SoftwareX 21 (2023) 101292

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

# MatCoupLy: Learning coupled matrix factorizations with Python

Marie Roald [*]

*Department of Data Science and Knowledge Discovery, Simula Metropolitan Center for Digital Engineering, c/o OsloMet – Storbyuniversitetet Postboks 4 St. Olavs Plass, 0130 Oslo, Norway*
*Faculty of Technology, Art and Design, Oslo Metropolitan University, Norway*

ARTICLE INFO

*Article history:*
Received 17 August 2022
Received in revised form 6 December 2022
Accepted 8 December 2022

*Keywords:*
Coupled matrix factorization
PARAFAC2
Constraints
AO-ADMM

ABSTRACT

Coupled matrix factorization (CMF) models jointly decompose a collection of matrices with one shared mode. For interpretable decompositions, constraints are often needed, and variations of constrained CMF models have been used in various fields, including data mining, chemometrics and remote sensing. Although such models are broadly used, there is a lack of easy-to-use, documented, and open-source implementations for fitting CMFs with user-specified constraints on all modes. We address this need with MatCoupLy, a Python package that implements a state-of-the-art algorithm for CMF and PARAFAC2 that supports any proximable constraint on any mode. This paper outlines the functionality of MatCoupLy, including three examples demonstrating the flexibility and extendibility of the package.

## Code metadata

| | |
|---|---|
| Current code version | v0.1.5 |
| Permanent link to code/repository used for this code version | https://github.com/ElsevierSoftwareX/SOFTX-D-22-00245 |
| Permanent link to reproducible capsule | https://zenodo.org/record/7233180 |
| Legal code license | MIT License |
| Code versioning system used | Git |
| Software code languages, tools and services used | Python version 3.7 or greater |
| Compilation requirements, operating environments and dependencies | Main functionality: • NumPy • SciPy • TensorLy Optional dependencies for loading data: • Pandas • Requests • tqdm Optional dependencies for unit test generation: • PyTest Optional dependencies for total variation regularization: • condat_tv |
| If available, link to developer documentation/manual | https://matcouply.readthedocs.io/ |
| Support email for questions | mariero@simula.no |

## 1. Motivation and significance

Data mining is the discovery of patterns and valuable insight from data. MatCoupLy is a Python package for a type of data mining model called coupled matrix factorization (CMF). CMF models jointly factorize a collection of data matrices, $\left\{\mathbf{X}^{(i)}\right\}_{i=1}^{I}$, with the same number of columns (e.g. samples) but possibly different numbers of rows (e.g. features or time points), on the form

$$\mathbf{X}^{(i)} \approx \mathbf{B}^{(i)}\mathbf{D}^{(i)}\mathbf{C}^{\mathsf{T}},$$

where $\mathbf{C}$ is a factor matrix shared for all $\mathbf{X}^{(i)}$-matrices, and $\left\{\mathbf{B}^{(i)}\right\}_{i=1}^{I}$ is a collection of factor matrices, one for each $\mathbf{X}^{(i)}$.

[*] Correspondence to: Department of Data Science and Knowledge Discovery, Simula Metropolitan Center for Digital Engineering, c/o OsloMet – Storbyuniversitetet Postboks 4 St. Olavs Plass, 0130 Oslo, Norway.
*E-mail address:* mariero@simula.no.

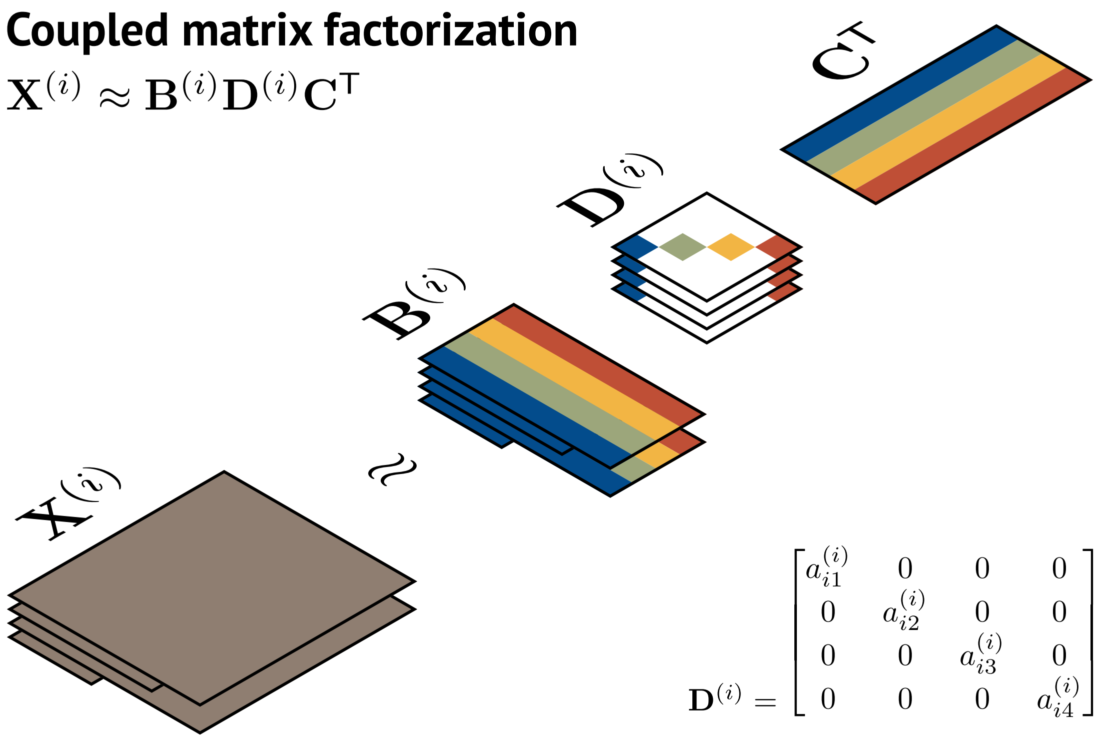
Coupled matrix factorization
$\mathbf{X}^{(i)} \approx \mathbf{B}^{(i)}\mathbf{D}^{(i)}\mathbf{C}^{\mathsf{T}}$
$\mathbf{C}^{\mathsf{T}}$
$\mathbf{D}^{(i)}$
$\mathbf{B}^{(i)}$
$\approx$
$\mathbf{X}^{(i)}$
$\mathbf{D}^{(i)} = \begin{bmatrix} a_{i1}^{(i)} & 0 & 0 & 0 \\ 0 & a_{i2}^{(i)} & 0 & 0 \\ 0 & 0 & a_{i3}^{(i)} & 0 \\ 0 & 0 & 0 & a_{i4}^{(i)} \end{bmatrix}$

**Fig. 1.** Illustration of a coupled matrix factorization.

The diagonal $\mathbf{D}^{(i)}$-matrices describe the signal strength of each component for each $\mathbf{X}^{(i)}$, and their diagonal entries are often collected into a single factor matrix, $\mathbf{A}$ (see Fig. 1 for an illustration). The columns of these factor matrices contain different types of patterns in the data. For example, if we, for each feature, $i$, have a time-by-sample matrix, $\mathbf{X}^{(i)}$, then the columns of $\mathbf{B}^{(i)}$ and $\mathbf{C}$ represent time courses for feature $i$ and sample-patterns, respectively, and $a_{ir}$ represents how prominent pattern $r$ is for feature $i$.

For the factor matrices to be unique and interpretable, it is often necessary to impose additional constraints. A prominent example of a constrained CMF model is PARAFAC2 [1,2], which uses a constant cross-product constraint (i.e. ${\mathbf{B}^{(i_1)}}^{\mathsf{T}}\mathbf{B}^{(i_1)} = {\mathbf{B}^{(i_2)}}^{\mathsf{T}}\mathbf{B}^{(i_2)} \forall i_1, i_2$) to achieve uniqueness under mild conditions [3]. PARAFAC2 has been successfully used to, e.g., extract phenotypes with time profiles that vary across patients from multi-patient health records [4], brain connectivity networks that vary across participants from multi-participant neuroimaging data [5], and student clusters with temporal resolutions that vary across sensor intrusiveness from multi-modal smartphone data [6]. Other examples of constrained CMF models are simultaneous non-negative matrix factorization, which has been used within systems biology [7,8]; muti-set multivariate curve resolution (multi-set MCR) [9], which extends the two-way MCR method used for analyzing a chemical sample (or chromatogram region) for simultaneously analyzing multiple experiments; coupled dictionary learning, which has been used in remote sensing with multi-source datasets [10]; and some variants of simultaneous component analysis, which has been used to analyze political questionnaires across countries [11].

Despite the wide use of CMF models, there is still a lack of software, in particular free and open-source software, to estimate constrained CMF models. The availability of free accessible software, such as scikit-learn [12] and PyTorch [13], has been essential for the rapid progress in machine learning research. Recently, TensorLy [14] has provided open-source software support to tensor decomposition models. However, there is no such software for constrained CMF models.

Several packages support matrix factorization with a single data matrix. For example, scikit-learn includes matrix decomposition models such as non-negative matrix factorization, principal component analysis, and dictionary learning [12]. pyMCR [15] implements matrix factorization with various constraints useful for performing MCR. However, these libraries do not support the joint factorization of multiple data matrices and are therefore limited in terms of finding shared patterns from related measurements. The Prince library supports jointly analyzing multiple data matrices with multiple factor analysis, which adapts PCA for multiple datasets [16]. However, this method is specific to PCA and cannot be used for other constrained CMF models. TensorLy can analyze multiple data matrices with PARAFAC2 but with the direct fitting algorithm [2], which uses the reformulation $\mathbf{B}^{(i)} = \mathbf{P}^{(i)}\mathbf{\Delta_B}$ with ${\mathbf{P}^{(i)}}^{\mathsf{T}}\mathbf{P}^{(i)} = \mathbf{I}$ to impose the constant cross-product constraint. However, this reformulation hinders the use of additional constraints for the $\mathbf{B}^{(i)}$ matrices, and TensorLy only supports non-negativity constraints for the other modes [14]. Thus, there is a need for easy-to-use, documented, and open-source software for fitting constrained CMF models that support flexible constraints.

The contribution of this work is to introduce the MatCoupLy software package, which addresses this need by building on top of TensorLy and implementing CMF fitted using alternating optimization with the alternating direction method of multipliers (AO-ADMM) [17,18]. Specifically, MatCoupLy solve non-convex optimization problems on the form

$$\min_{\{\mathbf{B}^{(i)},\mathbf{D}^{(i)}\}_{i=1}^{I},\mathbf{C}} \sum_{i=1}^{I} \left\{ \left\| \mathbf{B}^{(i)}\mathbf{D}^{(i)}\mathbf{C}^{\mathsf{T}} - \mathbf{X}^{(i)} \right\|_F^2 + \sum_{n=1}^{N_\mathbf{B}} g_{\mathbf{B}^{(i)}}^{(n)}\left(\mathbf{B}^{(i)}\right) + \sum_{n=1}^{N_\mathbf{A}} g_{\mathbf{D}^{(i)}}^{(n)}\left(\mathbf{D}^{(i)}\right) \right\} + \sum_{n=1}^{N_\mathbf{C}} g_{\mathbf{C}}^{(n)}\left(\mathbf{C}\right),$$

where $\left\| \mathbf{B}^{(i)}\mathbf{D}^{(i)}\mathbf{C}^{\mathsf{T}} - \mathbf{X}^{(i)} \right\|_F^2$ is the sum of squared errors for the $\mathbf{X}^{(i)}$ data matrix, and the $g_\star^{(n)}$-functions represent the $n$th regularization for the different factor matrices. Since the software

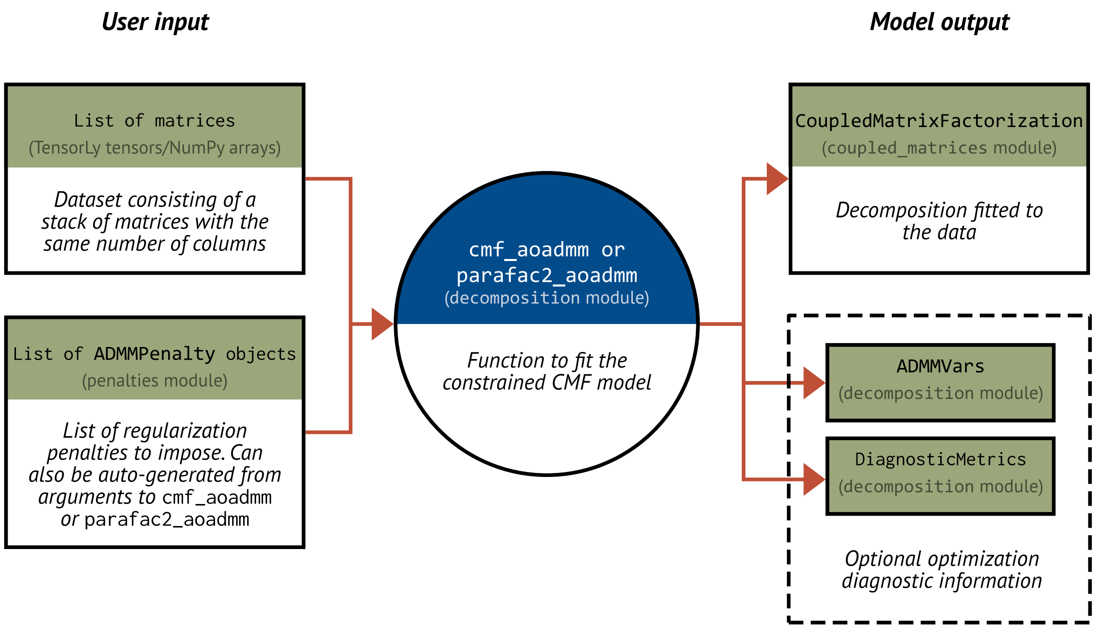
User input
Model output
List of matrices
(TensorLy tensors/NumPy arrays)
Dataset consisting of a stack of matrices with the same number of columns
List of ADMMPenalty objects
(penalties module)
List of regularization penalties to impose. Can also be auto-generated from arguments to cmf_aoadmm or parafac2_aoadmm
cmf_aoadmm or parafac2_aoadmm
(decomposition module)
Function to fit the constrained CMF model
CoupledMatrixFactorization
(coupled_matrices module)
Decomposition fitted to the data
ADMMVars
(decomposition module)
DiagnosticMetrics
(decomposition module)
Optional optimization diagnostic information

**Fig. 2.** Diagram showing an overview of the MatCoupLy API.

uses AO-ADMM, it supports any regularization penalty whose proximal operator, given by

$$\operatorname{prox}_{g_\star^{(n)}}(\mathbf{x}) = \min_{\mathbf{y}} g_\star^{(n)}(\mathbf{y}) + \frac{1}{2}\|\mathbf{x}-\mathbf{y}\|^2,$$

can be evaluated efficiently. For more details on the AO-ADMM algorithm implemented in MatCoupLy, we refer the reader to [18].

This software was developed as part of ongoing research on data mining with temporal data and constrained PARAFAC2 models. It uses the algorithm introduced in [18,19] and is based on the code that enabled these works.

## 2. Software description

MatCoupLy extends the TensorLy API with similar classes for decompositions and functions for fitting the models, supporting both the TensorLy NumPy and PyTorch backend. In the following sections, we outline the overarching software architecture and describe the modules and additional software features that most users of MatCoupLy will interface with.

### 2.1. Software architecture

Fig. 2 shows an overview of the software architecture. A user has a collection of data matrices they want to factorize simultaneously. The user then defines an appropriate set of constraints or regularization penalties by creating `ADMMPenalty` instances from the `penalties` module, either by using one of the many built-in penalty classes or by creating a custom penalty class. For ease of use, the built-in penalties can also be specified using the arguments of the decomposition function. Next, the dataset is factorized using either the `cmf_aoadmm` function (for a general CMF model) or the `parafac2_aoadmm` function (for PARAFAC2). These functions return a `CoupledMatrixFactorization` instance, which is a container for the extracted $\mathbf{A}$, $\mathbf{B}^{(i)}$ and $\mathbf{C}$ factor matrices. The decomposition functions can also return optional optimization diagnostics information in the form of two `NamedTuples`: `ADMMVars` (for auxiliary- and scaled dual variables for the inner ADMM iterations) and `DiagnosticMetrics` (for general optimization diagnostics, such as loss and feasibility gaps).

### 2.2. Software functionalities

***The* `coupled_matrices` *module:*** The `coupled_matrices` module provides functionality to interact with CMFs. CMFs can either be stored as a two-tuple containing a weight vector and a list of factor matrices or as a separate `CoupledMatrixFactorization` object. All functionality for forming dense datasets from factorizations is provided as both functions in the `coupled_matrices` module (for use with two-tuples) and methods of the `CoupledMatrixFactorization` class.

***The* `penalties` *module:*** The `penalties` module contains functionality for imposing constraints and regularization on CMFs. We refer to both hard constraints and regularization penalties as `ADMMPenalty` objects and divide them into three subsets: row-wise penalties, matrix-wise penalties, and multi-matrix penalties. Row-wise penalties can be directly imposed on all modes; matrix penalties can always be imposed on $\left\{\mathbf{B}^{(i)}\right\}_{i=1}^{I}$ and $\mathbf{C}$, but only on $\mathbf{A}$ if a constant feasibility penalty is imposed; and the multi-matrix penalties can only be imposed on $\left\{\mathbf{B}^{(i)}\right\}_{i=1}^{I}$. Each penalty type inherits from an abstract base class, which ensures that all penalties implement all needed functionality and that implementing new penalties requires a minimal amount of new code. E.g., if a user wants to implement a row-wise penalty, they only need to inherit from `RowVectorPenalty` and implement the `factor_matrix_row_update` and `penalty` methods, while the matrix-wise and multi-matrix updates are automatically generated. The `penalties` module also includes a wide variety of built-in `ADMMPenalty` objects, listed in Table 1.

***The* `decomposition` *module:*** All functionality for decomposing datasets is contained in the `decomposition` module. The user will generally interact with the `cmf_aoadmm` and the `parafac2_aoadmm` functions, which provide an easy interface to decompose datasets. To make it straightforward to impose multiple constraints, both functions support two ways to input constraints:

1. Each built-in penalty type has a corresponding function argument. MatCoupLy will automatically parse these, combine compatible penalties (e.g., non-negativity and unimodality) and create a minimal set of `ADMMPenalty` objects with sensible default values, thus increasing the efficiency of the algorithm.

**Table 1**
Overview of the constraints implemented in MatCoupLy v0.1.5.

| Penalty | Class name | Penalty type |
|---|---|---|
| $\iota_{NN}(\mathbf{x}) = \begin{cases} 0, & \text{if } x_i \geq 0 \, \forall x_i \in \mathbf{x} \\ \infty, & \text{otherwise} \end{cases}$ | `NonNegativity` | Row penalty |
| $\iota_{\text{box}(l,h)}(\mathbf{x}) = \begin{cases} 0, & \text{if } l \leq x_i \leq h \, \forall x_i \in \mathbf{x} \\ \infty, & \text{otherwise} \end{cases}$ | `Box` | Row penalty |
| $\Vert\mathbf{x}\Vert_1 = \sum_n \vert x_n \vert$ | `L1Penalty` | Row penalty |
| $\Vert\mathbf{M}\Vert_{\mathbf{L}} = \text{Tr}(\mathbf{M}^\mathsf{T}\mathbf{L}\mathbf{M})$ | `GeneralizedL2Penalty` | Matrix penalty |
| $\Vert\mathbf{M}\Vert_{\text{TV}} = \sum_r \sum_n \vert m_{n+1,r} - m_{n,r} \vert$ | `TotalVariationPenalty` | Matrix penalty |
| $\iota_{\mathcal{B}(R)}(\mathbf{M}) = \begin{cases} 0, & \text{if } \Vert\mathbf{m}\Vert_2 < R \text{ for all columns} \\ \infty, & \text{otherwise} \end{cases}$ | `L2Ball` | Matrix penalty |
| $\iota_{\Delta}(\mathbf{M}) = \begin{cases} 0, & \text{if } m_{n,r} \geq 0 \, \forall n, r \text{ and } \sum_n m_{n,r} = 1 \, \forall r \\ \infty, & \text{otherwise} \end{cases}$ | `UnitSimplex` | Matrix penalty |
| $\iota_{\mathcal{U}}(\mathbf{M}) = \begin{cases} 0, & \text{if all column vectors of } \mathbf{M} \text{ are unimodal} \\ \infty, & \text{otherwise} \end{cases}$ | `Unimodality` | Matrix penalty |
| $\iota_{\text{PF2}}\left(\left\{\mathbf{B}^{(i)}\right\}_{i=1}^{I}\right) = \begin{cases} 0, & \text{if } \mathbf{B}^{(i_1)\mathsf{T}}\mathbf{B}^{(i_1)} = \mathbf{B}^{(i_2)\mathsf{T}}\mathbf{B}^{(i_2)} \, \forall i_1, i_2 \\ \infty, & \text{otherwise} \end{cases}$ | `Parafac2` | Multi-matrix penalty |

2. If a user has created their own `ADMMPenalty` class or wants to change the default behavior (e.g., initialization of auxiliary variables), they can provide `ADMMPenalty` instances directly through the `regs`-argument.

***Automatic test suite:*** MatCoupLy is built following best software practices with automatic tests and continuous delivery. The extensive test suite contains both unit- and integration tests and covers 99% of all code statements. Additionally, it provides functionality for automatically creating a minimal number of unit tests for penalties. To create tests for a new penalty, a software user only needs to create a test class inheriting from `BaseTestFactorRowPenalty`, `BaseTestFactorMatrixPenalty`, or `BaseTestFactorMatricesPenalty` and implement functionality for generating data invariant to the proximal operator (i.e., $\text{prox}(x) = x$) and data not invariant to the proximal operator (i.e., $\text{prox}\,(x) \neq x$).

***Extensive documentation:*** MatCoupLy also includes extensive documentation. The documentation contains a primer on CMFs, API documentation, and extensive examples covering useful topics such as analysis of real and simulated data and how to extend MatCoupLy with custom penalty functions.

## 3. Illustrative examples

This section contains three examples, one of which demonstrates how the AO-ADMM algorithm can be used to extract insights from a dataset with bike-sharing data. This dataset was prepared for this publication and is published with an open license together with the code. The second example demonstrates how easy it is to create a new `ADMMPenalty` and how to use MatCoupLy's automatic unit test generation. Finally, the third example compares MatCoupLy with other related libraries. The examples in this section are abbreviated versions of three examples from the documentation.

### *3.1. Non-negative PARAFAC2 analysis on bike-sharing data*

Here, we consider a dataset consisting of two years of bike-sharing data from three major cities in Norway: Oslo, Bergen, and Trondheim. The dataset is organized as three `arrival_station_id` $\times$ `time` matrices where each city shares the same temporal profiles. The time is converted from UTC to local (CET) time. The $(j, k)$ entry of the data matrix for one city represents the number of trips that ended in station $j$ at time-point $k$.

Listing 1 contains code to load the dataset, fit a four-component non-negative PARAFAC2 model, and Fig. 3 shows a visualization of the components. The non-negative PARAFAC2 model clusters together four different types of bike trips (sorted in order of signal strength):

- biking home, which is mainly active at the end of the workday during weekdays;
- biking to work, which is mainly active at the start of the workday during weekdays;
- general trips, which are active during the whole day;
- leisure trips, which are mainly active during the summer and weekends.

The bike station components represent the areas across cities where people bike to work, home from work, and to leisurely activities. For example, for the leisure trip component, we see activation at Huk and Sukkerbiten in Oslo and Nordnes in Bergen, which are stations next to popular bathing places. Trondheim does not have any bike stations at large outdoor bathing places, but here we see activity at Lade idrettsanlegg, which is a popular sporting arena for children and within walking distance from a popular bathing spot.

### *3.2. Implementing an `ADMMPenalty`*

To implement an `ADMMPenalty`, we only need to inherit from the correct penalty class and implement the corresponding proximal operator and penalty function. In this example, we create a penalty class for the hard constraint with unimodality on all component vectors except one (the last), which is unconstrained. The code to create this penalty class and impose it when fitting a PARAFAC2 model is shown in Listing 2. To create a test for this penalty, we only need to inherit from `BaseTestFactorMatrixPenalty` and `MixinTestHardConstraint` and implement the `get_invariant_matrix` and `get_non_invariant_matrix` methods. The test code and output from running pytest are shown in Listing 3.

```python
import matplotlib.pyplot as plt
import matcouply.decomposition as decomposition
from matcouply.data import get_bike_data

bike_data = get_bike_data()
matrices = [bike_data["oslo"].values, bike_data["bergen"].values,
bike_data["trondheim"].values]
cmf, diagnostics = decomposition.parafac2_aoadmm(
    matrices,
    rank=4,
    non_negative=True,
    n_iter_max=1000,
    tol=1e-8,
    verbose=-1,  # Print only summary
    return_errors=True,
    random_state=0,
)
```
**Output:**
```
All regularization penalties (including regs list):
* Mode 0:
   - <'matcouply.penalties.NonNegativity' with aux_init='random_uniform', dual_init='random_uniform')>
* Mode 1:
   - <'matcouply.penalties.Parafac2' with svd='truncated_svd', update_basis_matrices=True,
update_coordinate_matrix=True, n_iter=1, aux_init='random_uniform', dual_init='random_uniform')>
   - <'matcouply.penalties.NonNegativity' with aux_init='random_uniform', dual_init='random_uniform')>
* Mode 2:
   - <'matcouply.penalties.NonNegativity' with aux_init='random_uniform', dual_init='random_uniform')>
converged in 221 iterations: FEASIBILITY GAP CRITERION AND RELATIVE LOSS CRITERION SATISFIED
```

*Listing 1: Code and output for analyzing the bike-sharing dataset.*

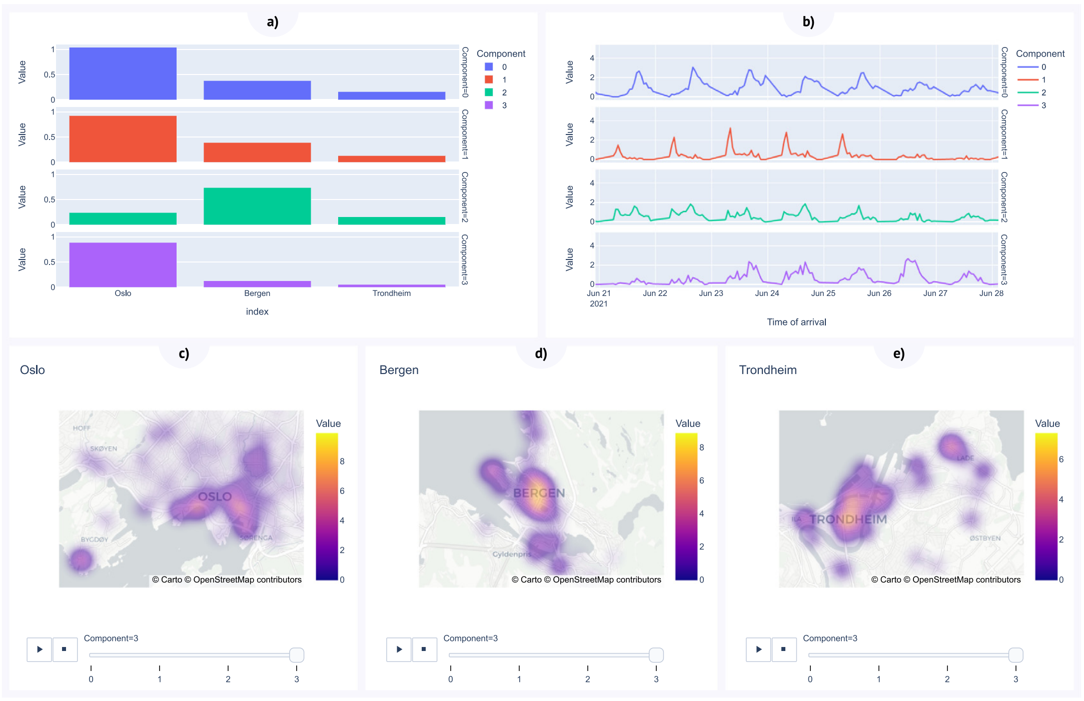

**Fig. 3.** Components for the bike sharing data. (a) shows the component strength for the different cities, (b) shows a zoomed in plot of one week of the temporal profile for the different components and (c)–(e) shows the spatial distribution of the leisure trip component (component 3) for the different cities. Instructions on how to make interactive versions of these plots are given in the documentation.

### *3.3. Evaluating constrained CMF models on simulated data*

To illustrate the advantage of MatCoupLy, we compared the performance of non-negative PARAFAC2 (N-PARAFAC2) from MatCoupLy, PARAFAC2 from TensorLy [14] (with non-negativity on **A** and **C**) and non-negative matrix factorization (NMF) from scikit-learn [12] in terms of recovering simulated components from noisy data. We simulated six components inspired by gas chromatography–mass spectrometry (GC–MS) data. The entries of the **A**-matrix were drawn from a uniform distribution between

```python
import tensorly as tl
# Decorator that makes it possible to inherit docstrings
from matcouply._doc_utils import copy_ancestor_docstring
# The unimodal regression implementation
from matcouply._unimodal_regression import unimodal_regression
from matcouply.penalties import HardConstraintMixin, MatrixPenalty, NonNegativity
from matcouply.decomposition

class UnimodalAllExceptLast(HardConstraintMixin, MatrixPenalty):
    def __init__(
        self,
        non_negativity=False,
        aux_init="random_uniform",
        dual_init="random_uniform",
    ):
        super().__init__(aux_init, dual_init)
        self.non_negativity = non_negativity

    @copy_ancestor_docstring
    def factor_matrix_update(self, factor_matrix, feasibility_penalty, aux):
        new_factor_matrix = tl.copy(factor_matrix)
        new_factor_matrix[:, :-1] = unimodal_regression(
            factor_matrix[:, :-1], non_negativity=self.non_negativity
        )

        if self.non_negativity:
            new_factor_matrix = tl.clip(new_factor_matrix, 0)
        return new_factor_matrix

# […] code to load data
cmf, diagnostics = parafac2_aoadmm(
    data_matrices,
    rank,
    n_iter_max=1000,
    regs=[
        [NonNegativity()],
        [UnimodalAllExceptLast(non_negativity=True)],
        [NonNegativity()]
    ],
    return_errors=True,
    random_state=init,
    verbose=True,
)
```

*Listing 2: Implementation of a custom `ADMMPenalty` that imposes unimodality on all component vectors except the last and a code snippet for using this penalty. Here, we impose this new constraint with the `regs` keyword argument. Since `UnimodalAllExceptLast` is a hard constraint, its penalty function is 0, so we use the `HardConstraintMixin` instead of implementing it.*

0 and 1, $\mathcal{U}(0, 1)$, and the entries of the **C**-matrix were drawn from a truncated normal distribution. Five out of the six $\mathbf{B}^{(i)}$-components were Gaussians whose average shifted for each $i$, and the entries of the final $\mathbf{B}^{(i)}$-component were drawn from $\mathcal{U}(0.5, 1.5)$ for each $i$.

Following this setup, we generated 20 simulated datasets, and added noise following

$$\mathbf{X}^{(i)}_{\text{noisy}} = \max\left(0, \mathbf{X}^{(i)} + \boldsymbol{\eta}^{(i)}\right),$$

where $\eta^{(i)}_{jk}$ was normally distributed and scaled so $\sqrt{\sum_{ijk} {\eta^{(i)}_{jk}}^2} = 0.1\sqrt{\sum_{ijk} {x^{(i)}_{jk}}^2}$. The values of $\mathbf{X}^{(i)}_{\text{noisy}}$ were truncated since the NMF function in scikit-learn only supports non-negative data. We used five random seeds for each fitting algorithm, selecting the initialization with the lowest loss among the feasible solutions.

Since the NMF model is a matrix factorization model, we converted the true factorization as well as the ones obtained with N-PARAFAC2 and PARAFAC2 to equivalent matrix factorization models by constructing a new data matrix, **X**, and factor matrix, $\tilde{\mathbf{B}}$, given by

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}^{(1)} \\ \mathbf{X}^{(2)} \\ \vdots \\ \mathbf{X}^{(I)} \end{bmatrix} \quad \text{and} \quad \tilde{\mathbf{B}} = \begin{bmatrix} \mathbf{B}^{(1)}\mathbf{D}^{(1)} \\ \mathbf{B}^{(2)}\mathbf{D}^{(2)} \\ \vdots \\ \mathbf{B}^{(I)}\mathbf{D}^{(I)} \end{bmatrix},$$

```
import numpy as np
import scipy.stats as stats
import tensorly as tl

from matcouply.testing import MixinTestHardConstraint, BaseTestFactorMatrixPenalty

class TestUnimodalAllExceptLast(
    MixinTestHardConstraint, BaseTestFactorMatrixPenalty
):
    PenaltyType = UnimodalAllExceptLast
    penalty_default_kwargs = {}
    min_rows = 3
    min_columns = 2

    def get_invariant_matrix(self, rng, shape):
        matrix = tl.zeros(shape)
        I, J = shape
        t = np.linspace(-10, 10, I)
        for j in range(J-1):
            sigma = rng.uniform(0.5, 1)
            mu = rng.uniform(-5, 5)
            matrix[:, j] = stats.norm.pdf(t, loc=mu, scale=sigma)
        matrix[:, J-1] = rng.uniform(size=I)
        return matrix

    def get_non_invariant_matrix(self, rng, shape):
        # There are at least 3 rows
        M = rng.uniform(size=shape)
        M[1, :-1] = -1  # M is positive, so setting the second element to -1 makes it
impossible for it to be unimodal
        return M
```

```
Output:
============================== test session starts ===============================
platform win32 -- Python 3.9.13, pytest-7.1.2, pluggy-1.0.0
Using—randomly-seed=2943073154
rootdir: C:\Users\marie\Programming\softwarex-paper, configfile: pyproject.toml
plugins: anyio-3.6.1, cov-3.0.0, randomly-3.12.0
collected 65 items

test_unimodal_except_last.py  [ 70%]
...................                                                         [100%]

=============================== 65 passed in 0.81s ===============================
```

Listing 3: Test code for the `UnimodalAllExceptLast` penalty and output obtained from running pytest.

respectively. Then, we computed a factor match score (FMS) for these matrix factorization models, given by

$$\mathrm{FMS}\left(\left(\tilde{\mathbf{B}}, \mathbf{C}\right), \left(\widehat{\mathbf{B}}, \hat{\mathbf{C}}\right)\right) = \frac{1}{R}\sum_{r=1}^{R} \frac{\tilde{\mathbf{b}}_r^{\mathsf{T}} \widehat{\mathbf{b}}_r}{\left\|\tilde{\mathbf{b}}_r\right\| \left\|\widehat{\mathbf{b}}_r\right\|} \frac{{\mathbf{c}_r}^{\mathsf{T}} \hat{\mathbf{c}}_r}{\|\mathbf{c}_r\| \left\|\hat{\mathbf{c}}_r\right\|},$$

where the hat represents the estimated factor matrices. To compute the FMS, we used the implementation in TensorLy-Viz [20].

Fig. 4 shows a boxplot of the FMS, where we see that N-PARAFAC2 outperformed both NMF and PARAFAC2. To ensure that the difference was significant, we used a paired nonparametric Wilcoxon signed-rank test implemented in SciPy [21], and obtained p-values $2.4 \times 10^{-5}$ (N-PARAFAC2 more accurate than PARAFAC2) and $2.0 \times 10^{-4}$ (N-PARAFAC2 more accurate than NMF). Thus, we see that by leveraging MatCoupLy's ability to combine constraints, we improve the recovery of the components.

## 4. Impact

MatCoupLy is a further development of the codebase used for two publications [18,19] and is used in an ongoing research project on blind source separation from hyperspectral data. The easy-to-use framework enables researchers to easily fit CMF models with constraints adapted to their specific needs. Proximal operators for several penalties are implemented in the package, including unimodality and the PARAFAC2 constraint, which are quite technical and are not found in any well-tested open-source Python package. Moreover, the flexibility of the package facilitates rapid prototyping of new constraints and regularization penalties with automatic unit test generation, which gives immediate feedback on the correctness of the implementations, speeding up the workflow of researchers using CMFs and PARAFAC2.

Finally, we note that while constrained PARAFAC2 models have gained increased interest in recent years [4,22,23], there is

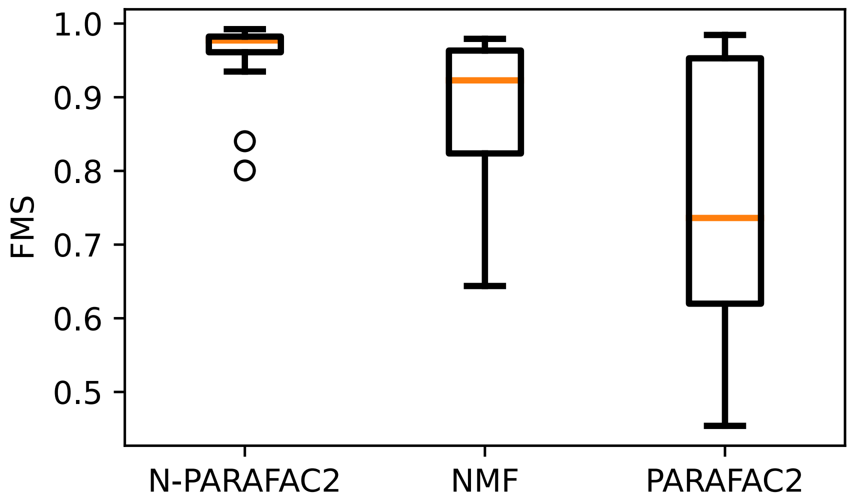

**Fig. 4.** Boxplot showing the FMS for the simulated datasets for each model evaluated in Section 3.3.

still a lack of accessible, easy-to-use, and open-source packages to fit such models.[1] MatCoupLy fills this gap, providing a straightforward and well-tested interface to the only available method for fitting PARAFAC2 models that supports proximable regularization on all factor matrices [18], thus facilitating the pursuit of new research questions within various fields, such as data mining, neuroscience, chemometrics, and remote sensing.

## 5. Conclusions

This paper introduces MatCoupLy, a package for fitting constrained CMF models with AO-ADMM in Python. MatCoupLy's flexibility makes it effortless to use a variety of constraints and regularization methods. Moreover, the extendible nature of MatCoupLy enables users to easily implement custom constraints or regularization methods in a well-tested environment. Thus, MatCoupLy provides researchers across different fields with a streamlined tool to extract insight from their datasets with CMF methods without needing detailed knowledge about non-convex optimization. Future work could improve MatCoupLy's flexibility further by implementing more constraints and regularization penalties, adding other loss functions (e.g., the Kullback–Leibler (KL)-divergence and cross-entropy loss) and adding support for decomposing datasets with missing data or incomplete matrices.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

All datasets can be loaded with functions in the software package.

## Acknowledgments

MatCoupLy was developed during a research visit to the Norwegian University of Life Sciences. I would like to thank Oliver Tomic, Kristian Hovde Liland, and Cecilia Marie Futsæther for valuable discussions and feedback during the development of this package.

[1] Despite requests for such functionality, see for example https://github.com/tensorly/tensorly/issues/104#issuecomment-561239139.

## References

[1] Harshman RA. PARAFAC2: Mathematical and technical notes. UCLA Work Pap Phonetics 1972;22:30–44.
[2] Kiers HAL, Ten Berge JMF, Bro R. PARAFAC2—Part, I. A direct fitting algorithm for the PARAFAC2 model. J Chemom 1999;13:275–94. http://dx.doi.org/10.1002/(SICI)1099-128X(199905/08)13:3/4%3C275::AID-CEM543%3E3.0.CO;2-B.
[3] Harshman RA, Lundy ME. Uniqueness proof for a family of models sharing features of Tucker's three-mode factor analysis and PARAFAC/CANDECOMP. Psychometrika 1996;61:133–54. http://dx.doi.org/10.1007/BF02296963.
[4] Afshar A, Perros I, Papalexakis EE, Searles E, Ho J, Sun J. COPA: Constrained PARAFAC2 for sparse & large datasets. In: Proc 27th ACM int conf inf knowl manag. 2018, p. 793–802. http://dx.doi.org/10.1145/3269206.3271775.
[5] Madsen KH, Churchill NW, Mørup M. Quantifying functional connectivity in multi-subject fMRI data using component models. Hum Brain Mapp 2017;38:882–99. http://dx.doi.org/10.1002/hbm.23425.
[6] Devineni P, Papalexakis EE, Michalska K, Faloutsos M. MIMiS: Minimally intrusive mining of smartphone user behaviors. In: IEEE/ACM int conf adv soc netw anal min. 2018, p. 568–9.
[7] Lee CM, Mudaliar MAV, Haggart DR, Wolf CR, Miele G, Vass JK, et al. Simultaneous non-negative matrix factorization for multiple large scale gene expression datasets in toxicology. PLoS One 2012;(7):1–21. http://dx.doi.org/10.1371/journal.pone.0048238.
[8] Badea L. Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization. In: Pac symp biocomput. 2008, p. 279–90.
[9] Ruckebusch C, Blanchet L. Multivariate curve resolution: A review of advanced and tailored applications and challenges. Anal Chimica Acta 2013;765:28–36. http://dx.doi.org/10.1016/j.aca.2012.12.028.
[10] Gong M, Zhang P, Su L, Liu J. Coupled dictionary learning for change detection from multisource data. IEEE Trans Geosci Remote Sens 2016;54:7077–91. http://dx.doi.org/10.1109/TGRS.2016.2594952.
[11] Kiers HAL, ten Berge JMF. Hierarchical relations between methods for simultaneous component analysis and a technique for rotation to a simple simultaneous structure. Br J Math Stat Psychol 1994;47:109–26. http://dx.doi.org/10.1111/j.2044-8317.1994.tb01027.x.
[12] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[13] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. PyTorch: An imperative style, high-performance deep learning library. In: Proc 32nd adv neural inf process syst. Curran Associates, Inc.; 2019, p. 8024–35.
[14] Kossaifi J, Panagakis Y, Anandkumar A, Pantic M. TensorLy: Tensor learning in Python. J Mach Learn Res 2019;20:1–6.
[15] Camp Jr CH. PyMCR: A Python library for multivariatecurve resolution analysis with alternating regression (MCR-AR). J Res National Inst Stand Technol 2019;124:1–10. http://dx.doi.org/10.6028/jres.124.018.
[16] Halford M. Prince (version 0.7.1). [Software]. 2021, Available from: https://github.com/MaxHalford/prince.
[17] Huang K, Sidiropoulos ND, Liavas AP. A flexible and efficient algorithmic framework for constrained matrix and tensor factorization. IEEE Trans Signal Process 2016;64(19):5052–65. http://dx.doi.org/10.1109/TSP.2016.2576427.

[18] Roald M, Schenker C, Calhoun V, Adali T, Bro R, Cohen JE, et al. An AO-ADMM approach to constraining PARAFAC2 on all modes. SIAM J Math Data Sci 2022;4(3):1191–222. http://dx.doi.org/10.1137/21M1450033.

[19] Roald M, Schenker C, Cohen JE, Acar E. PARAFAC2 AO-ADMM: Constraints in all modes. In: Proc 29th Eur signal process conf. 2021, http://dx.doi.org/10.23919/EUSIPCO54536.2021.9615927.

[20] Roald M, Moe YM. TLViz: Visualising and analysing tensor decomposition models with Python. J Open Source Softw 2022;7(79):4754. http://dx.doi.org/10.21105/joss.04754.

[21] Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, et al. SciPy 1.0: Fundamental algorithms for scientific computing in Python. Nature Methods 2020;17(3):261–72. http://dx.doi.org/10.1038/s41592-019-0686-2.

[22] Helwig NE. Estimating latent trends in multivariate longitudinal data via PARAFAC2 with functional and structural constraints. Biometrical J 2017;59(4):783–803. http://dx.doi.org/10.1002/bimj.201600045.

[23] Cohen JE, Bro R. Nonnegative PARAFAC2: A flexible coupling approach. In: Proc int conf latent var anal signal sep. 2018, p. 89–98. http://dx.doi.org/10.1007/978-3-319-93764-9_9.