x)$) is the polynomial in x associated with the coefficient vector $\boldsymbol{p}_{i+1,:}$ ($\boldsymbol{q}_{i+1,:}$). Since $p(x^*, y^*) = q(x^*, y^*) = 0$, $p(x^*, y)$ and $q(x^*, y)$ will have a common factor $k(y) := (y - y^*)^n$ for some positive integer n. We have that $\frac{p(x^*,y)}{k(y)} q(x^*, y) = \frac{q(x^*,y)}{k(y)} p(x^*, y)$. Set $u(y) := q(x^*, y)/k(y)$ and $v(y) := -p(x^*, y)/k(y)$. □

Equation (5.6) expresses the fact that if (x^*, y^*) satisfies $p(x^*, y^*) = q(x^*, y^*) = 0$, then there exist polynomials $u(y)$ and $v(y)$ such that all coefficients of the polynomial $p(x^*, y)u(y) + q(x^*, y)v(y)$ are zero. Writing p, q, u and v in the

basis $\{B_i(y)\}_{i=0}^{n-1}$, where $n := d_p^{(y)} + d_q^{(y)}$, these conditions are equivalent to the system

$$S^{p,q}(x^*) \cdot \boldsymbol{r} = \boldsymbol{0}, \tag{5.7}$$

where $\boldsymbol{r} := (\boldsymbol{u}, \boldsymbol{v})$ and $S^{p,q}(x)$ is a polynomial matrix called the Sylvester matrix. In other words, Lemma 5.1 states that $S^{p,q}(x)$ must be singular when evaluated at the x-coordinate of a common root. The Sylvester resultant is now defined as $\operatorname{res}^{p,q}(x) := \det(S^{p,q}(x))$. Clearly, the resultant is a polynomial in x and vanishes if and only if the Sylvester matrix is singular. The structure of the Sylvester matrix depends on the choice of basis corresponding to the variable y. Let $f(x, y)$ be a polynomial in the basis $\{B_i(y)\}$ with polynomial coefficients $f_i(x)$. Define the infinite Toeplitz, Hankel and shifted Toeplitz matrices

$$L^f(x) := \begin{bmatrix} f_0(x) & \\ \vdots & \ddots \\ f_{d_f^{(y)}}(x) & \ddots \\ & \ddots \end{bmatrix},$$

$$M^f(x) := \begin{bmatrix} f_0(x) & \cdots & f_{d_f^{(y)}}(x) & 0 & \cdots \\ \vdots & \iddots & \iddots & & \\ f_{d_f^{(y)}}(x) & \iddots & & & \\ 0 & & & & \\ \vdots & & & & \end{bmatrix} \text{ and}$$

$$N^f(x) := \begin{bmatrix} 0 & 0 & \cdots & & \\ 0 & f_0(x) & \cdots & f_{d_f^{(y)}}(x) & \\ \vdots & & \ddots & \ddots & \ddots \end{bmatrix},$$

respectively. For the monomial basis we have $y^i y^j = y^{i+j}$ so that the Sylvester matrix is given by

$$S^{p,q}(x) = \begin{bmatrix} L^p_{1:n,1:d_q^{(y)}}(x) & L^q_{1:n,1:d_p^{(y)}}(x) \end{bmatrix}.$$

For the Chebyshev basis we have $2T_i(y)T_j(y) = T_{i+j}(y) + T_{|i-j|}(y)$ so that the Sylvester matrix is, up to a scale factor, equal to

$$S^{p,q}(x) = \begin{bmatrix} L^{p\mathrm{T}}_{1:n,1:d_q^{(y)}}(x) + M^{p\mathrm{T}}_{1:n,1:d_q^{(y)}}(x) + N^{p\mathrm{T}}_{1:n,1:d_q^{(y)}}(x) \\ L^{q\mathrm{T}}_{1:n,1:d_p^{(y)}}(x) + M^{q\mathrm{T}}_{1:n,1:d_p^{(y)}}(x) + N^{q\mathrm{T}}_{1:n,1:d_p^{(y)}}(x) \end{bmatrix}^{\mathrm{T}}.$$

Importantly, the polynomial $\mathrm{res}^{p,q}(x)$ is in a sense a minimal characterization of the x-coordinates of the roots of the system $p(x,y) = q(x,y) = 0$. In fact, its roots are exactly the projected roots $V_{\mathbb{C}}^{(x)}$ together with roots at infinity of the form (x^*, ∞).

Theorem 5.1 (see, e.g., [19]). *If p and q are coprime, the distinct roots of $\mathrm{res}^{p,q}(x)$ are the roots of the bivariate system $p(x,y) = q(x,y) = 0$ projected onto the complex plane associated with x and the roots of the leading coefficient system $p_{d_p^{(y)}}(x) = q_{d_q^{(y)}}(x) = 0$. More precisely,*

$$\{x \in \mathbb{C} \,|\, \mathrm{res}^{p,q}(x) = 0\} \;=\; V_{\mathbb{C}}^{(x)} \cup \{x \in \mathbb{C} \,|\, p_{d_p^{(y)}}(x) = q_{d_q^{(y)}}(x) = 0\}.$$

The multiplicity of a root x^* of $\mathrm{res}^{p,q}(x)$ is the sum of the multiplicities of all solutions of $p(x,y) = q(x,y) = 0$ with x-coordinate x^*.

To compute the roots of the resultant, one possible approach is to explicitly compute the resultant $\mathrm{res}^{p,q}(x)$ as the determinant of a polynomial matrix [188], after which the roots can be obtained as the eigenvalues of its companion matrix. However, it is well known that a polynomial's roots can be very sensitive to small changes in the polynomial's coefficients [243]. For this reason, the accuracy required of the resultant's coefficients makes this approach prohibitively expensive. Instead, the resultant's roots can be obtained directly without constructing $\mathrm{res}^{p,q}(x)$ by noticing that (5.7) is a polynomial eigenvalue problem (PEP) [140] with matrix pencil

$$S^{p,q}(x^*) = \sum_{i=0}^{d} S^{(i)} B_i(x^*), \tag{5.8}$$

in the basis $\{B_i(x^*)\}$ and where $S^{(i)} \in \mathbb{C}^{n \times n}$ for $i = 0, \ldots, d$ and $d := \max(d_p^{(x)}, d_q^{(x)})$. It is well known that $p(x,y)$ and $q(x,y)$ share a nontrivial common factor if and only if $\mathrm{res}^{p,q}(x) \equiv 0$. In this case, the pencil (5.8) is said to be singular. If $p(x,y)$ and $q(x,y)$ don't share a common factor, the pencil is called regular. Unfortunately, the eigenvalues of a singular (or close to singular) pencil can be changed arbitrarily by small perturbations of its matrices [244]. However, the situation is not always as bad as this. Wilkinson gives the example of a singular pencil $A - \lambda B$, where

$$A := \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix} \qquad \text{and} \qquad B := \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}.$$

The perturbed pencil $\hat{A} - \lambda \hat{B}$ has characteristic equation $\det(\hat{A} - \lambda \hat{B}) = [(2+\Delta a_{11}) - \lambda(1+\Delta b_{11})](\Delta a_{22} - \lambda \Delta b_{22}) - (\Delta a_{12} - \lambda \Delta b_{12})(\Delta a_{21} - \lambda \Delta b_{21})$, where

$\hat{A} := A + \Delta A$ and $\hat{B} := B + \Delta B$. For almost all perturbations ΔA and ΔB, the characteristic equation has a root which is very close to 2 [244]. Hence, it may in some cases be possible to recover the eigenvalues of the regular part of a singular pencil. If this is not the case, the greatest common divisor of $p(x, y)$ and $q(x, y)$ or the singular part of the associated pencil should be extracted. Special care should be taken with both approaches, as they will inevitably be based on decisions concerning the ranks of the matrices involved.

A standard way of solving the PEP (5.7) is to linearize the polynomial pencil $S^{p,q}(x^*)$ into a linear pencil $A - x^*B$ and solve its associated generalized eigenvalue problem (GEP). The matrices $A, B \in \mathbb{C}^{dn \times dn}$ are chosen so that $A - x^*B$ has the same spectrum as $S^{p,q}(x^*)$. In practice, the monomial basis is used most often together with a companion linearization. For the Chebyshev basis, a colleague linearization can be used. In fact, both of these are confederate linearizations [142, 143], which are defined by the polynomial basis' recurrence relations. To construct such a linearization, consider as per example the recurrence relations

$$x \begin{bmatrix} 1 & x & \cdots \end{bmatrix} = \begin{bmatrix} 1 & x & \cdots \end{bmatrix} \cdot \begin{bmatrix} 0 & & \\ 1 & & \\ & \ddots & \end{bmatrix} \text{ and} \tag{5.9a}$$

$$x \begin{bmatrix} T_0(x) & T_1(x) & \cdots \end{bmatrix} = \begin{bmatrix} T_0(x) & T_1(x) & \cdots \end{bmatrix} \cdot \begin{bmatrix} 0 & \frac{1}{2} & & \\ 1 & 0 & \frac{1}{2} & \\ & \frac{1}{2} & 0 & \ddots \\ & & \frac{1}{2} & \ddots \\ & & & \ddots \end{bmatrix} \tag{5.9b}$$

for the monomial and Chebyshev basis, respectively. Let $\boldsymbol{v}^{(d)}(x)$ be the generalized Vandermonde vector $\begin{bmatrix} B_0(x) & \cdots & B_{d-1}(x) \end{bmatrix}^{\mathrm{T}}$ of length d and let $\boldsymbol{l}$ be a left eigenvector of the PEP (5.7) so that

$$\boldsymbol{l}^{\mathrm{T}} \cdot S^{p,q}(x^*) = \mathbf{0}. \tag{5.10}$$

Now truncate equations (5.9) up to the dth term and convert to block form by treating scalars as identity matrices, then use (5.10) to eliminate the last block row. Defining $\boldsymbol{w} := \boldsymbol{v}^{(d)}(x^*) \otimes \boldsymbol{l}$, we obtain the companion and colleague

linearizations

$$x^*\boldsymbol{w}^{\mathrm{T}} \cdot \begin{bmatrix} \mathbb{I} & & & \\ & \ddots & & \\ & & \mathbb{I} & \\ & & & S^{(d)} \end{bmatrix} = \boldsymbol{w}^{\mathrm{T}} \cdot \begin{bmatrix} 0 & & & -S^{(0)} \\ \mathbb{I} & & & -S^{(1)} \\ & \ddots & & \vdots \\ & & \mathbb{I} & -S^{(d-1)} \end{bmatrix} \text{ and} \tag{5.11a}$$

$$2x^*\boldsymbol{w}^{\mathrm{T}} \cdot \begin{bmatrix} \mathbb{I} & & & \\ & \ddots & & \\ & & \mathbb{I} & \\ & & & S^{(d)} \end{bmatrix} = \boldsymbol{w}^{\mathrm{T}} \cdot \begin{bmatrix} 0 & \mathbb{I} & & & -S^{(0)} \\ 2\mathbb{I} & 0 & \ddots & & \vdots \\ & \mathbb{I} & \ddots & \mathbb{I} & -S^{(d-3)} \\ & & \ddots & 0 & S^{(d)} - S^{(d-2)} \\ & & & \mathbb{I} & -S^{(d-1)} \end{bmatrix} \tag{5.11b}$$

for the monomial and Chebyshev basis, respectively. Here, $\boldsymbol{w}$ is a left eigenvector of the GEP (5.11). If $(x^*, y^*) \in \mathbb{C}^2$ is a root of $p(x, y) = q(x, y) = 0$, then $\boldsymbol{v}^{(n)}(y^*)$ is a left eigenvector $\boldsymbol{l}$ of (5.10) corresponding to the eigenvalue x^* since

$$\boldsymbol{v}^{(n)}(y^*)^{\mathrm{T}} \cdot S^{p,q}(x^*) = \begin{bmatrix} p(x^*, y^*)\boldsymbol{v}^{(d_q^{(y)})}(y^*)^{\mathrm{T}} & q(x^*, y^*)\boldsymbol{v}^{(d_p^{(y)})}(y^*)^{\mathrm{T}} \end{bmatrix} = \mathbf{0}^{\mathrm{T}}.$$

This means that if x^* is a simple eigenvalue of the GEP (5.11), we can recover y^* from its corresponding left eigenvector $\boldsymbol{w}$, which will be a multiple of $\boldsymbol{v}^{(d)}(x^*) \otimes \boldsymbol{v}^{(n)}(y^*)$. However, the situation becomes more difficult when (x^*, y^*) is a multiple root, or when there is a $\hat{y}^* \neq y^*$ for which $(x^*, \hat{y}^*)$ is also a root. Both of these cases increase the dimension of the kernel of $S^{p,q}(x^*)$ so that it is larger than one. Several solutions for recovering all values y^* under these circumstances have been proposed.

Some algorithms compute y^* by solving a smaller eigenproblem for each distinct x^* [31, 41, 141]. However, this approach requires estimating the multiplicities of the eigenvalues x^*, along with the dimension of their corresponding kernels. The former task is complicated by the fact that a multiple real eigenvalue x^* often splits into several nearby complex eigenvalues due to round-off errors or because the problem cannot be represented exactly in finite precision. Then, for each distinct x^* an eigenproblem of size equal to the dimension of its kernel should be solved. Another approach is to eliminate both x and y separately and then for each pair of projected coordinates check whether it constitutes a solution of the given system or not [23, 63, 189]. For many applications, a nontrivial final task is to determine which of the solutions are real.

To solve these issues, we propose to first project the solutions onto the real plane associated with x and y, which will allow us to compute both the x- and y-coordinates of the roots simultaneously and ensure they are real. Moreover, this

approach does not require computing any eigenvectors or solving any additional eigenproblems, which saves both computational effort and memory. Consider the change of variables

$$(x, y) = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ -i & i \end{bmatrix} \cdot (z, \omega) \tag{5.12}$$

which transforms the polynomials $p(x, y)$ and $q(x, y)$ into $\hat{p}(z, \omega)$ and $\hat{q}(z, \omega)$, respectively. This operation can destroy the coordinate degree structure of the polynomials, possibly changing the coefficient matrices from dense rectangular matrices to square upper anti-triangular matrices. However, the total degrees of the polynomials remain unchanged. We proceed as before, this time eliminating ω from the modified system $\hat{p}(z, \omega) = \hat{q}(z, \omega) = 0$. The roots of the resultant $\mathrm{res}^{\hat{p},\hat{q}}(z)$ now comprise the set

$$V_{\mathbb{C}}^{(z)} := \{z \in \mathbb{C} \,|\, \exists\, \omega \in \mathbb{C} : \hat{p}(z, \omega) = \hat{q}(z, \omega) = 0\}.$$

In fact, $\{z \in \mathbb{C} \,|\, \mathrm{res}^{\hat{p},\hat{q}}(z) = 0\}$ is often exactly equal to $V_{\mathbb{C}}^{(z)}$, since the transformation (5.12) generically gives rise to a constant leading coefficient system $\hat{p}_{d_p}(z) = \hat{q}_{d_q}(z) = 0$ with no solutions. Since the transformation in (5.12) is invertible, $V_{\mathbb{C}}^{(z)}$ can be characterized in terms of the roots of the original system $p(x, y) = q(x, y) = 0$ as

$$V_{\mathbb{C}}^{(z)} = \{z \in \mathbb{C} \,|\, \exists\, (x, y) \in V_{\mathbb{C}} : z = x + iy\}.$$

Clearly, the set of projected real roots

$$V_{\mathbb{R}^2}^{(z)} := \{z \in \mathbb{C} \,|\, \exists (x, y) \in V_{\mathbb{C}} \cap \mathbb{R}^2 : z = x + iy\}$$

is a subset of $V_{\mathbb{C}}^{(z)}$. As we are only interested in real roots $(x^*, y^*) \in \mathbb{R}^2$, we may also interpret the transformed system as a polyanalytic polynomial system $\hat{p}(z, \overline{z}) = \hat{q}(z, \overline{z}) = 0$. Hence, the real roots $(x^*, y^*) \in \mathbb{R}^2$ of a bivariate polynomial system may be obtained as the real and imaginary part of (a subset of) the eigenvalues of the GEP (5.11) associated with the elimination of $\overline{z}$ after converting the system into a polyanalytic system with (5.12). Moreover, the method allows multiple roots to be recovered without additional effort. Indeed, each eigenvalue's algebraic multiplicity is equal to the multiplicity of the corresponding root (x^*, y^*), so that multiple roots need not be treated separately from simple roots. In contrast, methods based on recovering y^* from the eigenspace belonging to x^* need to estimate the algebraic multiplicity of each eigenvalue and apply relatively complicated techniques to extract y^*.

In much the same way as the bivariate case, the complex roots $z^* \in \mathbb{C}$ of a polyanalytic polynomial system can be obtained by eliminating $\overline{z}$. For both types

of systems, it remains to identify which of the eigenvalues do not correspond to roots of the original system. In the final subsection, we present a robust method to remove these spurious solutions based on a Newton–Raphson refinement on the original polynomial system. The following subsection proposes several balancing steps that aim to maximize the accuracy of the computed roots.

5.3.2 Balancing the system and its associated pencil

Balancing the bivariate system

From here on, we restrict the discussion to polynomial systems given in the product monomial basis for simplicity. For both bivariate and polyanalytic polynomial systems, we center the exponents of the elements of the associated coefficient matrices P and Q around zero with the scaling

$$P \leftarrow 2^{-\operatorname{round}(\operatorname{median}(\log_2(|P|)))} P \tag{5.13a}$$

$$Q \leftarrow 2^{-\operatorname{round}(\operatorname{median}(\log_2(|Q|)))} Q. \tag{5.13b}$$

Here, $|\cdot|$ is the absolute value, $\log(\cdot)$ is the element-wise logarithm, $\operatorname{median}(\cdot)$ is the median over all finite elements of its argument and $\operatorname{round}(\cdot)$ rounds to the nearest integer to avoid introducing round-off error. The scaling (5.13) not only normalizes the coefficients of the two polynomials relative to each other, but also maximizes the available exponent range so that overflow is less likely to occur.

Bivariate systems $p(x, y) = q(x, y) = 0$ are then converted into polyanalytic polynomial systems $\hat{p}(z, \overline{z}) = \hat{q}(z, \overline{z}) = 0$ using (5.12). The entries of the coefficient matrices $\hat{P}$ and $\hat{Q}$ are linear combinations of the entries of the coefficient matrices P and Q. In order to minimize loss of precision due to rounding errors, we propose to first minimize the variance of the exponents of their elements by scaling the variables as $x \leftarrow 2^{r_x} x$ and $y \leftarrow 2^{r_y} y$ with

$$\hat{P} \leftarrow \operatorname{diag}(\boldsymbol{v}^{(d_p^{(y)}+1)}(2^{r_y})) \cdot P \cdot \operatorname{diag}(\boldsymbol{v}^{(d_p^{(x)}+1)}(2^{r_x})) \tag{5.14a}$$

$$\hat{Q} \leftarrow \operatorname{diag}(\boldsymbol{v}^{(d_q^{(y)}+1)}(2^{r_y})) \cdot Q \cdot \operatorname{diag}(\boldsymbol{v}^{(d_q^{(x)}+1)}(2^{r_x})) \tag{5.14b}$$

where r_x and r_y are real scalars that solve the least squares problem

$$\underset{r_x, r_y \in \mathbb{R}}{\text{minimize}} \quad \operatorname{var}(\log(|\hat{P}|)) + \operatorname{var}(\log(|\hat{Q}|)), \tag{5.15}$$

in which $\operatorname{var}(\cdot)$ is the sample variance over all finite elements of its argument. Again, we round r_x and r_y to their nearest integer values to avoid introducing

round-off error. After applying (5.14), we transform $\hat{P}$ and $\hat{Q}$ in-place with (5.12).

Balancing the polyanalytic system

Assuming we now dispose of a scaled polyanalytic system $\hat{p}(z,\overline{z}) = \hat{q}(z,\overline{z}) = 0$, our goal is to balance the problem such that its associated pencil $A - z^*B$ is closer to a well-conditioned normal pencil [128]. Ward suggests to scale all elements of A and B to the same order of magnitude [241]. To this end, a simple heuristic is to minimize the variance of the exponents of their elements by combining two balancing strategies. The first is to balance the coefficient matrices by scaling the variables z and $\overline{z}$ with a real scalar s_z as $z \leftarrow 2^{s_z} z$ and $\overline{z} \leftarrow 2^{s_z}\overline{z}$. The second is to balance the Sylvester blocks $S^{(i)}$ by scaling the variable z^* in the PEP (5.7) with a real scalar s_{z^*} as $z^* \leftarrow 2^{s_{z^*}} z^*$. We set

$$\hat{P} \leftarrow \operatorname{diag}(\boldsymbol{v}^{(d_p^{(\overline{z})}+1)}(2^{s_z})) \cdot \hat{P} \cdot \operatorname{diag}(\boldsymbol{v}^{(d_p^{(z)}+1)}(2^{s_z+s_{z^*}})) \tag{5.16a}$$

$$\hat{Q} \leftarrow \operatorname{diag}(\boldsymbol{v}^{(d_q^{(\overline{z})}+1)}(2^{s_z})) \cdot \hat{Q} \cdot \operatorname{diag}(\boldsymbol{v}^{(d_q^{(z)}+1)}(2^{s_z+s_{z^*}})) \tag{5.16b}$$

Analogously to (5.15), s_z and s_{z^*} are the solution of the least squares problem

$$\underset{s_z, s_{z^*} \in \mathbb{R}}{\text{minimize}} \quad \operatorname{var}(\log(|\hat{P}|)) + \operatorname{var}(\log(|\hat{Q}|)). \tag{5.17}$$

Again, we round the solution of (5.17) to the nearest integer before applying transformations (5.16) to avoid introducing round-off error. Finally, we center the exponents of the coefficient matrices by applying (5.13) to $\hat{P}$ and $\hat{Q}$.

Balancing the pencil

Solving the GEP (5.11) consists of three steps. Let A and B be the matrices in the right and left hand side of (5.11), respectively. The first step is to reduce B to an upper triangular matrix with a QR-factorization. Then, the pair (A, B) is reduced to generalized upper Hessenberg form using unitary left and right transformations so that A becomes an upper Hessenberg matrix and B remains upper triangular. Finally, the QZ algorithm is applied on the resulting pencil. Equation (5.11) is not the only way to linearize the PEP (5.7). However, compared to other linearizations it has the advantage that the pencil is already in a block generalized Hessenberg form. Moreover, many of the first Givens rotations reducing the pencil to a generalized Hessenberg form will be permutations. An alternative linearization is the second companion form [81],

which is perhaps less efficient but in our experience has a slight edge in accuracy over (5.11).

To improve the conditioning of the eigenvalues of the pencil $A - z^*B$, most balancing strategies are based on a two-sided diagonal transformation. The LAPACK [10] routine ZGGBAL implements Ward's algorithm [241], which aims to scale the elements of A and B such that their magnitudes are as close to unity as possible. Unfortunately, LAPACK's driver routine ZGGEV for computing the generalized eigenvalues of a pair of complex nonsymmetric matrices (and, consequently, MATLAB) disables Ward's algorithm by default. More recently, Lemonnier and Van Dooren [128] suggest to (approximately) solve the convex optimization problem

$$\underset{\det(D_l \cdot D_r)=1}{\text{minimize}} \quad \|D_l \cdot A \cdot D_r\|_{\mathrm{F}}^2 + \|D_l \cdot B \cdot D_r\|_{\mathrm{F}}^2 \tag{5.18}$$

for the real positive diagonal matrices D_l and D_r. In fact, this objective corresponds to searching for D_l and D_r so that $D_l^2 \cdot (|A|^2 + |B|^2) \cdot D_r^2$ is a doubly stochastic matrix, i.e., having row and column sums equal to one. The Sinkhorn–Knopp (SK) algorithm is perhaps the most simple method for finding a doubly stochastic scaling of a nonnegative matrix. Its simplicity has led to the repeated discovery of the method [111], of which the algorithm proposed in [128] is another variation. Given a nonnegative matrix $M := |A|^2 + |B|^2$, SK alternately solves for D_l and D_r by fixing D_r (D_l) and then setting the column (row) sums equal to one with the iterates

$$D_l^2 \leftarrow \operatorname{diag}(M \cdot D_r^2 \cdot \mathbf{1}_{dn\times 1})^{-1} \text{ and} \tag{5.19a}$$

$$D_r^2 \leftarrow \operatorname{diag}(\mathbf{1}_{1\times dn} \cdot D_l^2 \cdot M)^{-1}, \tag{5.19b}$$

respectively. The main cost of the SK iterates (5.19) are the left and right matrix-vector products with M, which are $O(d^2n^2)$ floating point operations (flop) each. Depending on the number of iterations, SK can be close to as expensive as solving the GEP (5.11) itself, which costs $O(d^3n^3)$ flop. To reduce the impact of SK on the total computational time, the block structure in M can be exploited in the matrix-vector products. Taking only the block structure and identity matrices into account reduces the computational complexity to $O(dn^2)$ flop per matrix-vector product, which is small enough in practice. Before applying D_l and D_r, their entries are rounded to the nearest power of two so that the transformation does not introduce round-off error.

5.3.3 Filtering spurious solutions with Newton–Raphson

Removing spurious solutions introduced by the resultant or by round-off error is not as easy as testing if the polynomials evaluated at the candidate roots are

(close to) zero, even after normalization. This is because evaluating a polynomial may be, and often is, ill-conditioned, especially as the modulus of the candidate root increases. Even taking the condition number into account does not improve such a filter's robustness much. We propose to refine the roots with a (complex) Newton–Raphson algorithm [206], and simultaneously remove spurious solutions based on the step length. For candidate roots close to a root of the polynomial system, the step length will be small relative to the modulus of the root, while the step length for spurious roots tends to be large relative to its modulus. We underline that this approach is only heuristic and hence not guaranteerd to remove all spurious solutions. Moreover, Newton–Raphson is known to fail for certain bivariate polynomial systems such as the Griewank–Osborne example [21, 87], where it iterates away from the isolated root at the origin for any starting point in a punctured neighbourhood of that root. Even though such systems are not common in practice, we limit the number of Newton–Raphson iterations and only invert the (numerically) nonsingular components of the Jacobian to prevent good candidate solutions from straying too far from the system's roots.

Writing the bivariate polynomial system $p(x, y) = q(x, y) = 0$ as a vector-valued residual function $F^{p,q}(x, y) := (p(x, y), q(x, y))$, the Newton–Rapshon step $(\Delta x, \Delta y)$ for a candidate root (x^*, y^*) is computed as

$$\begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix} = (J^{p,q}(x^*, y^*))^\dagger \cdot F^{p,q}(x^*, y^*), \tag{5.20a}$$

where $J^{p,q}$ is the Jacobian $\partial F^{p,q}/\partial(x, y)^{\mathrm{T}}$. The next iterate is then $(x^*, y^*) \leftarrow (x^*, y^*) + (\Delta x, \Delta y)$.

In the polyanalytic case $p(z, \overline{z}) = q(z, \overline{z}) = 0$, the standard way of computing the Newton–Raphson step at a candidate root z^* is to write $p(z, \overline{z}) = u_p(z, \overline{z}) + iv_p(z, \overline{z})$ and $q(z, \overline{z}) = u_q(z, \overline{z}) + iv_q(z, \overline{z})$, wherein u_p, v_p, u_q and v_q are real-valued polyanalytic polynomials. Define the vector-valued residual function $F_{\mathbb{R}}^{u_p,u_q,v_p,v_q}(z, \overline{z}) := (u_p(z, \overline{z}), u_q(z, \overline{z}), v_p(z, \overline{z}), v_q(z, \overline{z}))$ and let x and y be the real and imaginary part of z, respectively. The Newton–Raphson step $(\Delta x, \Delta y)$ at a point z^* can then be computed as

$$\begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix} = \left(J_{\mathbb{R}}^{u_p,v_p,u_q,v_q}(z^*, \overline{z}^*)\right)^\dagger \cdot F_{\mathbb{R}}^{u_p,u_q,v_p,v_q}(z^*, \overline{z}^*), \tag{5.20b}$$

where $J_{\mathbb{R}}^{u_p,u_q,v_p,v_q}$ is the real Jacobian $\partial F_{\mathbb{R}}^{u_p,u_q,v_p,v_q}/\partial(x, y)^{\mathrm{T}}$. With a step of this form, the next iterate is $z^* \leftarrow z^* + (\Delta x + i\Delta y)$. However, obtaining u_p, v_p, u_q and v_q and their partial derivatives given only $p(z, \overline{z})$ and $q(z, \overline{z})$ is cumbersome. The real Jacobian may be obtained much more elegantly from a complex Taylor series of the complex vector-valued residual function

$F^{p,q}(z,\overline{z}) := (p(z,\overline{z}), q(z,\overline{z}))$. First, we compute its complex Jacobian

$$J_{\mathbb{C}}^{p,q} := \begin{bmatrix} \dfrac{\partial p}{\partial z} & \dfrac{\partial p}{\partial \overline{z}} \\ \dfrac{\partial q}{\partial z} & \dfrac{\partial q}{\partial \overline{z}} \end{bmatrix},$$

where the partial derivatives are so-called Wirtinger derivatives and are with respect to z ($\overline{z}$), while treating $\overline{z}$ (z) as constant. By defining

$$J_{\mathbb{R}}^{p,q} := J_{\mathbb{C}}^{p,q} \cdot \begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix},$$

the real Jacobian can then be recovered from the identity [206]

$$J_{\mathbb{R}}^{u_p,v_p,u_q,v_q} \equiv \begin{bmatrix} \mathrm{Re}\{J_{\mathbb{R}}^{p,q}\} \\ \mathrm{Im}\{J_{\mathbb{R}}^{p,q}\} \end{bmatrix}. \tag{5.21}$$

The advantage of (5.21) is that evaluating $J_{\mathbb{C}}^{p,q}$ requires only four complex function evaluations, compared to eight real function evaluations for $J_{\mathbb{R}}^{u_p,u_q,v_p,v_q}$ together with the cost of constructing of the four polynomials u_p, v_p, u_q and v_q. It is interesting to note that $(J_{\mathbb{R}}^{p,q}(z^*,\overline{z}^*))^\dagger \cdot F^{p,q}(z^*,\overline{z}^*)$ computes a complex Newton–Raphson step, while the form (5.20b) ensures that the step will be real.

It is recommended to solve the least squares systems (5.20) either by regularizing the system or by approximating the Moore–Penrose pseudoinverse using the singular value decomposition (SVD). This is especially important near multiple roots, where the Jacobian tends to a singular matrix. Since the Jacobian is a small matrix of size 2×2 or 4×2, we choose the SVD and invert the first component if $\sigma_1 > \tau_\sigma$ and additionally invert the second component if $\sigma_2 > \sigma_1\tau_\sigma$, where σ_1 and σ_2 are the first two singular values of the Jacobian and τ_σ is a user-defined tolerance.

Next, we describe a method to remove spurious roots based on the Newton–Raphson step length which we have found to work well in practice. Let $(\Delta x, \Delta y)$ be a solution of (5.20a) or (5.20b) at a candidate root (x^*, y^*) or z^*, respectively. If the candidate root is close to a root of the polynomial system, it may be expected that its Newton–Raphson step will be small in comparison to a measure of the size of the root. Based on this observation, we identify a candidate root of a bivariate system as spurious if

$$\|(\Delta x, \Delta y)\| \geq \tau_{\mathrm{NR}} \cdot \max(\|(x^*, y^*)\|, 1) \tag{5.22a}$$

and of a polyanalytic system if

$$\|(\Delta x, \Delta y)\| \geq \tau_{\mathrm{NR}} \cdot \max(|z^*|, 1) \tag{5.22b}$$

holds for some tolerance τ_{NR}. This condition corresponds to a relative tolerance on the step length outside the unit circle, and an absolute tolerance inside the unit circle. The smaller τ_{NR}, the less likely spurious roots are returned as solutions, but also the more likely valid roots are discarded.

In summary, we describe the complete process for solving bivariate and polyanalytic univariate polynomial systems in Algorithm 5.1. In the following section, we investigate the performance of this and other algorithms on a set of bivariate systems, a subset of which are considered difficult due to the wide variation in exponents of the polynomial's coefficients. We will see that Algorithm 5.1 is not only able to recover the largest fraction of roots, but also does this accurately and efficiently.

5.4 Numerical experiments

We compare the relative performance of Algorithm 5.1, Bertini 1.4 [21], Chebfun 4.3.2987 [230], PHCpack 2.3.84 [237], the RealSolving C library [185, 186], the RegularChains package [12] and Maple's BivariatePolynomial method on a wide range of bivariate polynomial systems using the performance profiles proposed by Dolan and Moré [65]. Algorithm 5.1 was implemented as part of the MATLAB toolbox Tensorlab [211], which is free for academic and non-commercial use. Bertini has a variable precision (VP) and double precision (DP) mode, both of which are evaluated in the numerical experiments. Chebfun currently has two algorithms for computing the common roots of two bivariate functions. In our experiments, we use the recently added method based on the Bézout resultant [150]. Chebfun is the only software package in this comparison which chooses to represent the functions, in this case polynomials, in the product Chebyshev basis. Bertini and PHCpack are well-known homotopy continuation solvers. The RealSolving library is based on reducing the polynomial system to a rational univariate representation, from which the solutions can easily be recovered. The RegularChains package computes a triangular decomposition of the system, i.e., a set of simpler systems called regular chains which represent the solutions of the original system. Similar to but less advanced than Chebfun, Maple's BivariatePolynomial algorithm applies a random orthogonal change of variables, computes the y-coordinates of the isolated roots as the eigenvalues of the polynomial matrix associated with the Bézout resultant and recovers the x-coordinates from the corresponding eigenvectors. The first two algorithms were applied in MATLAB 8.1 (R2013a), Bertini and PHCpack as standalone executables, and the remaining three methods using Maple 17. In each experiment, all algorithms were applied without supplying any parameters so that each method uses its default parameter values. In the case of Bertini,

Input: Two bivariate or polyanalytic univariate polynomials p and q
Output: Isolated real roots (x^*, y^*) or complex roots z^* of the system $p = q = 0$

$\{P, Q\} \leftarrow$ Apply (5.13) to center the exponents of the coefficients
if *p and q are bivariate* **then**
  $\{r_x, r_y\} \leftarrow$ Solve (5.15)
  $\{r_x, r_y\} \leftarrow \{\text{round}(r_x), \text{round}(r_y)\}$
  $\{\hat{P}, \hat{Q}\} \leftarrow$ Apply the transformation (5.14) to balance the system
  $\{\hat{P}, \hat{Q}\} \leftarrow$ Apply the transformation (5.12) to obtain a polyanalytic system
end
$\{s_z, s_{z^*}\} \leftarrow$ Solve (5.17)
$\{s_z, s_{z^*}\} \leftarrow \{\text{round}(s_z), \text{round}(s_{z^*})\}$
$\{\hat{P}, \hat{Q}\} \leftarrow$ Apply the transformation (5.16) to balance the polyanalytic system
$\{\hat{P}, \hat{Q}\} \leftarrow$ Apply (5.13) to $\{\hat{P}, \hat{Q}\}$ to center the exponents of the coefficients
$A - z^*B \leftarrow$ Build (5.11) associated with eliminating $\overline{z}$ from $\hat{p} = \hat{q} = 0$
$\{D_l, D_r\} \leftarrow$ Approximately solve (5.18) with SK
$\{D_l, D_r\} \leftarrow \{\text{diag}(2^{\text{round}(\log_2(\text{diag}(D_l)))}), \text{diag}(2^{\text{round}(\log_2(\text{diag}(D_r)))})\}$
$\boldsymbol{z}^* \leftarrow \text{eig}(D_l \cdot A \cdot D_r, D_l \cdot B \cdot D_r)$
foreach *z^* in $\boldsymbol{z}^*$* **do**
  if *p and q are bivariate* **then**
    $(x^*, y^*) \leftarrow$ Decode $z^* = x^* + iy^*$
  end
  for $i = 1$ **to** maxiter *(e.g.,* maxiter $= 4$*)* **do**
    $(\Delta x, \Delta y) \leftarrow$ Solve (5.20) at (x^*, y^*) or z^* (for, e.g., $\tau_\sigma = 10^{-6}$)
    if *(5.22) holds (for, e.g., $\tau_{\text{NR}} = 10^{-2}$)* **then**
      Discard (x^*, y^*) or z^* and break
    end
    if *p and q are bivariate* **then**
      $(x^*, y^*) \leftarrow (x^*, y^*) + (\Delta x, \Delta y)$
    else
      $z^* \leftarrow z^* + (\Delta x + i\Delta y)$
    end
  end
end

Algorithm 5.1: Compute isolated roots of a bivariate or polyanalytic univariate polynomial system.

we supplied the parameter `MPTYPE: 0` to select its DP mode, and omitted this parameter to apply the default VP mode. All experiments were performed on a server with two hexacore Intel Xeon E5645 CPUs and 48 GB RAM.

Problem	**Total degree**	**Coefficient range**	**Singular**
`ex001`–`ex025` [31]	$(3, 2)$ to $(11, 10)$	$(0.30, 0.00)$ to $(8.79, 1.71)$	68%
`C1`–`C5`, `D1`–`D2`, `M1`–`M4`, `R1`–`R3` [63]	$(2, 2)$ to $(16, 5)$	$(0.00, 0.00)$ to $(8.05, 8.65)$	71%
`Q21`–`Q210`, `X` [177]	$(3, 3)$	$(0.60, 0.42)$ to $(2.40, 2.47)$	0%

Table 5.1: Properties of the initial low-degree problem set, containing a total of 50 systems.

Problem	**Total degree**	**Coefficient range**	**Condition range**
`13_sings_9`	$(9, 8)$	$(6.91, 6.26)$	$(0.54, 9.31)$
`compact_surf`	$(18, 17)$	$(3.76, 3.76)$	$(0.02, 16.28) \cup \infty$
`curve24`	$(29, 28)$	$(4.95, 4.68)$	$(0.13, 1.66)$
`curve_issac`	$(16, 15)$	$(3.43, 4.24)$	$(0.28, 16.23) \cup \infty$
`cusps_and_flexes`	$(9, 8)$	$(4.74, 4.29)$	$(0.22, 8.36)$
`deg16_7_curves`	$(32, 11)$	$(3.16, 1.08)$	$(0.15, 2.87) \cup \infty$
`dfold_10_6`	$(32, 31)$	$(4.97, 4.46)$	$(0.07, 1.41) \cup \infty$
`grid_deg_10`	$(10, 9)$	$(9.08, 8.16)$	$(0.11, 3.27)$
`hard_one`	$(27, 10)$	$(28.04, 7.82)$	$(6.65, 20.85)$
`huge_cusp`	$(8, 7)$	$(4.79, 4.16)$	$(0.37, 7.96)$
`L4_circles`	$(16, 15)$	$(8.05, 8.65)$	$(15.52, 19.43) \cup \infty$
`lebesgue`	$(20, 20)$	$(6.22, 6.53)$	$(0.10, 1.64)$
`spiral29_24`	$(29, 29)$	$(10.94, 4.95)$	$(0.13, 6.66)$
`ten_circles`	$(20, 19)$	$(6.10, 5.57)$	$(0.92, 31.39) \cup \infty$
`tryme`	$(34, 25)$	$(7.68, 6.96)$	$(3.50, 13.14)$
`vert_lines`	$(16, 14)$	$(2.15, 2.14)$	$(0.49, 1.75)$

Table 5.2: Properties of the initial moderate-degree problem set, containing a total of 16 systems. All of these systems with the exception of `lebesgue` were taken from [23].

Each algorithm's performance is evaluated on two problem sets[1]. The first set is generated using the low-degree bivariate polynomial systems listed in Table 5.1. The second set is more difficult and is generated using the bivariate polynomial systems listed in Table 5.2. The zero level lines and their intersections of two example systems of each type are displayed in Figure 5.1 and Figure

[1] Available at `http://esat.kuleuven.be/stadius/tensorlab/datasets/bivarsys.zip`.

5.2. Both tables show the total degrees of the system and the exponent ranges of the coefficients. For a polynomial system $p(x, y) = q(x, y) = 0$ the total degree is displayed as (d_p, d_q), while the exponent range is defined as $(\text{range}(p(x, y)), \text{range}(q(x, y)))$, where

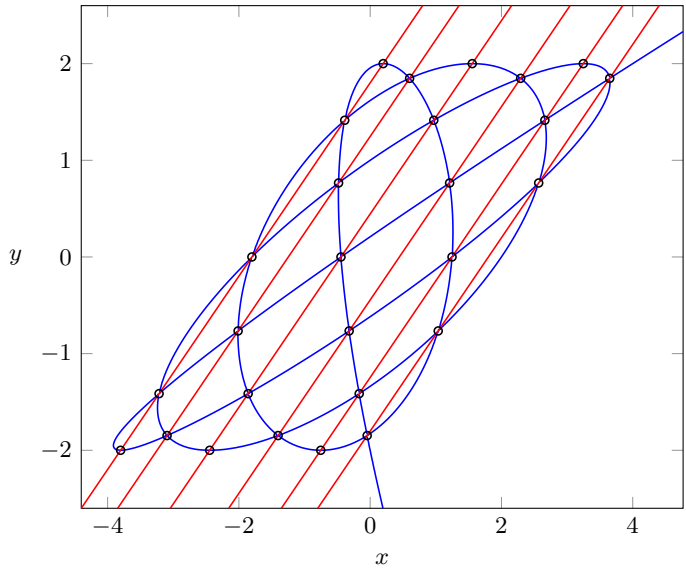
$$\text{range}(p(x, y)) := \max_{i,j} \log_{10}(|p_{ij}|) - \min_{\substack{i,j \\ p_{ij} \neq 0}} \log_{10}(|p_{ij}|).$$

The total degrees define the size of the eigenproblem, which is generically equal to $\max(d_p, d_q)(d_p + d_q)$. Large exponent ranges are usually associated with difficult problems since round-off errors tend to increase in magnitude as the exponent range increases. Table 5.1 also includes the fraction of systems with at least one solution where the Jacobian is singular. Furthermore, Table 5.2 shows the exponent range of the (finite) condition numbers of the Jacobian evaluated at all solutions of the corresponding system. If a system has at least one solution where the Jacobian is singular, we indicate this by taking the union with infinity.

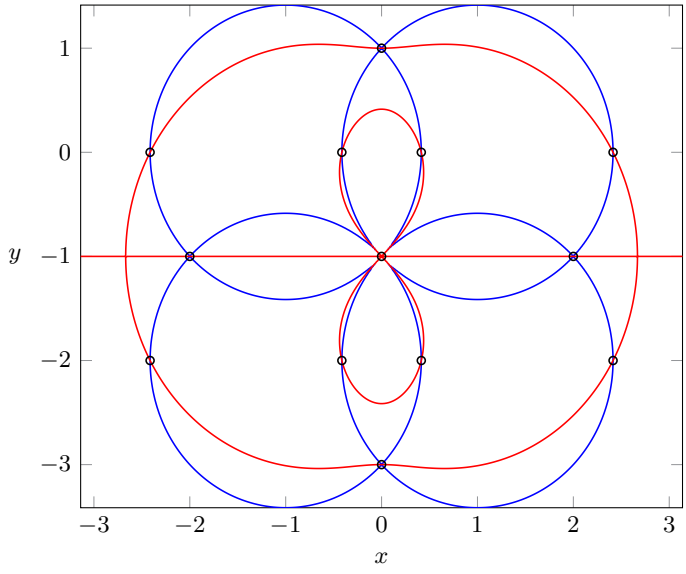
All systems are given in the product monomial basis, often with integer coefficients. In some cases these coefficients contain more digits than can be represented by double precision floating point numbers. For a fair comparison, each algorithm receives the systems in a normalized format. More specifically, each polynomial's coefficients are converted to hardware double precision floating point numbers, after which they are multiplied by an integer power of two such that their exponents are approximately centered around zero.

Some of the systems are of the form $p(x, y) = \frac{\partial p}{\partial y}(x, y) = 0$ and correspond to finding roots of $p(x, y)$ that are tangent to a vertical line. Taking inspiration from this, we artificially expand the low-degree and medium-degree problem sets by taking each system $p(x, y) = q(x, y) = 0$ and adding the systems $p(x, y) = \frac{\partial p}{\partial x}(x, y) = 0$ and $\frac{\partial p}{\partial x}(x, y) = \frac{\partial p}{\partial y}(x, y) = 0$, tripling the total number of systems. The only exception to this rule are the systems `Q21`–`Q210` and `X`, which are already of the form $\frac{\partial p}{\partial x}(x, y) = \frac{\partial p}{\partial y}(x, y) = 0$ and correspond to finding the stationary points of normal quartics [177].

For each system in the expanded low-degree and moderate-degree problem sets, we compute a reference solution set as follows. First, all solutions of all systems as computed by all eight algorithms are collected. There is no discrimination between real or complex solutions and valid or spurious solutions. For each such candidate root (x^*, y^*), we apply a high-precision Newton–Raphson method with 128 significant decimal digits to $\text{Re}\{(x^*, y^*)\}$. We terminate the refinement after 500 iterations or when the relative step size is less than $\epsilon_{\text{mach}} 10^{-3}$, where ϵ_{mach} is the machine epsilon for double precision. The refined root (x^*, y^*) is

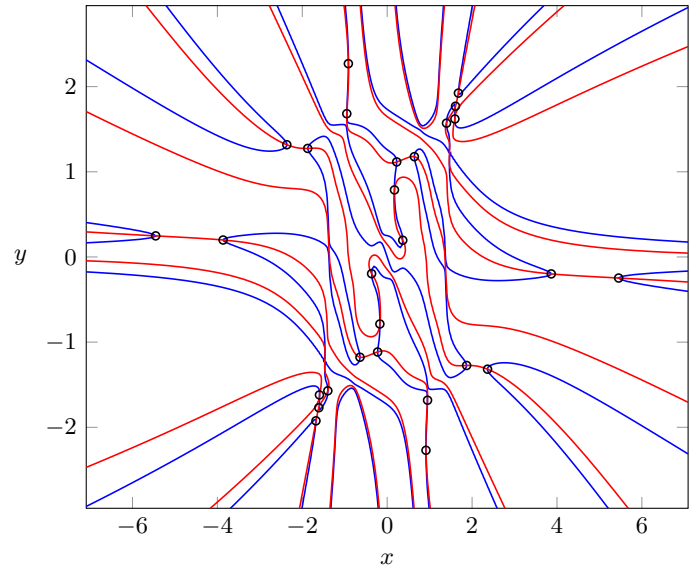


(a) `ex008`

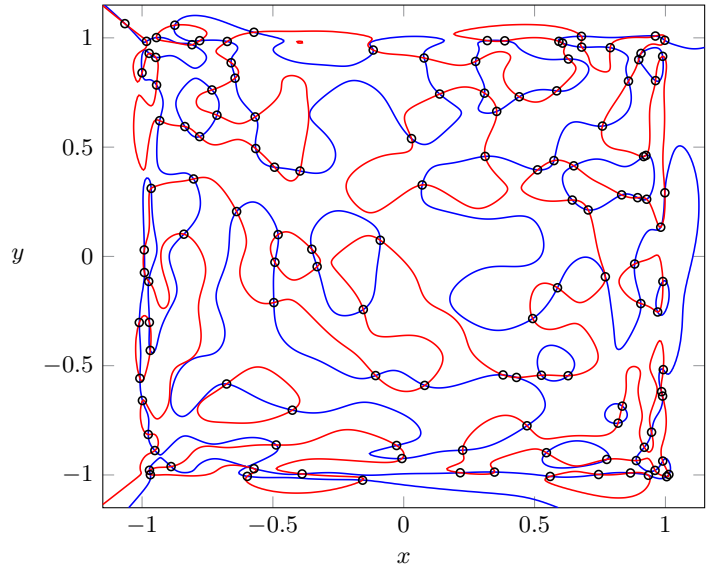


(b) `C3`

Figure 5.1: Two examples of the low-degree bivariate polynomial systems and their solutions.

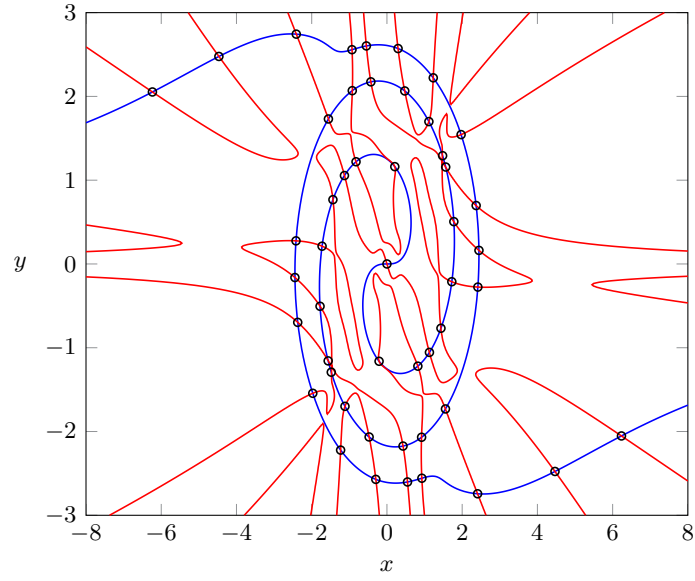


(a) `curve24`

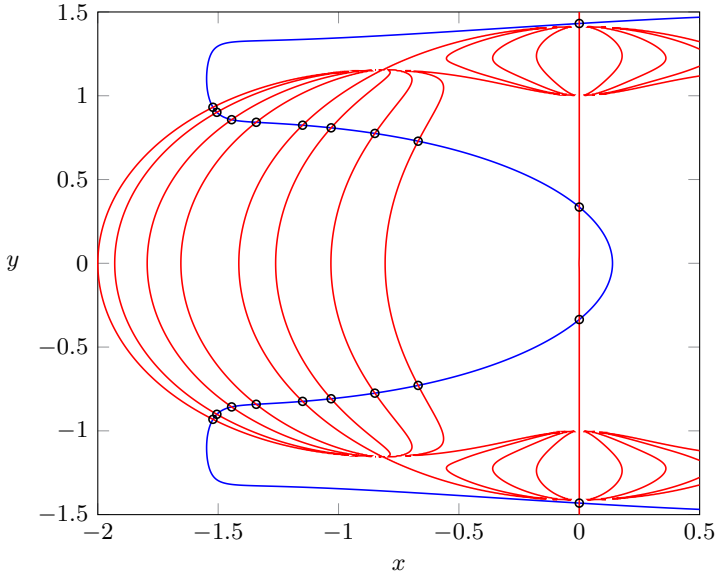


(b) `lebesgue`

Figure 5.2: Two examples of the moderate-degree bivariate polynomial systems and their solutions.



(a) `spiral29_24`



(b) `tryme`

Figure 5.3: Two more examples of the moderate-degree bivariate polynomial systems and their solutions.

accepted as a valid root if both conditions

$$|p(x^*, y^*)| \leq |p|(|x^*|, |y^*|)\epsilon_{\text{mach}} 10^3 \text{ and}$$

$$|q(x^*, y^*)| \leq |q|(|x^*|, |y^*|)\epsilon_{\text{mach}} 10^3,$$

hold when evaluated in high-precision. Here, the polynomials $|p|$ and $|q|$ correspond to the coefficient matrices $|P|$ and $|Q|$, respectively. Evaluated at the absolute value of (x^*, y^*), they are equal to the condition number of evaluating the polynomials p and q at (x^*, y^*). If accepted, the candidate root is truncated to double precision and added to the system's reference solution set. After all candidate roots have been processed, the reference solution sets are reduced to their distinct elements.

For each problem set P, the performance profiles are computed as follows. Let S be the set of solvers consisting of the previously mentioned eight algorithms and let $t_{p,s}$ denote the time required by solver s to solve problem p. If solver s did not successfully solve problem p, we set $t_{p,s} = \infty$. The performance profile for solver s,

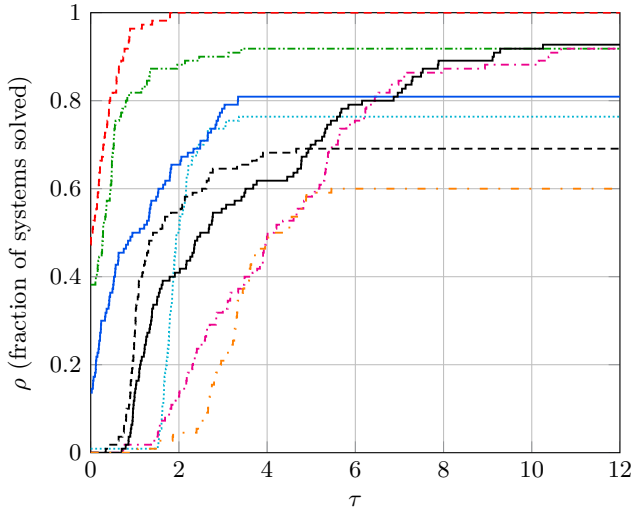
$$\rho_s(\tau) := \frac{|\{p \in P \,|\, t_{p,s} \leq 2^\tau \cdot (\min_{s \in S} t_{p,s})\}|}{|P|},$$

depicts the fraction of problems that solver s was able to solve within a factor 2^τ of the fastest solution time for that problem. We say that a polynomial system p has been solved successfully by solver s if for at least 99% of all roots (x^*, y^*) in the reference solution set, there is a candidate root $(\hat{x}^*, \hat{y}^*)$ computed by s which satisfies

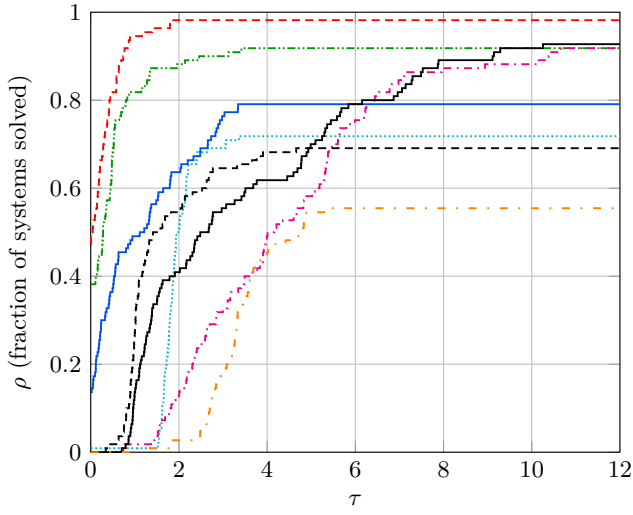
$$\|(x^*, y^*) - (\hat{x}^*, \hat{y}^*)\| \leq \max(\|(x^*, y^*)\|, 1)\epsilon_{\max}$$

for a certain maximal relative error $\epsilon_{\max}$. Chebfun computes roots on a rectangular domain specified by the user, which is $[-1, 1] \times [-1, 1]$ by default. For each system, we set this domain equal to the smallest axis-aligned bounding box of the corresponding reference solution set, plus a small margin to prevent edge cases.

Figures 5.4 and 5.5 show the performance profiles of the expanded low-degree and moderate-degree problem sets, respectively, for two choices of $\epsilon_{\max}$. The left hand side, where $\tau = 0$, of each plot shows the fraction of successfully solved problems for which a solver was the fastest solver. The right hand side, where τ is large, shows the fraction of problems a solver was able to successfully solve. Among the fixed precision solvers, Algorithm 5.1 was able to successfully solve the largest fraction of problems and was also considerably faster than most other algorithms for both problem sets and choices of $\epsilon_{\max}$, with the

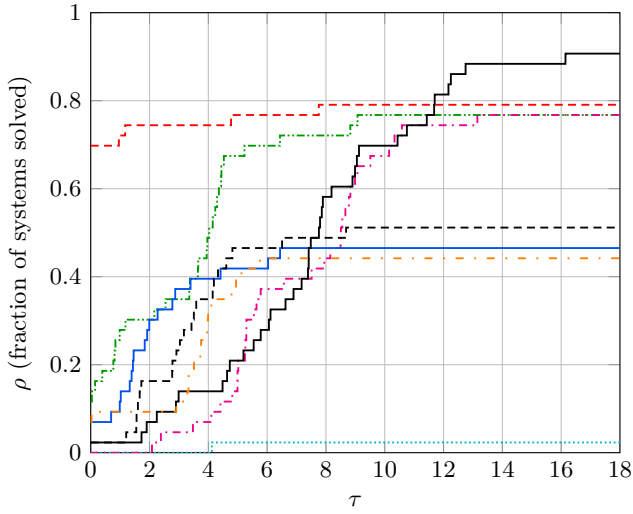


(a) $\epsilon_{\max} = 5\%$

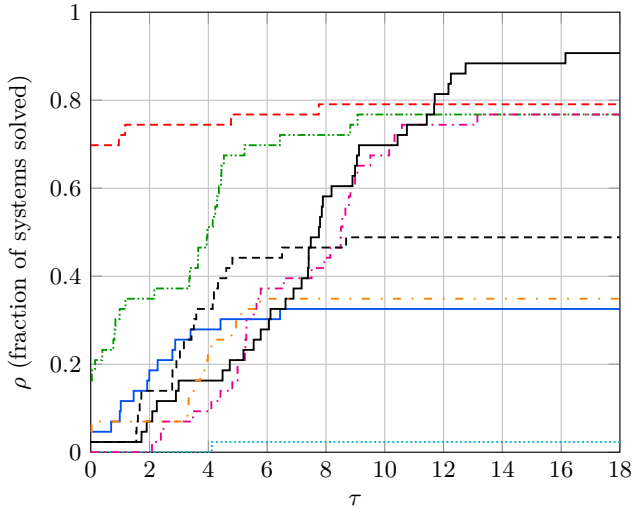


(b) $\epsilon_{\max} = 0.5\%$

Figure 5.4: Performance profiles for low-degree bivariate polynomial systems. The problem set consists of all these polynomial systems, and we say a system has been successfully solved if at least 99% of its roots have been found with a relative error that does not exceed $\epsilon_{\max}$. Legend: our method (----), PHCpack (——), Bertini VP (——), Bertini DP (----), Chebfun (- - -), RealSolving (-·-·-), RegularChains (-·-·-) and BivariatePolynomial (·····).



(a) $\epsilon_{\max} = 5\%$



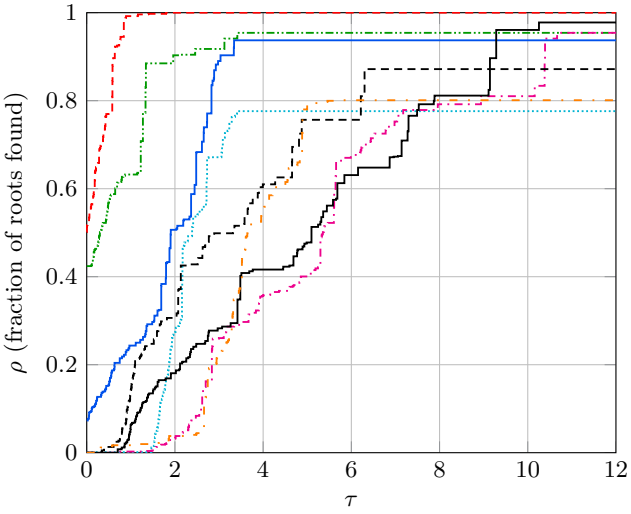
(b) $\epsilon_{\max} = 0.5\%$

Figure 5.5: Performance profiles for moderate-degree bivariate polynomial systems. The problem set consists of all these polynomial systems, and we say a system has been successfully solved if at least 99% of its roots have been found with a relative error that does not exceed $\epsilon_{\max}$. Legend: our method (- - - -), PHCpack (——), Bertini VP (——), Bertini DP (- - - -), Chebfun (- · -), RealSolving (-·-·-), RegularChains (-·-·-) and BivariatePolynomial (·····).

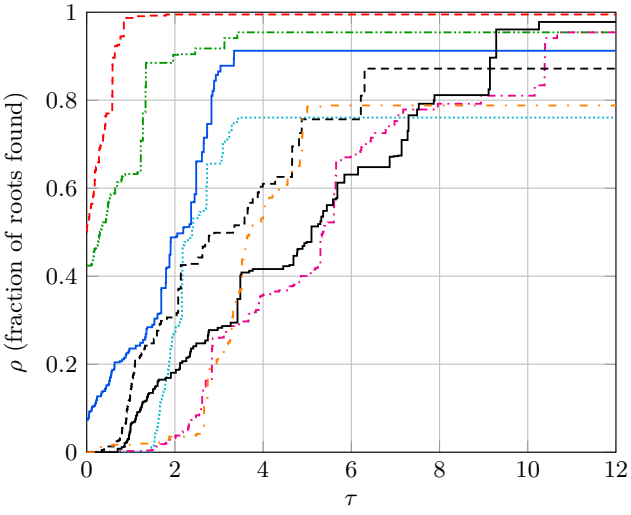
RealSolving library as a close second for the low-degree problem set. Bertini was able to solve an impressive 90% of the moderate degree problems in its variable precision mode, which is likely due to the increased flexibility that variable precision offers. Concerning the accuracy of the computed solutions, PHCpack and Chebfun seem to be sensitive to the tolerance $\epsilon_{\max}$, especially on the moderate-degree problem set. The other algorithms' performance profiles remain relatively static when $\epsilon_{\max}$ is decreased, indicating that their computed solutions are relatively more accurate.

Compared to the low-degree problem set, the fraction of moderate-degree systems that were successfully solved is much lower. These systems often have many more solutions than their low-degree counterparts. Since a solver by definition fails to solve a system as soon as over 1% of these roots is not found, we compute a second set of performance profiles to examine the fraction of successfully recovered roots as follows. We define two new problem sets containing all reference solutions of the expanded low-degree systems and moderate-degree systems, respectively. The time for solver s to solve problem p is then defined as the total time for solver s to compute all roots of the bivariate polynomial system associated with p, divided by the number of roots of the system's reference solution set. If solver s was not able to successfully recover the root p, we set $t_{p,s} = \infty$. The condition for successfully recovering a root is identical to that of the previous performance profiles.

The performance profiles based on the redefined problem sets corresponding to the expanded low-degree and moderate-degree systems are shown in Figure 5.6 and 5.7, respectively. Algorithm 5.1 was able to recover nearly all roots of both problem sets and for both values of ϵ_{mach}. The roots that it did not recover, were spread out among different systems, causing the fraction of successfully solved moderate-degree systems to drop to around 80% (cf. Figure 5.5). Interestingly, the RealSolving library successfully solved the (close to) second largest fraction of systems, yet other solvers recover a higher fraction of roots in Figure 5.7. This indicates that the RealSolving library is somewhat of a hit or miss algorithm, either recovering all solutions or very few. In contrast, Bertini's double precision mode, PHCpack and Chebfun are able to recover a larger fraction of roots, but fail to recover at least one root for many moderate-degree systems. Furthermore, we observe that for moderate-degree systems around 20% of the roots computed by PHCpack and Chebfun are only accurate up to $\epsilon_{\max} = 5\%$.

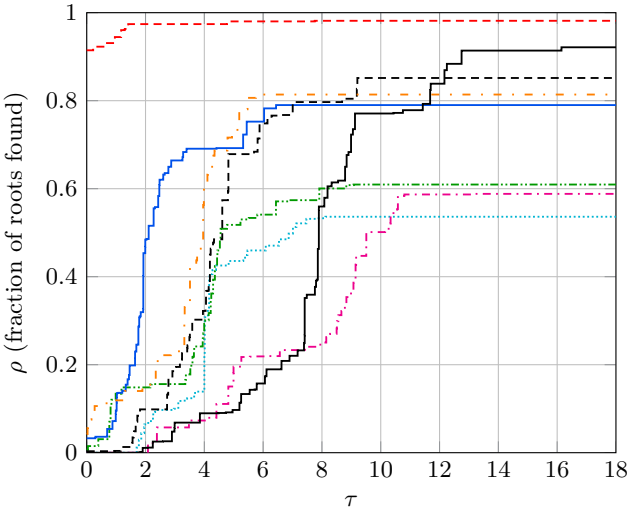


(a) $\epsilon_{\max} = 5\%$

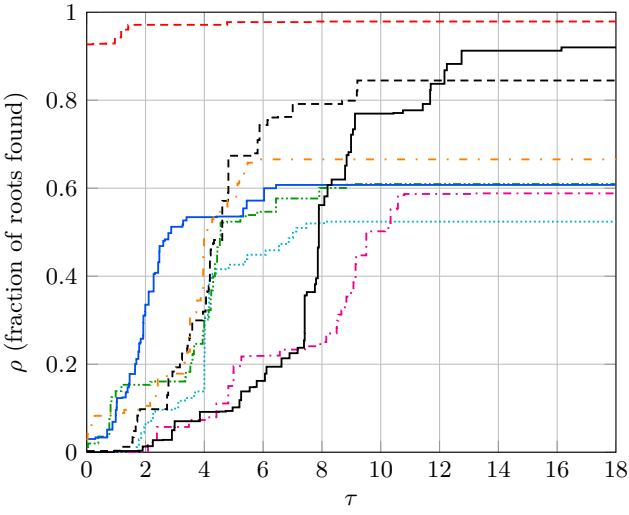


(b) $\epsilon_{\max} = 0.5\%$

Figure 5.6: Performance profiles for low-degree bivariate polynomial systems. The problem set consists of all distinct roots of these systems, and we say a root has been successfully found if its relative error does not exceed $\epsilon_{\max}$. Legend: our method (----), PHCpack (——), Bertini VP (——), Bertini DP (----), Chebfun (- - -), RealSolving (-·-·-), RegularChains (-·-·-) and BivariatePolynomial (·····).



(a) $\epsilon_{\max} = 5\%$



(b) $\epsilon_{\max} = 0.5\%$

Figure 5.7: Performance profiles for moderate-degree bivariate polynomial systems. The problem set consists of all distinct roots of these systems, and we say a root has been successfully found if its relative error does not exceed $\epsilon_{\max}$. Legend: our method (----), PHCpack (——), Bertini VP (——), Bertini DP (----), Chebfun (- - -), RealSolving (-·-·-), RegularChains (-··-··-) and BivariatePolynomial (·····).

5.5 Conclusion

We presented an algorithm for computing the isolated real solutions of bivariate polynomial systems, and the isolated complex solutions of polyanalytic polynomial systems. For bivariate systems in two variables, say x and y, the algorithm projects the solutions onto the real plane associated with x and y by combining a well chosen change of variables together with a resultant-based elimination of a hidden variable. In this way, the problem is reduced to computing the eigenvalues of a polynomial eigenvalue problem. In comparison to other eigen-based approaches, there is no need to compute the accompanying eigenvectors or solve any additional eigenproblems. To increase the algorithm's robustness, we presented three methods of balancing the eigenvalue problem and an elegant method to remove spurious roots. Though the algorithm was chiefly developed for the polynomials expressed in the product monomial basis, the generalization to other bases is straightforward. The numerical experiments show that our method often outperforms other algorithms in the fraction of roots successfully recovered, the accuracy of the estimated roots and computational time. The algorithms discussed in this chapter were implemented as part of the MATLAB toolbox Tensorlab [211] and are available online.

Chapter 6

Structured data fusion

We present structured data fusion (SDF) as an optimization-based framework for the rapid prototyping of knowledge discovery in one or more possibly incomplete data sets. In SDF, each data set—stored as a dense, sparse or incomplete tensor—is factorized with a tensor decomposition. Factorizations can be coupled, or fused, with each other by indicating which factors should be shared between data sets. At the same time, factors may be imposed to have any type of structure that can be constructed as an explicit function of some underlying variables. With the right choice of decomposition type and factor structure, even well-known matrix factorizations such as the eigenvalue decomposition, singular value decomposition and QR factorization can be computed with SDF. A domain specific language (DSL) for SDF is implemented as part of the software package Tensorlab, with which we offer a library of tensor decompositions and factor structures to choose from. The versatility of the SDF framework is demonstrated by means of four diverse applications, which are all solved entirely within Tensorlab's DSL. This chapter is a slightly adapted version of the technical report [209].

6.1 Introduction

Successor to the industrial age and fostered by the digital revolution, the information age is characterized by an ever increasing rate of data production and consumption. One of the greatest challenges we face is how to make sense out of the abundance of available information. Although many data sets take the form of matrices, they are increasingly surfacing as multidimensional arrays, also known as higher-order tensors. Due to the curse of dimensionality, aggregating, measuring, computing or storing all entries of high-dimensional tensors is often impracticable. Yet even incomplete tensors—of which only a fraction of entries are known—can require vast amounts of storage. The possibilities for leveraging *big data* continue to evolve rapidly, further fueling a growing demand for new algorithms and software to tap into this wealth of information.

Higher-order tensors also possess properties that are not present on the matrix level. In fact, there is an upcoming trend to (explicitly or implicitly) *tensorize* data sets which are otherwise naturally represented as matrices. Perhaps the most salient property of the jump to the third dimension is that tensor decompositions can be unique under mild conditions without imposing additional constraints. This is in stark contrast to matrix factorizations, which require constraints such as orthogonality or nonnegativity together with sparsity to obtain uniqueness. Factor uniqueness is a desirable property as it is a key facilitator of attaching meaning to factors, or in other words, extracting knowledge from data. However, tensor decompositions and imposing constraints on the factors are not the only means of achieving factor uniqueness.

Integrating and analyzing data from multiple sources, also known as *data fusion*, can be used to develop insights that are deeper and more accurate than those resulting from a single source of data. Similar to tensorization, data fusion provides a powerful means of introducing factor uniqueness. Where a single matrix factorization is not unique without additional constraints on the factors, the joint factorization of two matrices sharing at least one factor can be unique. Intuitively, this can be understood from the fact that if these matrices are of equal size, their joint factorization can be interpreted as a tensor decomposition. As such, tensor decompositions can be seen as a special case of data fusion where the data sets are homogeneous in size. Moreover, data fusion also allows the uniqueness of a single tensor decomposition to be transferred and reinforced between data sets by coupling their factorizations [66, 212–215].

Independent component analysis (ICA) is a form of blind source separation (BSS) which embodies the aforementioned concepts [38]. The goal is to separate a matrix of measurement data into two factors called the mixing matrix and source matrix, respectively. To fix the rotational freedom of the two factors,

matrix factorizations of the measurement data itself require constraints which may feel forced or unnatural depending on the application. For example, principal component analysis (PCA) is a related BSS technique that imposes orthogonality on the factors to obtain uniqueness. In contrast, one variety of ICA instead tensorizes the measurement data into a fourth-order cumulant tensor. The cumulant's factorization in symmetric rank-one tensors is unique under mild conditions and, hence, interpretable. In the case of ICA, the combined effect of tensorization and factorization expresses the desire that the underlying sources should be statistically independent. This can be seen as an implicit constraint which is often strong enough to elicit a unique result. Furthermore, the cumulant's information can be fused with second-order statistics such as the measurement data's covariance matrix to improve robustness and accuracy. Alternatively, a different type of ICA identifies the mixing matrix by computing several time-lagged covariance matrices and fusing them with a joint factorization [22, 51].

The concept of fusing data by joint factorization has found its way into many other disciplines. Canonical correlation analysis (CCA) is perhaps the first model aiming to capture the commonalities between two data sets [101]. Later, CCA was extended to the analysis of more than two Gramian matrices, which was dubbed multi-set CCA [42,129,133]. Data fusion is also known as multi-view [25] and multi-relational [135, 198] learning in artificial intelligence, as multi-set or multi-block data analysis in chemometrics and systems biology [201, 234, 242] and as integrative data analysis in psychometrics [43]. Multi-view data arises when an observation's attributes, or features, consist of two or more disjoint sets, or views. For example, a video's features could be separated into its visual and audio content. Relational data has a natural representation in the form of a tensor. For example, in the relation *student-performs-task*, the two entities *student* and *task* correspond to the tensor's modes or dimensions, while its entries are defined by the relation type *performs* [227]. This relation can then be coupled with the relations *task-requires-skill* and *student-has-skill* to form a multi-relational data set. Singh and Gordon proposed a framework for multi-relational learning called collective matrix factorization [198, 199] (CMF). Later, Acar et al. generalized CMF to coupled matrix and tensor factorization (CMTF) [6] for a Euclidian distance loss function.

Building on these foundations, we present structured data fusion (SDF) as a novel framework for the rapid prototyping of knowledge discovery in one or more possibly incomplete or sparse data sets. Each data set in an SDF problem is represented as a tensor and factorized with a tensor decomposition. The different factorizations are coupled with each other by indicating which (sub)factors should be shared between data sets. Factors may be imposed to have any type of structure that can be constructed as an explicit function of some

underlying variables. The scope of this framework reaches far beyond factor analysis alone, encompassing nearly the full breadth of applications resulting from matrix factorizations and tensor decompositions. It subsumes, for example, tasks based on dimensionality reduction such as feature extraction, subspace learning and model order reduction and tasks related to machine learning such as regression, classification, clustering, and imputation of missing data. Examples of applications that fit in the SDF framework and comprise at least two data sets include social network mining [166], (sub)classification of documents [132, 252, 257], images [258] and cancer [15, 137], sentiment analysis [130], cross-social media retrieval [89], collaborative and content-based filtering [73, 116, 198, 250], link prediction [70, 255], fusion of metabolomics data [4, 201, 234], predicting protein-protein interactions [246, 247, 254], multilingual text analysis [110], analysis of batch processes [202], spectral clustering [138], gene expression profiling [14, 172], gene function prediction [135], personalized medicine [125], multi-task learning [11], community discovery [134], compressed sensing [251], brain computer interfacing [126, 127], and a wide range of applications in signal processing [212, 214, 215]. Recent tensor methods and applications were surveyed in [8, 35, 112].

In SDF, the type of tensor decomposition, the coupling between factorizations and the structure imposed on the factors can all be chosen freely without any changes to the solver. By enabling the user to effortlessly switch between the countless combinations arising from these choices, he or she can rapidly iterate towards a solution for the problem at hand. The different types of tensor decompositions and factor structures are completely modularized, which opens the door to building separate libraries of these components. This goal is achieved in part by completely isolating the computational aspects related to the decomposition and factor structure from each other and the SDF solver. Furthermore, SDF is built with big data in mind—attaining linear time complexity in the total number of known elements in the data sets. As a result, the amount of effort required to add new tensor decompositions or factor structures to a software package implementing SDF such as Tensorlab [211] is kept to a minimum. The latter is a MATLAB toolbox that currently offers SDF with three types of tensor decompositions, two types of regularization and a library of 32 factor structures including nonnegativity, orthogonality and many matrix structures such as Toeplitz, Hankel and Vandermonde. With the right choice of decomposition type and factor structure, even well-known matrix factorizations such as the eigenvalue decomposition, singular value decomposition and QR factorization can be computed with SDF.

The chapter is organized as follows. Section 6.2 introduces SDF framework and reviews the most prominent tensor decompositions. In Section 6.3 we propose two families of algorithms for solving SDF problems with a computational

complexity per iteration which is linear in the number of known elements of the data sets. Moreover, the resulting algorithms are able to optimize over the complex domain just as easily as over the real domain. In Section 6.4 we apply the SDF framework on four applications: (a) accurately computing eigenvalues, (b) making personalized movie recommendations, (c) designing an alloy to exhibit certain physical properties and (d) predicting user participation in activities. We conclude the chapter in Section 6.5.

6.2 Structured data fusion

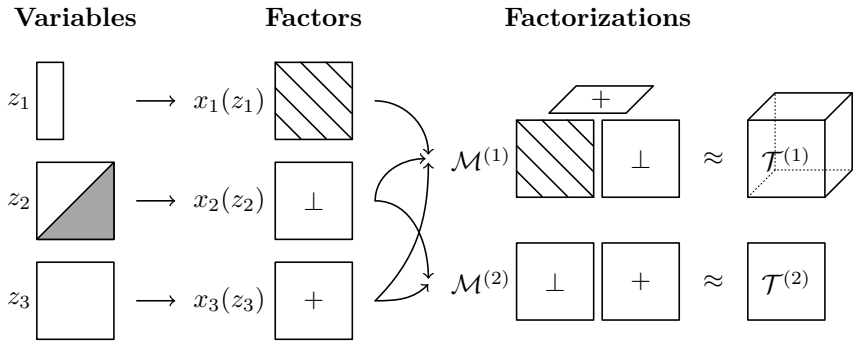


Figure 6.1: Schematic of structured data fusion. The vector z_1, upper triangular matrix z_2 (representing a sequence of Householder reflectors) and full matrix z_3 are transformed into a Toeplitz, orthogonal and nonnegative matrix, respectively. The resulting factors are then used to jointly factorize two coupled data sets.

Given D data sets $\mathcal{T}^{(d)}$ which are stored as possibly incomplete multidimensional arrays in $\mathbb{C}^{I_1 \times \cdots \times I_{N_d}}$, also called tensors, we are interested in jointly factorizing these data sets in factors which may depend (non)linearly on some underlying variables. The data sets need not be high-dimensional—vector and matrix data sets correspond to N_d equal to 1 and 2, respectively. With each tensor $\mathcal{T}^{(d)}$ we associate a so-called observation tensor $\mathcal{W}^{(d)}$ of the same size with ones in the positions corresponding to known entries of the data set and zeros elsewhere. We will use observation tensors mainly as a notational convenience for blanking out any unknown entries. The underlying variables $\boldsymbol{z}$ are stored as an ordered set of V variables $\{z_1, \ldots, z_V\}$, each defined as a real or complex scalar, vector, matrix or higher-order tensor. An ordered set of F factors $\mathcal{X}(\boldsymbol{z})$ is then defined as $\{x_1(z_{i_1}), \ldots, x_F(z_{i_F})\}$, in which the fth transformation x_f is

a smooth[1] function which maps the i_fth variable z_{i_f} to a tensor. For example, x_f could map a sequence of Householder reflectors z_{i_f} to a rectangular matrix with orthonormal columns. These factors will form the building blocks of the data sets' tensor decompositions. For each tensor $\mathcal{T}^{(d)}$, we choose a type of decomposition, or model, $\mathcal{M}^{(d)}$ to approximate that tensor with. For the moment, assume the model maps a subset of factors in $\mathcal{X}(\boldsymbol{z})$ to a tensor with dimensions equal to those of the approximated tensor. Examples of such models will be given in the following subsection. Two factorizations can be coupled with each other simply by requiring that their models share at least one factor. We then define structured data fusion as the optimization problem

$$\underset{\boldsymbol{z}}{\text{minimize}} \quad \sum_{d=1}^{D} \frac{\omega_d}{2} \left\| \mathcal{M}^{(d)}(\mathcal{X}(\boldsymbol{z})) - \mathcal{T}^{(d)} \right\|^2_{\mathcal{W}^{(d)}}, \tag{6.1}$$

where ω_d are scalar weights and $\|\cdot\|_{\mathcal{W}^{(d)}}$ is defined as the Frobenius norm $\|\mathcal{W}^{(d)} * \cdot\|_{\mathrm{F}}$, in which $*$ is the Hadamard or element-wise product. A good rule of thumb for choosing the weights ω_d may be to ensure they are inversely proportional to the number of known elements of the corresponding data sets $\mathcal{T}^{(d)}$ [242].

The motivation for this choice of norm is threefold. First, it is well-known that minimizing a sum of squares as in (6.1) gives rise to the maximum likelihood estimate of $\boldsymbol{z}$ in the presence of white Gaussian noise on the data sets $\mathcal{T}^{(d)}$. Second, the Frobenius norm is unique in that it enables a cheap Newton-like update with a computational complexity that is linear in the number of known elements, yet still retaining quadratic convergence near the solution. This is an especially important feature when dealing with big data, given that it takes about 8 hours to read a modern 4 TB hard drive from beginning to end. Third, let $\operatorname{vec}(\cdot)$ denote the column-wise vectorization of its argument and let n be the number of elements in a tensor $\mathcal{T}$. Then the equivalence of norms states that any vector norm can be used to bound another vector norm from above and below. For example, the L1-norm and infinity-norm are bounded by

$$\|\mathcal{T}\|_{\mathrm{F}} \leq \|\operatorname{vec}(\mathcal{T})\|_1 \leq \sqrt{n}\|\mathcal{T}\|_{\mathrm{F}}$$

$$\frac{1}{\sqrt{n}}\|\mathcal{T}\|_{\mathrm{F}} \leq \|\operatorname{vec}(\mathcal{T})\|_\infty \leq \|\mathcal{T}\|_{\mathrm{F}},$$

respectively. This suggests that even when a different norm is desired, it may be worthwhile to first minimize the computationally more tractable Frobenius norm as this will likely improve other measures of error as well.

[1] Here, smooth is to be understood in the sense of having continuous Wirtinger derivatives. In other words, the partial derivatives of x_f with respect to the elements of z_{i_f} and their complex conjugates $\overline{z_{i_f}}$ should exist and be continuous up to all orders.

6.2.1 Tensor decompositions

The rank of a matrix can be defined as the minimal number of rank-one matrices R whose sum $\sum_{r=1}^{R} \boldsymbol{u}_r \cdot \sigma_r \cdot \boldsymbol{v}_r^{\mathrm{T}}$ is equal to that matrix, or equivalently as the dimension of its column and row spaces, represented by the matrices U and V, respectively, in a factorization of the form $U \cdot S \cdot V^{\mathrm{T}}$. Together with some constraints on the factorization's constituents, we arrive at well-known matrix decompositions such as the singular value decomposition, eigenvalue decomposition and QR factorization, among many others.

Although these definitions of rank are compatible with each other in the matrix case, they disjoin into two different concepts for higher-order tensors. On the one hand we have the rank of a tensor, which is defined as the minimal number of so-called rank-one tensors whose sum is equal to that tensor. On the other hand, we have the multilinear rank of a tensor, which is defined as a tuple comprising the dimensions of the mode-n vector spaces of the tensor. In the matrix case, the dimension of the column space is equal to the dimension of the row space, but this is no longer true for higher-order tensors in general.

The two different views of tensor rank engender two archetypal tensor decompositions. The first is the canonical polyadic decomposition (CPD) [33, 95, 99, 100], which decomposes a tensor as the sum of a minimal[2] number of rank-one tensors (cf. Figure 6.2). The outer product of two tensors $\mathcal{A} \in \mathbb{C}^{I_1 \times \cdots \times I_P}$ and $\mathcal{B} \in \mathbb{C}^{J_1 \times \cdots \times J_Q}$ is defined by $(\mathcal{A} \circ \mathcal{B})_{i_1 \cdots i_P j_1 \cdots j_Q} := a_{i_1 \cdots i_P} \cdot b_{j_1 \cdots j_Q}$. A rank-one tensor is defined as an outer product of two or more nonzero vectors. For example, the outer product of two nonzero vectors $\boldsymbol{a} \circ \boldsymbol{b}$ is equivalent to the rank-one matrix $\boldsymbol{a} \cdot \boldsymbol{b}^{\mathrm{T}}$. The CPD model can then be written as

$$\mathcal{M}_{\mathrm{CPD}}(U^{(1)}, \ldots, U^{(N)}) := \sum_{r=1}^{R} \boldsymbol{u}_r^{(1)} \circ \cdots \circ \boldsymbol{u}_r^{(N)}, \tag{6.2}$$

where $\boldsymbol{u}_r^{(n)}$ denotes the rth column of the nth factor matrix $U^{(n)}$. The smallest integer R for which a tensor's CPD is exactly equal to that tensor is called the tensor's rank and the corresponding decomposition is called the tensor's rank decomposition. The rank is at most $(\prod_n I_n)/(\max_n I_n)$, where I_n is the length of the tensor's nth mode. In contrast to matrices, a higher-order tensor's rank can be much larger than any of its dimensions—though from a practical perspective many tensors of interest are approximated well by a CPD with only a small number of rank-one tensors.

[2] In practice, most decompositions of higher-order tensors are only approximate. Accordingly, minimal is to be understood in the sense of Occam's razor—i.e., the smallest value for which the model explains the data sufficiently well.

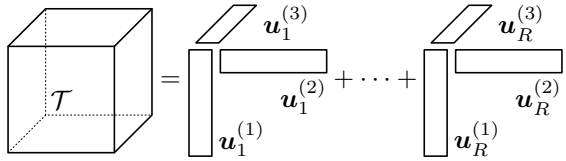


Figure 6.2: Canonical polyadic decomposition of a third-order tensor.

In addition, the rank decomposition of a higher-order tensor is unique under relatively mild conditions. Clearly, the rank-one terms in (3.1) may be arbitrarily permuted without affecting their sum. The vectors in a single rank-one term may also be arbitrarily scaled, as long as their product remains the same. A rank decomposition is called (essentially) unique when it is subject only to these permutation and scaling ambiguities. The most well-known sufficient condition for uniqueness is due to Kruskal [118, 119], which is generically satisfied if

$$R \leq \left\lceil \frac{1}{2} \sum_{n=1}^{N} \min(I_n - 1, R - 1) \right\rceil . \tag{6.3}$$

A more recent and powerful framework for the uniqueness of tensor decompositions can be found in [67, 68].

The low multilinear rank approximation (LMLRA) or Tucker decomposition [112, 232] is related to the second concept of tensor rank. It decomposes a tensor as a so-called core tensor $\mathcal{S}$, multiplied by matrices $U^{(n)}$ along each of its N modes (cf. Figure 4.1). A tensor's mode-n matricization is a matrix whose columns comprise the tensor's mode-n vectors—i.e., column vectors, row vectors, and so on. Formally, the mode-n matricization $T_{(n)}$ of a tensor $\mathcal{T} \in \mathbb{C}^{I_1 \times \cdots \times I_N}$ maps tensor element with indices $(i_1, \ldots, i_N)$ to matrix element (i_n, j) such that

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} (i_k - 1) J_k \quad \text{with} \quad J_k := \prod_{\substack{m=0 \\ m \neq n}}^{k-1} I_m, \tag{6.4}$$

wherein $I_0 := 1$. The mode-n rank of a tensor is then defined to be the rank of its mode-n matricization and the tensor's multilinear rank is represented by a tuple of all mode-n ranks. Furthermore, the mode-n tensor-matrix product $\mathcal{S} \bullet_n U^{(n)}$ of a tensor $\mathcal{S} \in \mathbb{C}^{R_1 \times \cdots \times R_N}$ and a matrix $U^{(n)} \in \mathbb{C}^{I_n \times R_n}$ is defined by the matricization $(\mathcal{S} \bullet_n U^{(n)})_{(n)} = U^{(n)} \cdot S_{(n)}$. In other words, in a mode-n tensor-matrix product each mode-n vector of a given tensor is premultiplied by a given matrix. For example, the expression $U \cdot S \cdot V^{\mathrm{T}}$ is equivalent to $S \bullet_1 U \bullet_2 V$. The LMLRA model can now be formulated as

$$\mathcal{M}_{\mathrm{LMLRA}}(U^{(1)}, \ldots, U^{(N)}, \mathcal{S}) := \mathcal{S} \bullet_1 U^{(1)} \cdots \bullet_N U^{(N)}. \tag{6.5}$$

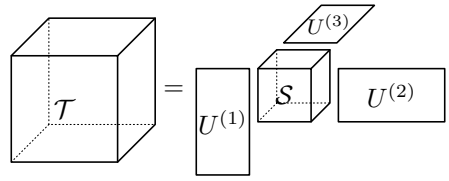


Figure 6.3: Low multilinear rank approximation of a third-order tensor.

Lastly, the block term decomposition (BTD) framework [46, 47, 55] unifies the CPD and LMLRA by combining the CPD's mild conditions for uniqueness with the more general low-rank structure of a LMLRA. A BTD decomposes a tensor as a sum of low multilinear rank terms (cf. Figure 6.4), each of which may have a different multilinear rank. Therefore, the BTD model is given by

$$\begin{aligned}\mathcal{M}_{\text{BTD}}(U_1^{(1)},\ldots,U_R^{(1)},\ldots,U_1^{(N)},\ldots,U_R^{(N)},\mathcal{S}_1,\ldots,\mathcal{S}_R)\\ := \sum_{r=1}^{R} \mathcal{S}_r \bullet_1 U_r^{(1)} \cdots \bullet_N U_r^{(N)}.\end{aligned} \tag{6.6}$$

In principle, it would suffice to exclusively implement the BTD model in the SDF framework since it subsumes both the CPD and LMLRA. For instance, the LMLRA can be seen as a single term BTD. However, as we will see there are computational advantages to be found in integrating the CPD model separately. One of the SDF framework's strengths is that it is decomposition agnostic. As such, support for other models such as the tensor train decomposition [161] can also be incorporated without having to change the underlying algorithms.

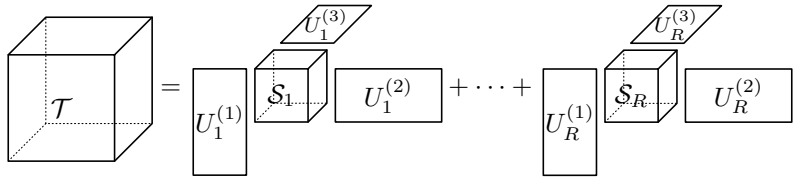


Figure 6.4: Block term decomposition of a third-order tensor.

6.3 Algorithms for SDF

The two most prevalent classes of algorithms to tackle (6.1) are quasi-Newton (QN) methods and nonlinear least squares (NLS) methods. Both iteratively improve an initial solution with additive updates obtained by minimizing a

second-order approximation of the objective function based solely on first-order derivatives. In the following, let $\mathcal{F}^{(d)}$ and $f^{(d)}$ be the dth residual tensor $\mathcal{M}^{(d)}(\mathcal{X}(\boldsymbol{z})) - \mathcal{T}^{(d)}$ and objective function $1/2\,\|\mathcal{F}^{(d)}\|^2_{\mathcal{W}^{(d)}}$, respectively. The SDF objective function is then $f := \sum_d \omega_d f^{(d)}$. Additionally, we identify tensors and ordered sets of variables and factors with their vectorized versions where convenient, especially in or around derivatives. For instance, the Jacobian $\partial\mathcal{F}^{(d)}/\partial\mathcal{X}^{\mathrm{T}}$ should be read as $\partial \operatorname{vec}(\mathcal{F}^{(d)})/\partial \operatorname{vec}(\mathcal{X})^{\mathrm{T}}$, where $\operatorname{vec}(\mathcal{X})$ is the vector $\begin{bmatrix}\operatorname{vec}(x_1)^{\mathrm{T}} & \cdots & \operatorname{vec}(x_F)^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$.

Given an objective function f and a current iterate $\boldsymbol{z}_k$, the key idea of QN and NLS is to minimize a second-order Taylor series approximation of f around $\boldsymbol{z}_k$. Like matrix factorizations, tensor decompositions may also involve complex-valued factors. Under these circumstances, the objective function f is necessarily nonanalytic in $\boldsymbol{z}$ due to the Frobenius norm. In other words, f depends on the complex conjugate $\overline{\boldsymbol{z}}$ and because of this the Taylor series of f around $\boldsymbol{z}$ does not exist in general. The de facto solution is to consider the Taylor series of f around the real and imaginary parts of the variables $\boldsymbol{z}$. However, this easily obfuscates any structure present in the objective function and its derivatives. In Chapter 2, a complex optimization framework has been introduced with which QN and NLS algorithms were generalized to both real and complex variables. In complex optimization, the real Taylor series is replaced by a complex Taylor series of f around $\boldsymbol{z}_k$ based on complex derivatives. More specifically, $f(\boldsymbol{z}_k + \boldsymbol{p})$ is approximated by the second-order complex Taylor series expansion

$$m_k^f(\boldsymbol{p}) := f(\boldsymbol{z}_k) + \overset{c}{\boldsymbol{p}}{}^{\mathrm{T}} \cdot \frac{\partial f}{\partial \overset{c}{\boldsymbol{z}}}(\boldsymbol{z}_k) + \frac{1}{2}\overset{c}{\boldsymbol{p}}{}^{\mathrm{H}} \cdot B_k \cdot \overset{c}{\boldsymbol{p}}, \tag{6.7}$$

wherein $\boldsymbol{z}$ and $\boldsymbol{p}$ are implicitly vectorized and $\overset{c}{\cdot}$ denotes the vertical concatenation of its argument and its complex conjugate, e.g., $\overset{c}{\boldsymbol{z}}$ is equivalent to $\begin{bmatrix}\boldsymbol{z}^{\mathrm{T}} & \overline{\boldsymbol{z}}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$. Analogously, the complex gradient $\partial f/\partial \overset{c}{\boldsymbol{z}}$ concatenates the so-called cogradient $\partial f/\partial \boldsymbol{z}$ and conjugate cogradient $\partial f/\partial \overline{\boldsymbol{z}}$. By definition, these derivatives are to be interpreted as partial derivatives with respect to elements of $\boldsymbol{z}$ ($\overline{\boldsymbol{z}}$), while treating $\overline{\boldsymbol{z}}$ ($\boldsymbol{z}$) as constant. They are also known, especially in the German literature, as Wirtinger derivatives [182]. For real-valued functions f, the cogradients are each other's complex conjugates. Consequently, only one of the two cogradients need be computed. In practice, most algorithms ask for the so-called scaled conjugate cogradient $2\partial f/\partial \overline{\boldsymbol{z}}$ for two reasons. First, the negative conjugate cogradient is the objective function's steepest descent direction. Second, the factor two ensures the scaled conjugate cogradient coincides with the real gradient if all variables are real-valued.

By construction, the matrix B_k is a positive semidefinite approximation of the complex Hessian $\partial^2 f/(\partial \overset{c}{\boldsymbol{z}} \partial \overset{c}{\boldsymbol{z}}{}^{\mathrm{T}})$. A good candidate step to take is the approximate Newton step $\boldsymbol{p}_{\mathrm{AN}} := \arg\min_{\boldsymbol{p}} m_k^f(\boldsymbol{p})$, which can be obtained by

setting the model's cogradients equal to zero and solving the resulting linear system

$$B_k \cdot \overset{c}{\boldsymbol{p}}_{\mathrm{AN}} = -\frac{\partial f}{\partial \overset{c}{\boldsymbol{z}}}(\boldsymbol{z}_k). \tag{6.8}$$

Notice that B_k is four times as large compared to the Hessian of the same objective function with real-valued variables. At first sight, it may seem that solving (6.8) requires significantly more effort than the equivalent task in real optimization, which is compounded by the fact that complex floating point operations are on average four times as expensive as real floating point operations. Moreover, depending on the total number of elements in $\boldsymbol{z}$, even storing B_k might become prohibitive. Fortunately, as we will see QN and NLS methods in complex optimization each have their own way of mitigating these obstacles. Assuming we have (approximately) solved (6.8), the next iterate can then be computed as $\boldsymbol{z}_{k+1} \leftarrow \boldsymbol{z}_k + \boldsymbol{p}_*$, where $\boldsymbol{p}_*$ is usually a linear combination of the steepest descent direction $\boldsymbol{p}_{\mathrm{SD}} := -2\partial f/\partial \overline{\boldsymbol{z}}$ at $\boldsymbol{z}_k$ and the approximate Newton step $\boldsymbol{p}_{\mathrm{AN}}$. The former is used to guarantee that the objective function value will improve and the latter enables super-linear or even quadratic convergence near the solution. Algorithms that compute $\boldsymbol{p}_*$ given the steepest descent direction and approximate Newton step are called globalization strategies and can be categorized into line search and trust-region approaches. Their name is derived from the fact that they ensure global convergence—i.e., the optimization algorithm will converge to a local minimizer given any initialization.

6.3.1 Quasi-Newton

Limited memory BFGS (L-BFGS) and, by extension, nonlinear conjugate gradient (NCG) are examples of QN methods [159]. Instead of storing B_k, L-BFGS represents its inverse B_k^{-1} as the sum of a scaled identity matrix plus m rank-two updates. The inverse Hessian approximation at the next iterate B_{k+1}^{-1} is computed as a rank-two correction of B_k^{-1}. This BFGS update is designed so that B_{k+1}^{-1} is, among all matrices that satisfy the so-called secant equation, in some sense as close as possible to B_k^{-1}. The secant equation expresses the desire that the gradient of the model m_{k+1}^f at the previous iterate $\boldsymbol{z}_k$ should match the objective function's gradient at $\boldsymbol{z}_k$, i.e.,

$$B_{k+1} \cdot \left(\overset{c}{\boldsymbol{z}}_{k+1} - \overset{c}{\boldsymbol{z}}_k\right) = \frac{\partial f}{\partial \overset{c}{\boldsymbol{z}}}(\boldsymbol{z}_{k+1}) - \frac{\partial f}{\partial \overset{c}{\boldsymbol{z}}}(\boldsymbol{z}_k). \tag{6.9}$$

NCG can be seen as a special case of L-BFGS where $m = 1$ together with a few other assumptions. In practice, it has been observed that L-BFGS often

performs better than NCG, which is likely due to its greater flexibility in representing second-order information.

Not only does a diagonal plus low-rank matrix representation of B_k^{-1} save memory, it also allows for a very efficient solution to (6.8). In fact, the L-BFGS two-loop recursion algorithm generalized to complex optimization [206] can compute $\boldsymbol{p}_{\mathrm{AN}}$ directly with the same amount of (albeit complex) floating point operations as the equivalent algorithm in real optimization.

In summary, QN methods require only two pieces of information: a way to evaluate the objective function f and its scaled conjugate cogradient $2\partial f/\partial \overline{\boldsymbol{z}}$.

6.3.2 Nonlinear least squares

The SDF problem given by (6.1) consists of a weighted sum of NLS objective functions $f^{(d)}$. NLS methods build a second-order model of the form (6.7) for each such $f^{(d)}$ by approximating the corresponding residual tensors $\mathcal{F}^{(d)}$ around $\boldsymbol{z}_k$ with the linear model

$$m_k^{\mathcal{F}^{(d)}}(\boldsymbol{p}) := \mathcal{F}^{(d)}(\boldsymbol{z}_k) + J_k^{(d)} \cdot \overset{\mathcal{C}}{\boldsymbol{p}}, \tag{6.10}$$

wherein $J_k^{(d)} := \partial \mathcal{F}^{(d)}/\partial \overset{\mathcal{C}}{\boldsymbol{z}}^{\mathrm{T}}$ is the residual tensor's complex Jacobian at $\boldsymbol{z}_k$. Substituting this model for $\mathcal{F}^{(d)}$ in the SDF objective function results in a complex second-order Taylor series expansion of f with

$$B_k = \sum_{d=1}^{D} \omega_d \left(\mathrm{vec}(\mathcal{W}^{(d)}) * J_k^{(d)}\right)^{\mathrm{H}} \cdot \left(\mathrm{vec}(\mathcal{W}^{(d)}) * J_k^{(d)}\right), \tag{6.11}$$

where the Hadamard product implicitly expands singleton dimensions of its operands. In other words, dimensions of length one of an operand that do not match the size of the corresponding dimensions in the other operand are repeated until their sizes match. Notice that the complex Jacobian $J_k^{(d)}$ does not depend on the data in $\mathcal{T}^{(d)}$. However, the approximate Hessian B_k does depend on the locations of the known entries of all data sets in the problem.

Depending on the model $\mathcal{M}^{(d)}$, a relatively sparse correction matrix can be added to the Gauss–Newton approximation (6.11) to obtain the objective function's complex Hessian [208]. Coupled with the fact that such corrections are scaled by residuals in $\mathcal{F}^{(d)}$, it may be expected that the Gauss–Newton approximation rapidly approaches the complex Hessian as the residuals decrease in magnitude. Consequently, NLS methods can converge close to quadratically near solutions with small residuals.

For many model parameterizations, the Gauss–Newton approximation of the Hessian is rank deficient due to indeterminacies such as the scaling ambiguity in the CPD. To solve (6.8) under these circumstances, one approach is to compute the Moore–Penrose pseudo-inverse of B_k. Another is to regularize the solution by using the Levenberg–Marquardt approximation of the Hessian, which adds a scaled identity matrix to the Gauss–Newton approximation.

However, tensor decompositions' Jacobians are also often very structured and we wish to exploit this structure to save memory and reduce the computational complexity of computing $\boldsymbol{p}_{\text{AN}}$. Inexact NLS methods only approximately solve (6.8) with the preconditioned conjugate gradient (PCG) algorithm. Since PCG only requires a function to evaluate $B_k \cdot \boldsymbol{y}$ for different $\boldsymbol{y}$, we can store B_k in a data-sparse way and easily exploit its rank structure in the matrix-vector product. Moreover, singular B_k no longer pose a problem because the solution can be regularized by limiting the number of PCG iterations. The convergence rate of PCG depends on how well the system's eigenvalues are clustered and the magnitude of its condition number. The eigenvalue distribution can be improved by applying a so-called preconditioner, which can be thought of as a cheap and approximate inverse of B_k. Designing preconditioners is a difficult task which is usually done one a case-by-case basis. We will focus on preconditioners for a single unstructured CPD or BTD, and leave the design of more advanced preconditioners as a topic for future work.

In summary, inexact NLS methods require four pieces of information: a way to evaluate the objective function f, its scaled conjugate cogradient $2\partial f/\partial \overline{\boldsymbol{z}}$, the approximate Hessian-vector product $B_k \cdot \boldsymbol{y}$ and, optionally, a preconditioner.

6.3.3 Modularizing decompositions and structure

Now that we have settled on which components need to be implemented, we will further subdivide these components into parts so as to isolate the decomposition models from any structure imposed on the factors. Modularizing decompositions and factor structure has two major advantages. First, it enables building libraries of both models and factor structures to be mixed and matched as desired. Second, separating the model and factor structure from each other ensures both remain transparent and independently exploitable.

Objective function By definition, $f(\boldsymbol{z}_k)$ is equal to the weighted sum $\sum_d \omega_d f^{(d)}(\boldsymbol{z}_k)$. To compute the latter, simply evaluate $\mathcal{X}(\boldsymbol{z}_k)$ once and then use the resulting factors to evaluate the models $\mathcal{M}^{(d)}$ at the known entries of the corresponding data sets $\mathcal{T}^{(d)}$.

Gradient Analogously to the objective function, we may compute the scaled conjugate cogradient $2\partial f/\partial \overline{\boldsymbol{z}}$ as $\sum_d \omega_d (2\,\partial f^{(d)}/\partial \overline{\boldsymbol{z}})$. To decouple the factor structure from the model, we apply the chain rule

$$\frac{\partial f^{(d)}}{\partial \overline{\boldsymbol{z}}} = \left(\frac{\partial \mathcal{X}}{\partial \boldsymbol{z}^{\mathrm{T}}}\right)^{\mathrm{H}} \cdot \frac{\partial f^{(d)}}{\partial \overline{\mathcal{X}}} + \left(\frac{\partial \mathcal{X}}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}\right)^{\mathrm{T}} \cdot \frac{\partial f^{(d)}}{\partial \mathcal{X}} \tag{6.12}$$

to separate the conjugate cogradient $\partial f^{(d)}/\partial \overline{\boldsymbol{z}}$ into partial derivative of the objective function $f^{(d)}$ with respect to the factors $\mathcal{X}$ and a partial derivative of the factors $\mathcal{X}$ with respect to the variables $\boldsymbol{z}$. As mentioned at the beginning of this section, we transparently interchange tensors and ordered sets with their vectorized equivalents in and around derivatives. Since $f^{(d)}$ is real-valued, its cogradient is the complex conjugate of its conjugate cogradient so that we can implement (6.12) by first computing an ordered set of factors $\mathcal{H}$ as $\partial f^{(d)}/\partial \overline{\mathcal{X}}$ and then applying the map

$$h_f \mapsto \left(\frac{\partial F}{\partial z_{i_f}^{\mathrm{T}}}\right)^{\mathrm{H}} \cdot h_f + \overline{\left(\frac{\partial F}{\partial \overline{z_{i_f}}^{\mathrm{T}}}\right)^{\mathrm{H}} \cdot h_f} \tag{6.13}$$

to each of its constituents h_f. Therefore, imposing a factor structure defined by the transformation x_f not only requires the ability to evaluate the expansion $x_f(z_{i_f})$, but also its associated linear contraction given by (6.13). For instance, a factorization may depend on a factor which is the matrix inverse of another factor. In that case, $x_f(z_{i_f}) := z_{i_f}^{-1}$ for certain indices f and i_f and it can be shown that the associated linear contraction is given by

$$h_f \mapsto -z_{i_f}^{-\mathrm{H}} \cdot h_f \cdot z_{i_f}^{-\mathrm{H}}. \tag{6.14}$$

When the model $\mathcal{M}^{(d)}$ is $\mathcal{M}_{\mathrm{CPD}}$ or $\mathcal{M}_{\mathrm{BTD}}$, the partial derivative of the objective function $f^{(d)}$ with respect to the factors $\mathcal{X}$ depends on the string of Khatri–Rao and Kronecker products

$$\mathrm{kr}_\sigma(\alpha_m) := \bigodot_{\substack{m=0 \\ N-m \notin \sigma}}^{N-1} \overline{\alpha_{N-m}} \quad \text{and} \tag{6.15a}$$

$$\mathrm{kron}_\sigma(\alpha_m) := \bigotimes_{\substack{m=0 \\ N-m \notin \sigma}}^{N-1} \overline{\alpha_{N-m}}, \tag{6.15b}$$

respectively, in which N is implicitly defined as the length of the sequence α_m. The Khatri–Rao product $A \odot B$ of two matrices A and B with an equal number of columns R is defined as the column-wise Kronecker product

$\begin{bmatrix}\boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_R \otimes \boldsymbol{b}_R\end{bmatrix}$. The functions kr_σ and kron_σ compute a string of Khatri–Rao or Kronecker products of the sequence α_m from back to front, with the exception of the indices specified by the set σ.

Assuming the dth data set $\mathcal{T}^{(d)}$ is modeled by $\mathcal{M}_{\mathrm{CPD}}$ defined by the factor matrices x_{f_n}, $n = 1, \ldots, N_d$, the partial derivative with respect to the complex conjugate of the nth factor matrix is given by

$$2\frac{\partial f^{(d)}}{\partial \overline{x_{f_n}}} = (W^{(d)}_{(n)} * F^{(d)}_{(n)}) \cdot \mathrm{kr}_{\{n\}}(x_{f_m}). \tag{6.16}$$

Similarly, if the dth data set $\mathcal{T}^{(d)}$ is modeled by $\mathcal{M}_{\mathrm{BTD}}$ defined by the factor matrices $x_{f_{(r,n)}}$ and core tensors $x_{f_{(r,S)}}$, $r = 1, \ldots, R_d$ and $n = 1, \ldots, N_d$, the partial derivative with respect to the complex conjugate of the nth factor matrix and core tensor of the rth term are given by

$$2\frac{\partial f^{(d)}}{\partial \overline{x_{f_{(r,n)}}}} = (W^{(d)}_{(n)} * F^{(d)}_{(n)}) \cdot \mathrm{kron}_{\{n\}}(x_{f_{(r,m)}}) \cdot (x_{f_{(r,S)}})^{\mathrm{H}}_{(n)} \quad \text{and} \tag{6.17a}$$

$$2\frac{\partial f^{(d)}}{\partial \overline{x_{f_{(r,S)}}}} = \mathrm{vec}(\mathcal{W}^{(d)} * \mathcal{F}^{(d)})^{\mathrm{T}} \cdot \mathrm{kron}_{\varnothing}(x_{f_{(r,m)}}), \tag{6.17b}$$

respectively. In both (6.16) and (6.17), the Hadamard product of the observation tensor $\mathcal{W}^{(d)}$ and residual tensor $\mathcal{F}^{(d)}$ ensures that only the residuals corresponding to known entries of the data set contribute to the gradient.

The matrices $\mathrm{kr}_{\{n\}}(x_{f_m})$ and $\mathrm{kron}_{\varnothing}(x_{f_{(r,m)}})$ are highly structured and often appear as the right operand in matricized tensor-matrix products, such as in (6.16) and (6.17). Concretely, they represent the matrices $\overline{x_{f_{N_d}}} \odot \cdots \odot \overline{x_{f_{n+1}}} \odot \overline{x_{f_{n-1}}} \odot \cdots \odot \overline{x_{f_1}}$ and $\overline{x_{f_{(r,N_d)}}} \otimes \cdots \otimes \overline{x_{f_{(r,1)}}}$, respectively. It is desirable to avoid explicitly forming these matrices where possible. Efficient algorithms computing the matricized-tensor times string of Khatri–Rao products are available [170,235], and can be generalized to strings of Kronecker products with little effort. Although these algorithms obviate the need to permute the tensor's elements in memory, they are ill-suited for big data because of their sizable intermediate results. For large data sets, a better approach may be to update the result while linearly scanning the data and computing the Khatri–Rao or Kronecker product's entries on the fly.

We now have the components necessary to solve SDF problems using gradient-based algorithms. Each model $\mathcal{M}^{(d)}$ should define how it can be evaluated given a set of factors and how to compute the gradient of its corresponding objective function $f^{(d)}$ with respect to those factors. Additionally, each transformation x_f should define how to expand the variable z_{i_f} into the factor $x_f(z_{i_f})$ and how to

apply the associated linear contraction (6.13). The final two components—the approximate Hessian-vector product $B_k \cdot \boldsymbol{y}$ and an optional preconditioner—are all that remain to enable the NLS family of algorithms as well.

Approximate Hessian-vector product Although the approximate Hessian B_k in (6.11) does not depend on the tensor's known entries, it does depend on their locations. In the case of big data, even reading the observation tensor $\mathcal{W}^{(d)}$ could take an unreasonable amount of time. Moreover, the presence of the observation tensor destroys the rank structure which would otherwise be present in the approximate Hessian. For these reasons, we redefine the approximate Hessian as

$$B_k := \sum_{d=1}^{D} \omega_d \, \dot{\boldsymbol{J}}_k^{(d)\mathrm{H}} \dot{\boldsymbol{J}}_k^{(d)}, \tag{6.18}$$

where the Jacobian $\dot{\boldsymbol{J}}_k^{(d)}$ is defined as $\partial \mathcal{F}^{(d)} / \partial \boldsymbol{z}^{\mathrm{T}}$ at $\boldsymbol{z}_k$. For the sake of simplicity, we have assumed that $\mathcal{F}^{(d)}$ is analytic in $\boldsymbol{z}$ so that (6.8) reduces to

$$B_k \cdot \boldsymbol{p}_{\mathrm{AN}} = -2 \frac{\partial f}{\partial \overline{\boldsymbol{z}}}(\boldsymbol{z}_k). \tag{6.19}$$

Because the observation tensor $\mathcal{W}^{(d)}$ was omitted, the approximate Hessian's (Frobenius) norm is now likely to be larger than that of the approximate Hessian in (6.11). As a result, the approximate Newton step $\boldsymbol{p}_{\mathrm{AN}}$ which solves (6.19) may be much smaller than intended. A reasonably effective yet simple solution is to instead solve the system

$$B_k \cdot \boldsymbol{p}_{\mathrm{AN}} = -\sum_{d=1}^{D} \sigma_d \omega_d \cdot 2 \frac{\partial f^{(d)}}{\partial \overline{\boldsymbol{z}}}(\boldsymbol{z}_k), \tag{6.20}$$

where σ_d is defined as the product of dth tensor's dimensions, divided by its number of known elements. The function of the scalars σ_d is to make up for the missing residuals corresponding to the unknown entries in the data sets. If each tensor's known entries are distributed more or less randomly across the tensor and if the residuals $\mathcal{F}^{(d)}$ are close to equal in magnitude, it may be expected that the right-hand side of (6.20) approaches the scaled conjugate cogradient of the same SDF problem wherein all tensors' entries are known.

Again, we use the chain rule to decouple the factor structure defined by the transformations x_f from the models $\mathcal{M}^{(d)}$. By the assumption that $\mathcal{F}^{(d)}$ is analytic in $\boldsymbol{z}$ we obtain the decomposition

$$B_k = \left(\sum_{d=1}^{D} \omega_d \left(\frac{\partial \mathcal{X}}{\partial \boldsymbol{z}^{\mathrm{T}}} \right)^{\mathrm{H}} \cdot \ddot{\boldsymbol{J}}_k^{(d)\mathrm{H}} \ddot{\boldsymbol{J}}_k^{(d)} \right) \cdot \frac{\partial \mathcal{X}}{\partial \boldsymbol{z}^{\mathrm{T}}}, \tag{6.21}$$

where the so-called (Jacobian's) Gramian $\ddot{J}_k^{(d)\mathrm{H}} \ddot{J}_k^{(d)}$ is defined as $(\partial \mathcal{F}^{(d)}/\partial \mathcal{X}^{\mathrm{T}})^{\mathrm{H}} \cdot \partial \mathcal{F}^{(d)}/\partial \mathcal{X}^{\mathrm{T}}$ at $\boldsymbol{z}_k$. Notice that we discarded the matrix multiplication symbol in the Gramian, emphasizing that it should be thought of as a structured matrix that can be multiplied by a vector in one action instead of two. The approximate Hessian-vector product $B_k \cdot \boldsymbol{y}$ can now be computed in three stages. First, for each transformation x_f expand the corresponding variable y_{i_f} in the ordered set of variables $\boldsymbol{y}$ into a factor g_f with the map

$$y_{i_f} \mapsto \frac{\partial F}{\partial z_{i_f}^{\mathrm{T}}} \cdot y_{i_f}. \tag{6.22}$$

Then, multiply each residual function's Gramian by the resulting ordered set of factors $\mathcal{G}$. Since the Gramian is square, the result can again be interpreted as an ordered set of factors $\mathcal{H}^{(d)}$. Finally, contract $\mathcal{H}^{(d)}$ into variables by reusing the previously defined map (6.13), multiply by ω_d and sum up the contributions over all data sets. Returning back to our example transformation $x_f(z_{i_f}) := z_{i_f}^{-1}$ which computes the fth factor as the matrix inverse of another factor, the linear expansion (6.22) can be implemented as

$$y_f \mapsto -z_{i_f}^{-1} \cdot y_f \cdot z_{i_f}^{-1}. \tag{6.23}$$

Depending on the model, the Gramian can exhibit a certain rank structure that we would like to exploit in its matrix-vector product. To this end, we introduce the string of Hadamard products

$$\mathrm{hdm}_\sigma(\alpha_m) := \overset{\substack{N-1 \\ m=0 \\ N-m \notin \sigma}}{\ast} \overline{\alpha_{N-m}}. \tag{6.24}$$

There is an interesting relationship between the Khatri–Rao and Hadamard product given by

$$\mathrm{kr}_\sigma(\alpha_m)^{\mathrm{H}} \cdot \mathrm{kr}_\sigma(\alpha_m) \equiv \mathrm{hdm}_\sigma(\alpha_m^{\mathrm{H}} \cdot \alpha_m), \tag{6.25}$$

which is a straightforward consequence of the fact that the columns of a string of Khatri–Rao products are vectorized rank-one tensors. Furthermore, we introduce the modified string of Hadamard and Kronecker products

$$\mathrm{hdm}_{\sigma,\rho}(\alpha_m, \beta_m) := \overset{\substack{N-1 \\ m=0 \\ N-m \notin \sigma}}{\ast} \begin{cases} \overline{\alpha_{N-m}} & N-m \notin \rho \\ \overline{\beta_{N-m}} & \text{otherwise} \end{cases} \tag{6.26a}$$

$$\mathrm{kron}_{\sigma,\rho}(\alpha_m, \beta_m) := \overset{\substack{N-1 \\ m=0 \\ N-m \notin \sigma}}{\bigotimes} \begin{cases} \overline{\alpha_{N-m}} & N-m \notin \rho \\ \overline{\beta_{N-m}} & \text{otherwise} \end{cases}, \tag{6.26b}$$

respectively. These functions generalize hdm_σ and kron_σ so that operands in the string corresponding to indices in the set ρ can be replaced by those of a second sequence β_m.

Assuming the dth data set $\mathcal{T}^{(d)}$ is modeled by $\mathcal{M}_{\mathrm{CPD}}$ defined by the factor matrices x_{f_n}, $n = 1, \ldots, N_d$, the factor matrix with index f_n of the Gramian-vector product is given by [208]

$$\begin{aligned}(\ddot{J}_k^{(d)\mathrm{H}} \ddot{J}_k^{(d)} \cdot \mathcal{G})_{f_n} &= g_{f_n} \cdot \mathrm{hdm}_{\{n\}}(x_{f_m}^{\mathrm{H}} \cdot x_{f_m}) \\ &\quad + x_{f_n} \cdot \sum_{\substack{\tilde{n}=1 \\ \tilde{n}\neq n}}^{N_d} \mathrm{hdm}_{\{n\},\{\tilde{n}\}}(x_{f_m}^{\mathrm{H}} \cdot x_{f_m}, g_{f_m}^{\mathrm{H}} \cdot x_{f_m}).\end{aligned} \tag{6.27}$$

Similarly, if the dth data set $\mathcal{T}^{(d)}$ is modeled by $\mathcal{M}_{\mathrm{BTD}}$ defined by the factor matrices $x_{f_{(r,n)}}$ and core tensors $x_{f_{(r,S)}}$, $r = 1, \ldots, R_d$ and $n = 1, \ldots, N_d$, it can be shown that the factor matrix with index $f_{(r,n)}$ and core tensor with index $f_{(r,S)}$ of the Gramian-vector product are given by

$$\begin{aligned}(\ddot{J}_k^{(d)\mathrm{H}} \ddot{J}_k^{(d)} \cdot \mathcal{G})_{f_{(r,n)}} &= \sum_{\tilde{r}=1}^{R_d} \Bigg(x_{f_{(\tilde{r},n)}} \cdot \Bigg((g_{f_{(\tilde{r},S)}})_{(n)} \cdot \mathrm{kron}_{\{n\}}(x_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}) \\ &\quad + (x_{f_{(\tilde{r},S)}})_{(n)} \cdot \sum_{\substack{\tilde{n}=1 \\ \tilde{n}\neq n}}^{N_d} \mathrm{kron}_{\{n\},\{\tilde{n}\}}(x_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}, g_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}) \Bigg) \\ &\quad + g_{f_{(\tilde{r},n)}} \cdot (x_{f_{(\tilde{r},S)}})_{(n)} \cdot \mathrm{kron}_{\{n\}}(x_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}) \Bigg) \cdot (x_{f_{(r,S)}})_{(n)}^{\mathrm{H}} \quad \text{and}\end{aligned} \tag{6.28a}$$

$$\begin{aligned}(\ddot{J}_k^{(d)\mathrm{H}} \ddot{J}_k^{(d)} \cdot \mathcal{G})_{f_{(r,S)}} &= \sum_{\tilde{r}=1}^{R_d} \Bigg(\mathrm{vec}(g_{f_{(\tilde{r},S)}})^{\mathrm{T}} \cdot \mathrm{kron}_{\varnothing}(x_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}) \\ &\quad + \mathrm{vec}(x_{f_{(\tilde{r},S)}})^{\mathrm{T}} \cdot \sum_{\tilde{n}=1}^{N_d} \mathrm{kron}_{\varnothing,\{\tilde{n}\}}(x_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}, g_{f_{(\tilde{r},m)}}^{\mathrm{H}} \cdot x_{f_{(r,m)}}) \Bigg),\end{aligned} \tag{6.28b}$$

respectively. The Gramian-vector products of (6.27) and (6.28) are designed to fully exploit the Gramian's rank structure. For example, in the case of a CPD the Gramian consists exclusively of diagonal and rank-one blocks [208]. Instead of computing and storing these blocks explicitly, (6.27) directly applies their action on the factors g_f. In much the same way, (6.28) takes advantage of the rank structure of the block term decomposition's Gramian.

Since the approximate Hessian-vector product will be applied several times for different $\boldsymbol{y}$ in each outer iteration of the optimization algorithm, it is worthwhile to store constants such as the sequence $x_{f_m}^{\mathrm{H}} \cdot x_{f_m}$ beforehand. Likewise, the linear expansion of (6.22) and linear contraction of (6.13) often depend on intermediate results that remain constant during the execution of the PCG algorithm. For example, the linear expansion and contraction corresponding to the transformation $x_f(z_{i_f}) := z_{i_f}^{-1}$ depend on the inverse of z_{i_f}, which does not change until the next update of the approximate Hessian.

Preconditioner The convergence rate of computing the approximate Newton step with PCG can be improved by modifying the system (6.20) with a preconditioner M_k which resembles B_k, yet is cheap to invert. Designing preconditioners for SDF problems is complicated by the fact that B_k is a weighted sum of Gramians transformed by linear expansion and contraction operators. In contrast to the previous components such as the gradient and approximate Hessian-vector product, there is no straightforward way of building a preconditioner for each of the countless combinations of decomposition types, coupling and factor structure. Instead, we restrict ourselves to the approximation of a single data set with a CPD or BTD in which there is no structure or symmetry imposed on the factors. The approach can be generalized to multiple data sets and structured factors, though it will be less obvious how to exploit any structure present in the preconditioner.

In the case of a single data set PCG computes the approximate Newton step from the modified system

$$M_k^{-1} \cdot B_k \cdot \boldsymbol{p}_{\mathrm{AN}} = -\sigma M_k^{-1} \cdot 2\frac{\partial f}{\partial \overline{\boldsymbol{z}}}(\boldsymbol{z}_k), \tag{6.29}$$

where M_k is a symmetric positive definite matrix and σ is the scaling coefficient introduced in (6.20). In truth, a system related to (6.29) is solved by PCG to overcome the fact that $M_k^{-1} \cdot B_k$ is not necessarily symmetric positive definite. However, the end result is the same as the algorithm can be rearranged so that only matrix-vector products of the form $B_k \cdot \boldsymbol{y}$ and $M_k^{-1} \cdot \boldsymbol{y}$ are necessary. Since the Gramian exhibits a block structure for many tensor decompositions, a block-diagonal preconditioner is a natural first choice. These so-called block-Jacobi type preconditioners can be interpreted as a divide-and-conquer approach to inverting the approximate Hessian. Not only are the diagonal blocks much smaller, and hence cheaper to invert, but they may also exhibit some structure which can be exploited in computing their inverse.

When the data set is approximated by a CPD defined by the factor matrices x_{f_n}, $n = 1, \ldots, N$, the factor matrix with index f_n of the block-Jacobi preconditioner-

vector product $M_k^{-1} \cdot \boldsymbol{y}$ can be computed as [208]

$$(M_k^{-1} \cdot \mathcal{Y})_{f_n} = y_{f_n} \cdot \mathrm{hdm}^{-1}_{\{n\}}(x^{\mathrm{H}}_{f_m} \cdot x_{f_m}). \tag{6.30}$$

Likewise, when the data set is approximated by a BTD defined by the factor matrices $x_{f_{(r,n)}}$ and core tensors $x_{f_{(r,S)}}$, $r = 1, \ldots, R_d$ and $n = 1, \ldots, N_d$, the factor matrix with index $f_{(r,n)}$ and core tensor with index $f_{(r,S)}$ of the block-Jacobi preconditioner-vector product are given by

$$(M_k^{-1} \cdot \mathcal{Y})_{f_{(r,n)}} = y_{f_{(r,n)}} \cdot \left((x_{f_{(r,S)}})_{(n)} \cdot \mathrm{kron}_{\{n\}}(x^{\mathrm{H}}_{f_{(r,m)}} \cdot x_{f_{(r,m)}}) \cdot (x_{f_{(r,S)}})^{\mathrm{H}}_{(n)}\right)^{-1} \tag{6.31a}$$

$$\text{and} \quad (M_k^{-1} \cdot \mathcal{Y})_{f_{(r,S)}} = \mathrm{vec}(y_{f_{(r,S)}})^{\mathrm{T}} \cdot \mathrm{kron}_{\varnothing}((x^{\mathrm{H}}_{f_{(r,m)}} \cdot x_{f_{(r,m)}})^{-1}), \tag{6.31b}$$

respectively.

6.3.4 Generalizations

The SDF framework as presented can be improved even further with a number of generalizations. One such improvement would be to allow chaining of transformations, i.e., allow factors of the form $x_g(x_f(z_{i_f}))$. This would enable combining factor structures in novel ways. For example, a nonnegative Toeplitz structure could be imposed by using x_f to square the elements of a generator vector z_{i_f} and then applying x_g to generate the Toeplitz matrix. Other convenient improvements could include allowing constant factors, which can be used to incorporate prior knowledge, and nested variables, which generalize the definition of a variable from a tensor to an ordered set of variables, each of which may contain another ordered set of variables or a tensor. The latter is especially useful when a single variable is most naturally represented by a set of vectors, matrices or tensors. For example, a matrix with orthonormal columns can be parametrized by a sequence of Householder reflectors, which could be stored as a set of vectors of increasing length.

With Tensorlab [211], we offer a domain specific language (DSL) for defining SDF problems which includes the above and other improvements such as the ability to compose factors out of subfactors. At the time of writing the supported models include the CPD, LMLRA and BTD and a library of 32 factor structures is provided. With the latter, the user can impose constraints such as nonnegativity, orthonormal columns, columns as sums of kernel functions and banded, Toeplitz, Hankel or Vandermonde structures, among others. As an example of the modularity of the framework, two additional models representing an L1- and L2-regularization term in the objective function are offered. Efficient

complex QN and NLS algorithms are included for solving the resulting SDF problems, as well as tools for estimating model parameters and generating high-quality initializations.

6.4 Numerical experiments

6.4.1 Eigenvalue decomposition as a structured tensor decomposition

We compare the relative accuracy of the eigenvalues of a certain matrix as computed by LAPACK's driver routine DGEEV with those computed by the Tensorlab implementation of SDF. Consider the colleague matrix

$$A = \begin{bmatrix} 0 & 1/2 & & \\ 1 & 0 & 1/2 & \\ & 1/2 & 0 & \ddots \\ & & \ddots & \ddots \end{bmatrix}$$

of order n, corresponding to the Chebyshev polynomial $T_n(x) := \cos(n \arccos(x))$. The eigenvalues of the matrix A are the zeros of $T_n(x)$, which are known to be given in closed form by

$$\lambda_i = \cos\left(\frac{\pi(i + \frac{1}{2})}{n}\right), \qquad i = 0, \ldots, n-1. \tag{6.32}$$

For reference, we evaluate (6.32) in high precision and then truncate the result to double precision. To compute the eigenvalue decomposition of A with SDF, we define two variables z_1 and z_2 as a matrix of order n and a row vector of length n, respectively. Then, we define three transformations $x_1(z_1) := z_1$, $x_2(z_1) := z_1^{-\mathrm{T}}$ and $x_3(z_2) := z_2$. Finally, we model the eigenvalue decomposition of A as a CPD defined by the factor matrices x_1, x_2 and x_3. Here, we interpret A as an $n \times n \times 1$ tensor wherein x_3 is associated with its third dimension. We compute the eigenvalue decomposition of A with LAPACK and use the resulting eigenvectors and eigenvalues to initialize the variables z_1 and z_2. The relative accuracy of the eigenvalues compared to the reference solution is shown in Figure 6.5 for $n = 200$. About 60% of the eigenvalues computed by SDF have a relative error which is at least 10 times smaller than those of LAPACK, and this increases to over 75% for $n = 500$.

The strength of SDF is not so much the efficiency with which the eigenvalue decomposition is computed, but rather the expressive power of the framework

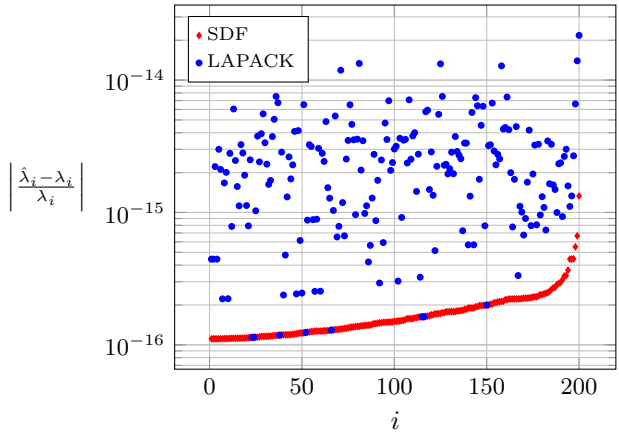


Figure 6.5: Relative accuracy of the eigenvalues of the colleague matrix A, as computed by LAPACK and Tensorlab's implementation of SDF.

and, in this example, the improved accuracy of the resulting eigenvalues. To make this more concrete, Listing 6.1 gives an impression of how an eigenvalue decomposition could be implemented using Tensorlab's DSL. The SDF problem is created as a MATLAB structure that defines the model's variables, factors and factorizations. The function `@struct_invtransp` encodes the transformation x_2 and its associated left and right Jacobian-vector products, wherein it can exploit any structure related to the transformation. Structure at the tensor decomposition level is exploited by their gradient and approximate Hessian-vector products. After solving the model with `sdf_minf` or `sdf_nls`—corresponding to the quasi-Newton and NLS family of algorithms discussed in the previous section—the resulting MATLAB structure `sol` contains the optimized variables and factors in `sol.variables` and `sol.factors`, respectively.

```
% Define and initialize variables.
model.variables.z1 = randn(n,n);
model.variables.z2 = randn(1,n);

% Define factors as transformed variables.
model.factors.x1 = 'z1';
model.factors.x2 = {'z1',@struct_invtransp};
model.factors.x3 = 'z2';

% Define factorizations using the previously defined factors.
model.factorizations.evd.data = A;
model.factorizations.evd.cpd  = {'x1','x2','x3'};
```

```
% Solve the SDF problem.
sol = sdf_nls(model);
```

Listing 6.1: Defining and solving an eigenvalue decomposition with Tensorlab's DSL for SDF.

6.4.2 Recommending movies: the Netflix prize

In 2006, the online DVD rental company Netflix organized a contest to improve its recommender system. The company released a training set of more than 100 million ratings given by about 480,000 customers to almost 18,000 movies over a period of 6 years. The goal of the contest is to predict unseen ratings more accurately than Netflix's proprietary Cinematch algorithm, enabling better personalized movie recommendations. The Netflix data set can be described by an incomplete user $\times$ movie $\times$ date tensor $\mathcal{R}$ containing integer ratings between 1 and 5 stars given by users to movies on certain dates. The data set fits in about 2 GB of memory, which is not considered big data, but still large enough to represent a formidable computational challenge. A standardized validation set of about 2 million ratings, called the probe set, was also supplied by Netflix.

Matrix factorization techniques were a key component in most solutions, though it took some time before participants understood how to effectively incorporate temporal data into their models. With a tensor representation of the data set, the temporal data is naturally taken into account by employing tensor factorizations. Here, we use SDF to extend the core matrix factorization model of the prize winning entry [113] to a tensor factorization. We will model each rating r_{umd} as

$$r_{umd} \approx \mu + b_u^{(1)} + b_m^{(2)} + b_d^{(3)} + \sum_{k=1}^{K} z_{uk}^{(1)} z_{mk}^{(2)} z_{dk}^{(3)}, \tag{6.33}$$

where μ is the mean rating and the vectors $\boldsymbol{b}^{(1)}$, $\boldsymbol{b}^{(2)}$ and $\boldsymbol{b}^{(3)}$ represent a bias due to users, movies and temporal effects, respectively. For example, the user bias models the fact that some users consistently rate movies a certain amount of stars more or less than the average user. Taking a closer look at (6.33), we notice that the bias terms are in fact structured rank-one tensors in which the outer product is taken with all-one vectors. Accordingly, after removing the mean, these terms can be assimilated together with the matrices $Z^{(1)}$, $Z^{(2)}$, $Z^{(3)}$ into factor matrices $x_1 = \begin{bmatrix} \boldsymbol{b}^{(1)} & 1_{U\times 2} & Z^{(1)} \end{bmatrix}$, $x_2 = \begin{bmatrix} 1_{M\times 1} & \boldsymbol{b}^{(2)} & 1_{M\times 1} & Z^{(2)} \end{bmatrix}$ and $x_3 = \begin{bmatrix} 1_{D\times 2} & \boldsymbol{b}^{(3)} & Z^{(3)} \end{bmatrix}$ corresponding to a structured CPD.

Model	RMSE
Mean	1.1296
Cinematch	0.9474
$K = 1$	0.9447
$K = 2$	0.9387
$K = 3$	0.9372
$K = 4$	0.9326
$K = 5$	0.9298
$K = 6$	0.9275

Table 6.1: RMSE on the probe set by the mean μ, Netflix's Cinematch algorithm and model (6.33).

We factorize the tensor $\mathcal{R}$ using SDF to implement the model (6.33) as a structured CPD and add a regularization term to the objective function of the form $\lambda(\|x_1\|_{\mathrm{F}}^2 + \|x_2\|_{\mathrm{F}}^2 + \|x_3\|_{\mathrm{F}}^2)$ to prevent overfitting. To determine the hyperparameter λ, we perform a simple grid search and select the value for which the model best explains the ratings in the validation set. Netflix chose to measure the predictive power of its models with the root mean square error (RMSE), defined as the square root of the mean squared error on the individual ratings. This measure has the tendency to amplify the penalization of false positives ("trust busters") and false negatives ("missed opportunities"). Table 6.1 shows the RMSE on the probe set of a baseline model that predicts each rating as the mean rating μ, Netflix's Cinematch algorithm and model (6.33) for various values of the rank parameter K. Even a relatively simple bias plus rank-one model can improve on Cinematch's performance, while larger values of K provide consistent improvements to the RMSE. The Netflix prize was awarded to the first team to improve on Cinematch's performance by 10%, which corresponds to an RMSE of 0.8527 on the probe set. Table 6.1 shows that a bias plus rank-6 model already accounts for a 2.1% improvement over Cinematch. For comparison, the prize winning entry consisted of a blend of 107 different models, the majority of which are matrix factorization models with ranks in the order of hundreds to thousands [113].

6.4.3 Designing an alloy

By varying the concentrations of an alloy's constituent metals, materials with different physical properties can be designed. For example, we might be interested in the tensile strength of an alloy comprising four metals. Since the concentrations should sum to one, the concentration of the fourth metal

is fixed once the concentrations of the first three are chosen. Assuming we discretize the concentrations of each of the constituent metals between 0% and 99% in steps of 1%, we can arrange the tensile strength of the resulting alloys in a $100 \times 100 \times 100$ tensor $\mathcal{T}$ where the nth dimension corresponds to the concentration of the nth constituent. Since measuring or computing each of the one million combinations is impractical, we would like to sample only a few of the tensor's entries. Based on these samples, the tensile strength of the other combinations can then be inferred.

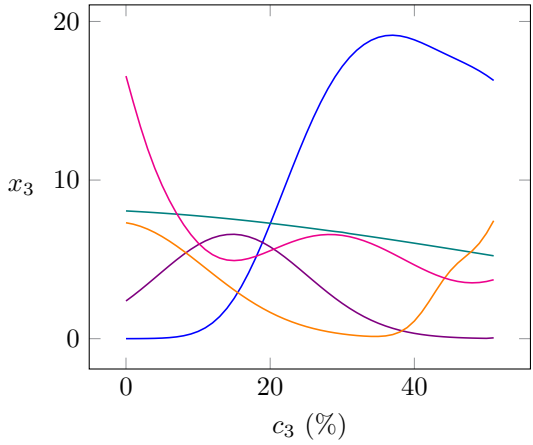
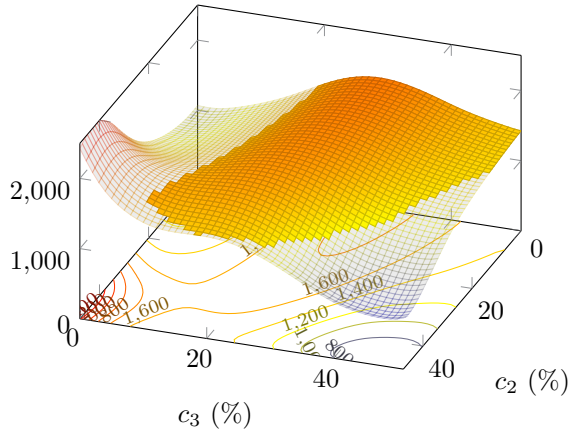


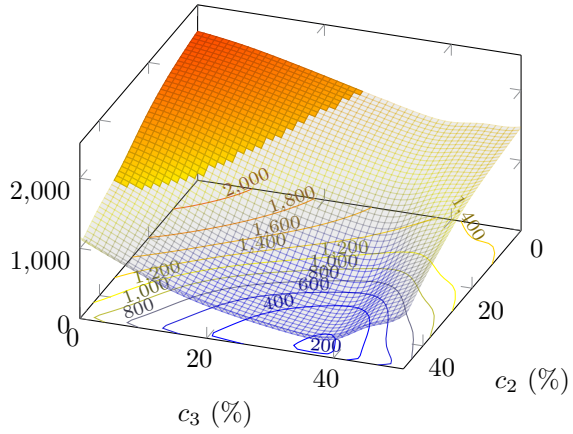
Figure 6.6: The 5 columns of the third factor matrix x_3 in the structured CPD of the InsPyro data set. Each column is a sum of 8 RBF kernels, sampled at different concentrations c_3.

InsPyro NV—a KU Leuven spin-off—kindly provided us with a data set containing more than 35,000 measurements of a certain material property of a four-material mixture. Although we cannot describe the material property or the material's constituents in more detail, we will refer to former as the tensile strength and to the latter as metals for the sake of this example. The metals' concentrations are varied in steps of 1% starting at 11% for the first metal and 1% for the remaining metals, up to a maximum of 95%, 49% and 52% for the three metals, respectively. As such, the data set can be represented as an incomplete $85 \times 49 \times 52$ tensor $\mathcal{T}$ in which 16.31% of the entries are known.

One fact we wish to exploit in modelling this data set is that the tensile strength is expected to vary relatively smoothly as the concentrations are varied. We can take advantage of this piece of information by approximating the tensor with a CPD in which the factor matrices' columns are sampled smoothly varying functions. Let $z^{(n)} \in \mathbb{R}^{3\times D\times R}$, $n = 1, \ldots, 3$, represent the parameters of R



(a) $c_1 = 20\%$



(b) $c_1 = 70\%$

Figure 6.7: The tensile strength of an alloy comprising four metals.

functions, each of which is the sum of D radial basis function (RBF) kernels. The ith element of the rth column of the nth factor matrix $x_n(z^{(n)})$ is then described by

$$(x_n(z^{(n)}))_{ir} := \sum_{d=1}^{D} z_{1dr}^{(n)} \exp\left(-\frac{(c_n(i) - z_{2dr}^{(n)})^2}{2(z_{3dr}^{(n)})^2}\right), \tag{6.34}$$

where $c_n(i)$ maps the index i to the ith concentration of the nth constituent. By modelling the columns of the factor matrices as functions, we have the

additional benefit of being able to accurately interpolate the melting point for non-integer concentrations.

We choose $D = 8$ and $R = 5$ and implement the structured CPD with factor matrices x_1, x_2 and x_3 defined by (6.34) with Tensorlab's DSL for SDF. The model is trained on two thirds of the data, and validated on the remaining one third. The relative error of the optimized model was 0.27% for both training and validation set. Figure 6.6 plots the columns of the third factor matrix x_3 as a function of the concentration c_3. Figure 6.7 shows two slices of the modelled tensor, where the concentration of the first constituent c_1 is kept fixed at 20% and 70%, respectively. The remaining two concentrations c_2 and c_3 are free. Because the sample data is not uniformly distributed over the tensor but rather clustered per slice, we have coloured the slices only partially opaque to indicate where the model is supported by nearby data points. Straying too far from the samples may lead to inaccurate extrapolations of the tensile strength even though the validation error is small.

6.4.4 Predicting user participation in activities

When visiting a city for the first time, it would be useful to be presented with a personalized list of recommended activities or points of interest (POI) near your current location. In [255], a multi-relational data set is presented in which 164 users' global positioning system (GPS) coordinates were tracked over a period of 2.5 years. The data set consists of 12,765 GPS trajectories with a total length of close to 140 megametres. The raw coordinates were clustered into 168 meaningful locations and the user comments attached to the GPS data were manually parsed into activity annotations for these locations. A total of five activity categories were defined: food and drink, shopping, movies and shows, sports and exercise, and tourism and amusement.

A binary user $\times$ location $\times$ activity tensor $\mathcal{T}^{(1)}$ is constructed from the above data in which element (u, l, a) has the value 1 if the user u participated in the activity a at location l, and 0 otherwise. The data set also includes four auxiliary sources of information: a user $\times$ user matrix $\mathcal{T}^{(2)}$ that measures the similarity between users obtained from social network data, a location $\times$ feature matrix $\mathcal{T}^{(3)}$ in which each feature represents a normalized number of POI at a location, an activity $\times$ activity matrix $\mathcal{T}^{(4)}$ that measures the similarity between activities obtained by analysis of internet search engine results, and a binary user $\times$ location matrix $\mathcal{T}^{(5)}$ in which element (u, l) has the value 1 if the user u has visited location l, and 0 otherwise. We preprocess the data by removing the 18 users who have not participated in any activity and normalize the columns

of $\mathcal{T}^{(3)}$ so that they sum to 1. In summary, we have 146 users, 168 locations, 5 activities and 14 location features.

In similar vein to the experiments in [70], we consider two scenarios. In a first scenario, we randomly select 80% of the entries in $\mathcal{T}^{(1)}$ and remove them. In a second scenario, all information of 50 randomly selected users is removed from $\mathcal{T}^{(1)}$. The task is then to predict the missing entries with the remaining information. We model each data set $\mathcal{T}^{(d)}$ with a simple rank-2 CPD $\mathcal{M}^{(d)}$, defined by the user factor $U \in \mathbb{R}^{146\times 2}$, location factor $L \in \mathbb{R}^{168\times 2}$, activity factor $A \in \mathbb{R}^{5\times 2}$ and feature factor $F \in \mathbb{R}^{14\times 2}$, depending on the data set. Furthermore, we introduce the factors $\lambda, \mu, \nu \in \mathbb{R}^{1\times 2}$ to absorb any differences in scale between factorizations. For example, we model the user $\times$ location matrix $\mathcal{T}^{(5)}$ by the CPD $U \cdot \operatorname{diag}(\nu) \cdot L^{\mathrm{T}}$. For the first scenario, two different models are computed by solving the regularized SDF problem

$$\begin{aligned}
\underset{U,L,A,F,\lambda,\mu,\nu}{\text{minimize}} \quad & \frac{\omega_1}{2}\left\|\mathcal{M}^{(1)}(U,L,A)-\mathcal{T}^{(1)}\right\|^2_{\mathcal{W}^{(1)}} \\
+\frac{\omega_2}{2}&\left\|\mathcal{M}^{(2)}(U,U,\lambda)-\mathcal{T}^{(2)}\right\|^2_{\mathrm{F}}+\frac{\omega_3}{2}\left\|\mathcal{M}^{(3)}(L,F)-\mathcal{T}^{(3)}\right\|^2_{\mathrm{F}} \\
+\frac{\omega_4}{2}&\left\|\mathcal{M}^{(4)}(A,A,\mu)-\mathcal{T}^{(4)}\right\|^2_{\mathrm{F}}+\frac{\omega_5}{2}\left\|\mathcal{M}^{(5)}(U,L,\nu)-\mathcal{T}^{(5)}\right\|^2_{\mathrm{F}} \\
+\frac{\omega_6}{2}&\left(\|U\|^2_{\mathrm{F}}+\|L\|^2_{\mathrm{F}}+\|A\|^2_{\mathrm{F}}+\|F\|^2_{\mathrm{F}}+\|\lambda\|^2_{\mathrm{F}}+\|\mu\|^2_{\mathrm{F}}+\|\nu\|^2_{\mathrm{F}}\right)
\end{aligned} \tag{6.35}$$

for two choices of the weights ω_d. In the first case, all weights are nonzero and we exploit all available information. In the second case we choose $\omega_2 = \omega_3 = \omega_4 = \omega_5 = 0$, corresponding to a regularized CPD of $\mathcal{T}^{(1)}$ without data fusion.

Figure 6.8 shows the receiver operating characteristic (ROC) curves of the resulting models. Each ROC curve is obtained by evaluating the corresponding model at the unknown entries of $\mathcal{T}^{(1)}$ and varying a threshold with which we discriminate between the negative class 0 and positive class 1. As the threshold value increases, the number of both false and true positives resulting from model's predictions decreases monotonically. The area under curve (AUC) is a standard measure of a binary classifier's quality. The SDF model, which exploits all available information, has an AUC of 93.78% and clearly outperforms the regularized CPD model, which only exploits information available in $\mathcal{T}^{(1)}$ and has an AUC of 85.59%.

The second scenario corresponds to the difficult case of making predictions for new users, also known as the cold start problem. In this scenario, we must make use of the auxiliary data sets since $\mathcal{T}^{(1)}$ by itself no longer has enough information to fully determine the factor U. To improve predictor robustness,

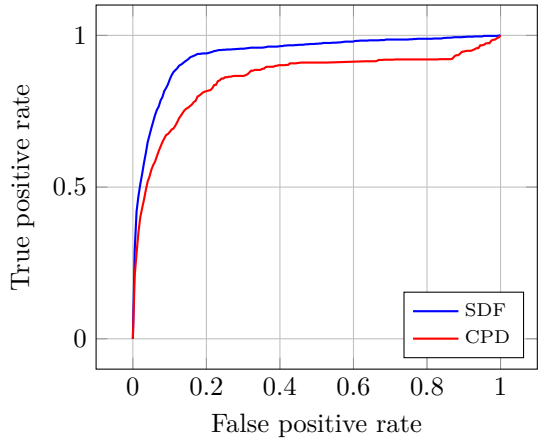


Figure 6.8: ROC curves for the prediction of the 80% missing user-location-activity links in $\mathcal{T}^{(1)}$. The SDF model fuses all five data sources with coupled rank-2 CPDs and has an AUC of 93.78%. The CPD model is a regularized CPD of $\mathcal{T}^{(1)}$ and has an AUC of 85.59%.

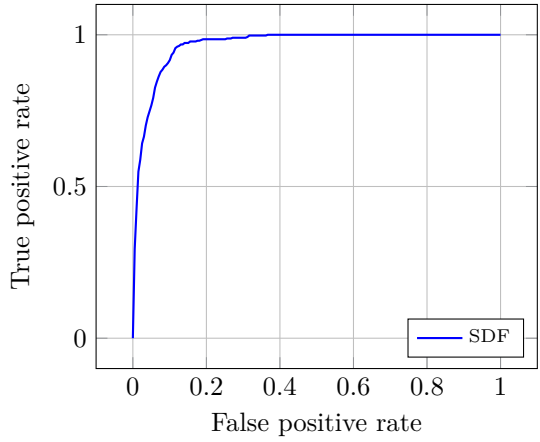


Figure 6.9: ROC curve for the prediction of the user-location-activity links of 50 missing users. The SDF model fuses all five data sources with coupled nonnegative rank-2 CPDs and has an AUC of 96.75%.

we impose the additional constraint that all factors should be nonnegative. This constraint is enforced by modifying the SDF problem (6.35) so that the transformation $x(z) := z * z$ is applied to each factor, effectively squaring all

entries. The ROC curve of the resulting model is shown in Figure 6.9 and has an AUC of 96.75%. Experimenting with different factor structures and tensor decompositions is as simple as indicating which transformations and factors should be used. For example, changing the model $\mathcal{M}^{(1)}$ from a CPD to a LMLRA only requires adding an extra factor in the SDF problem to represent the LMLRA's core tensor. However, in our experiments the LMLRA did not offer a significant advantage in performance compared to the CPD.

6.5 Conclusion

We presented structured data fusion (SDF) as a framework for the rapid prototyping of coupled tensor factorizations with structured factors. The ability to create libraries of both tensor decompositions and factor structures is facilitated by decoupling these components from the underlying optimization algorithm. New decompositions and constraints can be added to these libraries by implementing a small number of functions with which the structure inherent to each can be fully exploited by means of efficient matrix-vector products. The countless combinations of decomposition types, coupling between factorizations and factor structures and ease of switching between them allows users to rapidly iterate towards a solution. As a special case, well-known matrix factorizations such as the singular value decomposition can be computed by selecting the right decomposition type and factor structures. The SDF framework was implemented as a domain specific language (DSL) in Tensorlab [211], a MATLAB toolbox for tensor computations. The versatility of the framework was demonstrated by solving four applications ranging from the computation of eigenvalue decompositions to activity recommendation entirely within the Tensorlab's DSL for SDF.

Chapter 7

Conclusion

7.1 Contributions

In this thesis, we have developed a framework for optimization of complex variables (Chapter 2), structure-exploiting algorithms for tensor decompositions based on complex optimization (Chapter 3), exact line and plane search for accelerating many optimization problems related to tensor factorization (Chapter 4), a state-of-the-art numerical algorithm for solving bivariate and polyanalytic polynomial systems (Chapter 5) and a framework for data fusion by coupled tensor factorizations with structured factors (Chapter 6). Alongside these results, we have realized the development of a fully fledged software package by the name of Tensorlab that offers a concrete implementation of the proposed frameworks and algorithms. In the following, we give a chapter by chapter overview of our contributions:

Chapter 2:

- *A generic framework for complex optimization.* Complex optimization is a true generalization of real optimization in that its derivatives and iterates coincide with those generated by real optimization when applied to a function of real variables. Complex optimization has already been successfully applied in diverse application areas, of which one example is the computation of the Pythagoras number of sum of square magnitudes polynomials [124].

- *Generalized quasi-Newton algorithms for complex optimization.* We developed a complex generalization of the well known limited-memory BFGS algorithm and of several nonlinear conjugate gradient methods.
- *Generalized nonlinear least squares (NLS) algorithms for complex optimization.* Likewise, we generalized NLS algorithms to residual functions of complex variables and showed how the structure of the corresponding derivatives might be exploited by iteratively computing an approximate Newton step.
- *Generalized globalization strategies and numerical complex differentiation.* Moré–Thuente line search and dogleg trust-region globalization strategies are generalized to functions of complex variables. Furthermore, Tensorlab provides numerical complex differentiation schemes so that its complex optimization algorithms do not require user-supplied derivatives.
- *Fast simulation of nonlinear circuits.* We demonstrated how nonlinear circuits can be simulated efficiently in the frequency domain with complex optimization, significantly reducing the computational complexity and memory requirements. The resulting algorithm has been used as the underlying solver for the simulation of power electronics [259].

Chapter 3:

- *A generalized block term decomposition (BTD) that unifies two important tensor decompositions in signal processing.* The canonical polyadic decomposition (CPD) and rank-$(L_r, L_r, 1)$ BTD are some of the most promising techniques for separating measurements into their underlying sources. We introduced a BTD that generalizes these two decompositions, enabling the design of unified algorithms that can compute both of the former decompositions and opening perspectives for future research in signal processing.
- *Complex gradient-based optimization algorithms for tensor decompositions.* We developed an alternating least squares (ALS) algorithm for the generalized BTD, as well as complex quasi-Newton-like algorithms such as limited-memory BFGS and nonlinear conjugate gradient.
- *Complex matrix-free NLS optimization algorithms for tensor decompositions.* We exposed and exploited the structure in the Gauss–Newton approximation of the Hessian, resulting in matrix-free algorithms that substantially reduce the computational complexity and memory requirements of exact NLS algorithms. Our inexact NLS approach

inherently deals with the singularity of the Hessian approximation. Furthermore, we proposed an efficient preconditioner to accelerate convergence. Numerical experiments confirm that these improvements make the matrix-free NLS methods among the most efficient currently available.

Chapter 4:

- *Introduction of tensor optimization and classification of its line and plane search subproblems.* We introduced tensor optimization as a broad class of optimization problems comprising applications ranging from tensor decompositions to least squares support tensor machines. We classified the real and complex line and plane search subproblems used for accelerating tensor optimization solvers and showed how their global minimizers may be obtained by computing the solution of possibly polyanalytic univariate and bivariate polynomial systems.
- *Higher-order two-dimensional subspace minimization.* Many optimization algorithms generate iterates as a linear combination of two search directions: the steepest descent direction and an approximate Newton step. We introduced a higher-order two-dimensional subspace minimization that can be used to find the global minimizer of higher-order Taylor series approximation of the objective function within a specified region of the plane spanned by the two search directions. The method generalizes the two-dimensional subspace minimization method [159] and can be solved with a single generalized eigenvalue decomposition.
- *Scaled line and plane search.* In collaboration with Ignat Domanov, we introduced scaled line and plane search, which compute an optimal scaling of the solution simultaneously with the optimal line or plane search step. We showed that this scaling can be computed at almost no additional cost and also considered the complex extension of scaled line search, which requires minimizing a polyanalytic rational function.

Chapter 5:

- *State-of-the-art algorithm for solving bivariate and polyanalytic polynomial systems.* We proposed an efficient and numerically robust algorithm for solving bivariate and polyanalytic polynomial systems with a single generalized eigenvalue decomposition. Numerical experiments show that the algorithm consistently recovers a higher percentage of solutions and is

at the same time significantly faster and more accurate than competing double precision solvers.

- *Methods for numerical balancing and complex Newton–Raphson.* We presented several techniques for numerical balancing of bivariate polynomial systems and their associated generalized eigenvalue problem that can be transferred to other algorithms. Furthermore, we introduced a complex Newton–Raphson algorithm that simultaneously refines the candidate roots and filters spurious roots introduced by the method.
- *A benchmark for polynomial system solvers.* A benchmark of 150 low-degree polynomial systems and about 50 moderate-degree systems was compiled. There is a definite lack of standardized problem sets to test polynomial system solvers on. By providing this benchmark as an online resource, we hope to stimulate more quantitative comparisons between solvers.

Chapter 6:

- *A framework for structured data fusion (SDF).* We presented SDF as a framework for data fusion that enables the rapid prototyping of coupled tensor factorizations with structured factors. In SDF, each data set is represented as a dense, sparse or incomplete tensor and is factorized with a tensor decomposition. Factorizations can be coupled with each other by indicating which factors should be shared between data sets. At the same time, factors may be imposed to have any type of structure that can be constructed as an explicit function of some underlying variables.
- *A domain specific language (DSL) for modelling data fusion problems.* We designed and implemented a DSL for modelling SDF problems in Tensorlab. The DSL is intuitive to use, yet flexible enough to allow complicated data fusion models that depend on factors consisting of multiple subfactors, each of which may be the result of applying a chain of several factor structures. Together with the DSL, Tensorlab offers three types of tensor decomposition models, two types of regularization and a library of 32 types of factor structures to choose from.
- *Complex quasi-Newton and NLS optimization algorithms for coupled tensor factorizations with structured factors.* We have generalized the structure-exploiting algorithms developed in Chapter 3 to support sparse and incomplete tensors, new types of tensor decompositions, structured factors, regularization, and joint factorization of multiple data sets. Moreover, SDF allows new tensor decompositions, regularization models, and factor structures to be implemented without any changes to these algorithms.

7.2 Future research

Despite the considerable progress made in recent years, tensors and especially data fusion are still in their infancy. In the following, we discuss a few of the many possible directions for future research:

- *Probabilistic data fusion.* A probabilistic generalization of data fusion would significantly increase its flexibility and performance by enabling the integration of different noise models and prior knowledge into the optimization problem. Even more powerful and expressive factorizations can be obtained by allowing the addition of (in)equality constraints to the optimization problem. A framework for probabilistic coupled tensor factorizations has already been proposed in [248, 249], though it does not support complex factorizations or structured factors.

- *Data fusion by optimization on manifolds.* Optimization on manifolds has steadily been gaining traction as a potent method for embedding constraints into optimization problems [3]. In the case of data fusion, optimization on manifolds could join forces with standard optimization to provide an alternative way of imposing constraints or structure on the factors in a data fusion model. So far, optimization on manifolds has been limited to real-valued matrix manifolds such as the real Stiefel and low-rank matrix manifolds. Another compelling extension would be to consider the generalization to complex-valued matrix and tensor manifolds.

- *Automatic hyperparameter estimation.* Although structured data fusion enables rapidly exploring the vast search space of coupled tensor factorizations, it remains a chore to tune all of a model's parameters. Even simple data fusion models easily require more than four hyperparameters, making grid searches computationally prohibitive and warranting the need for more sophisticated model selection techniques. A correct estimation of the hyperparameters is often paramount for the model to generalize well to new data. Several approaches for automatic hyperparameter estimation are available, including cross-validation, placing priors on the hyperparameters and optimizing the log-posterior over both the model and its parameters simultaneously [160], and using Gaussian processes to model the generalization performance as a function of the hyperparameters [203].

- *Multi-objective optimization.* The different factorizations in a data fusion model need to be weighted and often it is not clear, a priori, what the relative importance of these factorizations should be. Multi-objective optimization provides a means of computing a Pareto-optimal set of

solutions to the problem. Alternatively, the weights can be treated as optimization variables and it could be attempted to find a solution along the Pareto front that is optimal in the sense of, e.g., maximum curvature or maximum volume under the Pareto surface. Finally, the weights may also be regarded as hyperparameters and determined with the techniques of the previous paragraph.

- *Online data fusion.* Big data is not only about volume and variety, but also about velocity. For example, in a biomedical setting real-time analysis of EEG measurements can be used to detect and predict brain events such as epileptic seizures. These measurements could be fused with other sensor data to make more accurate predictions and improved assessments. Fusion of real-time data requires the development of online algorithms for data fusion, which is as of yet an unexplored area of research.

Appendix A

Tensorlab user guide

1 Getting started

What is Tensorlab? In short, Tensorlab is a MATLAB toolbox for rapid prototyping of (coupled) tensor decompositions with structured factors. In Tensorlab, data sets are stored as (possibly incomplete) vectors, matrices and higher-order tensors. By mixing different types of decompositions and factor structures, a vast amount of factorizations can be computed. Users can define their own factor structures, or choose from the library of preimplemented structures, including nonnegativity, orthogonality, Toeplitz and Vandermonde matrices to name a few.

Section 2 covers how dense, incomplete and sparse data sets are represented as tensors in Tensorlab and the basic operations on such tensors. Tensor decompositions such as the canonical polyadic decomposition (CPD), low multilinear rank approximation (LMLRA) and block term decompositions (BTD) are discussed in Sections 3, 4 and 5, respectively. Structured data fusion (SDF), with which multiple data sets can be jointly factorized while imposing structure on the factors is covered in Section 6. Many of the algorithms to accomplish these tasks are based on complex optimization, that is, optimization of functions in complex variables. Section 7 introduces the necessary concepts and shows how to solve different types of complex optimization problems. Finally, Section 8 treats global optimization of bivariate (and polyanalytic) polynomials and rational functions, which appear as subproblems in tensor optimization.

Installation Unzip Tensorlab to any directory, browse to that location in MATLAB and run

```
addpath(pwd); % Add the current directory to the MATLAB search path.
savepath;     % Save the search path for future sessions.
```

Requirements Tensorlab requires MATLAB 7.9 (R2009b) or higher because of its dependency on the tilde operator `~`. If necessary, older versions of MATLAB can use Tensorlab by replacing the tilde in `[~` and `~,` with `tmp`. To do so on Linux/OS X, browse to Tensorlab and run

```
sed -i "" 's/\[~/[tmp/g;s/~,/tmp,/g' *.m
```

in your system's terminal. However, most of the functionality in Tensorlab requires at the very minimum MATLAB 7.4 (R2007a) because of its extensive use of `bsxfun`.

Octave is only partially supported, mainly because it coerces nested functions into subfunctions. The latter do not share the workspace of their parent function, which is a feature used by Tensorlab in certain algorithms.

Contents.m If you have installed Tensorlab to the directory `tensorlab`, run `doc tensorlab` from the command line for an overview of the toolboxes functionality (or, if that fails, try `help(pwd)`). Both commands display the file `Contents.m`, shown below. Although this user guide covers the most important aspects of Tensorlab, `Contents.m` shows a short one line description of all exported functions.

```
% BLOCK TERM DECOMPOSITION
% Algorithms
%    btd_minf     - BTD by unconstrained nonlinear optimization.
%    btd_nls      - BTD by nonlinear least squares.
% Initialization
%    btd_rnd      - Pseudorandom initialization for BTD.
% Utilities
%    btdgen       - Generate full tensor given a BTD.
%    btdres       - Residual of a BTD.
%
% CANONICAL POLYADIC DECOMPOSITION
% Algorithms
%    cpd          - Canonical polyadic decomposition.
%    cpd_als      - CPD by alternating least squares.
%    cpd_minf     - CPD by unconstrained nonlinear optimization.
%    cpd_nls      - CPD by nonlinear least squares.
```

```
%    cpd3_sd      - CPD by simultaneous diagonalization.
%    cpd3_sgsd    - CPD by simultaneous generalized Schur decomposition.
% Initialization
%    cpd_gevd     - CPD by a generalized eigenvalue decomposition.
%    cpd_rnd      - Pseudorandom initialization for CPD.
% Line and plane search
%    cpd_aels     - CPD approximate enhanced line search.
%    cpd_els      - CPD exact line search.
%    cpd_eps      - CPD exact plane search.
%    cpd_lsb      - CPD line search by Bro.
% Utilities
%    cpderr       - Errors between factor matrices in a CPD.
%    cpdgen       - Generate full tensor given a polyadic decomposition.
%    cpdres       - Residual of a polyadic decomposition.
%    rankest      - Estimate rank.
%
% COMPLEX OPTIMIZATION
% Nonlinear least squares
%    nls_gncgs    - Nonlinear least squares by Gauss-Newton with CG-
    Steihaug.
%    nls_gndl     - Nonlinear least squares by Gauss-Newton with dogleg
    trust region.
%    nls_lm       - Nonlinear least squares by Levenberg-Marquardt.
%    nlsb_gndl    - Bound-constrained NLS by projected Gauss-Newton
    dogleg TR.
% Unconstrained nonlinear optimization
%    minf_lbfgs   - Minimize a function by L-BFGS with line search.
%    minf_lbfgsdl - Minimize a function by L-BFGS with dogleg trust
    region.
%    minf_ncg     - Minimize a function by nonlinear conjugate gradient.
%    minf_sr1cgs  - Minimize a function by SR1 with CG-Steihaug.
% Utilities
%    deriv        - Approximate gradient and Jacobian.
%    ls_mt        - Strong Wolfe line search by More-Thuente.
%    mpcg         - Modified preconditioned conjugate gradients method.
%
% LOW MULTILINEAR RANK APPROXIMATION
% Algorithms
%    lmlra        - Low multilinear rank approximation.
%    lmlra_hooi   - LMLRA by higher-order orthogonal iteration.
%    lmlra_minf   - LMLRA by unconstrained nonlinear optimization.
%    lmlra_nls    - LMLRA by nonlinear least squares.
%    lmlra3_dgn   - LMLRA by a differential-geometric Newton method.
%    lmlra3_rtr   - LMLRA by a Riemannian trust region method.
%    mlsvd        - (Truncated) multilinear singular value decomposition
    .
```

```
% Initialization
%    lmlra_aca    - LMLRA by adaptive cross-approximation.
%    lmlra_rnd    - Pseudorandom initialization for LMLRA.
% Utilities
%    lmlraerr     - Errors between factor matrices in a LMLRA.
%    lmlragen     - Generate full tensor given a core tensor and factor
    matrices.
%    lmlrares     - Residual of a LMLRA.
%    mlrank       - Multilinear rank.
%    mlrankest    - Estimate multilinear rank.
%
% STRUCTURED DATA FUSION
% Algorithms
%   sdf_minf      - Structured data fusion by unconstrained nonlinear
    optimization.
%   sdf_nls       - Structured data fusion by nonlinear least squares.
% Structure
%   struct_abs        - Absolute value.
%   struct_band       - Band matrix.
%   struct_cell2mat   - Convert the contents of a cell array into a
    matrix.
%   struct_conj       - Complex conjugate.
%   struct_ctranspose - Complex conjugate transpose.
%   struct_diag       - Diagonal matrix.
%   struct_gram       - Gramian matrix.
%   struct_hankel     - Hankel matrix.
%   struct_inv        - Matrix inverse.
%   struct_invsqrtm   - Matrix inverse square root.
%   struct_invtransp  - Matrix inverse transpose.
%   struct_LL1        - Structure of the third factor matrix in a rank-(
    Lr,Lr,1) BTD.
%   struct_log        - Natural logarithm.
%   struct_matvec     - Matrix-vector and matrix-matrix product.
%   struct_nonneg     - Nonnegative array.
%   struct_normalize  - Normalize columns to unit norm.
%   struct_orth       - Rectangular matrix with orthonormal columns.
%   struct_plus       - Plus.
%   struct_poly       - Matrix with columns as polynomials.
%   struct_power      - Array power.
%   struct_rational   - Matrix with columns as rational functions.
%   struct_rbf        - Matrix with columns as sums of Gaussian RBF
    kernels.
%   struct_sigmoid    - Constrain array elements to an interval.
%   struct_sqrt       - Square root.
%   struct_sum        - Sum of elements.
%   struct_times      - Times.
```

```
%   struct_toeplitz   - Toeplitz matrix.
%   struct_transpose  - Transpose.
%   struct_tridiag    - Tridiagonal matrix.
%   struct_tril       - Lower triangular matrix.
%   struct_triu       - Upper triangular matrix.
%   struct_vander     - Vandermonde matrix.
%
% UTILITIES
% Clustering
%    gap          - Optimal clustering based on the gap statistic.
%    kmeans       - Cluster multivariate data using the k-means++
    algorithm.
% Polynomials
%    polymin      - Minimize a polynomial.
%    polymin2     - Minimize bivariate and real polyanalytic polynomials
    .
%    polyval2     - Evaluate bivariate and univariate polyanalytic
    polynomials.
%    polysol2     - Solve a system of two bivariate polynomials.
%    ratmin       - Minimize a rational function.
%    ratmin2      - Minimize bivariate and real polyanalytic rational
    functions.
% Statistics
%    cum4         - Fourth-order cumulant tensor.
%    scov         - Shifted covariance matrices.
% Tensors
%    dotk         - Dot product in K-fold precision.
%    fmt          - Format data set.
%    frob         - Frobenius norm.
%    ful          - Convert formatted data set to an array.
%    kr           - Khatri-Rao product.
%    kron         - Kronecker product.
%    mat2tens     - Tensorize a matrix.
%    mtkronprod   - Matricized tensor Kronecker product.
%    mtkrprod     - Matricized tensor Khatri-Rao product.
%    noisy        - Generate a noisy version of a given array.
%    sumk         - Summation in K-fold precision.
%    tens2mat     - Matricize a tensor.
%    tens2vec     - Vectorize a tensor.
%    tmprod       - Mode-n tensor-matrix product.
%    vec2tens     - Tensorize a vector.
% Visualization
%    slice3       - Visualize a third-order tensor with slices.
%    spy3         - Visualize a third-order tensor's sparsity pattern.
%    surf3        - Visualize a third-order tensor with surfaces.
%    voxel3       - Visualize a third-order tensor with voxels.
```

2 Data sets: dense, incomplete and sparse tensors

2.1 Representation

Dense tensors Scalars, vectors and matrices are zero-, one- and two-dimensional tensors, respectively. Arrays with three or more dimensions are called higher-order tensors. In Tensorlab, data sets are represented as dense, sparse or incomplete tensors. A dense tensor is simply a MATLAB array, e.g., `A = randn(10,10)` or `T = randn(5,5,5)`. All functions in Tensorlab accept dense tensors, and many (but not all) also accept incomplete and sparse tensors transparently.

Incomplete tensors An incomplete tensor is a data set in which some (or most) of the entries are unknown. To create an incomplete tensor, define a structure containing the tensor's known elements, their positions, the tensor's size and an incomplete flag as:

```
T.val = randn(3,1);
T.sub = [1 2 3; 4 5 6; 7 8 9];
T.size = [9 9 9];
T.incomplete = true;
```

and then call

```
T = fmt(T);
```

to convert the tensor to Tensorlab format. This creates a $9 \times 9 \times 9$ tensor `T` with three known elements at positions $(1, 2, 3)$, $(4, 5, 6)$ and $(7, 8, 9)$. Alternatively, the user may supply the linear indices in `T.ind` instead of the subindices `T.sub`. To convert linear indices to subindices or vice versa, see the MATLAB functions `ind2sub` and `sub2ind`, respectively.

Another way to create an incomplete tensor is to create a dense tensor in which the unknown values are `NaN`. For example

```
T = randn(5,5,5);
T(1:31:end) = NaN;
T = fmt(T);
```

creates a $5 \times 5 \times 5$ tensor `T` with diagonal elements equal to `NaN` and formats it as an incomplete tensor. If there are no entries equal to `NaN`, then `fmt(T)` returns `T` itself. To convert an incomplete tensor back to a MATLAB array, use `ful(T)`.

Sparse tensors A sparse tensor is a data set in which most of the entries are zero. To create a sparse tensor, define a structure containing the tensor's nonzero elements in the same way as for incomplete tensors and set the sparse flag:

```
T.val = randn(3,1);
T.sub = [1 2 3; 4 5 6; 7 8 9];
T.size = [9 9 9];
T.sparse = true;
```

before formatting the tensor with `T = fmt(T)`.

Another way to create a sparse tensor is to create a dense tensor in which at most 5% of the entries are nonzero. For example

```
T = zeros(5,5,5);
T(1:31:end) = 1;
T = fmt(T);
```

creates a $5 \times 5 \times 5$ diagonal tensor `T` with diagonal elements equal to 1 and formats it as a sparse tensor. If not enough entries are zero, then `fmt(T)` returns `T` itself. To convert a sparse tensor back to a MATLAB array, use `ful(T)`.

2.2 Tensor operations

For dense tensors

Matricization and tensorization A dense tensor `T` can be flattened into a matrix with `M = tens2mat(T,mode_row,mode_col)`. Let `size_tens = size(T)`, then the resulting matrix `M` is of size `prod(size_tens(mode_row))` by `prod(size_tens(mode_col))`. For example,

```
T = randn(3,5,7,9);
M = tens2mat(T,[1 3],[4 2]);
size(M)
```

outputs `[21 45]`. In `tens2mat`, a given column (row) of `M` is generated by fixing the indices corresponding to `mode_col` (`mode_row`) and then looping over the remaining indices in the order `mode_row` (`mode_col`). To transform a matricized tensor `M` back into its original size `size_tens`, use `mat2tens(M, size_tens,mode_row,mode_col)`.

The most common use case is to matricize a tensor by placing its mode-n vectors as columns in a matrix, also called a mode-n matricization. This can be

achieved by

```
T = randn(3,5,7);
n = 2;
M = tens2mat(T,n);
```

where the optional argument `mode_col` is implicitly equal to `[1:n-1 n+1:ndims(T)]`.

Tensor-matrix product In a mode-n tensor-matrix product, the tensor's mode-n vectors are premultiplied by a given matrix. In other words, `U*tens2mat(T,n)` is a mode-n matricization of the mode-n tensor-matrix product `T` $\bullet_n$ `U`. The function `tmprod(T,U,mode)` computes the tensor-matrix product `T` $\bullet_{\texttt{mode(1)}}$ `U{1}` $\bullet_{\texttt{mode(2)}}$ `U{2}` $\cdots$. For example,

```
T = randn(3,5,7);
U = {randn(11,3),randn(13,5),randn(15,7)};
S = tmprod(T,U,1:3);
size(S)
```

outputs `[11 13 15]`. To compute a single tensor-matrix product, it is not necessary to use a cell array. The following is an example of a single mode-2 tensor-matrix product:

```
T = randn(3,5,7);
S = tmprod(T,randn(13,5),2);
```

Kronecker and Khatri–Rao product Tensorlab includes a fast implementation of the Kronecker product $A \otimes B$ with the function `kron(A,B)`, which overrides MATLAB's built-in implementation. Let A and B be matrices of size I by J and K by L, respectively, then the Kronecker product of A and B is the IK by JL matrix

$$A \otimes B := \begin{bmatrix} a_{11}B & \cdots & a_{1J}B \\ \vdots & \ddots & \vdots \\ a_{I1}B & \cdots & a_{IJ}B \end{bmatrix}.$$

The Khatri–Rao product $A \odot B$ can be computed by `kr(A,B)`. Let A and B both be matrices with N columns, then the Khatri–Rao product of A and B is the column-wise Kronecker product

$$A \odot B := \begin{bmatrix} \boldsymbol{a}_1 \otimes \boldsymbol{b}_1 & \cdots & \boldsymbol{a}_N \otimes \boldsymbol{b}_N \end{bmatrix}.$$

More generally, `kr(A,B,C,...)` and `kr(U)` compute the string of Khatri–Rao products `((A` $\odot$ `B)` $\odot$ `C)` $\odot \cdots$ and `((U{1}` $\odot$ `U{2})` $\odot$ `U{3})` $\odot \cdots$, respectively.

Visualization Tensorlab offers three methods to visualize dense third-order tensors: `slice3`, `surf3` and `voxel3`. The following example demonstrates these methods on the amino acids dataset [28]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
figure(1); voxel3(X);
figure(2); surf3(X);
figure(3); slice3(X);
```

The resulting MATLAB figures can be seen in Figure A.1. By default, the `voxel3` plot uses a fast but not so accurate renderer. To enable the slow but accurate renderer, use `voxel3(X,'fast',false)`. See the `help` information for more rendering options.

For dense, incomplete and sparse tensors

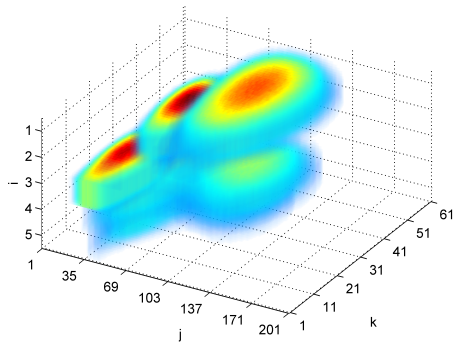
Frobenius norm The Frobenius norm of a tensor is the square root of the sum of square magnitudes of its (known) elements. Given a tensor `T`, its Frobenius norm can be computed with `frob(T)`. If the tensor is dense, this is equivalent with `norm(T(:))`, i.e., the two-norm of the vectorized tensor. The squared Frobenius norm can be computed with `frob(T,'squared')`.

Matricized tensor Kronecker product Often, algorithms for computing tensor decompositions do not explicitly need matricized tensors or Kronecker products, but rather the matricized tensor Kronecker product `tens2mat(T,n)*kron(U([end:-1:n+1 n-1:-1:1]))`, where the second operand represents the Kronecker product $\texttt{U\{end\}} \otimes \cdots \otimes \texttt{U\{n+1\}} \otimes \texttt{U\{n-1\}} \otimes \cdots \otimes \texttt{U\{1\}}$. The function `mtkronprod(T,U,n)` computes the result of this operation without explicitly computing either of the operands and without permuting the elements of the tensor `T` in memory.

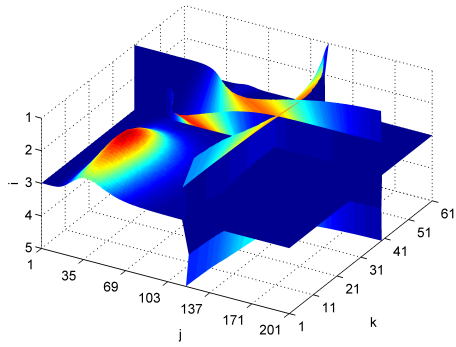
Another useful way of interpreting this product is as the result of

```
mode = [1:n-1 n+1:length(U)];
Ut = cellfun(@transpose,U,'UniformOutput',false);
tens2mat(tmprod(T,Ut(mode),mode),n)
```

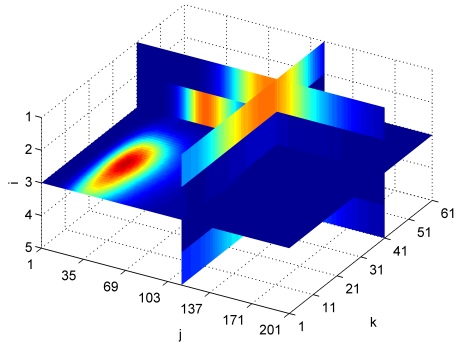
Matricized tensor Khatri-Rao product Similarly, algorithms for computing tensor decompositions usually don't explicitly need matricized tensors or Khatri-Rao products, but rather the matricized tensor Khatri-Rao product `tens2mat(T,`



(a) `voxel3`



(b) `surf3`



(c) `slice3`

Figure A.1: Three functions for visualizing third-order tensors in Tensorlab.

`n)*kr(U([end:-1:n+1 n-1:-1:1]))`. The function `mtkrprod(T,U,n)` computes the result of this operation without explicitly computing either of the operands and without permuting the elements of the tensor `T` in memory.

Another useful way of interpreting the rth column of this product is as the result of

```
mode = [1:n-1 n+1:length(U)];
Urt = cellfun(@(u)u(:,r).',U,'UniformOutput',false);
tens2mat(tmprod(T,Urt(mode),mode),n)
```

Creating noisy versions of (cell) arrays The `noisy` function can be used to create a noisy version of a MATLAB array, of a (nested) cell array of arrays and of incomplete or sparse tensors. For example,

```
U = {randn(10,2),randn(15,2),randn(20,2)};
SNR = 20;
Uhat = noisy(U,SNR);
```

creates a cell array `U` and a noisy version of that cell array called `Uhat` with a signal-to-noise ratio of 20 dB.

3 Canonical polyadic decomposition

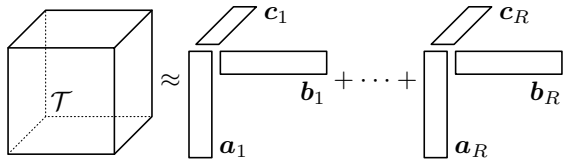


Figure A.2: Canonical polyadic decomposition of a third-order tensor.

The canonical polyadic decomposition (CPD) [33, 95, 99, 100, 112] approximates a tensor with a sum of R rank-one tensors. Let $\mathcal{A} \circ \mathcal{B}$ denote the outer product between an N-dimensional tensor $\mathcal{A}$ and an M-dimensional tensor $\mathcal{B}$, then $\mathcal{A} \circ \mathcal{B}$ is the $(N+M)$-dimensional tensor defined by $(\mathcal{A} \circ \mathcal{B})_{i_1 \cdots i_N \, j_1 \cdots j_M} = a_{i_1 \cdots i_N} \cdot b_{j_1 \cdots j_M}$. For example, let $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ be nonzero vectors in $\mathbb{R}^n$, then $\boldsymbol{a} \circ \boldsymbol{b} \equiv \boldsymbol{a} \cdot \boldsymbol{b}^{\mathrm{T}}$ is a rank-one matrix and $\boldsymbol{a} \circ \boldsymbol{b} \circ \boldsymbol{c}$ is defined to be a rank-one tensor. Let `T` be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$, and let `U{n}` be matrices of size $I_n \times R$,

then

$$\mathtt{T} \approx \sum_{r=1}^{R} \mathtt{U\{1\}(:,r)} \circ \mathtt{U\{2\}(:,r)} \circ \cdots \circ \mathtt{U\{N\}(:,r)}$$

is a CPD of T in R rank-one terms. A visual representation of this decomposition in the third-order case is shown in Figure A.2.

3.1 Problem and tensor generation

Generating pseudorandom factor matrices A cell array of pseudorandom factor matrices `U = {U{1},U{2},...}` corresponding to a CPD of a tensor of dimensions `size_tens` in R rank-one terms can be generated with

```
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
```

By default `cpd_rnd` generates `U{n}` as `randn(size_tens(n),R)`. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @rand;
U = cpd_rnd(size_tens,R,options);
```

and the inline equivalent

```
U = cpd_rnd(size_tens,R,struct('Real',@rand,'Imag',@rand));
```

generate `U{n}` as `rand(size_tens(n),R)+rand(size_tens(n),R)*1i`.

Generating the associated full tensor Given a cell array of factor matrices `U = {U{1},U{2},...}`, its associated full tensor T can be computed with

```
T = cpdgen(U);
```

This is equivalent to

```
M = U{1}*kr(U(end:-1:2)).';
size_tens = cellfun('size',U,1);
T = mat2tens(M,size_tens,1);
```

3.2 Computing the CPD

The principal method To compute the CPD of a dense, sparse or incomplete tensor `T` in R rank-one terms, call `cpd(T,R)`. For example,

```
% Generate pseudorandom factor matrices U and their associated full
    tensor T.
size_tens = [7 8 9]; R = 4;
U = cpd_rnd(size_tens,R);
T = cpdgen(U);

% Compute the CPD of the full tensor T.
Uhat = cpd(T,R);
```

generates a real rank-4 tensor and decomposes it. Internally, `cpd` first *compresses the tensor* using a low multilinear rank approximation (see Section 4) if it is worthwhile, then chooses a method to *generate an initialization* `U0` (e.g., `cpd_gevd`), after which it *executes an algorithm* to compute the CPD given the initialization (e.g., `cpd_nls`) and finally decompresses the tensor and *refines the solution* (if compression was applied).

Setting the options The different steps in `cpd` are customizable by supplying the method with an options structure (see `help cpd` for more information), e.g.,

```
options.Display = true; % Show progress on the command line.
options.Initialization = @cpd_rnd; % Select pseudorandom initialization.
options.Algorithm = @cpd_als; % Select ALS as the main algorithm.
options.AlgorithmOptions.LineSearch = @cpd_els; % Add exact line search.
options.AlgorithmOptions.TolFun = 1e-12; % Set algorithm stop criteria.
options.AlgorithmOptions.TolX = 1e-12;
Uhat = cpd(T,R,options);
```

The structures `options.InitializationOptions`, `options.AlgorithmOptions` and `options.RefinementOptions` will be passed as options structures to the algorithms corresponding to initialization, algorithm and refinement steps, respectively. For example, in the example above `cpd` will call `cpd_als` as `cpd_als(T,options.Initialization(T,R),options.AlgorithmOptions)`.

Viewing the algorithm output Each step may also output additional information specific to that step. For instance, most CPD algorithms such as `cpd_als` will keep track of the number of iterations and objective function value. To obtain this information, capture the second output:

```
[Uhat,output] = cpd(T,R,options);
```

and inspect its fields, for example by plotting the objective function value:

```
semilogy(0:output.Algorithm.iterations,sqrt(2*output.Algorithm.fval));
xlabel('iteration');
ylabel('frob(cpdres(T,U))');
grid on;
```

Computing the error The residual between a tensor `T` and its CPD approximation defined by `Uhat` can be computed with

```
res = cpdres(T,Uhat);
```

If the tensor is dense, then the result is equivalent to `cpdgen(Uhat)-T`. The relative error between the tensor and its CPD approximation can then be computed as

```
relerr = frob(cpdres(T,Uhat))/frob(T);
```

One can also consider the relative error between the factor matrices `U{n}` that generated `T` and their approximations `Uhat{n}` computed by `cpd`. Due to the permutation and scaling indeterminacies of the CPD, the columns of `Uhat` may need to be permuted and scaled to match those of `U` before comparing them to each other. The function `cpderr` takes care of these indeterminacies and then computes the relative error between the given two sets of factor matrices. I.e.,

```
relerr = cpderr(U,Uhat);
```

returns a vector in which the nth entry is the relative error between `U{n}` and `Uhat{n}`. This method is also applicable when `Uhat{n}` is an under- or overfactoring of the solution, meaning `Uhat{n}` comprises fewer or more rank-one terms than the tensor's rank, respectively. In the underfactoring case, `Uhat{n}` is padded with zero-columns, and in the overfactoring case only the columns in `Uhat{n}` that best match those in `U{n}` are kept. See the `help` information for more details.

3.3 Choosing the number of rank-one terms R

To help choose the number of rank-one terms R, use the `rankest` tool. Running `rankest(T)` on a dense, sparse or incomplete tensor `T` plots an L-curve which represents the balance between the relative error of the CPD and the number of

rank-one terms R. The following example applies `rankest` on the amino acids dataset [28]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
rankest(X);
```

The resulting figure is shown in Figure A.3. The algorithm computes the CPD of the given tensor for various values of R, starting at the smallest integer for which the lower bound on the relative error (displayed as a solid blue line) is smaller than the specified `options.MinRelErr`. The lower bound is based on the truncation error of the tensor's multilinear singular values [54]. For incomplete and sparse tensors, this lower bound is not available and the first value to be tried for R is 1. The number of rank-one terms is increased until the relative error of the approximation is less than `options.MinRelErr`. In a sense, the corner of the resulting L-curve makes an optimal trade-off between accuracy and compression. The `rankest` tool computes the number of rank-one terms R corresponding to the L-curve corner and marks it on the plot with a square. This optimal number of rank-one terms is also `rankest`'s first output. By capturing it as `R = rankest(X)`, the L-curve is no longer plotted.

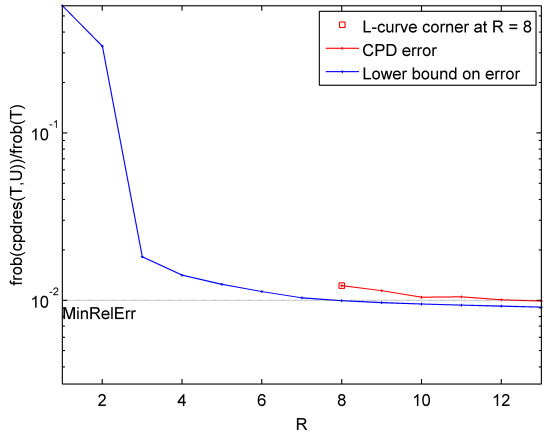


Figure A.3: The `rankest` tool for choosing the number of rank-one terms R in a CPD.

4 Low multilinear rank approximation

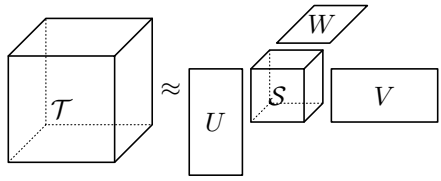


Figure A.4: Low multilinear rank approximation of a third-order tensor.

A low multilinear rank approximation (LMLRA) [112, 232] approximates a tensor by a smaller (core) tensor and a set of factor matrices. Let `T` and `S` be tensors of dimensions $I_1 \times I_2 \times \cdots \times I_N$ and $J_1 \times J_2 \times \cdots \times J_N$, respectively, let `U{n}` be matrices of size $I_n \times J_n$ ($J_n \leq I_n$) for $1 \leq n \leq N$, and let $\bullet_n$ denote the mode-n tensor-matrix product (see Section 2.2), then

$$\mathtt{T} \approx \mathtt{S} \bullet_1 \mathtt{U\{1\}} \bullet_2 \mathtt{U\{2\}} \bullet_3 \cdots \bullet_{\mathtt{N}} \mathtt{U\{N\}}$$

is a LMLRA of the tensor `T` in the core tensor `S` and factor matrices `U{n}`. A visual representation of this decomposition in the third-order case is shown in Figure A.4.

4.1 Problem and tensor generation

Generating pseudorandom factor matrices and core tensor A cell array of pseudorandom factor matrices `U = {U{1},U{2},...}` with orthonormal columns and a core tensor `S` of dimensions `size_core` corresponding to a LMLRA of a tensor of dimensions `size_tens` can be generated with

```
size_tens = [17 19 21];
size_core = [3 5 7];
[U,S] = lmlra_rnd(size_tens,size_core);
```

By default `lmlra_rnd` generates `U{n}` and `S` using `randn`, after which `U{n}` is orthogonalized. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @rand;
[U,S] = lmlra_rnd(size_tens,size_core,options);
```

and its inline equivalent

```
[U,S] = lmlra_rnd(size_tens,size_core,struct('Real',@rand,'Imag',@rand))
```

Generating the associated full tensor Given a cell array of factor matrices `U = {U{1},U{2},...}` and core tensor `S`, its associated full tensor `T` can be computed with

```
T = lmlragen(U,S);
```

This is equivalent to

```
T = tmprod(S,U,1:length(U));
```

4.2 Computing a LMLRA

The principal method To compute a LMLRA of a dense, sparse or incomplete tensor `T` with a core tensor of size `size_core`, call `lmlra(T,size_core)`. For example,

```
% Generate pseudorandom LMLRA (U,S) and associated full tensor T.
size_tens = [17 19 21];
size_core = [3 5 7];
[U,S] = lmlra_rnd(size_tens,size_core);
T = lmlragen(U,S);

% Compute a LMLRA of a noisy version of T with 20dB SNR.
[Uhat,Shat] = lmlra(noisy(T,20),size_core);
```

generates a real rank-(3,5,7) tensor and computes its low multilinear rank approximation. Internally, `lmlra` chooses a method to *generate an initialization* `U0` and `S0` (e.g., `lmlra_aca`), after which it *executes an algorithm* to compute the LMLRA given the initialization (e.g., `lmlra_nls`).

Setting the options The two steps in `lmlra` are customizable by supplying the method with an options structure (see `help lmlra` for more information), e.g.,

```
options.Display = true; % Show progress on the command line.
options.Initialization = @lmlra_rnd; % Select pseudorandom
    initialization.
options.Algorithm = @lmlra_nls; % Select NLS as the main algorithm.
options.AlgorithmOptions.TolFun = 1e-12; % Set stop criteria.
options.AlgorithmOptions.TolX = 1e-12;
```

```
[Uhat,Shat] = lmlra(T,size_core,options);
```

The structures `options.InitializationOptions` and `options.AlgorithmOptions` will be passed as options structures to the algorithms corresponding to initialization and algorithm steps, respectively.

Viewing the algorithm output Each step may also output additional information specific to that step. For instance, most LMLRA algorithms such as `lmlra_hooi` will keep track of the number of iterations and the difference in subspace angle of every two successive iterates `sangle`. To obtain this information, capture the third output:

```
[Uhat,Shat,output] = lmlra(T,size_core,options);
```

and inspect its fields, for example by plotting the objective function value:

```
semilogy(0:output.Algorithm.iterations,sqrt(2*output.Algorithm.fval));
xlabel('iteration');
ylabel('frob(lmlrares(T,U))');
grid on;
```

Computing the error The residual between a tensor `T` and its LMLRA defined by `Uhat` and `Shat` can be computed with

```
res = lmlrares(T,Uhat,Shat);
```

If the tensor is dense, then the result is equivalent to `lmlragen(Uhat,Shat)-T`. The relative error between the tensor and its LMLRA can then be computed as

```
relerr = frob(lmlrares(T,Uhat,Shat))/frob(T);
```

If the factor matrices `U{n}` that generated `T` are known, the subspace angle between them and their approximations `Uhat{n}` computed by `lmlra` is also a measure of the approximation error. The function `lmlraerr` computes the subspace angle between the given two sets of factor matrices. In other words,

```
sangle = lmlraerr(U,Uhat);
```

returns a vector in which the nth entry is `subspace(U{n},Uhat{n})`.

4.3 Choosing the size of the core tensor

For dense tensors, use the `mlrankest` tool to help determine the size of the core tensor `size_core`. Running `mlrankest(T)` plots an L-curve which represents the balance between an upper bound [54] on the relative error of a LMLRA of `T` for different core tensor sizes, and the LMLRA's compression ratio. The following example applies `mlrankest` on the amino acids dataset [28]:

```
url = 'http://www.models.life.ku.dk/go-engine?filename=amino.mat';
urlwrite(url,'amino.mat'); % Download amino.mat in this directory.
load amino X;
mlrankest(X);
```

The resulting figure is shown in Figure A.5. The corner of this L-curve is often a good estimate of the optimal trade-off between accuracy and compression. The core tensor size `size_core` corresponding to the L-curve corner is marked on the plot with a square and is also `mlrankest`'s first output. By capturing it as in `size_core = mlrankest(X)`, the L-curve is no longer plotted. Additionally, the compression and relative error of other choices of `size_core` can be viewed by using the figure's Data Cursor tool.

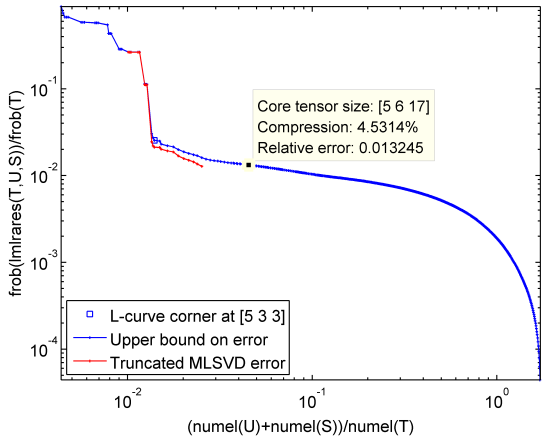


Figure A.5: The `mlrankest` tool for choosing the core tensor size `size_core` in a LMLRA.

5 Block term decomposition

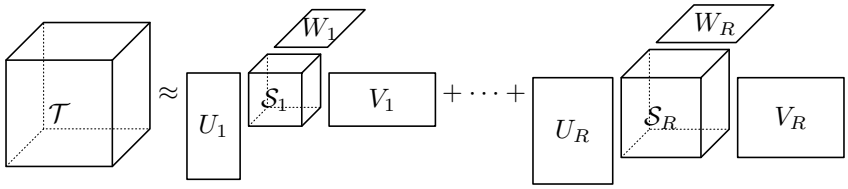


Figure A.6: Block term decomposition of a third-order tensor.

A block term decomposition (BTD) [112, 232] approximates a tensor by a sum of low multilinear rank terms. Let `T` be a tensor of dimensions $I_1 \times I_2 \times \cdots \times I_N$ and let `U{r}` represent the rth block term in the BTD. More specifically, let `U{r}{N+1}` be a core tensor of dimensions $J_1^{(r)} \times J_2^{(r)} \times \cdots \times J_N^{(r)}$ and let `U{r}{n}` be matrices of size $I_n \times J_n^{(r)}$ $(J_n^{(r)} \leq I_n)$ for $1 \leq n \leq N$, then

$$\texttt{T} \approx \sum_{r=1}^{R} \texttt{U\{r\}\{N+1\}} \bullet_1 \texttt{U\{r\}\{1\}} \bullet_2 \texttt{U\{r\}\{2\}} \bullet_3 \cdots \bullet_{\texttt{N}} \texttt{U\{r\}\{N\}}$$

is a BTD of the tensor `T` in the block terms `U{r}`. A visual representation of this decomposition in the third-order case is shown in Figure A.6.

5.1 Problem and tensor generation

Generating pseudorandom factor matrices and core tensor A cell array of block terms `U = {U{1},U{2},...}` in which the rth term has a core tensor `U{r}{end}` of size `size_core{r}` corresponding to a BTD of a tensor of dimensions `size_tens` can be generated with

```
size_tens = [17 19 21];
size_core = {[3 5 7],[6 3 5],[4 3 4]};
U = btd_rnd(size_tens,size_core);
```

By default `btd_rnd` generates `U{r}{n}` using `randn`, after which the factor matrices `U{r}{1:N}` are orthogonalized. Other generators can also be specified with an options structure, e.g.,

```
options.Real = @rand;
options.Imag = @rand;
[U,S] = btd_rnd(size_tens,size_core,options);
```

and its inline equivalent

```
[U,S] = btd_rnd(size_tens,size_core,struct('Real',@rand,'Imag',@rand));
```

Generating the associated full tensor Given a cell array of block terms `U = {U{1},U{2},...}`, its associated full tensor `T` can be computed with

```
T = btdgen(U);
```

This is equivalent to

```
R = length(U);
N = length(U{1})-1;
size_tens = cellfun('size',U{1}(1:N),1);
T = zeros(size_tens);
for r = 1:R
    T = T+tmprod(U{r}{N+1},U{r}(1:N),1:N);
end
```

5.2 Computing a BTD

With a specific algorithm Currently, the user must generate his or her own initialization and select a specific (family of) algorithm(s) such as `btd_nls` or `btd_minf` to compute the BTD given the initialization.

To compute a BTD of a dense, sparse or incomplete tensor `T` given an initialization `U0`, run `btd_nls(T,U0)`. For example,

```
% Generate pseudorandom BTD U and associated full tensor T.
size_tens = [17 19 21];
size_core = {[2 2 2],[2 2 2],[2 2 2]};
U = btd_rnd(size_tens,size_core);
T = btdgen(U);

% Generate an initialization U0 and compute the BTD with nonlinear least
     squares.
U0 = noisy(U,20);
Uhat = btd_nls(T,U0);
```

generates a tensor `T` as the sum of three real rank-(2,2,2) tensors and then computes its BTD.

Setting the options The selected algorithm may be customizable by supplying the method with an options structure (see the relevant `help` for more information), e.g.,

```
options.Display = 5; % Show convergence progress every 5 iterations.
options.MaxIter = 200; % Set stop criteria.
options.TolFun = 1e-12;
options.TolX = 1e-12;
Uhat = btd_nls(T,U0,options);
```

Viewing the algorithm output The selected method also returns output specific to the algorithm, such as the number of iterations and the algorithm's objective function value. To obtain this information, capture the second output:

```
[Uhat,output] = btd_nls(T,U0,options);
```

and inspect its fields, for example by plotting the objective function value:

```
semilogy(0:output.iterations,sqrt(2*output.fval));
xlabel('iteration');
ylabel('frob(btdres(T,U))');
grid on;
```

Computing the error The residual between a tensor `T` and its BTD defined by `Uhat` can be computed with

```
res = btdres(T,Uhat);
```

If the tensor is dense, then the result is equivalent to `btdgen(Uhat)-T`. The relative error between the tensor and its BTD can then be computed as

```
relerr = frob(btdres(T,Uhat))/frob(T);
```

6 Structured data fusion

Structured data fusion (SDF) [209] is the practice of jointly factorizing one or more coupled data sets while optionally imposing structure on the factors.

Each data set—stored as a dense, sparse or incomplete tensor, cf. Section 2—in a data fusion problem can be factorized with a different tensor decomposition. Currently, the user has the choice of the CPD and BTD models, and with a bit of effort it is easy to add new models as well.

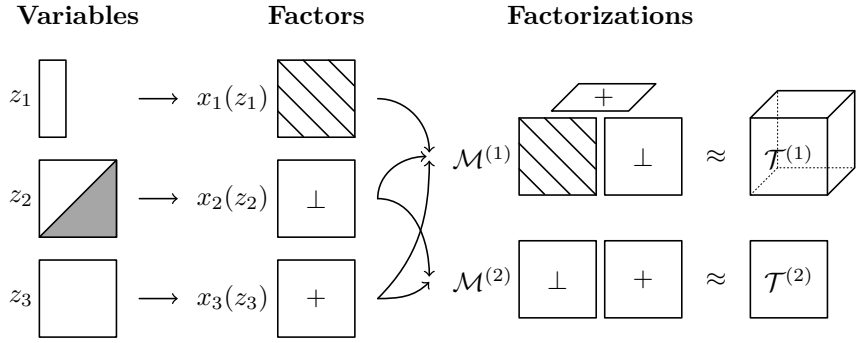


Figure A.7: Schematic of structured data fusion. The vector z_1, upper triangular matrix z_2 and full matrix z_3 are transformed into a Toeplitz, orthogonal and nonnegative matrix, respectively. The resulting factors are then used to jointly factorize two coupled data sets.

Structure can be imposed on the factors in a modular way and the user can choose from a library of predefined structures such as nonnegativity, orthogonality, Hankel, Toeplitz, Vandermonde, matrix inverse, and many more. See `Contents.m` for a complete list. By selecting the right structures you can even compute classical matrix decompositions such as the QR factorization, eigenvalue decomposition and singular value decomposition.

6.1 Domain specific language for SDF

Tensorlab uses a domain specific language (DSL) for modelling structured data fusion problems. The three key ingredients of an SDF model are (1) defining variables, (2) defining factors as transformed variables and (3) defining the data sets and which factors to use in their factorizations (cf. Figure A.7).

Example 1: nonnegative symmetric CPD. The DSL is best explained by example. To start, we'll compute the nonnegative symmetric CPD of an incomplete tensor `T`. First, generate the tensor:

```
% Generate a nonnegative symmetric CPD.
I = 15;
R = 4;
U = rand(I,R);
U = {U,U,U};
T = cpdgen(U);
```

```
% Remove 10% of its entries.
T(randperm(numel(T),round(0.1*numel(T)))) = NaN;

% Format as incomplete tensor.
T = fmt(T);
```

Next, create a structure `model` which defines the variables of the SDF problem. In this case, there is only one variable and its size is equal to that of the factor `U`. The `'variables'` field defines the parameters which are optimized, and is also used as initialization for the SDF algorithm.

```
% Define model variables.
model.variables.u = randn(I,R);
```

Here, the variable `u` is defined as a MATLAB array. It is also perfectly valid to define variables as (nested) cell arrays of arrays, if desired. Now we need to define the factors. There is only one factor in this CPD, which we will define as a transformation of the variable `u`. More specifically, we will require the factor to be nonnegative:

```
% Define model factors as transformed variables.
model.factors.U = {'u',@struct_nonneg}; % Create U as struct_nonneg(u).
```

The factor `U` is built taking the variable `u`, and applying the transformation `@struct_nonneg`. In fact, factors can be built with a much more complex structure using subfactors, see the following examples for details. Finally, we define the data set to be factorized and which factors to use:

```
% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'U','U','U'};
```

Each factorization in the SDF problem should be given a new name. In this case there is only one factorization `'myfac'` and it contains two fields. The first is `'data'` and contains the tensor to be factorized. The second should be either `'cpd'` or `'btd'`, depending on which model to use, and should define the factors to be used in the decomposition.

Note that it is not necessary to use fields to describe the names of the variables and factors. Instead, one may also create cell arrays of variables and factors and use indices to refer to them. In this format, the model would be written as

```
% Equivalent SDF model without using names for variables and factors.
model.variables = { randn(I,R) };
model.factors = { {1,@struct_nonneg} };
model.factorizations.myfac.data = T;
```

```
model.factorizations.myfac.cpd = {1,1,1};
```

The model can now be solved with one of the two families of algorithms for SDF problems: `sdf_minf` and `sdf_nls`. In the case of many missing entries, the `sdf_minf` family is likely to perform best. Their first output contains the optimized variables and factors in the fields `'variables'` and `'factors'`, respectively:

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 2: structured coupled matrix factorization. Let X and Y be square matrices of order N that we wish to jointly factorize as $U \cdot \Lambda_X \cdot V^T$ and $U \cdot \Lambda_Y \cdot W^T$, respectively. The factor U is common to both X and Y, and the extent to which the columns are shared is given by the absolute values of the diagonal matrices Λ_X and Λ_Y. Furthermore, we will require the elements of W to lie in the interval (-3,5) and impose a Vandermonde structure on V, so that

$$V = \begin{bmatrix} v_1^0 & v_1^1 & \cdots & v_1^d \\ \vdots & \vdots & \ddots & \vdots \\ v_N^0 & v_N^1 & \cdots & v_N^d \end{bmatrix}$$

for some positive integer d. Notice that V depends only on a so-called generator vector $\boldsymbol{v} = \begin{bmatrix} v_1 & \cdots & v_N \end{bmatrix}^{\mathrm{T}}$.

First, we'll create the matrices X and Y:

```
% Generate structured coupled matrices X and Y.
N = 10;
R = 4;
U = randn(N,R);
V = bsxfun(@power,randn(N,1),0:R-1); % Vandermonde factor.
W = rand(N,R)*(5+3)-3; % Elements in (-3,5).
lambdaX = 0:3; % X does not share first column of U.
lambdaY = 3:-1:0; % Y does not share last column of U.
X = U*diag(lambdaX)*V.';
Y = U*diag(lambdaY)*W.';
```

Then, we define a structure containing the model's variables

```
% Define model variables.
model.variables.u = randn(N,R);
model.variables.v = randn(N,1);
```

```
model.variables.w = randn(N,R);
model.variables.lambdaX = randn(1,R);
model.variables.lambdaY = randn(1,R);
```

Here, the variables are the matrices U and W, the generator vector $\boldsymbol{v}$ for the Vandermonde matrix V and the diagonals $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$. The model assumes that X and Y share at most $R = 4$ vectors in U. Next, we define the factors as transformed variables with

```
% Define the structure for V by creating an anonymous function which
    stores deg.
deg = [0 R-1];
vanderm = @(z,task)struct_vander(z,task,deg);

% Define the structure for W by creating an anonymous function which
    stores rng.
rng = [-3 5];
sigmoid = @(z,task)struct_sigmoid(z,task,rng);

% Define model factors as transformed variables.
model.factors.U = 'u';
model.factors.V = {'v',vanderm}; % Create V as vanderm(v).
model.factors.W = {'w',sigmoid}; % Create W as sigmoid(w).
model.factors.LX = 'lambdaX';
model.factors.LY = 'lambdaY';
```

In this example the transformations depend on parameters. To pass along these parameters, we encapsulate them inside anonymous functions. For example, the sigmoid transformation `struct_sigmoid` requires the interval `rng` to constrain its argument in.

Now we add the two factorizations of X and Y to the model with

```
% Define the joint factorization of the matrices X and Y.
model.factorizations.xfac.data = X;
model.factorizations.xfac.cpd = {'U','V','LX'};
model.factorizations.yfac.data = Y;
model.factorizations.yfac.cpd = {'U','W','LY'};
```

Here, we use the CPD to describe the factorizations $X = U \cdot \Lambda_X \cdot V^T$ and $Y = U \cdot \Lambda_Y \cdot W^T$ by associating the factor matrices $\boldsymbol{\lambda}_X$ and $\boldsymbol{\lambda}_Y$ with the third dimension of X and Y. Alternatively, we could describe the two factorizations with the BTD model by imposing the core tensor to be a diagonal matrix with `struct_diag`, but this is less efficient than using the CPD model.

The SDF problem can now be solved with

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
options.TolFun = 1e-9; % Stop earlier.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Although the algorithm converges to a solution, it usually does not converge to the solution we used to generate the problem. This indicates that there are not enough constraints to make the solution to this SDF problem unique (in contrary to the first example).

Example 3: an orthogonal factor. We show how to compute a simple CPD in which one of the factors is constrained to have orthonormal colums. To this end, we will create a variable q that parameterizes a matrix with orthonormal columns Q. The structure `struct_orth` can then be used to transform the variable q into the factor Q (cf. Figure A.7). First, we set up the problem:

```
% Generate CPD, wherein one factor has orthonormal columns.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
U{1} = orth(U{1}); % Enforce orthonormal columns.
T = cpdgen(U);
```

The help information provided by `struct_orth` tells us that the variable q should be a vector of length $IR - 0.5R(R-1)$. To transform q into a matrix with orthonormal columns, we need to pass the size of Q to `struct_orth`. One way to do this is with an anonymous function, as shown below. Now we are ready to define and solve the SDF problem:

```
% Define model variables.
model.variables.q = randn(I*R-0.5*R*(R-1),1);
model.variables.b = randn(I,R);
model.variables.c = randn(I,R);

% Define a function which will transform q into Q.
% This anonymous function stores the size of Q as its last argument.
orthq = @(z,task)struct_orth(z,task,[I R]);

% Define model factors.
model.factors.Q = {'q',orthq}; % Create Q as orthq(q).
model.factors.B = 'b';
model.factors.C = 'c';
```

```
% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'Q','B','C'};

% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);

% Check that Q indeed has orthonormal columns
sol.factors.Q'*sol.factors.Q
```

Example 4: rank-$(L_r, L_r, 1)$ BTD. This example shows how to use the BTD model and how to use subfactors to build more complex factors in SDF models. We want to compute a rank-$(L_r, L_r, 1)$ BTD [46, 47, 208] of a tensor `T` in two terms, i.e.,

$$\mathsf{T} \approx (\mathsf{A1} \cdot \mathsf{B1}^{\mathrm{T}}) \circ \mathsf{c1} + (\mathsf{A2} \cdot \mathsf{B2}^{\mathrm{T}}) \circ \mathsf{c2}$$

where `c1` and `c2` are vectors and `size(A1,2)` equals `size(B1,2)` equals L_1 and analogously for L_2.

Example 4a: modelling a rank-$(L_r, L_r, 1)$ BTD with a BTD. We could compute this decomposition by formulating it as a more general BTD

$$\begin{aligned}\mathsf{T} &\approx (\mathsf{A1} \cdot \mathsf{S1} \cdot \mathsf{B1}^{\mathrm{T}}) \circ \mathsf{c1} + (\mathsf{A2} \cdot \mathsf{S2} \cdot \mathsf{B2}^{\mathrm{T}}) \circ \mathsf{c2} \\ &= \mathsf{S1} \bullet_1 \mathsf{A1} \bullet_2 \mathsf{B1} \bullet_3 \mathsf{c1} + \mathsf{S2} \bullet_1 \mathsf{A2} \bullet_2 \mathsf{B2} \bullet_3 \mathsf{c2}\end{aligned}$$

where we have introduced the core tensors `S1` and `S2`. First we create the tensor `T` with

```
% Generate rank-(Lr,Lr,1) BTD.
size_tens = [10 10 10];
L1 = 2;
L2 = 2;
U = btd_rnd(size_tens,{[L1 L1 1],[L2 L2 1]});
T = btdgen(U);
```

then we define the model as

```
% Define rank-(Lr,Lr,1) BTD model using the BTD.
model.variables.A1 = randn(size_tens(1),L1);
model.variables.B1 = randn(size_tens(2),L1);
model.variables.c1 = randn(size_tens(3),1);
model.variables.S1 = randn(L1,L1,1);
model.variables.A2 = randn(size_tens(1),L2);
model.variables.B2 = randn(size_tens(2),L2);
```

```
model.variables.c2 = randn(size_tens(3),1);
model.variables.S2 = randn(L2,L2,1);
model.factors = { 'A1','B1','c1','S1', 'A2','B2','c2','S2' };
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.btd = {{1,2,3,4},{5,6,7,8}};
```

where, for the sake of brevity, we have used the index-notation for the `'factors'` field. Notice that the BTD is specified in the same format as for other BTD algorithms, except in place of factor matrices and core tensors, there are references to factors. The BTD can then be computed by

```
% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 4b: modelling a rank-$(L_r, L_r, 1)$ BTD with a CPD. A different and more efficient approach to compute the rank-$(L_r, L_r, 1)$ BTD would be to cast it as a CPD as in

$$\begin{aligned}\mathsf{T} &\approx (\mathsf{A1} \cdot \mathsf{B1}^{\mathrm{T}}) \circ \mathsf{c1} + (\mathsf{A2} \cdot \mathsf{B2}^{\mathrm{T}}) \circ \mathsf{c2} \\ &= \sum_{j=1}^{\mathsf{L1}} \mathsf{A1}_{:j} \circ \mathsf{B1}_{:j} \circ \mathsf{c1} + \sum_{j=1}^{\mathsf{L2}} \mathsf{A2}_{:j} \circ \mathsf{B2}_{:j} \circ \mathsf{c2}.\end{aligned}$$

Here, the CPD's third factor matrix looks like `[repmat(c1,1,L1) repmat(c2,1,L2)]`, while the first two would be `[A1 A2]` and `[B1 B2]`, respectively. These factor matrices are all built out of subfactor matrices, which we can implement in the DSL as follows:

```
% Define rank-(Lr,Lr,1) BTD model using the CPD.
model.variables.A1 = randn(size_tens(1),L1);
model.variables.B1 = randn(size_tens(2),L1);
model.variables.c1 = randn(size_tens(3),1);
model.variables.A2 = randn(size_tens(1),L2);
model.variables.B2 = randn(size_tens(2),L2);
model.variables.c2 = randn(size_tens(3),1);
% The factors A, B and C are built out of subfactors.
% Structure may also be imposed on subfactors if desired.
model.factors.A = { {'A1'},{'A2'} };
model.factors.B = { {'B1'},{'B2'} };
model.factors.C = { {'c1'},{'c1'},{'c2'},{'c2'} };
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.cpd = {'A','B','C'};
```

Let's deconstruct the meaning of this syntax: `model.factors.A = { {'A1'},{ 'A2'} }` says that the factor `A` is a matrix consisting of two subfactors placed horizontally next to each other. Subfactors may also be placed vertically by using a semicolon `;` instead of a comma `,` to separate them. The SDF algorithm will first build the subfactors, which are in this case simply references to the variables `A1` and `A2`, and then call `cell2mat` on the cell array of subfactors to create the factor `A`. Like factors, structure may also be imposed on subfactors. For instance, we could have written `model.factors.A = { {'A1',@struct_sigmoid},{'A2'} }` to constrain the elements of the first submatrix in the factor `A` to the interval $(-1, 1)$ (the elements of the variable `A1` itself are not restricted to this interval however).

Because computing the rank-$(L_r, L_r, 1)$ BTD is a relatively common task, we show a final compact and efficient way to do so using the predefined structure `struct_LL1`:

```
% Define rank-(Lr,Lr,1) BTD model using the CPD (more efficient).
L = [L1 L2];
LL1 = @(z,task)struct_LL1(z,task,L);
model.variables.A = randn(size_tens(1),L1+L2);
model.variables.B = randn(size_tens(2),L1+L2);
model.variables.c = randn(size_tens(3),2);
model.factors.A = 'A';
model.factors.B = 'B';
model.factors.C = {'c',LL1};
model.factorizations.mybtd.data = T;
model.factorizations.mybtd.cpd = {'A','B','C'};
```

Example 5: constants. When a factor or subfactor is intended to be constant, then instead of referring to a variable by its name or index, use the constant array itself.

Example 5a: known factor matrix. The following example computes a CPD in which one factor matrix is known:

```
% Generate a CPD.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
T = cpdgen(U);

% Model the CPD of T, where the second half of U{3} is known.
model.variables.a = randn(size(U{1}));
model.variables.b = randn(size(U{1}));
model.variables.c = randn(size(U{1},1),2);
model.factors.A = 'a';
```

```
model.factors.B = 'b';
model.factors.C = U{3}; % The third factor is constant.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF model.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 5b: partially known factor matrix. To create a factor of which some of the columns are known, simply define the factor to consist of a number of subfactors where one of the subfactors is a numeric array. The following example computes a CPD in which part of one factor matrix is known:

```
% Generate a CPD.
I = 15;
R = 4;
U = cpd_rnd([I I I],R);
T = cpdgen(U);

% Model the CPD of T, where the second half of U{3} is known.
model.variables.a = randn(size(U{1}));
model.variables.b = randn(size(U{1}));
model.variables.c = randn(size(U{1},1),2);
model.factors.A = 'a';
model.factors.B = 'b';
% The third factor is partially known.
model.factors.C = {'c',U{3}(:,1:2)};
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF model.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
sol.variables
sol.factors
```

Example 6: chaining factor structures. Factor structures can be chained so that a variable is transformed by a sequence of functions. In the following example, we compute a CPD in which a subfactor of the first factor matrix is a nonnegative Toeplitz matrix and the last factor is known. To create a nonnegative Toeplitz subfactor, we will create a generator vector for the Toeplitz matrix, transform it with `struct_nonneg` so that it is nonnegative, and then transform it with `struct_toeplitz` to create a nonnegative Toeplitz

subfactor. The factor A is defined as a matrix consisting of two subfactors: the top subfactor transforms the variable `atop` with `struct_nonneg`, followed by `struct_toeplitz`, and the bottom subfactor is simply `abtm`.

```
% Generate CPD, wherein
% - the first factor has a nonnegative Toeplitz subfactor.
% - the last factor is known.
I = 20;
R = 4;
U = cpd_rnd([I I I],R);
U{1}(1:R,1:R) = toeplitz(rand(4,1));
T = cpdgen(U);

% Define model variables.
model.variables.atop = randn(2*R-1,1);
model.variables.abtm = randn(I-R,R);
model.variables.b = randn(I,R);

% Define model factors.
% The first factor vertically concatenates two subfactors.
% The top subfactor is generated as a nonnegative Toeplitz matrix,
% i.e., A = [ struct_toeplitz(struct_nonneg(atop)) ; abtm ].
model.factors.A = {{'atop',@struct_nonneg,@struct_toeplitz}; ...
                   {'abtm'}};
model.factors.B = 'b';
% Third factor matrix is known.
model.factors.C = U{3};

% Define model factorizations.
model.factorizations.myfac.data = T;
model.factorizations.myfac.cpd = {'A','B','C'};

% Solve the SDF problem.
options.Display = 5; % View convergence progress every 5 iterations.
sol = sdf_nls(model,options);
```

Example 7: regularization. Next to the CPD and BTD models, SDF also includes two models which represent L1- and L2-regularization terms. By including the factorizations

```
% Use L2-regularization on the factors A and B.
model.factorizations.myreg2.data = {zeros(size(U{1})),zeros(size(U{2}))
    };
model.factorizations.myreg2.regL2 = {'A','B'};

% Use L1-regularization on C-ones(size(C)).
```

```
model.factorizations.myreg1.data = ones(size(U{3}));
model.factorizations.myreg1.regL1 = {'C'};
```

to the previous example, the terms $\frac{1}{2}\|\begin{bmatrix}\mathrm{vec}(A)^{\mathrm{T}} & \mathrm{vec}(B)^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}} - \mathbf{0}\|_2^2$ and $\frac{1}{2}\|\mathrm{vec}(C) - \mathbf{1}\|_1$ are added to the objective function. The `data` field defaults to all zeros, if omitted.

6.2 Implementing a new factor structure

Function signature To add your own structure to impose on factors in an SDF model, create a function with the following function signature

```
function [x,state] = struct_mystruct(z,task)
end
```

Function evaluation This function will be called in three different ways by SDF algorithms. The first is simply evaluating the structure, given the variable `z`:

```
[x,state] = struct_mystruct(z)
```

From here on, we will assume the transformation maps `z` to `x = sqrt(z)`. Thus, an early implementation could look like

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task), x = sqrt(z); state = []; end
end
```

Right Jacobian-vector product Once the function has been evaluated at a certain `z`, it will be called several times to evaluate the Jacobian-vector product

$$\frac{\mathrm{d}\,\mathrm{vec}(\sqrt{z})}{\mathrm{d}\,\mathrm{vec}(z)^{\mathrm{T}}} \cdot \boldsymbol{r}$$

at `z`, given a vector $\boldsymbol{r}$ stored in `task.r`. The latter will not actually be stored as a vector, but rather in the same format as the input variable `z`, be it an array or (nested) cell array of arrays. Likewise, the result of the right Jacobian-vector product `x` should be in the same format as the output of the evaluation `struct_mystruct(z)`. Moreover, the second output `state` from the function evaluation stage can be used to store computations in, which will then be made available as fields in `task`. Taking this into account, our transformation could look like

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task)
    x = sqrt(z);
    state.dz = 1./(2*x);
elseif ~isempty(task.r)
    % Here, task.dz is equal to 1/(2*sqrt(z)), which we stored earlier.
    x = task.dz.*task.r;
    state = [];
end
end
```

Left Jacobian-vector product Finally, the function will also be called to evaluate the left Jacobian-vector product

$$\left(\frac{\partial \operatorname{vec}(\sqrt{z})}{\partial \operatorname{vec}(z)^{\mathrm{T}}}\right)^{\mathrm{H}} \cdot \ell + \overline{\left(\frac{\partial \operatorname{vec}(\sqrt{z})}{\partial \operatorname{vec}(\overline{z})^{\mathrm{T}}}\right)^{\mathrm{H}} \cdot \ell}$$

where the partial derivative with respect to z ($\overline{z}$) treats $\overline{z}$ (z) as constant (cf. Section 7.1) and the vector ℓ is stored in `task.l`. The output `x` should be of the same format as the input `z`. Here, the second term in the left Jacobian-vector product is zero, and the full implementation of the structure becomes

```
function [x,state] = struct_mystruct(z,task)
if nargin < 2 || isempty(task)
    x = sqrt(z);
    state.dz = 1./(2*x);
elseif ~isempty(task.r)
    x = task.dz.*task.r;
    state = [];
elseif ~isempty(task.l)
    x = conj(task.dz).*task.l;
    state = [];
end
end
```

Remark Note that currently the `sdf_nls` family of algorithms does not accept nonanalytic structures, i.e., structures for which the second term in the left Jacobian-vector product is nonzero. The `sdf_minf` family does support nonanalytic structures, however. For example, `sdf_nls` currently does not support a structure of the form $z * \overline{z}$, but does support the structure $z * z$. Since `sdf_minf` does not use the right Jacobian-vector product, a partial workaround is to implement the structure $z * z$ for the right Jacobian-vector product so that `sdf_nls` can still use this structure if z is real-valued. At the same time, the left Jacobian-vector product can be based on the original structure $z * \overline{z}$ so that `sdf_minf` can be used for both real and complex z.

7 Complex optimization

Optimization problems An integral part of Tensorlab comprises optimization of real functions in complex variables [206]. Tensorlab offers algorithms for complex optimization that solve unconstrained nonlinear optimization problems of the form

$$\underset{\boldsymbol{z} \in \mathbb{C}^n}{\text{minimize}} \quad f(\boldsymbol{z}, \overline{\boldsymbol{z}}), \tag{minf}$$

where $f : \mathbb{C}^n \to \mathbb{R}$ is a smooth function (cf. `minf_lbfgs`, `minf_lbfgsdl`, `minf_ncg` and `minf_sr1cgs`) and nonlinear least squares problems of the form

$$\underset{\boldsymbol{z} \in \mathbb{C}^n}{\text{minimize}} \quad \frac{1}{2}\|\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})\|_F^2, \tag{nls}$$

where $\|\cdot\|_F$ is the Frobenius norm and $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$ is a smooth function that maps n complex variables to $\prod I_n$ complex residuals (cf. `nls_gndl` , `nls_gncgs` and `nls_lm`). For nonlinear least squares problems, simple bound constraints of the form

$$\begin{aligned} \underset{\boldsymbol{z} \in \mathbb{C}^n}{\text{minimize}} \quad & \frac{1}{2}\|\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})\|_F^2 \\ \text{subject to} \quad & \text{Re}\{\boldsymbol{l}\} \leq \text{Re}\{\boldsymbol{z}\} \leq \text{Re}\{\boldsymbol{u}\} \\ & \text{Im}\{\boldsymbol{l}\} \leq \text{Im}\{\boldsymbol{z}\} \leq \text{Im}\{\boldsymbol{u}\} \end{aligned} \tag{nlsb}$$

are also supported (cf. `nlsb_gndl`). Furthermore, when a real solution $\boldsymbol{z} \in \mathbb{R}^n$ is sought, complex optimization reduces to real optimization and the algorithms are computationally equivalent to their real counterparts.

Prototypical example Throughout this section, we will use the Lyapunov equation

$$A \cdot X + X \cdot A^{\mathrm{H}} + Q = 0,$$

which has important applications in control theory and model order reduction, as a prototypical example. In this matrix equation, the matrices $A, Q \in \mathbb{C}^{n \times n}$ are given and the objective is to compute the matrix $X \in \mathbb{C}^{n \times n}$. Since the equation is linear in X, there exist direct methods to compute X. However, these are relatively expensive, requiring $O(n^6)$ floating point operations (flop) to compute the solution. Instead, we will focus on a nonlinear extension of this equation to low-rank solutions X, which enables us to solve large-scale Lyapunov equations.

From here on, X is represented as the matrix product $U \cdot V$, where $U \in \mathbb{C}^{n \times k}$, $V \in \mathbb{C}^{k \times n}$ $(k < n)$. In the framework of (minf) and (nls), we define the objective function and residual function as

$$f_{\mathrm{lyap}}(U, V) := \frac{1}{2} \|\mathcal{F}_{\mathrm{lyap}}(U, V)\|_F^2 \tag{f-lyap}$$

and

$$\mathcal{F}_{\mathrm{lyap}}(U, V) := A \cdot (U \cdot V) + (U \cdot V) \cdot A^{\mathrm{H}} + Q, \tag{F-lyap}$$

respectively.

Remark Please note that this example serves mainly as an illustration and that computing a good low-rank solution to a Lyapunov equation proves to be quite difficult in practice due to an increasingly large amount of local minima as k increases.

7.1 Complex derivatives

Pen & paper differentiation

Scalar functions To solve optimization problems of the form (minf), many algorithms require first-order derivatives of the real-valued objective function f. For a function of real variables $f_R : \mathbb{R}^n \to \mathbb{R}$, these derivatives can be captured in the gradient $\frac{\partial f_R}{\partial \boldsymbol{x}}$. For example, let

$$f_R(\boldsymbol{x}) := \sin(\boldsymbol{x}^{\mathrm{T}} \boldsymbol{x} + 2\boldsymbol{x}^{\mathrm{T}} \boldsymbol{a})$$

for $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$, then its gradient is given by

$$\frac{df_R}{d\boldsymbol{x}} = \cos(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x} + 2\boldsymbol{x}^{\mathrm{T}}\boldsymbol{a}) \cdot (2\boldsymbol{x} + 2\boldsymbol{a}).$$

Things get more interesting for real-valued functions of complex variables. Let

$$f(\boldsymbol{z}) := \sin(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^{\mathrm{T}}\boldsymbol{a}),$$

where $\boldsymbol{z} \in \mathbb{C}^n$ and an overline denotes the complex conjugate of its argument. It is clear that $f(\boldsymbol{x}) \equiv f_R(\boldsymbol{x})$ for $\boldsymbol{x} \in \mathbb{R}^n$ and hence f is a generalization of f_R to the complex plane. Because f is now a function of both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$, the limit $\lim_{\boldsymbol{h}\to\boldsymbol{0}} \frac{f(\boldsymbol{z}+\boldsymbol{h})-f(\boldsymbol{z})}{\boldsymbol{h}}$ no longer exists in general and so it would seem a complex gradient does not exist either. In fact, this only tells us the function f is not analytic in $\boldsymbol{z}$, i.e., its Taylor series in $\boldsymbol{z}$ alone does not exist. However, it can be shown that f is analytic in $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$ as a whole, meaning f has a Taylor series in the variables $\boldsymbol{z}_C := \begin{bmatrix}\boldsymbol{z}^{\mathrm{T}} & \overline{\boldsymbol{z}}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$ with a complex gradient

$$\frac{df}{d\boldsymbol{z}_C} = \begin{bmatrix}\dfrac{\partial f}{\partial \boldsymbol{z}} \\ \dfrac{\partial f}{\partial \overline{\boldsymbol{z}}}\end{bmatrix},$$

where $\frac{\partial f}{\partial \boldsymbol{z}}$ and $\frac{\partial f}{\partial \overline{\boldsymbol{z}}}$ are the cogradient and conjugate cogradient, respectively. The (conjugate) cogradient is a Wirtinger derivative and is to be interpreted as a partial derivative of f with respect to the variables $\boldsymbol{z}$ ($\overline{\boldsymbol{z}}$), while treating the variables $\overline{\boldsymbol{z}}$ ($\boldsymbol{z}$) as constant. For the example above, we have

$$\frac{\partial f}{\partial \boldsymbol{z}} = \cos(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^{\mathrm{T}}\boldsymbol{a}) \cdot (\overline{\boldsymbol{z}} + \boldsymbol{a})$$

$$\frac{\partial f}{\partial \overline{\boldsymbol{z}}} = \cos(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z} + (\overline{\boldsymbol{z}} + \boldsymbol{z})^{\mathrm{T}}\boldsymbol{a}) \cdot (\boldsymbol{z} + \boldsymbol{a}).$$

First, we notice that $\overline{\frac{\partial f}{\partial \boldsymbol{z}}} = \frac{\partial f}{\partial \overline{\boldsymbol{z}}}$, which holds for any real-valued function $f(\boldsymbol{z}, \overline{\boldsymbol{z}})$. A consequence is that any algorithm that optimizes f will only need one of the two cogradients, since the other is just its complex conjugate. Second, we notice that the cogradients evaluated in real variables $\boldsymbol{z} \in \mathbb{R}^n$ are equal to the real gradient $\frac{df_R}{d\boldsymbol{x}}$ up to a factor 2. Taking these two observations into account, the unconstrained nonlinear optimization algorithms in Tensorlab require only the *scaled conjugate cogradient*

$$\boldsymbol{g}(\boldsymbol{z}) := 2\frac{\partial f}{\partial \overline{\boldsymbol{z}}} \equiv 2\overline{\frac{\partial f}{\partial \boldsymbol{z}}}$$

and can optimize f over both $\boldsymbol{z} \in \mathbb{C}^n$ and $\boldsymbol{z} \in \mathbb{R}^n$.

Vector-valued functions To solve optimization problems of the form (nls), first-order derivatives of the vector-valued, or more generally tensor-valued, residual function $\mathcal{F}$ are often required. For a tensor-valued function $\mathcal{F} : \mathbb{R}^n \to \mathbb{R}^{I_1 \times \cdots \times I_N}$, these derivatives can be captured in the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{x}^{\mathrm{T}}}$. For example, let

$$\mathcal{F}_R(\boldsymbol{x}) := \begin{bmatrix} \sin(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) & \boldsymbol{x}^{\mathrm{T}}\boldsymbol{b} \\ \boldsymbol{x}^{\mathrm{T}}\boldsymbol{a} & \cos(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) \end{bmatrix}$$

for $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^n$, then its Jacobian is given by

$$\frac{d \operatorname{vec}(\mathcal{F}_R)}{d\boldsymbol{x}^{\mathrm{T}}} = \begin{bmatrix} \cos(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) \cdot (2\boldsymbol{x}^{\mathrm{T}}) \\ \boldsymbol{a}^{\mathrm{T}} \\ \boldsymbol{b}^{\mathrm{T}} \\ -\sin(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) \cdot (2\boldsymbol{x}^{\mathrm{T}}) \end{bmatrix}.$$

But what happens when we allow $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$? For example,

$$\mathcal{F}(\boldsymbol{z}) := \begin{bmatrix} \sin(\boldsymbol{z}^{\mathrm{T}}\boldsymbol{z}) & \boldsymbol{z}^{\mathrm{T}}\boldsymbol{b} \\ \overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{a} & \cos(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z}) \end{bmatrix},$$

where $\boldsymbol{z} \in \mathbb{C}^n$ could be a generalization of $\mathcal{F}_R$ to the complex plane. Following a similar reasoning as for scalar functions f, we can define a *Jacobian* and *conjugate Jacobian* as $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$, respectively. For the example above, we have

$$\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}} = \begin{bmatrix} \cos(\boldsymbol{z}^{\mathrm{T}}\boldsymbol{z}) \cdot (2\boldsymbol{z}^{\mathrm{T}}) \\ \boldsymbol{0}^{\mathrm{T}} \\ \boldsymbol{b}^{\mathrm{T}} \\ -\sin(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z}) \cdot \overline{\boldsymbol{z}}^{\mathrm{T}} \end{bmatrix} \quad \text{and} \quad \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}} = \begin{bmatrix} \boldsymbol{0}^{\mathrm{T}} \\ \boldsymbol{a}^{\mathrm{T}} \\ \boldsymbol{0}^{\mathrm{T}} \\ -\sin(\overline{\boldsymbol{z}}^{\mathrm{T}}\boldsymbol{z}) \cdot \boldsymbol{z}^{\mathrm{T}} \end{bmatrix}.$$

Because $\mathcal{F}$ maps to the complex numbers, it is no longer true that the conjugate Jacobian is the complex conjugate of the Jacobian. In general, algorithms that solve (nls) require both the Jacobian and conjugate Jacobian. In some cases only one of the two Jacobians is required, e.g., when $\mathcal{F}$ is analytic in $\boldsymbol{z}$, which implies $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}} \equiv 0$. Tensorlab offers nonlinear least squares solvers for both the general nonanalytic case and the latter analytic case.

Numerical differentiation

Real scalar functions (with the i-trick) The real gradient can be numerically approximated with `deriv` using the so-called i-trick [216]. For example, define the scalar functions

$$f_1(x) := \frac{10^{-20}}{1 - 10^3 x} \qquad f_2(\boldsymbol{x}) := \sin(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{a})^3 \qquad f_3(X, Y) := \arctan(\operatorname{trace}(X^{\mathrm{T}} \cdot Y)),$$

where $x \in \mathbb{R}$, $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$ and $X, Y \in \mathbb{R}^{n \times n}$. Their first-order derivatives are

$$\frac{df_1}{dx} = \frac{10^{-17}}{(1 - 10^3 x)^2}$$

$$\frac{df_2}{d\boldsymbol{x}} = 3\sin(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{a})^2 \cos(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{a}) \cdot \boldsymbol{a}$$

$$\begin{cases} \dfrac{\partial f_3}{\partial X} &= \frac{1}{1+\mathrm{trace}(X^{\mathrm{T}} \cdot Y)^2} \cdot Y \\ \dfrac{\partial f_3}{\partial Y} &= \frac{1}{1+\mathrm{trace}(X^{\mathrm{T}} \cdot Y)^2} \cdot X \end{cases}.$$

An advantage of using the i-trick is that it can compute first-order derivatives accurately up to the order of machine precision. The disadvantages are that this requires an equivalent of about 4 (real) function evaluations per variable (compared to 2 for finite differences) and that certain requirements must be met. First, only the real gradient can be computed, meaning the gradient can only be computed where the variables are real. Second, the function must be real-valued when evaluated in real variables. Third, the function must be analytic on the complex plane. In other words, the function may not be a function of the complex conjugate of its argument. For example, the i-trick can be used to compute the gradient of the function `@(x)x.'*x`, but not of the function `@(x)x'*x` because the latter depends on $\overline{\boldsymbol{x}}$. As a last example, note that `@(x)real(x)` is not analytic in $x \in \mathbb{C}$ because it can be written as `@(x)(x+conj(x))/2`.

Choosing $\boldsymbol{a}$ as `ones(size(x))` in the example functions above, the following example uses `deriv` to compute the real gradient of these functions using the i-trick:

```
% Three test functions.
f1 = @(x)1e-20/(1-1e3*x);
f2 = @(x)sin(x.'*ones(size(x)))^3;
f3 = @(XY)atan(trace(XY{1}.'*XY{2}));

% Their first-order derivatives.
g1 = @(x)1e-17/(1-1e3*x)^2;
g2 = @(x)3*sin(x.'*ones(size(x)))^2*cos(x.'*ones(size(x)))*ones(size(x))
    ;
g3 = @(XY){1/(1+trace(XY{1}.'*XY{2})^2)*XY{2}, ...
           1/(1+trace(XY{1}.'*XY{2})^2)*XY{1}};

% Approximate the real gradient with the i-trick and compute its
    relative error.
x = randn;
```

```
relerr1 = abs(g1(x)-deriv(f1,x))/abs(g1(x))
x = randn(10,1);
relerr2 = norm(g2(x)-deriv(f2,x))/norm(g2(x))
XY = {randn(10),randn(10)};
relerr3 = cellfun(@(a,b)frob(a-b)/frob(a),g3(XY),deriv(f3,XY))
```

In Tensorlab, derivatives of scalar functions are returned in the same format as the function's argument. Notice that `f3` is function of a cell array `XY`, containing the matrix X in `XY{1}` and the matrix Y in `XY{2}`. In similar vein, the output of `deriv(f3,XY)` is a cell array containing the matrices $\frac{\partial f_3}{\partial X}$ and $\frac{\partial f_3}{\partial Y}$. Often, this allows the user to conveniently write functions as a function of a cell array of variables (containing vectors, matrices or tensors) instead of coercing all variables into one long vector which must then be disassembled in the respective variables.

Scalar functions (with finite differences) If the conditions for the i-trick are not satisfied, or if a scaled conjugate cogradient is required, an alternative is using finite differences to approximate first-order derivatives. In both cases, the finite difference approximation can be computed using `deriv(f,x,[],'gradient')`. As a first example, we compute the relative error of the finite difference approximation of the real gradient of f_1, f_2 and f_3:

```
% Approximate the real gradient with finite differences and compute its
    relative error.
x = randn;
relerr1 = abs(g1(x)-deriv(f1,x,[],'gradient'))/abs(g1(x))
x = randn(10,1);
relerr2 = norm(g2(x)-deriv(f2,x,[],'gradient'))/norm(g2(x))
XY = {randn(10),randn(10)};
relerr3 = cellfun(@(a,b)frob(a-b)/frob(a),g3(XY),deriv(f3,XY,[],'
    gradient'))
```

If $f(\boldsymbol{z})$ is a real-valued scalar function of complex variables, `deriv` can compute its scaled conjugate cogradient $\boldsymbol{g}(\boldsymbol{z})$. For example, let

$$f(\boldsymbol{z}) := \boldsymbol{a}^{\mathrm{T}}(\boldsymbol{z}+\overline{\boldsymbol{z}}) + \log(\boldsymbol{z}^{\mathrm{H}}\boldsymbol{z}) \qquad \boldsymbol{g}(\boldsymbol{z}) := 2\frac{\partial f}{\partial \overline{\boldsymbol{z}}} \equiv 2\overline{\frac{\partial f}{\partial \boldsymbol{z}}} = 2\cdot\boldsymbol{a} + \frac{2}{\boldsymbol{z}^{\mathrm{H}}\boldsymbol{z}}\cdot\boldsymbol{z},$$

where $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{z} \in \mathbb{C}^n$. Since f is real-valued and $\boldsymbol{z}$ is complex, calling `deriv(f,z)` is equivalent to `deriv(f,z,[],'gradient')` and uses finite differences to approximate the scaled conjugate cogradient. In the following example $\boldsymbol{a}$ is chosen as `ones(size(z))` and the relative error of the finite difference approximation of the scaled conjugate cogradient $\boldsymbol{g}(\boldsymbol{z})$ is computed:

```
% Approximate the scaled conjugate cogradient with finite differences
```

```
% and compute its relative error.
f = @(z)ones(size(z)).'*(z+conj(z))+log(z'*z);
g = @(z)2*ones(size(z))+2/(z'*z)*z;
z = randn(10,1)+randn(10,1)*1i;
relerr = norm(g(z)-deriv(f,z))/norm(g(z))
```

Remark In case of doubt, use `deriv(f,z,[],'gradient')` to compute the scaled conjugate cogradient. Running `deriv(f,z)` will attempt to use the i-trick when `z` is real, which can be substantially more accurate, but should only be applied when `f` is analytic.

Real vector-valued functions (with the i-trick) Analogously to scalar functions, the real Jacobian of tensor-valued functions can also be numerically approximated with `deriv` using the i-trick. Take for example the following matrix-valued function

$$\mathcal{F}_1(\boldsymbol{x}) := \begin{bmatrix} \log(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) & \mathbf{0} \\ 2\boldsymbol{a}^{\mathrm{T}}\boldsymbol{x} & \sin(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) \end{bmatrix},$$

where $\boldsymbol{a}, \boldsymbol{x} \in \mathbb{R}^n$, and its real Jacobian

$$J_1(\boldsymbol{x}) := \frac{d\,\mathrm{vec}(\mathcal{F}_1)}{d\boldsymbol{x}^{\mathrm{T}}} = 2\begin{bmatrix} \frac{1}{\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}} \cdot \boldsymbol{x}^{\mathrm{T}} \\ \boldsymbol{a}^{\mathrm{T}} \\ \mathbf{0} \\ \cos(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{x}) \cdot \boldsymbol{x}^{\mathrm{T}} \end{bmatrix}.$$

Set $\boldsymbol{a}$ equal to `ones(size(x))`. Since each entry in $\mathcal{F}_1(\boldsymbol{x})$ is real-valued and not a function of $\overline{\boldsymbol{x}}$, we can approximate the real Jacobian $J_1(\boldsymbol{x})$ with the i-trick:

```
% Approximate the Jacobian of a tensor-valued function with the i-trick
% and compute its relative error.
F1 = @(x)[log(x.'*x) 0; 2*ones(size(x)).'*x sin(x.'*x)];
J1 = @(x)2*[1/(x.'*x)*x.'; ones(size(x)).'; zeros(size(x)).'; cos(x.'*x)
    *x.'];
x = randn(10,1);
relerr1 = frob(J1(x)-deriv(F1,x))/frob(J1(x))
```

Analytic vector-valued functions (with finite differences) The Jacobian of an analytic tensor-valued function $\mathcal{F}(\boldsymbol{z})$ can be approximated with finite differences by calling `deriv(F,z,[],'Jacobian')`. Functions that are analytic when their argument is real, may no longer be analytic when their argument is complex. For example, $\mathcal{F}(\boldsymbol{z}) := \begin{bmatrix} \boldsymbol{z}^{\mathrm{H}}\boldsymbol{z} & \mathrm{Re}\{\boldsymbol{z}\}^{\mathrm{T}}\boldsymbol{z} \end{bmatrix}$ is not analytic in $\boldsymbol{z} \in \mathbb{C}^n$ because it

depends on $\overline{\boldsymbol{z}}$, but is analytic when $\boldsymbol{z} \in \mathbb{R}^n$. An example of a function that is analytic for both real and complex $\boldsymbol{z}$ is the function $\mathcal{F}_1(\boldsymbol{z})$. The following two examples compute the relative error of the finite differences approximation of the Jacobian $J_1(\boldsymbol{x})$ in a real vector $\boldsymbol{x}$:

```
% Approximate the Jacobian of an analytic tensor-valued function
% with finite differences and compute its relative error.
x = randn(10,1);
relerr1 = frob(J1(x)-deriv(F1,x,[],'Jacobian'))/frob(J1(x))
```

and the relative error of the finite differences approximation of $J_1(\boldsymbol{z})$ in a complex vector $\boldsymbol{z}$:

```
% Approximate the Jacobian of an analytic tensor-valued function
% with finite differences and compute its relative error.
z = randn(10,1)+randn(10,1)*1i;
relerr1 = frob(J1(z)-deriv(F1,z,[],'Jacobian'))/frob(J1(z))
```

Nonanalytic vector-valued functions (with finite differences) In general, a tensor-valued function may be function of its argument and its complex conjugate. The matrix-valued function

$$\mathcal{F}_2(X,Y) := \begin{bmatrix} \log(\operatorname{trace}(X^{\mathrm{H}} \cdot Y)) & \mathbf{0} \\ \boldsymbol{a}^{\mathrm{T}}(X+\overline{X})\boldsymbol{a} & \boldsymbol{a}^{\mathrm{T}}(Y-\overline{Y})\boldsymbol{a} \end{bmatrix},$$

where $\boldsymbol{a} \in \mathbb{R}^n$ and $X, Y \in \mathbb{C}^{n\times n}$ is an example of such a nonanalytic function because it depends on X, Y and $\overline{X}$, $\overline{Y}$. Its Jacobian and conjugate Jacobian are given by

$$J_2(X,Y) := \frac{\partial \operatorname{vec}(\mathcal{F}_2)}{\partial \begin{bmatrix} \operatorname{vec}(X)^{\mathrm{T}} & \operatorname{vec}(Y)^{\mathrm{T}} \end{bmatrix}} = \begin{bmatrix} 0 & \frac{1}{\operatorname{trace}(X^{\mathrm{H}}\cdot Y)} \cdot \operatorname{vec}(\overline{X})^{\mathrm{T}} \\ (\boldsymbol{a}\otimes\boldsymbol{a})^{\mathrm{T}} & 0 \\ 0 & 0 \\ 0 & (\boldsymbol{a}\otimes\boldsymbol{a})^{\mathrm{T}} \end{bmatrix}$$

$$J_2^c(X,Y) := \frac{\partial \operatorname{vec}(\mathcal{F}_2)}{\partial \begin{bmatrix} \operatorname{vec}(\overline{X})^{\mathrm{T}} & \operatorname{vec}(\overline{Y})^{\mathrm{T}} \end{bmatrix}} = \begin{bmatrix} \frac{1}{\operatorname{trace}(X^{\mathrm{H}}\cdot Y)} \cdot \operatorname{vec}(Y)^{\mathrm{T}} & 0 \\ (\boldsymbol{a}\otimes\boldsymbol{a})^{\mathrm{T}} & 0 \\ 0 & 0 \\ 0 & -(\boldsymbol{a}\otimes\boldsymbol{a})^{\mathrm{T}} \end{bmatrix},$$

respectively. For a nonanalytic tensor-valued function $\mathcal{F}(\boldsymbol{z})$, `deriv(F,z,[], 'Jacobian-C')` computes a finite differences approximation of the *complex Jacobian*

$$\begin{bmatrix} \dfrac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}} & \dfrac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}} \end{bmatrix},$$

comprising both the Jacobian and conjugate Jacobian. The complex Jacobian of $\mathcal{F}_2(X, Y)$ is the matrix $\begin{bmatrix} J_2(X,Y) & J_2^c(X,Y) \end{bmatrix}$. In the following example $\boldsymbol{a}$ is equal to `ones(length(z{1}))` and the relative error of the complex Jacobian's finite differences approximation is computed:

```
% Approximate the complex Jacobian of a nonanalytic tensor-valued
    function
% with finite differences and compute its relative error.
F2 = @(z)[log(trace(z{1}'*z{2})) 0; ...
          sum(sum(z{1}+conj(z{1}))) sum(sum(z{2}-conj(z{2})))];
J2 = @(z)[zeros(1,numel(z{1})) 1/trace(z{1}'*z{2})*reshape(conj(z{1})
    ,1,[]) ...
          1/trace(z{1}'*z{2})*reshape(z{2},1,[]) zeros(1,numel(z{2}));
              ...
          ones(1,numel(z{1})) zeros(1,numel(z{1})) ...
          ones(1,numel(z{2})) zeros(1,numel(z{2})); ...
          zeros(1,2*numel(z{1})) zeros(1,2*numel(z{2})); ...
          zeros(1,numel(z{1})) ones(1,numel(z{1})) ...
          zeros(1,numel(z{1})) -ones(1,numel(z{1}))];
z = {randn(10)+randn(10)*1i,randn(10)+randn(10)*1i};
relerr2 = frob(J2(z)-deriv(F2,z,[],'Jacobian-C'))/frob(J2(z))
```

7.2 Nonlinear least squares

Algorithms Tensorlab offers three algorithms for unconstrained nonlinear least squares: `nls_gndl`, `nls_gncgs` and `nls_lm`. The first is Gauss–Newton with a dogleg trust region strategy, the second is Gauss–Newton with CG-Steihaug for solving the trust region subproblem and the last is Levenberg–Marquardt. A bound-constrained method, `nlsb_gndl`, is also included and is a projected Gauss–Newton method with dogleg trust region. All algorithms are applicable to both analytic and nonanalytic residual functions and offer various ways of exploiting the structure available in its (complex) Jacobian.

With numerical differentiation The complex optimization algorithms that solve (nls) require the Jacobian of the residual function $\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})$, which will be $\mathcal{F}_{\text{lyap}}(U, V)$ for the remainder of this section (cf. Section 7). The second argument `dF` of the nonlinear least squares optimization algorithms `nls_gndl`, `nls_gncgs` and `nls_lm` specifies how the Jacobian should be computed. To approximate the Jacobian with finite differences, set `dF` equal to `'Jacobian'` or `'Jacobian-C'`.

The first case, `'Jacobian'`, corresponds to approximating the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$, assuming $\mathcal{F}$ is analytic in $\boldsymbol{z}$. The second case, `'Jacobian-C'`, corresponds to approximating the complex Jacobian consisting of the Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and conjugate Jacobian $\frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$, where $\boldsymbol{z} \in \mathbb{C}^n$. Since $\mathcal{F}_{\text{lyap}}$ does not depend on $\overline{U}$ or $\overline{V}$, we may implement a nonlinear least squares solver for the low-rank solution of the Lyapunov equation as

```
function z = lyap_nls_deriv(A,Q,z0)

F = @(z)(A*z{1})*z{2}+z{1}*(z{2}*A')+Q;
z = nls_gndl(F,'Jacobian',z0);

end
```

The residual function `F(z)` is matrix-valued and its argument is a cell array `z`, consisting of the two matrices U and V. The output of the optimization algorithm, in this case `nls_gndl`, is a cell array of the same format as the argument of the residual function `F(z)`.

With the Jacobian Using the property $\operatorname{vec}(A \cdot X \cdot B) \equiv (B^{\mathrm{T}} \otimes A) \cdot \operatorname{vec}(X)$, it is easy to verify that the Jacobian of $\mathcal{F}_{\text{lyap}}(U, V)$ is given by

$$\frac{\partial \operatorname{vec}(\mathcal{F}_{\text{lyap}})}{\partial \begin{bmatrix} \operatorname{vec}(U)^{\mathrm{T}} & \operatorname{vec}(V)^{\mathrm{T}} \end{bmatrix}} = \begin{bmatrix} (V^{\mathrm{T}} \otimes A) + (\overline{A} \cdot V^{\mathrm{T}} \otimes \mathbb{I}) & (\mathbb{I} \otimes A \cdot U) + (\overline{A} \otimes U) \end{bmatrix}. \tag{J-lyap}$$

To supply the Jacobian to the optimization algorithm, set the field `dF.dz` as the function handle of the function that computes the Jacobian at a given point $\boldsymbol{z}$. For problems for which the residual function $\mathcal{F}$ depends on both $\boldsymbol{z}$ and $\overline{\boldsymbol{z}}$, the complex Jacobian can be supplied with the field `dF.dzc`. See Section 7.1 or the `help` page of the selected algorithm for more information. Applied to the Lyapunov equation, we have

```
function z = lyap_nls_J(A,Q,z0)

dF.dz = @J;
z = nls_gndl(@F,dF,z0);

    function F = F(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
    end

    function J = J(z)
```

```
        U = z{1}; V = z{2}; I = eye(size(A));
        J = [kron(V.',A)+kron(conj(A)*V.',I) kron(I,A*U)+kron(conj(A),U)
            ];
    end

end
```

With the Jacobian's Gramian When the residual function $\mathcal{F} : \mathbb{C}^n \to \mathbb{C}^{I_1 \times \cdots \times I_N}$ is analytic[1] in $\boldsymbol{z}$ (i.e., it is not a function of $\overline{\boldsymbol{z}}$) and the number of residuals $\prod I_n$ is large compared to the number of variables n, it may be beneficial to compute the Jacobian's Gramian $J^{\mathrm{H}}J$ instead of the Jacobian $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ itself. This way, each iteration of the nonlinear least squares algorithm no longer requires computing the (pseudo-)inverse $J^{\dagger}$, but rather the less expensive (pseudo-)inverse $(J^{\mathrm{H}}J)^{\dagger}$. In the case of the Lyapunov equation, this can lead to a significantly more efficient method if the rank k of the solution is small with respect to its order n. Along with the Jacobian's Gramian, the objective function $f := \frac{1}{2}\|\mathcal{F}\|_F^2$ and its gradient $\frac{\partial f}{\partial \boldsymbol{z}} \equiv J^{\mathrm{H}} \cdot \operatorname{vec}(\mathcal{F})$ are also necessary. Skipping the derivation of the gradient and Jacobian's Gramian, the implementation could look like

```
function z = lyap_nls_JHJ(A,Q,z0)

AHA = A'*A;
dF.JHJ = @JHJ;
dF.JHF = @grad;
z = nls_gndl(@f,dF,z0);

    function f = f(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
        f = 0.5*(F(:)'*F(:));
    end

    function g = grad(z)
        U = z{1}; V = z{2};
        gU = AHA*(U*(V*V'))+A'*(U*(V*A'*V'))+A'*(Q*V')+ ...
             A*(U*(V*A*V'))+U*(V*AHA*V')+Q*(A*V');
        gV = (U'*AHA*U)*V+((U'*A'*U)*V)*A'+(U'*A')*Q+ ...
             ((U'*A*U)*V)*A+((U'*U)*V)*AHA+(U'*Q)*A;
        g = {gU,gV};
    end
```

[1] In the more general case of a nonanalytic residual function, the structure in its complex Jacobian can be exploited by computing an inexact step. See the following paragraph for more details.

```
    function JHJ = JHJ(z)
        U = z{1}; V = z{2}; I = eye(size(A));
        tmpa = kron(conj(V*A'*V'),A); tmpb = kron(conj(A),U'*A'*U);
        JHJ11 = kron(conj(V*V'),AHA)+kron(conj(V*AHA*V'),I)+tmpa+tmpa';
        JHJ22 = kron(I,U'*AHA*U)+kron(conj(AHA),U'*U)+tmpb+tmpb';
        JHJ12 = kron(conj(V),AHA*U)+kron(conj(V*A),A'*U)+ ...
                kron(conj(V*A'),A*U)+kron(conj(V*AHA),U);
        JHJ = [JHJ11 JHJ12; JHJ12' JHJ22];
    end

end
```

By default, the algorithm `nls_gndl` uses the Moore–Penrose pseudoinverse of either J or $J^{\mathrm{H}}J$ to compute a new descent direction. However, if it is known that these matrices always have full rank, a more efficient least squares inverse can be computed. To do so, use the option

```
% Compute a more efficient least squares step instead of using the
    pseudoinverse.
options.JHasFullRank = true;
z = nls_gndl(@f,dF,z0,options);
```

The other nonlinear least squares algorithms `nls_gncgs` and `nls_lm` use a different approach for calculating the descent direction and do not have such an option.

With an inexact step The most computationally intensive part of most nonlinear least squares problems is computing the next descent direction, which involves inverting either the Jacobian $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ or its Gramian $J^{\mathrm{H}}J$ in the case of an analytic residual function. With iterative solvers such as preconditioned conjugate gradient (PCG) and LSQR, the descent direction can be approximated using only matrix-vector products. The resulting descent directions are said to be inexact. Many problems exhibit some structure in the Jacobian which can be exploited in its matrix-vector product, allowing for an efficient computation of an inexact step. Concretely, the user has the choice of supplying the matrix vector products $J \cdot \boldsymbol{x}$ and $J^{\mathrm{H}} \cdot \boldsymbol{y}$, or the single matrix-vector product $(J^{\mathrm{H}}J) \cdot \boldsymbol{x}$. An implementation of an inexact nonlinear least squares solver using the former method can be

```
function z = lyap_nls_Jx(A,Q,z0)

dF.dzx = @Jx;
z = nls_gndl(@F,dF,z0);
```

```
    function F = F(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
    end

    function b = Jx(z,x,transp)
        U = z{1}; V = z{2};
        switch transp
            case 'notransp' % b = J*x
                Xu = reshape(x(1:numel(U)),size(U));
                Xv = reshape(x(numel(U)+1:end),size(V));
                b = (A*Xu)*V+Xu*(V*A')+(A*U)*Xv+U*(Xv*A');
                b = b(:);
            case 'transp'   % b = J'*x
                X = reshape(x,size(A));
                Bu = A'*(X*V')+X*(A*V');
                Bv = (U'*A')*X+(U'*X)*A;
                b = [Bu(:); Bv(:)];
        end
    end

end
```

where the Kronecker structure of the Jacobian (J-lyap) is exploited by reducing the computations to matrix-matrix products. Under suitable conditions on A and Q, this implementation can achieve a complexity of $O(nk^2)$, where n is the order of the solution $X = U \cdot V$ and k is its rank.

In the case of a nonanalytic residual function $\mathcal{F}(\boldsymbol{z}, \overline{\boldsymbol{z}})$, computing an inexact step requires matrix-vector products $J \cdot \boldsymbol{x}$, $J^{\mathrm{H}} \cdot \boldsymbol{y}$, $J_c \cdot \boldsymbol{x}$ and $J_c^{\mathrm{H}} \cdot \boldsymbol{y}$, where $J := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \boldsymbol{z}^{\mathrm{T}}}$ and $J_c := \frac{\partial \operatorname{vec}(\mathcal{F})}{\partial \overline{\boldsymbol{z}}^{\mathrm{T}}}$ are the residual function's Jacobian and conjugate Jacobian, respectively. For more information on how to implement these matrix-vector products, read the `help` information of the desired (nls) solver.

Setting the options The parameters of the selected optimization algorithm can be set by supplying the method with an options structure, e.g.,

```
% Set algorithm tolerances.
options.TolFun = 1e-12;
options.TolX = 1e-6;
options.MaxIter = 100;
dF.dz = @J;
z = nls_gndl(@F,dF,z0,options);
```

Remark It is important to note that since the objective function is the square of a residual norm, the objective function tolerance `options.TolFun` can be set as small as 10^{-32} for a given machine epsilon of 10^{-16}. See the `help` information on the selected algorithm for more details.

Viewing the algorithm output The second output of the optimization algorithms returns additional information pertaining to the algorithm. For example, the algorithms keep track of the objective function value in `output.fval` and also output the circumstances under which the algorithm terminated in `output.info`. As an example, the norm of the residual function of each iterate can be plotted with

```
% Plot each iterate's objective function value.
dF.dz = @J;
[z,output] = nls_gndl(@F,dF,z0);
semilogy(0:output.iterations,sqrt(2*output.fval));
```

Since the objective function is $\frac{1}{2}\|\mathcal{F}\|_F^2$, we plot `sqrt(2*output.fval)` to see the norm $\|\mathcal{F}\|_F$. See the `help` information on the selected algorithm for more details.

7.3 Unconstrained nonlinear optimization

Algorithms Tensorlab offers three algorithms for unconstrained complex optimization: `minf_lbfgs`, `minf_lbfgsdl` and `minf_ncg`. The first two are limited-memory BFGS with Moré–Thuente line search and dogleg trust region, respectively, and the last is nonlinear conjugate gradient with Moré–Thuente line search. Instead of the supplied Moré–Thuente line search, the user may optionally supply a custom line search method. See the `help` information for details.

With numerical differentiation The complex optimization algorithms that solve (minf) require the (scaled conjugate co-)gradient of the objective function $f(\boldsymbol{z}, \overline{\boldsymbol{z}})$, which will be $f_{\text{lyap}}(\boldsymbol{z}, \overline{\boldsymbol{z}})$ for the remainder of this section (cf. Section 7). The second argument of the unconstrained nonlinear minimization algorithms `minf_lbfgs`, `minf_lbfgsdl` and `minf_ncg` specifies how the gradient should be computed. To approximate the (scaled conjugate co-)gradient with finite differences, set the second argument equal to the empty matrix `[]`. An implementation for the Lyapunov equation could look like

```
function z = lyap_minf_deriv(A,Q,z0)

f = @(z)frob((A*z{1})*z{2}+z{1}*(z{2}*A')+Q);
z = minf_lbfgs(f,[],z0);

end
```

As with the nonlinear least squares algorithms, the argument of the objective function is a cell array `z`, consisting of the two matrices U and V. Likewise, the output of the optimization algorithm, in this case `minf_lbfgs`, is a cell array of the same format as the argument of the objective function `f(z)`.

With the gradient If an expression for the (scaled conjugate co-)gradient is available, it can be supplied to the optimization algorithm in the second argument. For the Lyapunov equation, the implementation could look like

```
function z = lyap_minf_grad(A,Q,z0)

AHA = A'*A;
z = minf_lbfgs(@f,@grad,z0);

    function f = f(z)
        U = z{1}; V = z{2};
        F = (A*U)*V+U*(V*A')+Q;
        f = 0.5*(F(:)'*F(:));
    end

    function g = grad(z)
        U = z{1}; V = z{2};
        gU = AHA*(U*(V*V'))+A'*(U*(V*A'*V'))+A'*(Q*V')+ ...
             A*(U*(V*A*V'))+U*(V*AHA*V')+Q*(A*V');
        gV = (U'*AHA*U)*V+((U'*A'*U)*V)*A'+(U'*A')*Q+ ...
             ((U'*A*U)*V)*A+((U'*U)*V)*AHA+(U'*Q)*A;
        g = {gU,gV};
    end

end
```

The function `grad(z)` computes the scaled conjugate cogradient $2\frac{\partial f_{\mathrm{lyap}}}{\partial \bar{\boldsymbol{z}}}$, which coincides with the real gradient for $\boldsymbol{z} \in \mathbb{R}^n$. See Section 7.1 for more information on complex derivatives.

Remark Note that the gradient `grad(z)` is returned in the same format as the solution `z`, i.e., as a cell array containing matrices of the same size as U and V. However, the gradient may also be returned as a vector if this is more convenient for the user. In that case, the scaled conjugate cogradient should be of the format $2\frac{\partial f_{\text{lyap}}}{\partial \overline{\boldsymbol{z}}}$ where $\boldsymbol{z} := \begin{bmatrix}\text{vec}(U)^{\text{T}} & \text{vec}(V)^{\text{T}}\end{bmatrix}^{\text{T}}$.

Setting the options The parameters of the selected optimization algorithm can be set by supplying the method with an options structure, e.g.,

```
% Set algorithm tolerances.
options.TolFun = 1e-6;
options.TolX = 1e-6;
options.MaxIter = 100;
z = minf_lbfgs(@f,@grad,z0,options);
```

See the `help` information on the selected algorithm for more details.

Viewing the algorithm output The second output of the optimization algorithms returns additional information pertaining to the algorithm. For example, the algorithms keep track of the objective function value in `output.fval` and also output the circumstances under which the algorithm terminated in `output.info`. As an example, the objective function value of each iterate can be plotted with

```
% Plot each iterate's objective function value.
[z,output] = minf_lbfgs(@f,@grad,z0);
semilogy(0:output.iterations,output.fval);
```

See the `help` information on the selected algorithm for more details.

8 Global minimization of bivariate functions

Analytic bivariate polynomials Consider the problem of minimizing a bivariate polynomial

$$\underset{x,y\,\in\,\mathbb{R}}{\text{minimize}} \;\; p(x,y), \tag{bipol}$$

or more generally, a rational function

$$\underset{x,y\,\in\,\mathbb{R}}{\text{minimize}} \;\; \frac{p(x,y)}{q(x,y)}. \tag{birat}$$

Since all local minimizers (x^*, y^*) are stationary points, they are roots of the system of bivariate polynomials

$$\frac{\partial p}{\partial x}q - p\frac{\partial q}{\partial x} = 0$$

$$\frac{\partial p}{\partial y}q - p\frac{\partial q}{\partial y} = 0,$$

where $q(x, y) \equiv 1$ in the case of minimizing a bivariate polynomial. With `polysol2`, Tensorlab offers a numerically robust way of computing isolated real roots of a system of bivariate polynomials

$$\begin{aligned} f(x, y) &= 0 \\ g(x, y) &= 0 \end{aligned} \tag{bisys}$$

as the eigenvalues of a generalized eigenvalue problem [205, 210]. Stationary points of bivariate polynomials and rational functions may be computed with `polymin2` and `ratmin2`, respectively.

Polyanalytic univariate polynomials Closely related is the problem of minimizing a polyanalytic univariate polynomial

$$\underset{z \in \mathbb{C}}{\text{minimize}} \;\; p(z, \overline{z}), \tag{unipol}$$

or more generally, a rational function

$$\underset{z \in \mathbb{C}}{\text{minimize}} \;\; \frac{p(z, \overline{z})}{q(z, \overline{z})}. \tag{unirat}$$

Analogously to the analytic bivariate case, all local minimizers are roots of the system

$$\frac{\partial p}{\partial z}q - p\frac{\partial q}{\partial z} = 0$$

$$\frac{\partial p}{\partial \overline{z}}q - p\frac{\partial q}{\partial \overline{z}} = 0,$$

where the derivatives are Wirtinger derivatives (cf. Section 7.1). The method `polysol2` can also solve systems of polyanalytic polynomials

$$\begin{aligned} f(z, \overline{z}) &= 0 \\ g(z, \overline{z}) &= 0. \end{aligned} \tag{unisys}$$

In fact, given a system of bivariate polynomials (bisys), `polysol2` will first convert it to the form (unisys) before computing the roots as the eigenvalues of a (complex) generalized eigenvalue problem. Stationary points of real-valued polyanalytic polynomials and rational functions may be computed with `polymin2` and `ratmin2`, respectively.

8.1 Stationary points of polynomials and rational functions

Polynomials and rational functions In MATLAB, a polynomial $p(x)$ is represented by a row vector `p = [ad ... a2 a1 a0]` as

$$p(x) = \begin{bmatrix} a_d & \cdots & a_2 & a_1 & a_0 \end{bmatrix} \cdot \begin{bmatrix} x^d & \cdots & x^2 & x & 1 \end{bmatrix}^{\mathrm{T}}.$$

For example, the polynomial $p(x) = x^3 + 2x^2 + 3x + 4$ is represented by the row vector `p = [1 2 3 4]`. Its derivative $\frac{dp}{dx}$ can be computed with `polyder(p)` and its zeros can be computed with `roots(p)`.

The stationary points of $p(x)$, i.e., all x^* which satisfy $\frac{dp}{dx}(x^*) = 0$, are the output of `roots(polyder(p))`. However, some of these solutions may have a small imaginary part which correspond to a numerical zero. The stationary points can also be computed as roots of the polynomial's derivative with `polymin(p)`, which deals with solutions with small imaginary part and returns only real solutions.

Analogously, the stationary points of a a rational function $\frac{p(x)}{q(x)}$ are given by

```
roots(conv(polyder(p),q)-conv(p,polyder(q)))
```

where `conv(p,q)` is the convolution of the two row vectors `p` and `q` and is equivalent to computing the coefficients of the polynomial $p(x) \cdot q(x)$. As in the polynomial case, there are a few numerical issues which can be dealt with by computing the stationary points of $\frac{p(x)}{q(x)}$ with `ratmin(p,q)`.

Bivariate polynomials and rational functions In Tensorlab, a bivariate polynomial $p(x, y)$ is represented by a matrix `p` as

$$p(x,y) = \begin{bmatrix} 1 & y & \cdots & y^{d_y} \end{bmatrix} \cdot \begin{bmatrix} a_{00} & \cdots & a_{0d_x} \\ \vdots & \ddots & \vdots \\ a_{d_y 0} & \cdots & a_{d_y d_x} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ \vdots \\ x^{d_x} \end{bmatrix}.$$

For example, the six-hump camel back function [64]

$$p(x,y) = 4x^2 - 2.1x^4 + \frac{1}{3}x^6 + xy - 4y^2 + 4y^4$$

is represented by the matrix

```
p = [ 0   0   4   0  -2.1 0   1/3; ...
      0   1   0   0   0   0   0;   ...
     -4   0   0   0   0   0   0;   ...
      0   0   0   0   0   0   0;   ...
      4   0   0   0   0   0   0];
```

and can be seen in Figure A.8. The stationary points of the polynomial $p(x,y)$ can be computed as the solutions of the system $\frac{\partial p}{\partial x} = \frac{\partial p}{\partial y} = 0$ with `polymin2(p)`. To obtain a global minimum, select the solution with smallest function value using

```
[xy,v] = polymin2(p);
[vmin,idx] = min(v);
xymin = xy(idx,:);
```

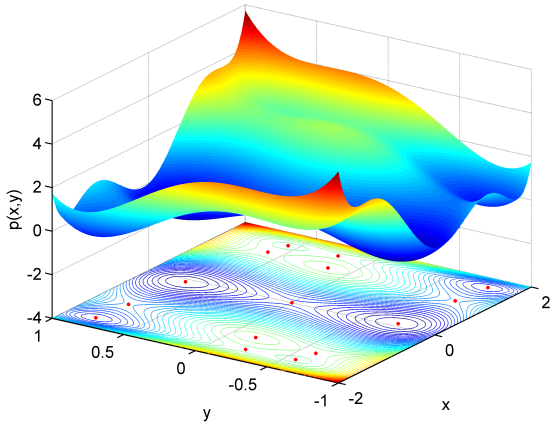


Figure A.8: The six-hump camel back function and its stationary points as red dots.

To visualize the level zero contour lines of $\frac{\partial p}{\partial x}$ and $\frac{\partial p}{\partial y}$ in the neighbourhood of the stationary points, set `options.Plot = true` as follows

```
p = randn(6); % Generate random bivariate polynomial of coordinate
    degree 5.
```

```
options.Plot = true;
xy = polymin2(p,options);
```

or inline as `polymin2(randn(6),struct('Plot',true))`.

The corresponding system of polynomials is solved by `polysol2`. The latter includes several balancing steps to improve the accuracy of the solution. For some poorly scaled problems, the method may fail to find all solutions of the system. In that case, try decreasing the `polysol2` balancing tolerance `options.TolBal` to a smaller value, e.g.,

```
options.TolBal = 1e-4;
xy = polymin2(p,options);
```

Computing the stationary points of a bivariate rational function $\frac{p(x,y)}{q(x,y)}$ is completely analogous to the polynomial case. The following example generates a random bivariate rational function and computes its stationary points:

```
p = randn(6); % Random bivariate polynomial of coordinate degree 5.
q = randn(4); % Random bivariate polynomial of coordinate degree 3.
options.Plot = true;
xy = ratmin2(p,q,options);
```

The inline equivalent of this example is `ratmin2(randn(6),randn(4),struct('Plot',true))`.

Polyanalytic polynomials and rational functions Polyanalytic polynomials $p(z,\overline{z})$ are represented by a matrix `p` as

$$p(z,\overline{z}) = \begin{bmatrix}1 & \overline{z} & \cdots & \overline{z}^{d_y}\end{bmatrix} \cdot \begin{bmatrix} a_{00} & \cdots & a_{0d_x} \\ \vdots & \ddots & \vdots \\ a_{d_y 0} & \cdots & a_{d_y d_x} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ z \\ \vdots \\ z^{d_x} \end{bmatrix}.$$

For example, the polynomial $p(z,\overline{z}) = 1 + 2z + 3z^2 + 4\overline{z} + 5z\overline{z} + 6z^2\overline{z} + 7\overline{z}^2$ is represented by the matrix

```
p = [1 2 3; ...
     4 5 6; ...
     7 0 0];
```

However, minimizing a polyanalytic polynomial $p(z,\overline{z})$ only makes sense if $p(z,\overline{z})$ is real-valued for all $z \in \mathbb{C}$. A polyanalytic polynomial is real-valued if and only if its matrix representation is Hermitian, i.e., `p == p'`. As with bivariate

polynomials, the stationary points of a real-valued polyanalytic polynomial $p(z,\overline{z})$ can be computed with `polymin2(p)`.

As an example, the stationary points of a pseudorandom real-valued polyanalytic polynomial can be computed with

```
p = rand(6)+rand(6)*1i;
p = p*p';
options.Plot = true;
xy = polymin2(p,options);
```

Computing the stationary points of a polyanalytic rational function $\frac{p(z,\overline{z})}{q(z,\overline{z})}$ is completely analogous to the polynomial case. For example,

```
p = rand(6)+rand(6)*1i; p = p*p';
q = rand(4)+rand(4)*1i; q = q*q';
options.Plot = true;
xy = ratmin2(p,q,options);
```

Remark If the matrix `p` contains complex coefficients and is Hermitian, `polymin2` will treat `p` as a real-valued polyanalytic polynomial. Otherwise, it will be treated as a bivariate polynomial. In some cases it may be necessary to specify what type of polynomial `p` is. In that case, set `options.Univariate` to `true` if `p` is a real-valued polyanalytic polynomial and `false` otherwise. The same option also applies to `ratmin2`.

8.2 Solving bivariate polynomial systems

The functions `polymin2` and `ratmin2` depend on the lower level function `polysol2` to compute the isolated solutions of systems of bivariate polynomials (bisys) or systems of polyanalytic univariate polynomials (unisys). In the case (bisys), `polysol2(p,q)` computes the isolated real solutions of the system $p(x,y) = q(x,y) = 0$. A solution (x^*, y^*) is said to be isolated if there exists a neighbourhood of (x^*, y^*) in $\mathbb{C}^2$ that contains no solution other than (x^*, y^*). Some systems may have solutions that are isolated in $\mathbb{R}^2$, but not in $\mathbb{C}^2$.

Remark By default, `polysol2(p,q)` assumes `p` and `q` are polyanalytic univariate if at least one of `p` and `q` contains complex coefficients. If this is not the case, the user can specify the type of polynomial by setting `options.Univariate` to `true` if both polynomials are polyanalytic univariate or `false` otherwise.

The algorithm in `polysol2` applies several balancing steps to the problem in order to improve the accuracy of the computed roots, before refining them with Newton-Raphson. If the system is poorly scaled, it may be necessary to decrease the balancing tolerance `options.TolBal` to a smaller value. In Figure A.9, the solutions of a relatively difficult bivariate system $p(x, y) = q(x, y) = 0$ are plotted. The polynomials $p(x, y)$ and $q(x, y)$ are both of total degree 20 and have coefficients of which the exponents (in base 10) range between 1 and 7.

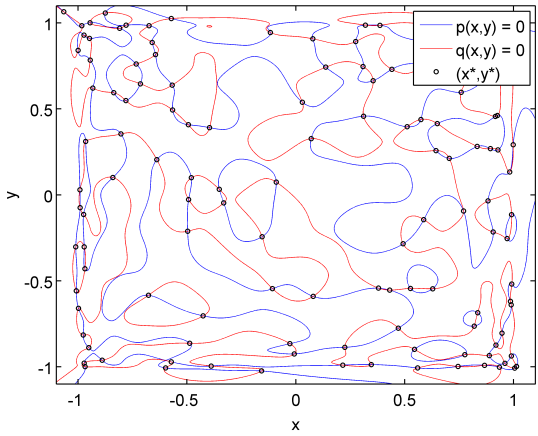


Figure A.9: A system of bivariate polynomials $p(x, y) = q(x, y) = 0$ of total degree 20.

The following example generates a random bivariate system, computes its isolated solutions and plots the results:

```
p = tril(randn(5)); q = triu(randn(5));
options.Plot = true; % Note that plotting can take a while.
xy = polysol2(p,q,options);
```

Acknowledgements

We would like to acknowledge Mariya Ishteva and Nick Vannieuwenhoven for their important contributions to Tensorlab. Additionally, some code is based on a modified version of GenRTR, by Pierre-Antoine Absil, Christopher G. Baker, and Kyle A. Gallivan. The Moré–Thuente line search implemented in `ls_mt` was translated and adapted for complex optimization from the original Fortran MINPACK-2 implementation by Brett M. Averick, Richard G. Carter and Jorge J. Moré.

Bibliography

[1] Abatzoglou, T. J. Tensor-based techniques for the blind separation of DS-CDMA signals. *Signal Processing 87*, 2 (Feb. 2007), 322–336.

[2] Abatzoglou, T. J., Mendel, J. M., and Harada, G. A. The constrained total least squares technique and its applications to harmonic superresolution. *IEEE Transactions on Signal Processing 39*, 5 (May 1991), 1070–1087.

[3] Absil, P.-A., Mahony, R., and Sepulchre, R. *Optimization Algorithms on Matrix Manifolds.* Princeton University Press, Princeton, NJ, 2008.

[4] Acar, E., Gurdeniz, G., Rasmussen, M. A., Rago, D., Dragsted, L. O., and Bro, R. Coupled matrix factorization with sparse factors to identify potential biomarkers in metabolomics. In *Proc. of the 12th IEEE International Conference on Data Mining Workshops (ICDMW)* (2012).

[5] Acar, E., Kolda, T. G., and Dunlavy, D. M. An optimization approach for fitting canonical tensor decompositions. Tech. rep., Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2009.

[6] Acar, E., Kolda, T. G., and Dunlavy, D. M. All-at-once optimization for coupled matrix and tensor factorizations. In *Proc. of Mining and Learning with Graphs* (2011).

[7] Acar, E., Rasmussen, M. A., Savorani, F., Næs, T., and Bro, R. Understanding data fusion within the framework of coupled matrix and tensor factorizations. Submitted, Nov. 2012.

[8] Acar, E., and Yener, B. Unsupervised multiway data analysis: A literature survey. *Knowledge and Data Engineering, IEEE Transactions on 21*, 1 (2009), 6–20.

[9] Albersmeyer, J., and Diehl, M. The lifted Newton method and its application in optimization. *SIAM Journal on Optimization 20*, 3 (2010), 1655–1684.

[10] Anderson, E., Bai, Z., Bischof, C. H., Blackford, S., Demmel, J. W., Dongarra, J. J., Du Croz, J. J., Greenbaum, A., Hammarling, S. J., McKenney, A., and Sorensen, D. C. *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[11] Argyriou, A., Evgeniou, T., and Pontil, M. Convex multi-task feature learning. *Machine Learning 73*, 3 (2008), 243–272.

[12] Aubry, P., Lazard, D., and Moreno Maza, M. On the theories of triangular sets. *Journal of Symbolic Computation 28*, 1-2 (July 1999), 105–124.

[13] Auzinger, W., and Stetter, H. J. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In *Proceedings of the International Conference on Numerical Mathematics* (Birkhäuser, Basel, 1988), vol. 86 of *Intern. Series in Numer. Math.*, pp. 12–30.

[14] Badea, L. Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization. In *Pacific Symposium on Biocomputing* (2008), vol. 290, pp. 279–290.

[15] Badea, L. Multi-relational factorizations for cancer subclassification. In *Proc. of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)* (2010).

[16] Bader, B. W., and Kolda, T. G. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing 30*, 1 (2006), 205–231.

[17] Bai, Z., Day, D., Demmel, J. W., Dongarra, J. J., Gu, M., Ruhe, A., and van der Vorst, H. A. Templates for linear algebra problems. In *Computer Science Today*, J. van Leeuwen, Ed., vol. 1000 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 1995, pp. 115–140.

[18] Bajaj, C., Garrity, T., and Warren, J. On the applications of multi-equational resultants. Technical Report 88-826, Department of Computer Science, Purdue University, Nov. 1988.

[19] Basu, S., Pollack, R., and Roy, M.-F. *Algorithms in Real Algebraic Geometry*, second ed., vol. 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag, 2006.

[20] Bates, D. J., Hauenstein, J. D., Sommese, A. J., and Wampler, C. W. Adaptive multiprecision path tracking. *SIAM Journal on Numerical Analysis 46*, 2 (2008), 722–746.

[21] Bates, D. J., Hauenstein, J. D., Sommese, A. J., and Wampler, C. W. *Numerically solving polynomial systems with Bertini*, vol. 25 of *Software, Environments and Tools*. SIAM, 2013.

[22] Belouchrani, A., Abed-Meraim, K., Cardoso, J.-F., and Moulines, E. A blind source separation technique using second order statistics. *IEEE Transactions on Signal Processing 45*, 2 (1997), 434–444.

[23] Berberich, E., Emeliyanenko, P., and Sagraloff, M. An elimination method for solving bivariate polynomial systems: Eliminating the usual drawbacks. In *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX)* (San Francisco, CA, USA, Jan. 2011), M. Müller-Hannemann and R. Werneck, Eds., SIAM, pp. 35–47.

[24] Beyer, M. A., and Laney, D. The importance of 'big data': A definition. `http://www.gartner.com/doc/2057415`, June 2012.

[25] Blum, A., and Mitchell, T. Combining labeled and unlabeled data with co-training. In *Proc. of the Conference on Computational Learning Theory* (1998).

[26] Borlase, S. *Smart Grids: Infrastructure, Technology, and Solutions*. CRC Press, 2012.

[27] Brandwood, D. H. A complex gradient operator and its application in adaptive array theory. *IEEE Proceeding H 130*, 1 (1983), 11–16.

[28] Bro, R. *Multi-way analysis in the food industry: models, algorithms, and applications*. PhD thesis, University of Amsterdam, 1998.

[29] Broyden, C. G. The convergence of a class of double-rank minimization algorithms: 2. The new algorithm. *Journal of the Institute of Mathematics and its Applications 6*, 3 (1970), 222–231.

[30] Buchberger, B. Gröbner bases and systems theory. *Multidimensional Systems and Signal Processing 12* (2001), 223–251.

[31] Busé, L., Khalil, H., and Mourrain, B. Resultant-based methods for plane curves intersection problems. In *Proceedings of the 8th international conference on Computer Algebra in Scientific Computing* (2005), vol. 3718 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 75–92.

[32] Byrd, R. H., Schnabel, R. B., and Schultz, G. A. Approximate solution of the trust region problem by minimization over two-dimensional subspaces. *Mathematical Programming 40*, 1 (1988), 247–263.

[33] Carroll, J. D., and Chang, J.-J. Analysis of individual differences in multidimensional scaling via an n-way generalization of "Eckart–Young" decomposition. *Psychometrika 35*, 3 (1970), 283–319.

[34] Chen, Y., Han, D., and Qi, L. New ALS methods with extrapolating search directions and optimal step size for complex-valued tensor decompositions. *IEEE Transactions on Signal Processing 59*, 12 (Dec. 2011), 5888–5898.

[35] Cichocki, A., Mandic, D., Caiafa, C., Phan, A. H., Zhou, G., Zhao, Q., and De Lathauwer, L. Multiway component analysis: Tensor decompositions for signal processing applications. *IEEE Signal Processing Magazine* (2014). Accepted.

[36] Cichocki, A., Zdunek, R., Phan, A. H., and Amari, S.-I. *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-way Data Analysis and Blind Source Separation*. Wiley, 2009.

[37] Comon, P. Independent component analysis, a new concept? *Signal Processing 36*, 3 (1994), 287–314.

[38] Comon, P., and Jutten, C. *Handbook of Blind Source Separation: Independent Component Analysis and Applications*. Academic Press, 2010.

[39] Comon, P., Luciani, X., and de Almeida, A. L. F. Tensor decompositions, alternating least squares and other tales. *Journal of Chemometrics 23* (2009), 393–405.

[40] Conn, A. R., Gould, N. I. M., and Toint, P. L. *Trust-region methods*. MPS-SIAM Series on Optimization. SIAM and MPS, 2000.

[41] Corless, R. M., Gianni, P. M., and Trager, B. M. A reordered Schur factorization method for zero-dimensional polynomial systems with multiple roots. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation* (1997), ACM Press, pp. 133–140.

[42] Correa, N. M., Adali, T., Li, Y.-O., and Calhoun, V. D. Canonical correlation analysis for data fusion and group inferences. *IEEE Signal Processing Magazine 27*, 4 (2010), 39–50.

[43] Curran, P. J., and Hussong, A. M. Integrative data analysis: The simultaneous analysis of multiple data sets. *Psychological Methods 14*, 2 (2009), 81–100.

[44] Danova, T. Morgan Stanley: 75 billion devices will be connected to the Internet Of Things by 2020. `http://www.businessinsider.com/75-billion-devices-will-be-connected-to-the-internet-by-2020-2013-10`, Oct. 2013.

[45] De Lathauwer, L. A link between the canonical decomposition in multilinear algebra and simultaneous matrix diagonalization. *SIAM Journal on Matrix Analysis and Applications 28*, 3 (2006), 642–666.

[46] De Lathauwer, L. Decompositions of a higher-order tensor in block terms — Part I: Lemmas for partitioned matrices. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1022–1032.

[47] De Lathauwer, L. Decompositions of a higher-order tensor in block terms — Part II: Definitions and uniqueness. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1033–1066.

[48] De Lathauwer, L. Blind separation of exponential polynomials and the decomposition of a tensor in rank-$(L_r, L_r, 1)$ terms. *SIAM Journal on Matrix Analysis and Applications 32*, 4 (Dec. 2011), 1451–1474.

[49] De Lathauwer, L. A short introduction to tensor-based methods for factor analysis and blind source separation. In *Proc. of the 7th International Symposium on image and signal processing and analysis (ISPA 2011)* (Sept. 2011), pp. 558–563.

[50] De Lathauwer, L. Block component analysis, a new concept for source separation and factor analysis. In *Latent Variable Analysis and Signal Separation* (2012), vol. 7191 of *Lecture Notes in Computer Science*, Proc. of 10th International Conference, LVA/ICA 2012, Tel Aviv, Israel, March 12-15, 2012., pp. 1–8.

[51] De Lathauwer, L., and Castaing, J. Blind identification of underdetermined mixtures by simultaneous matrix diagonalization. *IEEE Transactions on Signal Processing 56*, 3 (2008), 1096–1105.

[52] De Lathauwer, L., Comon, P., and Mastronardi, N. Special issue on tensor decompositions and applications. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1–1279.

[53] De Lathauwer, L., and de Baynast, A. Blind deconvolution of DS-CDMA signals by means of decomposition in rank-$(1, L, L)$ terms. *IEEE Transactions on Signal Processing 56*, 4 (2008), 1562–1571.

[54] De Lathauwer, L., De Moor, B. L. R., and Vandewalle, J. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications 21*, 4 (2000), 1253–1278.

[55] De Lathauwer, L., and Nion, D. Decompositions of a higher-order tensor in block terms — Part III: Alternating least squares algorithms. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1067–1083.

[56] De Lathauwer, L., and Vandewalle, J. Dimensionality reduction in higher-order signal processing and rank-$(R_1, R_2, \ldots, R_N)$ reduction in multilinear algebra. *Linear Algebra and its Applications 391* (2004), 31–55.

[57] de Silva, V., and Lim, L.-H. Tensor rank and the ill-posedness of the best low-rank approximation problem. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1084–1127.

[58] De Vos, M., Vergult, A., De Lathauwer, L., De Clercq, W., Van Huffel, S., Dupont, P., Palmini, A., and Van Paesschen, W. Canonical decomposition of ictal EEG reliably detects seizure onset zone. *Neuroimage 37*, 3 (Sept. 2007), 844–854.

[59] Delort, P. Harnessing data as a new source of growth: Big data analytics and policies. `http://www.oecd.org/sti/ieconomy/Session_3_Delort.pdf#page=6`.

[60] Dennis, Jr., J. E., and Mei, H. H. W. Two new unconstrained optimization algorithms which use function and gradient values. *Journal of Optimization Theory and Applications 28*, 4 (1979), 453–482.

[61] Dennis, Jr., J. E., and Moré, J. J. Quasi-Newton methods, motivation and theory. *SIAM Review 19*, 1 (1977), 46–89.

[62] Ding, Z., Johnson, C. R., and Kennedy, R. A. On the (non)existence of undesirable equilibria of godard blind equalizers. *IEEE Transactions on Signal Processing 40*, 10 (Oct. 1992), 2425–2432.

[63] Diochnos, D. I., Emiris, I. Z., and Tsigaridas, E. P. On the asymptotic and practical complexity of solving bivariate systems over the reals. *Journal of Symbolic Computation 44*, 7 (July 2009), 818–835.

[64] Dixon, L. C., and Szegő, G. P. *The global optimization problem: an introduction.* Towards global optimisation II. North-Holland Pub. Co., 1978, pp. 1–15.

[65] Dolan, E. D., and Moré, J. J. Benchmarking optimization software with performance profiles. *Mathematical Programming 91*, 2 (2002), 201–213.

[66] Domanov, I., and De Lathauwer, L. Canonical polyadic decomposition of third-order tensors: reduction to generalized eigenvalue decomposition. ESAT-STADIUS Internal Report 13-36, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[67] Domanov, I., and De Lathauwer, L. On the uniqueness of the canonical polyadic decomposition — Part I: Basic results and uniqueness of one factor matrix. *SIAM Journal on Matrix Analysis and Applications 34*, 3 (2013), 855–875.

[68] Domanov, I., and De Lathauwer, L. On the uniqueness of the canonical polyadic decomposition — Part II: Overall uniqueness. *SIAM Journal on Matrix Analysis and Applications 34*, 3 (2013), 876–903.

[69] Emiris, I. Z., and Mourrain, B. Matrices in elimination theory. *Journal of Symbolic Computation 28*, 1-2 (1999), 3–43.

[70] Ermiş, B., Acar, E., and Cemgil, A. T. Link prediction in heterogeneous data via generalized coupled tensor factorization. *Data Mining and Knowledge Discovery* (2013), 1–34.

[71] Evans, D. The Internet of Things. `http://blogs.cisco.com/diversity/the-internet-of-things-infographic/`, July 2011.

[72] Facebook. Facebook reports fourth quarter and full year 2013 results. `http://investor.fb.com/releasedetail.cfm?ReleaseID=821954`, Jan. 2014.

[73] Fang, Y., and Si, L. Matrix co-factorization for recommendation with rich side information and implicit feedback. In *Proc. of the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems* (2011).

[74] Faugère, J.-C. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In *Proceedings of the 2002 international symposium on Symbolic and algebraic computation* (2002), ISSAC '02, ACM, pp. 75–83.

[75] Finnegan, M. Boeing 787s to create half a terabyte of data per flight, says Virgin Atlantic. http://www.computerworlduk.com/news/infrastructure/3433595/, Mar. 2013.

[76] Fletcher, R. A new approach to variable metric algorithms. *Computer Journal 13*, 3 (1970), 317–322.

[77] Fletcher, R., and Reeves, C. M. Function minimization by conjugate gradients. *Computer Journal 7* (1964), 149–154.

[78] Gilbert, J. C., and Lemaréchal, C. Some numerical experiments with variable-storage quasi-Newton algorithms. *Mathematical Programming 45*, 1 (1989), 407–435.

[79] Gilbert, J. C., and Nocedal, J. Global convergence properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization 2*, 1 (1992), 21–42.

[80] Godard, D. N. Self-recovering equalization and carrier tracking in two-dimensional data communication systems. *IEEE Transactions on Communications 28*, 11 (Nov. 1980), 1867–1875.

[81] Gohberg, I. C., Lancaster, P., and Rodman, L. *Matrix Polynomials*. Academic Press, New York, 1982.

[82] Goldfarb, D. A family of variable-metric methods derived by variational means. *Mathematics of Computation 24*, 109 (1970), 23–26.

[83] Gonzales-Vega, L., and Necula, I. Efficient topology determination of implicitly defined algebraic plane curves. *Computer Aided Geometric Design 19*, 9 (Dec. 2002), 719–743.

[84] Google. Google.com anno 2002. http://web.archive.org/web/20021107070412/http://www.google.com/, Nov. 2002.

[85] Google. Inside search. http://www.google.com/intl/nl/insidesearch/howsearchworks/thestory/, Feb. 2014.

[86] Grasedyck, L. Hierarchical singular value decomposition of tensors. *SIAM Journal on Matrix Analysis and Applications 31*, 4 (2010), 2029–2054.

[87] Griewank, A., and Osborne, M. R. Analysis of newton's method at irregular singularities. *SIAM Journal on Numerical Analysis 20*, 4 (1983), 747–773.

[88] Gross, B. Google's self driving car gathers nearly 1 GB/sec. http://www.linkedin.com/today/post/article/20130502024505-9947747-google-s-self-driving-car-gathers-nearly-1-gb-per-second, May 2013.

[89] Gupta, S. K., Phung, D., Adams, B., and Venkatesh, S. A matrix factorization framework for jointly analyzing multiple nonnegative data. In *Proc. of the 2011 SIAM International Conference on Data Mining (SDM)* (2011).

[90] Haardt, M., Roemer, F., and Del Galdo, G. Higher-order SVD-based subspace estimation to improve the parameter estimation accuracy in multidimensional harmonic retrieval problems. *IEEE Transactions on Signal Processing 56*, 7 (2008), 3198–3213.

[91] Hackbusch, W. *Tensor Spaces and Numerical Tensor Calculus*, vol. 42 of *Springer Series in Computational Mathematics*. Springer-Verlag Berlin, 2012.

[92] Hackbusch, W., and Kühn, S. A new scheme for the tensor representation. *Journal Fourier Analysis and Applications 15*, 5 (2009), 706–722.

[93] Hager, W. W., and Zhang, H. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on Optimization 16*, 1 (2006), 170–192.

[94] Hager, W. W., and Zhang, H. A survey of nonlinear conjugate gradient methods. *Pacific Journal of Optimization 2*, 1 (2006), 35–58.

[95] Harshman, R. A. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics 16*, 1 (1970), 84–84.

[96] Hayashi, C., and Hayashi, F. A new algorithm to solve PARAFAC-model. *Behaviormetrika 9*, 11 (1982), 49–60.

[97] Hayden, E. Is the $1,000 genome for real? http://www.nature.com/news/is-the-1-000-genome-for-real-1.14530, Jan. 2014.

[98] Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards 49* (1952), 409–436.

[99] Hitchcock, F. L. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematical Physics 6*, 1 (1927), 164–189.

[100] Hitchcock, F. L. Multiple invariants and generalized rank of a p-way matrix or tensor. *Journal of Mathematical Physics 7*, 1 (1927), 39–79.

[101] Hotelling, H. Relations between two sets of variates. *Biometrika 28*, 3/4 (1936), 321–377.

[102] IATA. 2012 Best in history of continuous safety improvements. `http://www.iata.org/pressroom/pr/Pages/2013-02-28-01.aspx`, Feb. 2013.

[103] James, R. C. *Advanced calculus*. Wadsworth, 1966.

[104] Jiang, T., and Sidiropoulos, N. D. Kruskal's permutation lemma and the identification of CANDECOMP/PARAFAC and bilinear models with constant modulus constraints. *IEEE Transactions on Signal Processing 52* (2004), 2625–2636.

[105] Jónsson, G. F., and Vavasis, S. A. Accurate solution of polynomial equations using macaulay resultant matrices. *Mathematics of Computation 74*, 249 (July 2004), 221–262.

[106] Kailath, T., and Sayed, A. H., Eds. *Fast reliable algorithms for matrices with structure*. SIAM, Philadelphia, PA, USA, May 1999.

[107] Kajiya, J. T. Ray tracing parametric patches. *ACM SIGGRAPH Computer Graphics 16*, 3 (July 1982), 245–254.

[108] Karami, A., Yazdi, M., and Zolghadre Asli, A. Noise reduction of hyperspectral images using kernel non-negative Tucker decomposition. *IEEE Journal of selected topics in signal processing 5*, 3 (June 2011), 487–493.

[109] Khatri, C. G., and Rao, C. R. Solutions to some functional equations and their applications to characterization of probability distributions. *Sankhya: The Indian Journal of Statistics, Series A 30*, 2 (1968), 167–180.

[110] Kim, J., Monteiro, D. C, R., and Park, H. Group sparsity in nonnegative matrix factorization. In *Proc. of the 2012 SIAM International Conference on Data Mining (SDM)* (2012).

[111] Knight, P. A. The Sinkhorn-Knopp algorithm: Convergence and applications. *SIAM Journal on Matrix Analysis and Applications 30*, 1 (Mar. 2008), 261–275.

[112] Kolda, T. G., and Bader, B. W. Tensor decompositions and applications. *SIAM Review 51*, 3 (Sept. 2009), 455–500.

[113] Koren, Y., Bell, R., and Volinsky, C. Matrix factorization techniques for recommender systems. *Computer 42*, 8 (2009), 30–37.

[114] Kreutz-Delgado, K. The complex gradient operator and the CR-calculus. *ArXiv e-prints* (June 2009). arXiv:0906.4835v1.

[115] Krijnen, W. P., Dijkstra, T. K., and Stegeman, A. On the non-existence of optimal solutions and the occurrence of "degeneracy" in the CANDECOMP/PARAFAC model. *Psychometrika 73*, 3 (2008), 431–439.

[116] Krohn-Grimberghe, A., Drumond, L., Freudenthaler, C., and Schmidt-Thieme, L. Multi-relational matrix factorization using bayesian personalized ranking for social network data. In *Proc. of the 5th ACM international conference on Web search and data mining* (2012).

[117] Kroonenberg, P. M. *Applied Multiway Data Analysis*. Wiley, 2008.

[118] Kruskal, J. B. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear Algebra and its Applications 18*, 2 (1977), 95–138.

[119] Kruskal, J. B. *Rank, decomposition, and uniqueness for 3-way and N-way arrays*. Elsevier Science Publishers B.V., 1989, pp. 7–18.

[120] Kruskal, J. B., Harshman, R. A., and Lundy, M. E. How 3-MFA data can cause degenerate PARAFAC solutions, among other relationships. In *Multiway data analysis*, R. Coppi and S. Bolasco, Eds. North-Holland Publishing Co., 1989, pp. 115–121.

[121] Landsberg, J. M. *Tensors: Geometry and Applications*, vol. 128 of *Graduate Studies in Mathematics*. AMS, 2012.

[122] Lasserre, J. B. Global optimization with polynomials and the problems of moments. *SIAM Journal on Optimization 11*, 3 (2001), 796–817.

[123] Lazard, D. Résolution des systèmes d'équations algébriques. *Theoretical Computer Science 15* (1981), 77–110.

[124] Le, T., H., Sorber, L., and Van Barel, M. The Pythagoras number of real sum of squares polynomials and sum of square magnitudes of polynomials. *Calcolo 50*, 4 (Dec. 2013), 283–303.

[125] Lee, C. H., Alpert, B. O., Sankaranarayanan, P., and Alter, O. GSVD comparison of patient-matched normal and tumor aCGH profiles reveals global copy-number alterations predicting glioblastoma multiforme survival. *PLoS One 7*, 1 (2012). Article e30098.

[126] Lee, H., and Choi, S. Group nonnegative matrix factorization for EEG classification. In *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)* (2009).

[127] Lee, H., Yoo, J., and Choi, S. Semi-supervised nonnegative matrix factorization. *IEEE Signal Processing Letters 17*, 1 (2010), 4–7.

[128] Lemonnier, D., and Van Dooren, P. Balancing regular matrix pencils. *SIAM Journal on Matrix Analysis and Applications 28*, 1 (2006), 253–263.

[129] Levin, J. Simultaneous factor analysis of several Gramian matrices. *Psychometrika 31*, 3 (1966), 413–419.

[130] Li, T., Zhang, Y., and Sindhwani, V. A non-negative matrix tri-factorization approach to sentiment classification with lexical prior knowledge. In *Proc. of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP* (2009).

[131] Li, T.-Y. Numerical solution of multivariate polynomial systems by homotopy continuation methods. *Acta Numerica 6* (1997), 399–436.

[132] Li, W.-J., and Yeung, D.-Y. Relation regularized matrix factorization. In *Proc. of the 21st International Joint Conference on Artificial Intelligence* (2009).

[133] Li, Y.-O., Adali, T., Wei, W., and Calhoun, V. D. Joint blind source separation by multiset canonical correlation analysis. *IEEE Signal Processing Magazine 57*, 10 (2009), 3918–3929.

[134] Lin, Y.-R., Sun, J., Castro, P., Konuru, R., Sundaram, H., and Kelliher, A. Metafac: Community discovery via relational hypergraph factorization. In *Proc. of the 15th ACM SIGKDD international conference on Knowledge Discovery and Data mining* (2009).

[135] Lippert, C., Weber, S. H., Huang, Y., Tresp, V., Schubert, M., and Kriegel, H.-P. Relation prediction in multi-relational domains using matrix factorization. In *Proc. of the NIPS 2008 Workshop: Structured Input-Structured Output* (2008).

[136] Liu, D. C., and Nocedal, J. On the limited memory BFGS method for large scale optimization. *Mathematical Programming 45*, 1 (1989), 503–528.

[137] Liu, W., Chan, J., Bailey, J., Leckie, C., and Ramamohanarao, K. Mining labelled tensors by discovering both their common and

discriminative subspaces. In *Proc. of the 2013 SIAM International Conference on Data Mining (SDM)* (2013).

[138] Long, B., Zhang, Z., Wú, X., and Yu, P. S. Spectral clustering for multi-type relational data. In *Proc. of the 23rd International Conference on Machine Learning* (2006).

[139] Lu, H., Plataniotis, K. N., and Venetsanopoulos, A. N. A survey of multilinear subspace learning for tensor data. *Pattern Recognition 44*, 7 (2011), 1540–1551.

[140] Manocha, D. Solving systems of polynomial equations. *IEEE Computer Graphics and Applications 14*, 2 (Mar. 1994), 46–55.

[141] Manocha, D., and Demmel, J. W. Algorithms for intersecting parametric and algebraic curves II: Multiple intersections. *Graphical Models and Image Processing 57*, 2 (Mar. 1995), 81–100.

[142] Maroulas, J., and Barnett, S. Polynomials with respect to a general basis. I. Theory. *Journal of Mathematical Analysis and Applications 72* (1979), 177–194.

[143] Maroulas, J., and Barnett, S. Polynomials with respect to a general basis. II. Applications. *Journal of Mathematical Analysis and Applications 72* (1979), 599–614.

[144] Mitchell, B. C., and Burdick, D. S. Slowly converging PARAFAC sequences: swamps and two-factor degeneracies. *Journal of Chemometrics 8*, 2 (Mar. 1994), 155–168.

[145] Möller, H. M., and Stetter, H. J. Multivariate polynomial equations with multiple zeros solved by matrix eigenproblems. *Numerische Mathematik 70*, 3 (May 1995), 311–329.

[146] Moore, R. E. *Methods and applications of interval analysis.* SIAM, 1987.

[147] Moré, J. J., and Thuente, D. J. Line search algorithms with guaranteed sufficient decrease. *ACM Transactions on Mathematical Software 20*, 3 (1994), 286–307.

[148] Morgan, A. P. *Polynomial continuation and its relationship to the symbolic reduction of polynomial systems.* Symbolic and Numerical Computation for Artificial Intelligence. Academic Press, New York, 1992, pp. 23–45.

[149] Nagel, L. W. SPICE2: A computer program to simulate semiconductor circuits. Memorandum ERL-M520, Electronics Research Lab, University of California, Berkeley, 1975.

[150] Nakatsukasa, Y., Noferini, V., and Townsend, A. Computing the common zeros of two bivariate functions via Bézout resultants. Submitted, 2013.

[151] Nakhla, M. S., and Vlach, J. A piecewise harmonic balance technique for determination of periodic response of nonlinear systems. *IEEE Transactions on Circuits and Systems 23*, 2 (1976), 85–91.

[152] Navasca, C., De Lathauwer, L., and Kindermann, S. Swamp reducing technique for tensor decomposition. In *Proceedings of the 16th European Signal Processing Conference (EUSIPCO 2008)* (2008).

[153] Nehari, Z. *Introduction to complex analysis*. Allyn & Bacon, Inc, 1961.

[154] Netflix. Grand prize awarded to team BellKor's Pragmatic Chaos. http://www.netflixprize.com/community/viewtopic.php?id=1537, Sept. 2009.

[155] Nion, D., and De Lathauwer, L. Block component model based blind DS-CDMA receivers. *IEEE Transactions on Signal Processing 56*, 11 (Nov. 2008), 5567–5579.

[156] Nion, D., and De Lathauwer, L. An enhanced line search scheme for complex-valued tensor decompositions. Application in DS-CDMA. *Signal Processing 88*, 3 (2008), 749–755.

[157] Nion, D., and De Lathauwer, L. Tensor algebra and multidimensional harmonic retrieval in signal processing for MIMO radar. *IEEE Transactions on Signal Processing 58*, 11 (Nov. 2010), 5693–5705.

[158] Nocedal, J. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation 35*, 151 (1980), 773–782.

[159] Nocedal, J., and Wright, S. J. *Numerical Optimization*, second ed. Springer Series in Operations Research. Springer, 2006.

[160] Nowlan, S. J., and Hinton, G. E. Simplifying neural networks by soft weight-sharing. *Neural Computation 4* (1992), 473–493.

[161] Oseledets, I. V. Tensor-train decomposition. *SIAM Journal on Scientific Computing 33*, 5 (June 2011), 2295–2317.

[162] Paatero, P. A weighted non-negative least squares algorithm for three-way 'PARAFAC' factor analysis. *Chemometrics and Intelligent Laboratory Systems 38*, 2 (1997), 223–242.

[163] Paatero, P. The multilinear engine: A table-driven, least squares program for solving multilinear problems, including the n-way parallel factor analysis model. *Journal of Computational and Graphical Statistics 8*, 4 (Dec. 1999), 854–888.

[164] Paatero, P. Construction and analysis of degenerate PARAFAC models. *Journal of Chemometrics 14*, 3 (2000), 285–299.

[165] Paige, C. C., and Saunders, M. A. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software 8*, 1 (1982), 43–71.

[166] Papalexakis, E. E., Mitchell, T. M., Sidiropoulos, N. D., Faloutsos, C., Talukdar, P. P., and Murphy, B. Scoup-SMT: Scalable coupled sparse matrix-tensor factorization. *Computing Research Repository abs/1302.7* (2013).

[167] Parrilo, P. A. *Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization*. PhD thesis, California Institute of Technology, 2000.

[168] Phan, A. H., Tichavský, P., and Cichocki, A. Low complexity damped Gauss-Newton algorithms for CANDECOMP/PARAFAC. *Computing Research Repository abs/1205.2* (2012).

[169] Phan, A. H., Tichavský, P., and Cichocki, A. On fast computation of gradients for CANDECOMP/PARAFAC algorithms. *Computing Research Repository abs/1204.1* (2012).

[170] Phan, A. H., Tichavský, P., and Cichocki, A. Fast alternating LS algorithms for high order CANDECOMP/PARAFAC tensor factorizations. *IEEE Transactions on Signal Processing 61*, 19 (2013), 4834–4846.

[171] Polak, E., and Ribière, G. Note sur la convergence de methods de directions conjugres. *Revue Française d'Informatique et de Recherche Operationnelle 16* (1969), 35–43.

[172] Ponnapalli, S. P., Saunders, M. A., Van Loan, C. F., and Alter, O. A higher-order generalized singular value decomposition for comparison of global mRNA expression from multiple organisms. *PLoS One 6*, 11 (2011). Article e28072.

[173] Powell, M. J. D. A hybrid method for nonlinear equations. In *Numerical methods for nonlinear algebraic equations* (1970), P. Robinowitz, Ed., Gordon and Breach Science, London, pp. 87–144.

[174] Powell, M. J. D. A new algorithm for unconstrained optimization. In *Nonlinear programming* (1970), J. B. Rosen, O. L. Mangasarian, and K. Ritter, Eds., Academic Press, New York, pp. 31–66.

[175] Powell, M. J. D. Restart procedures for the conjugate gradient method. *Mathematical Programming 12*, 1 (1977), 241–254.

[176] Powell, M. J. D. *Nonconvex minimization calculations and the conjugate gradient method*. Numerical Analysis. Springer-Verlag, Berlin, 1984.

[177] Qi, L., Wan, Z., and Yang, Y.-F. Global minimization of normal quartic polynomials based on global descent directions. *SIAM Journal on Optimization 15*, 1 (2006), 275–302.

[178] Quarles, T. L. Analysis of performance and convergence issues for circuit simulation. Memorandum ERL-M89/42, Electronics Research Lab, University of California, Berkeley, 1989.

[179] Rajih, M., Comon, P., and Harshman, R. A. Enhanced line search: a novel method to accelerate PARAFAC. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 1128–1147.

[180] Rao, C. R., and Rao, M. B. *Matrix algebra and its applications to statistics and econometrics*. World Scientific, 1998.

[181] Reber, P. What is the memory capacity of the human brain? `http://www.scientificamerican.com/article/what-is-the-memory-capacity/`, Apr. 2010.

[182] Remmert, R. *Theory of Complex Functions*. Springer-Verlag, 1991.

[183] Ricci-Curbastro, G. Résumé de quelques travaux sur les systèmes variables de fonctions associés à une forme différentielle quadratique. *Bulletin des Sciences Mathématiques 2*, 16 (1892), 167–189.

[184] Rizzoli, V., Mastri, F., Cecchetti, C., and Sgallari, F. Fast and robust inexact Newton approach to the harmonic-balance analysis of nonlinear microwave circuits. *IEEE Microwave and Guided Wave Letters 7*, 10 (Oct. 1997), 359–361.

[185] Rouillier, F. Solving zero-dimensional systems through the rational univariate representation. *Applicable Algebra in Engineering, Communication and Computing 9*, 5 (1999), 433–461.

[186] Rouillier, F., and Zimmermann, P. Efficient isolation of polynomial real roots. *Journal of Computational and Applied Mathematics 162*, 1 (2003), 33–50.

[187] Schnabel, R. B., and Dennis, Jr., J. E. A new derivation of symmetric positive definite secant updates. Tech. rep., Department of Computer Science, Colorado University, Aug. 1980.

[188] Šebek, M., Pejchová, S., Henrion, D., and Kwakernaak, H. Numerical methods for zeros and determinant of polynomial matrix. In *Proceedings of the IEEE Mediterranean Symposium on New Directions in Control and Automation* (Chania, Crete, Greece, June 1996), pp. 488–491.

[189] Seidel, R., and Wolpert, N. On the exact computation of the topology of real algebraic curves. In *Proceedings of the twenty-first annual symposium on Computational geometry* (New York, NY, USA, 2005), SCG 2005, ACM, pp. 107–115.

[190] Shanno, D. F. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation 24*, 111 (1970), 647–656.

[191] Sherman, J., and Morrison, W. J. Adjustment of an inverse matrix corresponding to changes in the elements of a given column or a given row of the original matrix. *Annals of Mathematical Statistics 20* (1949), 621–621.

[192] Sherman, J., and Morrison, W. J. Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics 21* (1950), 124–127.

[193] Shor, N. Z. Class of global minimum bounds of polynomial functions. *Cybernetics and Systems Analysis 23*, 6 (1987), 731–734.

[194] Sidiropoulos, N. D., and Bro, R. On the uniqueness of multilinear decomposition of N-way arrays. *Journal of Chemometrics 14*, 3 (2000), 229–239.

[195] Sidiropoulos, N. D., Bro, R., and Giannakis, G. B. Parallel factor analysis in sensor array processing. *IEEE Transactions on Signal Processing 48*, 8 (Aug. 2000), 2377–2388.

[196] Sidiropoulos, N. D., and Dimić, G. Z. Blind multiuser detection in W-CDMA systems with large delay spread. *IEEE Signal Processing Letters 8*, 3 (Mar. 2001), 87–89.

[197] Sidiropoulos, N. D., Giannakis, G. B., and Bro, R. Blind PARAFAC receivers for DS-CDMA systems. *IEEE Transactions on Signal Processing 48*, 3 (2000), 810–823.

[198] Singh, A. P., and Gordon, G. J. Relational learning via collective matrix factorization. In *Proc. of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2008).

[199] Singh, A. P., and Gordon, G. J. A unified view of matrix factorization models. In *Machine Learning and Knowledge Discovery in Databases*, vol. 5212 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008, pp. 358–373.

[200] Smilde, A. K., Bro, R., and Geladi, P. *Multi-Way Analysis with Applications in the Chemical Sciences*. Wiley, 2004.

[201] Smilde, A. K., van der Werf, M. J., Bijlsma, S., van der Werff-van der Vat, B. J. C., and Jellema, R. H. Fusion of mass spectrometry-based metabolomics data. *Analytical Chemistry 77*, 20 (2005), 6729–6736.

[202] Smilde, A. K., Westerhuis, J. A., and Boqué, R. Multiway multiblock component and covariates regression models. *Journal of Chemometrics 14*, 3 (2000), 301–331.

[203] Snyman, J. A., Laroche, J., and Adams, S. S. Practical bayesian optimization of machine learning algorithms. In *Neural Information Processing Systems* (2012).

[204] Sommese, A. J., and Wampler, C. W. *The Numerical Solution Of Systems Of Polynomials Arising In Engineering And Science*. World Scientific Press, Singapore, 2005.

[205] Sorber, L., Domanov, I., Van Barel, M., and De Lathauwer, L. Exact line and plane search for tensor optimization. ESAT-STADIUS Internal Report 13-02, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[206] Sorber, L., Van Barel, M., and De Lathauwer, L. Unconstrained optimization of real functions in complex variables. *SIAM Journal on Optimization 22*, 3 (2012), 879–898.

[207] Sorber, L., Van Barel, M., and De Lathauwer, L. Complex Optimization Toolbox v1.0. Available online at `http://esat.kuleuven.be/stadius/cot/`, Feb. 2013.

[208] Sorber, L., Van Barel, M., and De Lathauwer, L. Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms and a new generalization. *SIAM Journal on Optimization 23*, 2 (2013), 695–720.

[209] Sorber, L., Van Barel, M., and De Lathauwer, L. Structured data fusion. ESAT-STADIUS Internal Report 13-177, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[210] Sorber, L., Van Barel, M., and De Lathauwer, L. Numerical solution of bivariate and polyanalytic polynomial systems. *SIAM Journal on Numerical Analysis* (2014). Accepted.

[211] Sorber, L., Van Barel, M., and De Lathauwer, L. Tensorlab v2.0. Available online at `http://www.tensorlab.net`, Jan. 2014.

[212] Sorensen, M., and De Lathauwer, L. Array signal processing based on coupled tensor decompositions. ESAT-STADIUS Internal Report 13-241, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[213] Sorensen, M., and De Lathauwer, L. Coupled canonical polyadic decomposition — part I: Uniqueness. ESAT-STADIUS Internal Report 13-143, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[214] Sorensen, M., and De Lathauwer, L. Double coupled CPD with applications. ESAT-STADIUS Internal Report 13-242, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[215] Sorensen, M., and De Lathauwer, L. Multidimensional harmonic retrieval via coupled canonical polyadic decompositions. ESAT-STADIUS Internal Report 13-240, Department of Electrical Engineering (ESAT), KU Leuven, Leuven, Belgium, 2013.

[216] Squire, W., and Trapp, G. Using complex variables to estimate derivatives of real functions. *SIAM Review 10*, 1 (Mar. 1998), 110–112.

[217] Stegeman, A. Degeneracy in Candecomp/Parafac explained for $p \times p \times 2$ arrays of rank $p + 1$ or higher. *Psychometrika 71*, 3 (2006), 483–501.

[218] Stegeman, A. Low-rank approximation of generic $p \times q \times 2$ arrays and diverging components in the CANDECOMP/PARAFAC model. *SIAM Journal on Matrix Analysis and Applications 30*, 3 (2008), 988–1007.

[219] Steihaug, T. The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis 20*, 3 (June 1983), 626–637.

[220] Stelter, B. Netflix hits milestone and raises its sights. http://www.nytimes.com/2013/10/22/business/media/netflix-hits-subscriber-milestone-as-shares-soar.html, Oct. 2013.

[221] Stetter, H. J. Matrix eigenproblems are at the heart of polynomial system solving. *ACM SIGSAM Bulletin 30*, 4 (Dec. 1996), 22–25.

[222] Suárez, A. *Analysis and Design of Autonomous Microwave Circuits*, vol. 190 of *Wiley Series in Microwave and Optical Engineering*. Wiley-IEEE Press, Dec. 2009.

[223] Sullivan, D. Google: 100 billion searches per month, search to integrate Gmail, launching enhanced search app for iOS. http://searchengineland.com/google-search-press-129925, Apr. 2012.

[224] Suykens, J. A. K., Van Gestel, T., De Brabanter, J., De Moor, B. L. R., and Vandewalle, J. *Least Squares Support Vector Machines*. World Scientific Pub. Co., Singapore, 2002.

[225] Sylvester, J. J. Additions to the articles in the september number of this journal, "On a new class of theorems," and on Pascal's theorem. *The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science 37* (1850), 363–370.

[226] Tao, D., Li, X., Wu, X., Hu, W., and Maybank, J. Supervised tensor learning. *Knowledge and Information Systems 13*, 1 (2007), 1–42.

[227] Thai-Nghe, N., Drumond, L., Horváth, T., and Schmidt-Thieme, L. Multi-relational factorization models for predicting student performance. In *Proc. of the KDD Workshop on Knowledge Discovery in Educational Data* (2011).

[228] Tomasi, G. *Practical and computational aspects in chemometric data analysis*. PhD thesis, The Royal Veterinary and Agricultural University, May 2006.

[229] Tomasi, G., and Bro, R. A comparison of algorithms for fitting the PARAFAC model. *Computational Statistics and Data Analysis 50*, 7 (Apr. 2006), 1700–1734.

[230] Townsend, A., and Trefethen, L. N. An extension of Chebfun to two dimensions. Submitted, 2013.

[231] Tucker, L. R. The extension of factor analysis to three-dimensional matrices. In *Contributions to Mathematical Psychology*, H. Gulliksen and N. Frederiksen, Eds. Holt, Rinehart and Winston, New York, 1964, pp. 110–127.

[232] Tucker, L. R. Some mathematical notes on three-mode factor analysis. *Psychometrika 31*, 3 (1966), 279–311.

[233] van den Bos, A. Complex gradient and Hessian. *IEE Proceedings Vision, Image & Signal Processing 141*, 6 (Dec. 1994), 380–383.

[234] Van Mechelen, I., and Smilde, A. K. A generic linked-mode decomposition model for data fusion. *Chemometrics and Intelligent Laboratory Systems 104*, 1 (2010), 83–94.

[235] Vannieuwenhoven, N., Vanbaelen, N., Meerbergen, K., and Vandebril, R. The dense multiple-vector tensor-vector product: An initial study. Technical Report TW635, KU Leuven, 2013.

[236] Vasilescu, M. A. O., and Terzopoulos, D. Multilinear analysis of image ensembles: Tensorfaces. In *Proc. 7th European Conference on Computer Vision* (2002), vol. 2350 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 447–460.

[237] Verschelde, J. Algorithm 795: PHCpack: a general-purpose solver for polynomial systems by homotopy continuation. *ACM Transactions on Mathematical Software 25*, 2 (June 1999), 251–276.

[238] Verschelde, J., Verlinden, J., and Cools, R. Homotopies exploiting newton polytopes for solving sparse polynomial systems. *SIAM Journal on Numerical Analysis 31* (1994), 915–930.

[239] Wall, M. First-of-its-kind NASA satellite launches to track Earth weather, climate. `http://www.space.com/13435-nasa-npp-weather-climate-satellite-launch-wrap.html`, Oct. 2011.

[240] Wampler, C. W., and Sommese, A. J. Numerical algebraic geometry and algebraic kinematics. *Acta Numerica 20*, 1 (2011), 469–567.

[241] Ward, R. C. Balancing the generalized eigenvalue problem. *SIAM Journal on Statistical Computing 2*, 2 (June 1981), 141–152.

[242] Wilderjans, T., Ceulemans, E., and Van Mechelen, I. Simultaneous analysis of coupled data blocks differing in size: A comparison of two weighting schemes. *Computational Statistics and Data Analysis 53*, 4 (2009), 1086–1098.

[243] Wilkinson, J. H. The evaluation of the zeros of ill-conditioned polynomials. Part I. *Numerische Mathematik 1*, 1 (1959), 150–166.

[244] Wilkinson, J. H. Kronecker's canonical form and the QZ algorithm. *Linear Algebra and its Applications 28* (1979), 285–303.

[245] Wolfe, P. Convergence conditions for ascent methods. *SIAM Review 11*, 2 (Apr. 1969), 226–235.

[246] Xu, Q., Xiang, H., and Yang, Q. Protein-protein interaction prediction via collective matrix factorization. In *Proc. of the 2010 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (2010).

[247] Xu, Q., and Yang, Q. A survey of transfer and multitask learning in bioinformatics. *Journal of Computing Science and Engineering 5*, 3 (2011), 257–268.

[248] Yilmaz, Y. K., and Cemgil, A. T. Probabilistic latent tensor factorisation. In *Proc. of International Conference on Latent Variable analysis and Signal Separation* (2010).

[249] Yilmaz, Y. K., Cemgil, A. T., and Simsekli, U. Generalized coupled tensor factorization. In *Advances in Neural Information Processing Systems* (2011).

[250] Yoo, J., and Choi, S. Weighted nonnegative matrix co-tri-factorization for collaborative prediction. In *Advances in Machine Learning*, vol. 5828 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 396–411.

[251] Yoo, J., and Choi, S. Matrix co-factorization on compressed sensing. In *Proc. of the Twenty-Second International Joint Conference on Artificial Intelligence* (2011).

[252] Yu, K., Yu, S., and Tresp, V. Multi-label informed latent semantic indexing. In *Proc. of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2005).

[253] Zarzoso, V., and Comon, P. Optimal step-size constant modulus algorithm. *IEEE Transactions on Communications 56*, 1 (Jan. 2008), 10–13.

[254] Zhang, K., Gray, J. W., and Parvin, B. Sparse multitask regression for identifying common mechanism of response to therapeutic targets. *Bioinformatics 26*, 12 (2010), 97–105.

[255] Zheng, V. W., Cao, B., Zheng, Y., Xie, X., and Yang, Q. Collaborative filtering meets mobile recommendation: A user-centered approach. In *Proc. of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)* (2010).

[256] Zhou, K., and Roumeliotis, S. I. Multirobot active target tracking with combinations of relative observations. *IEEE Transactions on Robotics 27*, 4 (Aug. 2011), 678–695.

[257] Zhu, S., Yu, K., Chi, Y., and Gong, Y. Combining content and link for classification using matrix factorization. In *Proc. of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (2007).

[258] Zhu, Y., Chen, Y., Lu, Z., Pan, S. J., Xue, G.-R., Yu, Y., and Yang, Q. Heterogeneous transfer learning for image classification. In *Proc. of the 25th AAAI Conference on Artificial Intelligence* (2011).

[259] Zwysen, J., Jacqmaer, P., Gelagaev, R., and Driesen, J. An electromagnetic circuit simulator for power electronics. *IEEE Transactions on Magnetics 48*, 2 (2012), 799–802.

Curriculum vitae

Laurent Sorber was born on August 11, 1987 in Wilrijk, Belgium. In high school, he studied mathematics and sciences at the Onze-Lieve-Vrouw-van-Lourdescollege (OLVE), Edegem-Mortsel. He received a Bachelor of Science (B.Sc.) degree in Electrical Engineering and Computer Science from KU Leuven, Leuven in 2008, and a Master of Science (M.Sc.) degree in Mathematical Engineering from the same university in 2010. Later that year, he started his doctoral studies at the Department of Electrical Engineering and the Department of Computer Science under the supervision of prof. dr. ir. Lieven De Lathauwer and prof. dr. ir. Marc Van Barel, supported by a doctoral fellowship of the Flanders agency for Innovation by Science and Technology (IWT). On August 9, 2014 he will marry his fiancée Clara Mertens, whom he loves very much.

Publications

Journal papers

- Sorber, L., Van Barel, M., De Lathauwer, L. Structured data fusion. ESAT-STADIUS Internal Report 13-177, KU Leuven. Submitted.
- Sorber, L., Van Barel, M., De Lathauwer, L. Numerical solution of bivariate and polyanalytic polynomial systems. *SIAM Journal on Numerical Analysis*. Accepted.
- Sorber, L., Domanov, I., Van Barel, M., De Lathauwer, L. Exact line and plane search for tensor optimization. ESAT-STADIUS Internal Report 13-02, KU Leuven. Submitted.
- Sorber, L., Van Barel, M., De Lathauwer, L. Optimization-based algorithms for tensor decompositions: canonical polyadic decomposition, decomposition in rank-$(L_r, L_r, 1)$ terms and a new generalization. *SIAM Journal on Optimization 23*, 2 (2013), 695–720.
- Sorber, L., Van Barel, M., De Lathauwer, L. Unconstrained optimization of real functions in complex variables. *SIAM Journal on Optimization 22*, 3 (2012), 879–898.
- Le, T., Sorber, L., Van Barel, M. The Pythagoras number of real sum of squares polynomials and sum of square magnitudes of polynomials. *Calcolo 50*, 4 (2013), 283–303.
- Van Barel, M., Humet, M., Sorber, L. Approximating optimal point configurations for multivariate polynomial interpolation. *Electronic Transactions on Numerical Analysis*, 42 (2014), 41–63.
- Hunyadi, B., Camps, D., Sorber, L., Van Paesschen, W., De Vos, M., Van Huffel, S., De Lathauwer, L. Block term decomposition

for modelling epileptic seizures. ESAT-STADIUS Internal Report 14-04, KU Leuven. Submitted.

- Vervliet, N., Debals, O., Sorber, L., De Lathauwer, L. Breaking the curse of dimensionality using decompositions of incomplete tensors. ESAT-STADIUS Internal Report 13-169, KU Leuven. Submitted.

Abstracts

- Sorber, L., Van Barel, M., De Lathauwer, L. Structured data fusion. Householder Symposium XIX. Spa, Belgium, June 8-13 2014.
- De Lathauwer, L., Sorber, L., Sorensen, M., Van Barel, M. Coupled matrix/tensor decompositions: an introduction. Householder Symposium XIX. Spa, Belgium, June 8-13 2014.
- Sorber, L., Van Barel, M., De Lathauwer, L. Structured data fusion. SIAM Conference on Optimization. San Diego, California, USA, May 19-22 2014.
- Sorber, L., Sorensen, M., Van Barel, M., De Lathauwer, L. From tensor to coupled matrix/tensor decomposition. International Symposium on Mathematical Theory of Networks and Systems (MTNS). Groningen, The Netherlands, 7-11 July 2014.
- Sorber, L., Van Barel, M., De Lathauwer, L. Computation of tensor decompositions via numerical optimization. Workshop: numerical methods for high-dimensional problems. Marne-la-Vallée, France, 14-18 April 2014.
- Sorber, L., Van Barel, M., De Lathauwer, L. Exact line and plane search for tensor optimization by global minimization of bivariate polynomials and rational functions. Numerical Analysis and Scientific Computation with Applications. Calais, France, 24-26 June 2013.
- Sorber, L., Domanov, I., Van Barel, M., De Lathauwer, L. Exact line and plane search for tensor optimization by global minimization of bivariate polynomials and rational functions. SIAM Conference on Computational Science and Engineering. Boston, Massachusetts, USA, February 25-March 1 2013.
- Van Barel, M., Humet, M., Sorber, L. "Good" points for multivariate polynomial interpolation and approximation. TUM-IAS Workshop on Novel Numerical Methods. TUM-IAS Workshop on Novel Numerical

Methods. Institute for Advanced Study (TUM-IAS), Garching, Germany, 29-31 July 2013.

- Van Barel, M., Humet, M., Sorber, L. Structured matrices and multivariate orthogonal polynomials. Master Course in Applied Linear Algebra. Athens, Greece, 19 December 2012.
- Van Barel, M., Humet, M., Sorber, L. Structured matrices and multivariate orthogonal polynomials. Structured Matrix Computations in Non Euclidean Geometries: Algorithms and Applications. Luminy, France, 8-12 October 2012.
- Sorber, L., Van Barel, M., De Lathauwer, L. State of the art overview of algorithms for tensor decompositions. TRICAP. Bruges, Belgium, 2-7 June 2012.
- Sorber, L., Van Barel, M., De Lathauwer, L. State of the art overview of algorithms for tensor decompositions. WSC Spring Meeting. University of Antwerp, Antwerp, Belgium, 11 May 2012.
- Sorber, L., Van Barel, M., De Lathauwer, L. Optimization-based algorithms for the decomposition of a tensor in rank-$(L_r, L_r, 1)$ terms. Data Mining, Networks and Dynamics (Dagstuhl Seminar 11451): vol. 1 (11). Dagstuhl Seminar on Data Mining, Neworks and Dynamics. Schloss Dagstuhl, Wadern, Germany, 6-11 November 2011, 23-38.
- Le, T., Van Barel, M., Sorber, L. Low-rank representation of sum of squares polynomials and applications. International Congress on Computational and Applied Mathematics. Ghent, 9-12 July 2012.
- Sorber, L., Van Barel, M., De Lathauwer, L. Efficient algorithms for tensor decompositions. SIAM Conference on Applied Linear Algebra. Valencia, Spain, 18-22 June 2012.
- Sorber, L., Van Barel, M., De Lathauwer, L. Tensor decompositions by unconstrained optimization of real functions in complex variables. International conference on Matrix Methods in Mathematics and Applications. Moscow, Russia, 22-25 June 2011.
- Sorber, L., Van Barel, M., De Lathauwer, L. Optimization-based algorithms for the rank-$(L_r, L_r, 1)$ block term decomposition and related decompositions. International Linear Algebra Society. Braunschweig, Germany, 22-26 August 2011.
- Sorber, L., De Lathauwer, L. Algorithms for the rank-$(L_r, L_r, 1)$ block term decomposition. Tensor Decompositions and Applications. Monopoli, Italy, 13-17 September 2010.



FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
STADIUS
Kasteelpark Arenberg 10
BE-3001 Leuven
http://laurent.sorber.be