BlockTensorDecompositions.jl: A Unified Constrained Tensor Decomposition Julia Package

Nicholas J. E. Richardson
Department of Mathematics

Noah Marusenko
Department of Computer Science

Michael P. Friedlander
Departments of Mathematics and Computer Science

Table of contents

1 Introduction

- Tenors are useful in many applications
- Need tools for fast and efficient decompositions

For the scientific user, it would be most useful for there to be a single piece of software that can take as input 1) any reasonable type of factorization model and 2) constraints on the individual factors, and produce a factorization. Details like what rank to select, how the constraints should be enforced, and convergence criteria should be handled automatically, but customizable to the knowledgable user. These are the core specification for BlockTensorDecompositions.jl.

1.1 Related tools

- Packages within Julia
- Other languages
- Hint at why I developed this

Beyond the external usefulness already mentioned, this package offers a playground for fair comparisons of different parameters and options for performing tensor factorizations across various decomposition models. There exist packages for working with tensors in languages like Python (TensorFlow [1], PyTorch [2], and TensorLy [3]), MATLAB (Tensor Toolbox [4]), R (rTensor [5]), and Julia (TensorKit.jl [6], Tullio.jl [7], OMEinsum.jl [8], and TensorDecompositions.jl [9]). But they only provide a groundwork for basic manipulation of tensors and the most common tensor decomposition models and algorithms, and are not equipped to handle arbitrary user defined constraints and factorization models.

Some progress towards building a unified framework has been made [10–12]. But these approaches don't operate on the high dimensional tensor data natively and rely on matricizations of the problem, or only consider nonnegative constraints. They also don't provide an all-in-one package for executing their frameworks.

1.2 Contributions

- Fast and flexible tensor decomposition package
- Framework for creating and performing custom
  - tensor decompositions
  - constrained factorization (the what)
  - iterative updates (the how)
- Implement new "tricks"
  - a (Lipschitz) matrix step size for efficient sub-block updates
  - multi-scaled factorization when tensor entries are discretizations of a continuous function

  - partial projection and rescaling to enforce linear constraints (rather than Euclidean projection)
- ?? rank detection ??

The main contribution is a description of a fast and flexible tensor decomposition package, along with a public implementation written in Julia: BlockTensorDecompositions.jl. This package provides a framework for creating and performing custom tensor decompositions. To the author's knowledge, it is the first package to provide automatic factorization to a large class of constrained tensor decompositions problems, as well as a framework for implementing new constraints and iterative algorithms. This paper also describes three new techniques not found in the literature that empirically convergence faster than traditional block-coordinate descent.

2 Tensor Decompositions

- the math section of the paper

This section reviews the notation used throughout the paper and commonly used tensor decompositions.

2.1 Notation

- tensor notation, use MATLAB notation for indexing so subscripts can be used for a sequence of tensors

2.1.a Sets

The set of real number is denoted as $\mathbb{R}$ and its restrictions to nonnegative numbers is denoted as $\mathbb{R}_+ = \mathbb{R}_{\geq 0} = \{x \in \mathbb{R} \mid x \geq 0\}$.

We use $[N] = \{1, 2, ..., N\} = \{n\}_{n=1}^{N}$ to denote integers from 1 to N.

Usually, lower case symbols will be used for the running index, and the capitalized letter will be the maximum letter it runs to. This leads to the convenient shorthand $i \in [I]$, $j \in [J]$, etc.

We use a capital delta Δ to denote sets of vectors or higher order tensors where the slices or fibres along a specified dimension sum to 1, i.e. generalized simplexes.

Usually, we use script letters ($\mathcal{A}, \mathcal{B}, \mathcal{C}$, etc.) for other sets.

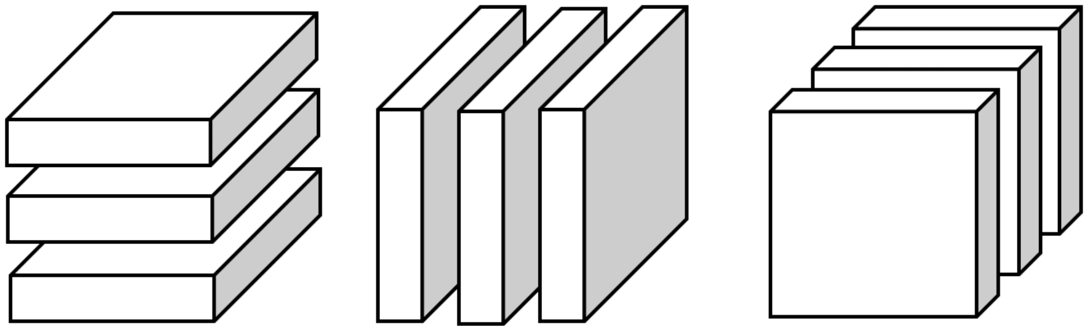
2.1.b Vectors, Matrices, and Tensors

Vectors are denoted with lowercase letters (x, y, etc.), and matrices and higher order tensors with uppercase letters (commonly A, B, C and X, Y, Z). The order of a tensor is the number of axes it has. We would call vectors "order-1" or "1st order" tensors, and matrices "order-2" or "2nd order" tensors.

To avoid confusion between entries of a vector/matrix/tensor and indexing a list of objects, we use square brackets to denote the former, and subscripts to denote the later. For example, the entry in the ith row and jth column of a matrix $A \in \mathbb{R}$ is $A[i, j]$. This follows MATLAB/Julia notation where `A[i,j]` points to the entry $A[i, j]$. We contrast this with a list of I objects being denoted as $a_1, ..., a_I$, or more compactly, $\{a_i\}$ when it is clear the index $i \in [I]$.

The transpose $A^\top \in \mathbb{R}^{J\times I}$ of a matrix $A \in \mathbb{R}^{I\times J}$ flips entries along the main diagonal: $A^\top[j,i] = A[i,j]$. In Julia, the transpose of a matrix is typed with a single apostrophe `A'`.

The n-slices, nth mode slices, or mode n slices of an Nth order tensor A are notated with the slice $A[:, ..., :, i_n, :, ..., :]$. For a 3rd order tensor A, the 1st, 2nd, and 3rd mode slices $A[i, :, :]$, $A[:, j, :]$, and $A[:, :, k]$ have special names and are called the horizontal, lateral, and frontal slices and are displayed in Figure 1. In Julia, the 1-, 2-, and 3-slices of a third order array `A` would be `eachslice(A, dims=1)`, `eachslice(A, dims=2)`, and `eachslice(A, dims=3)`.



(a) horizontal slices *A[i, :, :]* (b) lateral slices *A[:, j, :]* (c) frontal slices *A[:, :, k]*

Figure 1: Slices of an order 3 tensor A.

The n-fibres, nth mode fibres, or mode n fibres of an Nth order tensor A are denoted $A[i_1, ..., i_{n-1}, :, i_{n+1}, ..., i_N]$. For example, the 1-fibres of a matrix M are the column vectors $M[:, j]$, and the 2-fibres are the row vectors $M[i, :]$. For order-3 tensors, the 1st, 2nd, and 3rd mode fibres $A[:, j, k]$, $A[i, :, :]$, and $A[i, j, :]$ are called the vertical/column, horizontal/row, and depth/tube fibres respectively and are displayed in Figure 2. Natively in Julia, the 1-, 2-, and 3-fibres of a third order array `A` would be `eachslice(A, dims=(2,3))`, `eachslice(A, dims=(1,3))`, and `eachslice(A, dims=(1,2))`. BlockTensorDecomposition.jl defines the function `eachfibre(A; n)` to do exactly this. For example, the 1-fibres of an array `A` would be `eachfibre(A, n=1)`.

For matrices, the 1-fibres are the same as the 2-slices (and vice versa), but for Nth order tensors in general, fibres are always vectors, whereas n-slices are $(N-1)$th order tensors.

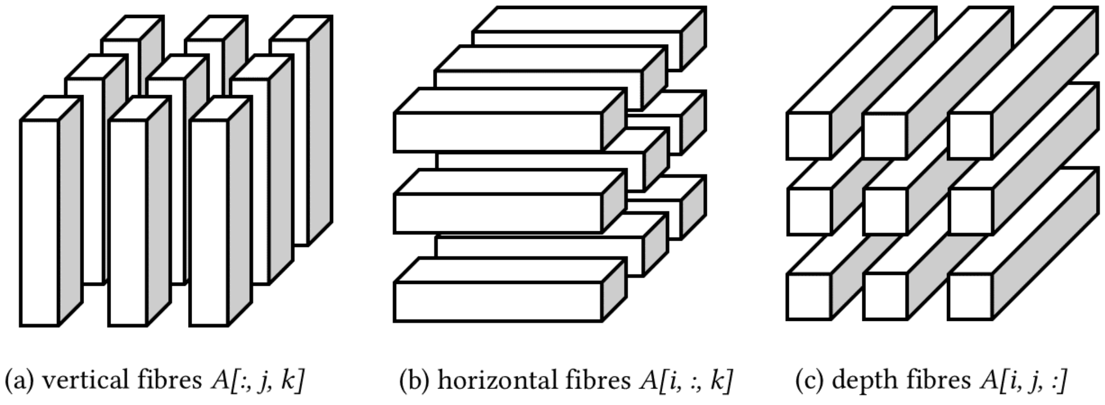

(a) vertical fibres A[:, j, k]
(b) horizontal fibres A[i, :, k]
(c) depth fibres A[i, j, :]

Figure 2: Fibres of an order 3 tensor A.

Since we commonly use I as the size of a tensor's dimension, we use id_I to denote the identity tensor of size I (of the appropriate order). When the order is 2, id_I is an $I \times I$ matrix with ones along the main diagonal, and zeros elsewhere. For higher orders N, this is an $\underbrace{I \times \dots \times I}_{N \text{ times}}$ tensor where $\mathrm{id}_I[i_1, ..., i_N] = 1$ when $i_1 = ... = i_N \in [I]$, and is zero otherwise.

BlockTensorDecomposition.jl defines `identity_tensor(I, ndims)` to construct id_I.

For a vector, matrix, or tensor filled with ones, we use $\mathbb{1} \in \mathbb{R}^{I_1 \times \dots \times I_N}$. This can be constructed in Julia with `ones(I₁, ..., Iₙ)`.

2.1.c Products of Tensors

Definition 2.1: The outer product $\otimes$ between two tensors $A \in \mathbb{R}^{I_1 \times \dots \times I_M}$ and $B \in \mathbb{R}^{J_1 \times \dots \times J_N}$ yields an order $M + N$ tensor $A \otimes B \in \mathbb{R}^{I_1 \times \dots \times I_M \times J_1 \times \dots \times J_N}$ that is entry-wise

$$(A \otimes B)[i_1, ..., i_M, j_1, ..., j_N] = A[i_1, ..., i_M]B[j_1, ..., j_N].$$

TODO Define in BlockTensorDecomposition.jl

The Frobenius inner product between two tensors $A, B \in \mathbb{R}^{I_1 \times \dots \times I_N}$ yields a real number $A \cdot B \in \mathbb{R}$ and is defined as

$$\langle A, B \rangle = A \cdot B = \sum_{i_1=1}^{I_1} \dots \sum_{i_N=1}^{I_N} A[i_1, ..., i_N]B[i_1, ..., i_N].$$

Julia's standard library package LinearAlgebra implements the Frobenius inner product with `dot(A, B)` or `A ⋅ B`.

The n-slice dot product $\cdot_n$ between two tensors $A \in \mathbb{R}^{K_1, ..., K_{n-1}, I, K_{n+1}, ..., K_N}$ and $B \in \mathbb{R}^{K_1, ..., K_{n-1}, J, K_{n+1}, ..., K_N}$ returns a matrix $(A \cdot_n B) \in \mathbb{R}^{I \times J}$ with entries

$$(A \cdot_n B)[i,j] = \sum_{k_1 \dots k_{n-1} k_{n+1} \dots k_N} A[k_1, \dots, k_{n-1}, i, k_{n+1}, \dots, k_N] B[k_1, \dots, k_{n-1}, j, k_{n+1}, \dots, k_N].$$

This product can also be thought of as taking the dot product $(A \cdot_n B)[i,j] = A_i \cdot B_j$ between all pairs of nth order slices of A and B, which exactly how BlockTensorDecomposition.jl defines the operation.

```julia
function slicewise_dot(A::AbstractArray, B::AbstractArray; dims=1)
    C = zeros(size(A, dims), size(B, dims))
    if A === B # use faster routine if they are the same
        return _slicewise_self_dot!(C, A; dims)
    end

    for (i, A_slice) in enumerate(eachslice(A; dims))
        for (j, B_slice) in enumerate(eachslice(B; dims))
            C[i, j] = A_slice ⋅ B_slice
        end
    end
    return C
end

function _slicewise_self_dot!(C, A; dims=1)
    enumerated_A_slices = enumerate(eachslice(A; dims))
    for (i, Ai_slice) in enumerated_A_slices
        for (j, Aj_slice) in enumerated_A_slices
            if i > j
                continue
            else # only compute the upper triangle entries of C
                C[i, j] = Ai_slice ⋅ Aj_slice
            end
        end
    end
    return Symmetric(C) # indexing C[2,1] points to the entry in C[1,2]
end
```

BlockTensorDecomposition.jl defines this operation with `slicewise_dot(A, B, n)`. In the special case where $A = B$, a more efficient method that only computes entries where $i \leq j$ is defined since $A \cdot_n A$ is a symmetric matrix.

The n-slice product of a tensor with itself $X \cdot_n X$ should be thought of as a generalization of the Gram matrix $X^\top X$ since it considers the matrix generated by taking the dot product between every nth mode slice, just like how the Gram matrix considers the dot product between every pair of columns.

The n-mode product $\times_n$ between a tensor $A \in \mathbb{R}^{I_1 \times \dots \times I_N}$ and matrix $B \in \mathbb{R}^{I_n \times J}$, returns a tensor $(A \times_n B) \in \mathbb{R}^{I_1 \times \dots \times I_{n-1} \times J \times I_{n+1} \times \dots \times I_N}$ with entries

$$(A \times_n B)[i_1, ..., i_{n-1}, j, i_{n+1}, ..., i_N] = \sum_{i_n=1}^{I_n} A[i_1, ..., i_{n-1}, i_n, i_{n+1}, ..., i_N] B[i_n, j].$$

BlockTensorDecomposition.jl defines this operation with `nmode_product(A, B, n)`.

```julia
function nmode_product(A::AbstractArray, B::AbstractMatrix, n::Integer)
    # convert the problem to the mode-1 product
    Aperm = swapdims(A, n)
    Cperm = Aperm ×₁ B
    return swapdims(Cperm, n) # swap back
end

function ×₁(A::AbstractArray, B::AbstractMatrix)
    # Turn the 1-mode product into matrix-matrix multiplication
    sizeA = size(A)
    Amat = reshape(A, sizeA[1], :)

    # Initialize the output tensor
    C = zeros(size(B, 1), sizeA[2:end]...)
    Cmat = reshape(C, size(B, 1), prod(sizeA[2:end]))

    # Perform matrix-matrix multiplication Cmat = B*Amat
    mul!(Cmat, B, Amat)

    return C # Output entries of Cmat in tensor form
end

function swapdims(A::AbstractArray, a::Integer, b::Integer=1)
    # Construct a permutation where a and b are swapped
    # e.g. [4, 2, 3, 1, 5, 6] when a=4 and b=1
    dims = collect(1:ndims(A))
    dims[a] = b; dims[b] = a
    return permutedims(A, dims)
end
```

! Note

If we were only working with a fixed order of tensors, we could have defined `×₁` entry-wise with `Tullio.jl`. The function definition `tullio×₁` below gives an example for order three tensors.

```julia
function tullio×₁(A::AbstractArray{_,3}, B::AbstractMatrix)
  @tullio C[i, j, k] := A[r, j, k] * B[i, r]
  return C
end
```

But we would need a new definition for each ordered tensor, or use Julia's meta programming to write a method for each order at runtime.

The n-mode product and n-slice product can be thought of as opposites of each other. The n-mode product sums over just the nth dimension of the first tensor, whereas the n-slice product sums over all but the nth dimension.

We can extend the n-mode product to sum over multiple indices between two tensors.

The multi-mode product $\times_{1,\dots,n} = \times_{1:n} = \times_{[n]}$ between a tensor $A \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and tensor $B \in \mathbb{R}^{I_1 \times \cdots \times I_n}$, returns a tensor $\left(A \times_{[n]} B\right) \in \mathbb{R}^{I_{n+1} \times \cdots \times I_N}$ with entries

$$\left(A \times_{[n]} B\right)[i_{n+1}, \dots, i_N] = \sum_{i_1,\dots,i_n} A[i_1, \dots, i_n, i_{n+1}, \dots, i_N] B[i_1, \dots, i_n].$$

This product contracts the first n indexes of A with every index of B.

More generally, we can contract any number of indexes such as the last n indexes of A with every index of B with $\times_{N-n+1,\dots,N} = \times_{(N-n+1):N} = \times_{[-n]}$,

$$\left(A \times_{[-n]} B\right)[i_1, \dots, i_n] = \sum_{i_{n+1},\dots,i_N} A[i_1, \dots, i_{N-n}, i_{N-n+1}, \dots, i_N] B[i_{N-n+1}, \dots, i_N],$$

or specific indexes. For example, we would define $\left(A \times_{1,3,5} B\right) \in \mathbb{R}^{I_2 \times I_4 \times I_6}$ where $A \in \mathbb{R}^{I_1 \times \cdots \times I_6}$ and $B \in \mathbb{R}^{I_1 \times I_3 \times I_5}$ to be

$$\left(A \times_{1,3,5} B\right)[i_2, i_4, i_6] = \sum_{i_1,i_3,i_5} A[i_1, i_2, i_3, i_4, i_5, i_6] B[i_1, i_3, i_5].$$

When A a *half-symmetric*[1] tensor of order $2N$

$$A[i_1, \dots, i_N, i_{N+1}, \dots, i_{2N}] = A[i_{N+1}, \dots, i_{2N}, i_1, \dots, i_N], \tag{1}$$

we have

[1]For example, the Hessian of a scalar function (see Definition 2.3).

$$A \times_{[N]} B = A \times_{[-N]} B$$

for tensors B of order N.

2.1.d Gradients, Norms, and Lipschitz Constants

Definition 2.2 (Gradient): The gradient $\nabla f : \mathbb{R}^{I_1 \times \cdots \times I_N} \to \mathbb{R}^{I_1 \times \cdots \times I_N}$ of a (differentiable) function $f : \mathbb{R}^{I_1 \times \cdots \times I_N} \to \mathbb{R}$ is defined entry-wise for a tensor $A \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ by

$$\nabla f(A)[i_1, ..., i_N] = \frac{\partial f}{\partial A[i_1, ..., i_N]}(A).$$

Definition 2.3 (Hessian): The Hessian $\nabla^2 f : \mathbb{R}^{I_1 \times \cdots \times I_N} \to \mathbb{R}^{(I_1 \times \cdots \times I_N)^2}$ of a second-differentiable function $f : \mathbb{R}^{I_1 \times \cdots \times I_N} \to \mathbb{R}$ is the gradient of the gradient and is defined for a tensor $A \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ entry-wise by

$$\nabla^2 f(A)[i_1, ..., i_N, j_1, ..., j_N] = \frac{\partial^2 f}{\partial A[i_1, ..., i_N] \partial A[j_1, ..., j_N]}(A).$$

For a tensor input A of order N, the Hessian tensor $\nabla^2 f(A)$ is of order $2N$.

See Section 8.1 for how this definition can be reproduced by performing two gradients $\nabla^2 f = \nabla(\nabla f)$.

Definition 2.4: The Frobenius norm of a tensor A is the square root of its dot product with itself

$$\|A\|_F = \sqrt{\langle A, A \rangle}.$$

For vectors v, this is equivalent to the (Euclidean) 2-norm

$$\|v\|_F = \|v\|_2 = \sqrt{\langle v, v \rangle}.$$

For matrices M, the ($2 \to 2$) operator norm is defined as

$$\|M\|_{\text{op}} = \sup_{\|v\|_2 = 1} \|Mv\|_2 = \sigma_1(M)$$

where $\sigma_1(M)$ is the largest singular value of M.

For tensors T, the operator-norm is ambiguous since there are multiple ways we can treat tensors as function on other tensors. There is a canonical way to do this for vectors $x \mapsto v^\top x$ and matrices

$x \mapsto Mx$, but not tensors. In the case of the Hessian tensor $\nabla^2 f(A) \in \mathbb{R}^{(I_1 \times \cdots \times I_N)^2}$ evaluated at $A \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, it is natural to consider the function $X \mapsto \nabla^2 f(A) \times_{[N]} X$ for $X \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. This gives us our definition of the operator norm on tensors.

Definition 2.5 (Operator Norm): The operator norm of a half-symmetric tensor $A \in \mathbb{R}^{(I_1 \times \cdots \times I_N)^2}$ (Equation 1) is defined as

$$\|A\|_{\mathrm{op}} = \sup_{\|X\|_F = 1} \left\| A \times_{[N]} X \right\|_F. \tag{2}$$

In Equation 2, $X \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. Note that this definition agrees with the usual operator norm on matrices when $N = 1$.

Theorem 2.1 (Norm of an outer product): Let $T = A_1 \otimes \cdots \otimes A_N \in \mathbb{R}^{(I_1 \times \cdots \times I_N)^2}$ where $A_n \in \mathbb{R}^{I_n \times I_n}$ are symmetric matrices.

Then T is half-symmetric and

$$\|T\|_{\mathrm{op}} = \prod_{n=1}^{N} \|A_n\|_{\mathrm{op}}.$$

⚠ Warning

According to how the outer product $\otimes$ is defined in Definition 2.1, the product $A_1 \otimes \cdots \otimes A_N$ shown in Theorem 2.1 is really an element of $\mathbb{R}^{I_1 \times I_1 \times \cdots \times I_N \times I_N}$. Note how the indexes are ordered differently than an element of $\mathbb{R}^{(I_1 \times \cdots \times I_N)^2} = \mathbb{R}^{I_1 \times \cdots \times I_N \times I_1 \times \cdots \times I_N}$. Correcting for this with explicit notation becomes cumbersome and would require tensor transposes, a new definition of an outer product, or reordering of indexes in the definition of a half-symmetric tensor. These can have knock-on effects to the definition of the Hessian, multi-mode product, and the operator norm.

To avoid the headache, the equality

$$T = A_1 \otimes \cdots \otimes A_N$$

in Theorem 2.1 should be thought of as the following entry-wise equation

$$T[i_1, \ldots, i_N, j_1, \ldots, j_N] = A_1[i_1, j_1] \cdots A_N[i_N, j_N]. \tag{3}$$

With the outer product understood as Equation 3, the results of Theorem 2.1 that T is half-symmetric and its operator norm is the product of the operator norms of the constituent matrices is true.

Definition 2.6 (Lipschitz Function): A function $g : \mathbb{R}^{I_1\times\cdots\times I_N} \to \mathbb{R}^{I_1\times\cdots\times I_N}$ is L-Lipschitz when

$$\|g(A) - g(B)\|_F \leq L\|A - B\|_F, \quad \forall A, B \in \mathbb{R}^{I_1\times\cdots\times I_N}.$$

We call the smallest such L *the* Lipschitz constant of g.

Definition 2.7 (Smooth Function): A differentiable function $f : \mathbb{R}^{I_1\times\cdots\times I_N} \to \mathbb{R}$ is L-smooth when its gradient $g = \nabla f$ is L-Lipschitz.

Theorem 2.2 (Quadratic Smoothness): Let $f : \mathbb{R}^{I_1\times\cdots\times I_N} \to \mathbb{R}$ be a quadratic function

$$f(X) = \frac{1}{2}A(X, X) + B(X) + C$$

of X with bilinear function $A : \left(\mathbb{R}^{I_1\times\cdots\times I_N}\right)^2 \to \mathbb{R}$, linear function $B : \mathbb{R}^{I_1\times\cdots\times I_N} \to \mathbb{R}$, and constant $C \in \mathbb{R}^{I_1\times\cdots\times I_N}$.

Then:

1. the Hessian $\nabla^2 f$ is a constant function that evaluates to $\nabla^2 f(X) = D$ at every point X for some $D \in \mathbb{R}^{(I_1\times\cdots\times I_N)^2}$,
2. the tensor D only depends on the bilinear function A (and not on B and C), and
3. the quadratic function f is L-smooth with constant

$$L = \|D\|_{\text{op}}.$$

2.2 Common Decompositions

- Extensions of PCA/ICA/NMF to higher dimensions
- talk about the most popular Tucker, Tucker-n, CP
- other decompositions
  - high order SVD (see Kolda and Bader)
  - HOSVD (see Kolda, Shifted power method for computing tensor eigenpairs)

A tensor decomposition is a factorization of a tensor into multiple (usually smaller) tensors, that can be recombined into the original tensor. To make a common interface for decompositions, we make an abstract subtype of Julia's `AbstractArray`, and subtype `AbstractDecomposition` for our concrete tensor decompositions.

```
abstract type AbstractDecomposition{T, N} <: AbstractArray{T, N} end
```

Computationally, we can think of a generic decomposition as storing factors $(A, B, C, ...)$ and operations $(\times_a, \times_b, ...)$ for combining them. This is what we do in BlockTensorDecomposition.jl.

```
struct GenericDecomposition{T, N} <: AbstractDecomposition{T, N}
    factors::Tuple{Vararg{AbstractArray{T}}} # e.g. (A, B, C)
    contractions::Tuple{Vararg{Function}} # e.g. (×₁, ×₂)
end
# Y = A ×₁ B ×₂ C
array(G::GenericDecomposition) = multifoldl(contractions(G), factors(G))
```

The function `multifoldl` applies the given operations between each factor, from left to right.

```
function multifoldl(ops, args)
    @assert (length(ops) + 1) == length(args)
    x, xs... = args
    for (op, arg) in zip(ops, xs)
        x = op(x, arg)
    end
    return x
end
```

Different types of decompositions define different operations, and different "ranks" of the same decomposition specific the sizes of the factors used.

A commonly used family of decompositions can be derived from the Tucker decomposition.

Definition 2.8: A rank-$(R_1, ..., R_N)$ Tucker decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ produces N matrices $A_n \in \mathbb{R}^{I_n \times R_n}$, $n \in [N]$, and core tensor $B \in \mathbb{R}^{R_1 \times \cdots \times R_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r_1=1}^{R_1} \cdots \sum_{r_N=1}^{R_N} A_1[i_1, r_1] \cdots A_r[i_N, r_N] B[r_1, ..., r_N] \tag{4}$$

entry-wise. More compactly, this decomposition can be written using the n-mode product, or with double brackets

$$Y = B \times_1 A_1 \times_2 \ldots \times_N A_N = B \bigtimes_n A_n = [\![B; A_1, ..., A_N]\!]. \tag{5}$$

The *Tucker Product* defined by Equation 5 is implemented in BlockTensorDecomposition.jl with `tuckerproduct(B, (A1, ..., AN))` and computes

$$B \bigtimes_n A_n = [\![B; A_1, ..., A_N]\!].$$

It can also optionally “exclude” one of the matrix factors with the call `tuckerproduct(B, (A1, ..., AN); exclude=n)` to compute

$$B \bigtimes_{m \neq n} A_m = [\![B; A_1, ..., A_{n-1}, \mathrm{id}_{R_n}, A_{n+1}, ..., A_N]\!].$$

```julia
function tuckerproduct(core, matrices; exclude=nothing)
    N = ndims(core)
    if isnothing(exclude)
        return multifoldl(tucker_contractions(N), (core, matrices...))
    else
        return multifoldl(getnotindex(tucker_contractions(N), exclude), (core, 
getnotindex(matrices, exclude)...))
    end
end

tucker_contractions(N) = Tuple((B, A) -> nmode_product(B, A, n) for n in 1:N)
```

Sometimes we write $A_0 = B$ to ease notation, and suggest the “zeroth” factor of the tucker decomposition is the core tensor B. In the special case when $N = 3$, we can visualize Tucker decomposition as multiplying the core tensor by matrices on all three sides as shown in Figure 3.

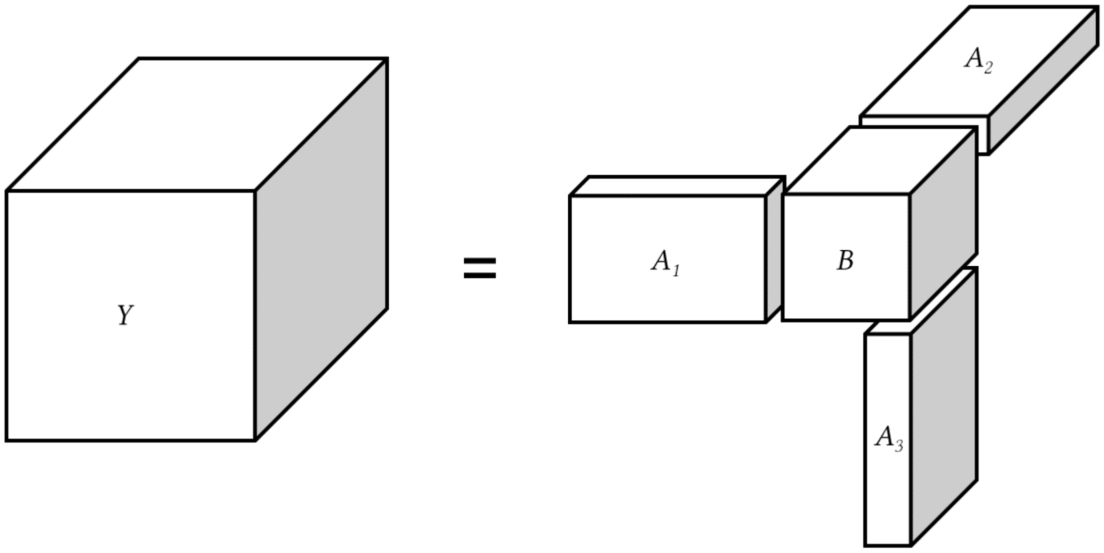


Figure 3: Tucker factorization of a 3rd order tensor Y.

Setting all the matrices of a Tucker decomposition to the identity matrix but the first gives the Tucker-1 decomposition.

Definition 2.9: A rank-R Tucker-1 decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ produces a matrix $A \in \mathbb{R}^{I_1 \times R}$, and core tensor $B \in \mathbb{R}^{R \times I_2 \times \cdots \times I_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r=1}^{R} A[i_1, r] B[r, i_2, ..., i_N] \tag{6}$$

entry-wise or more compactly,

$$Y = AB = B \times_1 A = [\![B; A]\!].$$

Note we extend the usual definition of matrix-matrix multiplication

$$(AB)[i, j] = \sum_{r=1}^{R} A[i, r] B[r, j]$$

to tensors B in the compact notation for Tucker-1 decomposition $Y = AB$.

More generally, any number of matrices can be set to the identity matrix giving the Tucker-n decomposition.

Definition 2.10: A rank-$(R_1, ..., R_n)$ Tucker-n decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ produces n matrices $A_1, ..., A_n$, and core tensor $B \in \mathbb{R}^{R_1 \times \cdots \times R_n \times I_{n+1} \times \cdots \times I_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r_1=1}^{R_1} ... \sum_{r_N=1}^{R_n} A_1[i_1, r_1] \cdots A_n[i_N, r_n] B[r_1, ..., r_n, i_{n+1}, ..., i_N] \tag{7}$$

entry-wise, or compactly written in the following three ways,

$$\begin{aligned} &Y = B \times_1 A_1 \times_2 ... \times_n A_n \times_{n+1} \mathrm{id}_{I_{n+1}} \times_{n+2} ... \times_N \mathrm{id}_{I_N} \\ &Y = B \times_1 A_1 \times_2 ... \times_n A_n \\ &Y = [\![B; A_1, ..., A_n]\!]. \end{aligned}$$

Lastly, if we set the core tensor B to the identity tensor id_R, we obtain the **can**onical **decomp**osition/**para**llel **fac**tors model (CANDECOMP/PARAFAC or CP for short).

Definition 2.11: A rank-R CP decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ produces N matrices $A_n \in \mathbb{R}^{I_n \times R}$, such that

$$Y[i_1, ..., i_N] = \sum_{r=1}^{R} A_1[i_1, r] \cdots A_r[i_N, r] \tag{8}$$

entry-wise. More compactly, this decomposition can be written using the n-mode product, or with double brackets

$$Y = \mathrm{id}_R \times_1 A_1 \times_2 ... \times_N A_N = \mathrm{id}_R \bigtimes_n A_n = [\![A_1, ..., A_N]\!].$$

Note CP decomposition is sometimes referred to as Kruskal decomposition, and requires the core only be diagonal (and not necessarily identity) and the factors A_n have normalized columns $\|A_n[:, r]\|_2 = 1$.

Other factorization models are used that combine aspects of CP and Tucker decomposition [13], are specialized for order 3 tensors [14, 15], or provide alternate decomposition models entirely like tensor-trains [16]. But the (full) Tucker, and its special cases Tucker-n, and CP decomposition are most commonly used extensions of the low-rank matrix factorization to tensors. These factorizations are summarized in Table 1.

Table 1: Summary of common tensor factorizations. Here, N is the order of the factorized tensor.

Name	Bracket Notation	n-mode Product	Entry-wise
Tucker	$[\![A_0; A_1, ..., A_N]\!]$	$A_0 \times_1 A_1 \times_2 ... \times_N A_N$	Equation 4
Tucker-1	$[\![A_0; A_1]\!]$	$A_0 \times_1 A_1$	Equation 6
Tucker-n	$[\![A_0; A_1, ..., A_n]\!]$	$A_0 \times_1 A_1 \times_2 ... \times_n A_n$	Equation 7
CP	$[\![A_1, ..., A_N]\!]$	$\mathrm{id}_R \times_1 A_1 \times_2 ... \times_N A_N$	Equation 8

TODO add discussion on other decompositions - high order SVD (see Kolda and Bader) - HOSVD (see Kolda, Shifted power method for computing tensor eigenpairs)

Tensor decompositions are not nessisarily unique. It should be clear that scaling one factor by $x \neq 0$ and dividing another by x yields the same original tensor. Furthermore, fibres and slices can be permuted without affecting the the original tensor. Up to these manipulations, for a fixed rank, there exist criteria that ensures their decompositions are unique [13, 17, 18].

2.2.a Representing Tucker Decompositions

There are implemented in BlockTensorDecomposition.jl and can be called, for a third order tensor, with `Tucker((B, A₁, A₂, A₃))`, `Tucker1((B, A₁))`, and `CPDecomposition((A₁, A₂, A₃))`. These Julia `structs` store the tensor in its factored form. We could define the contractions for these types and use the common interface provided by `array`, but it turns out we can reconstruct the whole tensor more efficiently. If the recombined tensor or particular entries are requested, Julia dispatches on the type of decomposition and calls a particular method of `array` or `getindex`. The implementations for efficient array construction and index access are provided below.

```
array(T::Tucker) = multifoldl(tucker_contractions(ndims(T)), factors(T))
tucker_contractions(N) = Tuple((G, A) -> nmode_product(G, A, n) for n in 1:N)
```

TODO add getindex method for Tucker type

```
function array(T::Tucker1)
    B, A = factors(T)
    return B ×₁ A
end

function getindex(T::Tucker1, I::Vararg{Int})
    B, A = factors(T)
    i, J... = I # (i, J) = (I[1], I[begin+1:end])
    return (@view A[i, :]) ⋅ view(B, :, J...)
end
```

```
array(CPD::CPDecomposition) =
  mapreduce(vector_outer, +, zip((eachcol.(factors(CPD)))...))

vector_outer(v) = reshape(kron(reverse(v)...),length.(v))

getindex(CPD::CPDecomposition, I::Vararg{Int}) =
  sum(reduce(.*, (@view f[i,:]) for (f,i) in zip(factors(CPD), I)))
```

2.3 Tensor rank

- tensor rank
- constrained rank (nonnegative etc.)

The rank of a matrix $Y \in \mathbb{R}^{I\times J}$ can be defined as the smallest $R \in \mathbb{Z}_+$ such that there exists an exact factorization $Y = AB$ for some $A \in \mathbb{R}^{I\times R}$ and $B \in \mathbb{R}^{R\times J}$.

Although this can be extended to higher order tensors, we must specify under which factorization model we are using. For example, the *CP-rank* R of a tensor Y is the smallest such R that omits an exact CP decomposition of Y.

Definition 2.12: The CP rank of a tensor $Y \in \mathbb{R}^{I_1\times\cdots\times I_N}$ is the smallest R such that there exist factors $A_n \in \mathbb{R}^{I_n\times R}$ and $Y = [\![A_1, ..., A_N]\!]$,

$$\operatorname{rank}_{\mathrm{CP}}(Y) = \min\{R \mid \exists A_n \in \mathbb{R}^{I_n\times R}, n \in [N] \quad \text{s.t.} \quad Y = [\![A_1, ..., A_N]\!]\}.$$

In a similar way, we can define the *Tucker-1-rank R*.

Definition 2.13: The Tucker-1 rank of a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is the smallest R such that there exist factors $A \in \mathbb{R}^{I_1 \times R}$ and $B \in \mathbb{R}^{R \times I_2 \times \cdots \times I_N}$ where $Y = AB$

$$\operatorname{rank}_{\text{Tucker-1}}(Y) = \min\{R \mid \exists A_n \in \mathbb{R}^{I_n \times R}, B \in \mathbb{R}^{R \times I_2 \times \cdots \times I_N} \quad \text{s.t.} \quad Y = AB\}$$

For the Tucker and Tucker-n decompositions, we instead call a particular factorization **a** rank-$(R_1, ..., R_N)$ Tucker factorization or **a** rank-$(R_1, ..., R_n)$ Tucker-n factorization, rather than **the** CP- or Tucker-1-rank of a tensor or **the** rank of a matrix.

One reason CP and Tucker-1 only need a single rank R can be explained by considering the case when the order of the tensor $N = 2$ (matrices). The two factorizations become equivalent and are equal to low-rank R matrix factorization $Y = AB$. In fact, Tucker-1 is always equivalent to a low-rank matrix factorization, if you consider a flattening of the tensor to arrange the entries as a matrix.

The idea of tensor rank can be generalized further to constrained rank. These are the smallest rank R such that the factors in the decomposition obey the given set of constraints.

For example, the nonnegative Tucker-1 rank is defined as

$$\operatorname{rank}^{+}_{\text{Tucker-1}}(Y) = \min\left\{R \,\middle|\, \exists A_n \in \mathbb{R}_{+}^{I_n \times R}, B \in \mathbb{R}_{+}^{R \times I_2 \times \cdots \times I_N} \quad \text{s.t.} \quad Y = AB\right\}.$$

More restrictive constraints increase the rank of the tensor since there is less freedom in selecting the factors.

Most tensor decomposition algorithms require the rank as input [CITE] since calculating the rank of the tensor can be NP-hard in general [19]. For applications where the rank is not known a priori, a common strategy is to attempt a decomposition for a variety of ranks, and select the model with smallest rank that still achieves good fit between the factorization and the original tensor.

3 Computing Decompositions

- Given a data tensor and a model, how do we fit the model?

Many tensor decompositions algorithms exist in the literature. Usually, they cyclically (or in a random order) update factors until their reconstruction satisfies some convergence criterion. The base algorithm described in Section 3.2 provides flexible framework for wide class of constrained tensor factorization problems. This framework was selected based on empirical observations where it outperforms other similar algorithms, and has also been observed in the literature [10].

3.1 Optimization Problem

- Least squares (can use KL, 1 norm, etc.)

Ideally, we would be given a data tensor Y and decomposition model, and compute an exact factorization of Y into its factors. Because there is often measurement, numerical, or modeling error, an exact factorization of Y for a particular rank may not exist. To over come this, we instead try to fit the model to the data. Let X be the reconstruction of factors $A_1, ..., A_N$ according to some decomposition for a fixed rank. We assume we know the size of the factors $A_1, ..., A_N$ and how they are combined to produce a tensor the same size of Y, i.e. the map $g : (A_1, ..., A_N) \mapsto X$.

There are many loss functions that can be used to determine how close the model X is to the data Y. In principle, any distance or divergence $d(Y, X)$ could be used. We use the L_2 loss or least-squares distance between the tensors $\|X - Y\|_F^2$, but other losses are used for tensor decomposition in practice such as the KL divergence [CITE].

The main optimization we must solve is now given.

Definition 3.1: The constrained least-squares tensor factorization problem is to solve

$$\min_{A_1,...,A_N} \frac{1}{2}\|g(A_1, ..., A_N) - Y\|_F^2 \quad \text{s.t.} \quad (A_1, ..., A_N) \in \mathcal{C}_1 \times \cdots \times \mathcal{C}_N \tag{9}$$

for a given data tensor Y, constraints $\mathcal{C}_1, ..., \mathcal{C}_N$, and decomposition model g with fixed rank.

Note the problem would have the same solutions as simply using the objective $\|g(A_1, ..., A_N) - Y\|$ without squaring and dividing by 2. We define the objective in Equation 9 to make computing the function value and gradients faster.

3.2 Base algorithm

- Use Block Coordinate Descent / Alternating Proximal Descent
  - do *not* use alternating least squares (slower for unconstrained problems, no closed form update for general constrained problems)

Let $f(A_1, ..., A_N) := \frac{1}{2}\|g(A_1, ..., A_N) - Y\|_F^2$ be the objective function we wish to minimize in Equation 9. Following Xu and Yin [10], the general approach we take to minimize f is to apply block coordinate descent using each factor as a different block. Let A_n^t be the tth iteration of the nth factor, and let

$$f_n^t(A_n) := \frac{1}{2}\left\|g\left(A_1^{t+1}, ..., A_{n-1}^{t+1}, A_n, A_{n+1}^t, ..., A_N^t\right) - Y\right\|_F^2$$

be the (partially updated) objective function at iteration t for factor n.

Given initial factors $A_1^0, ..., A_N^0$, we cycle through the factors $n \in [N]$ and perform the update

$$A_n^{t+1} \leftarrow \arg\min_{A_n \in \mathcal{C}_n} \langle \nabla f_n^t(A_n^t), A_n - A_n^t \rangle + \frac{L_n^t}{2}\left\|A_n - A_n^t\right\|_F^2,$$

for $t = 1, 2, \ldots$ until some convergence criterion is satisfied (see Section 4.2.a).

This implicit update has the *projected gradient descent* closed form solution for convex constraints $\mathcal{C}_n$,

$$A_n^{t+1} \leftarrow P_{\mathcal{C}_n}\left(A_n^t - \frac{1}{L_n^t}\nabla f_n^t(A_n^t)\right). \tag{10}$$

We typically choose L_n^t to be the Lipschitz constant of ∇f_n^t, since it is a sufficient condition to guarantee $f_n^t(A_n^{t+1}) \leq f_n^t(A_n^t)$, but other step sizes can be used in theory [20 (Sec. 1.2.3)].

?ASIDE? To write ∇f_n^t, we have assumed (block) differentiability of the decomposition model g. In practice, most decompositions are “block-linear” (freeze all factors but one and you have a linear function) and in rare cases are “block-affine”. “block-affine” is enough to ensure f_n^t is convex (i.e. f is “block-convex”) so the updates Equation 10 converge to a Nash equilibrium (block minimizer).

3.2.a High level code

To ensure the code stays flexible, the main algorithm of BlockTensorDecomposition.jl, `factorize`, is defined at a very high level.

```julia
factorize(Y; kwargs...) =
    _factorize(Y; (default_kwargs(Y; kwargs...))...)

"""
Inner level function once keyword arguments are set
"""
function _factorize(Y; kwargs...)
    decomposition, previous, updateprevious!, parameters, updateparameters!,
    update!, stats_data, getstats, converged, kwargs = initialize(Y, kwargs)

    while !converged(stats_data; kwargs...)
        # Usually one cycle of updates through each factor in the decomposition
        update!(decomposition; parameters...)

        # This could be the next stepsize or other info used by update!
        updateparameters!(parameters, decomposition, previous)

        push!(stats_data,
            getstats(decomposition, Y, previous, parameters, stats_data))

        # Update one or two previous iterates. For example, used for momentum
        updateprevious!(previous, parameters, decomposition)
    end

     kwargs = postprocess!(decomposition, Y, previous, parameters, stats_data,
updateparameters!, getstats, kwargs)
```

```
    return decomposition, stats_data, kwargs
end
```

The magic of the code is in defining the functions at runtime for a particular decomposition requested, from a reasonable set of default keyword arguments. This is discussed further in Section 4.2.

3.2.b Computing Gradients

- Use Auto diff generally
- But hand-crafted gradients and Lipschitz calculations *can* be faster (e.g. symmetrized slice-wise dot product)

Generally, we can use automatic differentiation on f to compute gradients. Some care needs to be taken otherwise the forward or backwards pass will have to be recompiled every iteration since the factors are updated every iteration.

But for Tucker decompositions, we can compute gradients faster than what an automatic differentiation scheme would give, by taking advantage of symmetry and other computational shortcuts.

Starting with the Tucker-1 decomposition (Definition 2.9), we would like to compute $\nabla_B f(B, A)$ and $\nabla_A f(B, A)$ for $f(B, A) = \frac{1}{2}\|AB - Y\|_F^2$ for a given input Y. We have the gradient

$$\nabla_B f(B, A) = A^\top(AB - Y) = (B \times_1 A - Y) \times_1 A^\top \tag{11}$$

by chain rule, but it is more efficient to calculate the gradient as

$$\nabla_B f(B, A) = (A^\top A)B - A^\top Y = B \times_1 (A^\top A) - Y \times_1 A^\top. \tag{12}$$

[2]For $A \in \mathbb{R}^{I\times R}$, $B \in \mathbb{R}^{R\times J\times K}$, and $Y \in \mathbb{R}^{I\times J\times K}$, Equation 11 requires

$$\underbrace{2IJKR}_{AB-Y} + \underbrace{IJK(2I-1)}_{A^\top(AB-Y)} \sim 2IJKR + 2I^2JK$$

floating point operations (FLOPS) whereas Equation 12 only uses

$$\underbrace{\frac{R(R+1)}{2}(2I-1)}_{A^\top A} + \underbrace{RJK(2I-1)}_{A^\top Y} + \underbrace{2R^2JK}_{(A^\top A)B-(A^\top Y)} \sim 2IJKR + 2R^2JK + IR^2$$

FLOPS[3]. So for small ranks $R \ll I$, Equation 12 is cheaper.

A similar story can be said about $\nabla_A f(B, A)$ which is most efficiently computed as

$$\nabla_A f(B, A) = A(B \cdot_1 B) - Y \cdot_1 B.$$

[2]Seeing Equation 11 and Equation 12 written using the 1-mode product shows how it is "backwards" to normal matrix-matrix multiplication.

[3]Note we have the smaller factor $R(R+1)/2$ and not the expected R^2 number of entries needed to compute $A^\top A$. The product is a symmetric matrix so only the upper or lower triangle of entries needs to be computed.

! Note

For the family of Tucker decompositions, the objective function f is "block-quadratic" with respect to the factors. This means the gradient with respect to a factor is an affine function of that factor. This is exactly what we see in Equation 12 where B is multiplied by the "slope" $A^\top A$ plus a shift of $-Y \times_1 A^\top$.

The associated implementation with BlockTensorDecomposition.jl is shown below. We define a `make_gradient` which takes the decomposition, factor index `n`, and data tensor `Y`, and creates a function that computes the gradient for the same type of decomposition. This lets us manipulate the function that computes the gradient, rather than just the computed gradient.

```julia
function make_gradient(T::Tucker1, n::Integer, Y::AbstractArray; objective::L2,
kwargs...)
    if n==0 # the core is the zeroth factor
        function gradient0(X::Tucker1; kwargs...)
            (B, A) = factors(X)
            AA = A'A
            YA = Y×₁A'
            grad = B×₁AA - YA
            return grad
        end
        return gradient0
    elseif n==1 # the matrix is the first factor
        function gradient1(X::Tucker1; kwargs...)
            (B, A) = factors(X)
            BB = slicewise_dot(B, B)
            YB = slicewise_dot(Y, B)
            grad = A*BB - YB
            return grad
        end
        return gradient1
    else
        error("No $(n)th factor in Tucker1")
    end
end
```

Similarly, we also have special methods for the Tucker and CP Decomposition.

The gradient with respect to the core for a full Tucker factorization is

$$\nabla_B f(B, A_1, ..., A_N) = B \bigtimes_n A_n^\top A_n - Y \bigtimes_n A_n^\top,$$

and the gradient with respect to the matrix factor A_n is

$$\nabla_{A_n} f(B, A_1, ..., A_N) = A_n (\tilde{X}_n \cdot_n \tilde{X}_n) - Y \cdot_n \tilde{X}_n$$

where

$$\tilde{X}_n = \left(B \bigtimes_{m\neq n} A_m \right) = [\![B; A_1, ..., A_{n-1}, \mathrm{id}_{R_n}, A_{n+1}, ..., A_N]\!].$$

```julia
function make_gradient(T::Tucker, n::Integer, Y::AbstractArray; objective::L2,
kwargs...)
    N = ndims(T)
    if n==0 # the core is the zeroth factor
        function gradient_core(X::AbstractTucker; kwargs...)
            B = core(X)
            matrices = matrix_factors(X)
            gram_matrices = map(A -> A'A, matrices) # gram matrices AA = A'A,
                                                    # BB = B'B, ...
            grad = tuckerproduct(B, gram_matrices)
                 - tuckerproduct(Y, adjoint.(matrices))
            return grad
        end
        return gradient_core

    elseif n in 1:N # the matrix factors start at m=1
        function gradient_matrix(X::AbstractTucker; kwargs...)
            B = core(X)
            matrices = matrix_factors(X)
            Aₙ = factor(X, n)
            X̃ₙ = tuckerproduct(B, matrices; exclude=n)
            grad = Aₙ * slicewise_dot(X̃ₙ, X̃ₙ; dims=n)
                   - slicewise_dot(Y, X̃ₙ; dims=n)
            return grad
        end
        return gradient_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

For the CP Decomposition, we can simply treat the core as $B = \mathrm{id}_R$ and compute the gradient with respect to the matrix factors similarly to the Tucker decomposition:

$$\nabla_{A_n} f(A_1, ..., A_N) = A_n\left(\tilde{X}_n \cdot_n \tilde{X}_n\right) - Y \cdot_n \tilde{X}_n$$

where

$$\tilde{X}_n = \left(\mathrm{id}_R \bigtimes_{m\neq n} A_m \right) = [\![\mathrm{id}_R; A_1, ..., A_{n-1}, \mathrm{id}_R, A_{n+1}, ..., A_N]\!].$$

```julia
function   make_gradient(T::CPDecomposition,   n::Integer,   Y::AbstractArray;
objective::L2, kwargs...)
    N = ndims(T)
    if n in 1:N # the matrix factors start at m=1
        function gradient_matrix(X::AbstractTucker; kwargs...)
            B = core(X)
            matrices = matrix_factors(X)
            Aₙ = factor(X, n)
            X̃ₙ = tuckerproduct(B, matrices; exclude=n)
            grad = Aₙ * slicewise_dot(X̃ₙ, X̃ₙ; dims=n)
                   - slicewise_dot(Y, X̃ₙ; dims=n)
            return grad
        end
        return gradient_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

3.2.c Computing Lipschitz Step-sizes

Similar to automatic differentiation, there exist "automatic Lipschitz" calculations to upper bound the Lipschitz constant of a function [21].

For the family of Tucker decompositions, we can compute the Lipschitz constants of the gradient efficiently similar to how we compute the gradient in Section 3.2.b with the following corollaries of Theorem 2.2.

Corollary 3.1: Let $B \in \mathbb{R}^{R_1 \times \cdots \times R_N}$, $A_m \in \mathbb{R}^{I_m \times R_m}$, and $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. The function

$$f(A) = \frac{1}{2}\left\| [B; A_1, ..., A_{n-1}, A, A_{n+1}, ..., A_N] - Y \right\|_F^2$$

is quadratic, and L-smooth with constant

$$L_{A_n} = \left\| \tilde{X}_n \cdot_n \tilde{X}_n \right\|_{\mathrm{op}}$$

where

$$\tilde{X}_n = \left(B \bigtimes_{m \neq n} A_m \right) = \left[\!\left[B; A_1, ..., A_{n-1}, \mathrm{id}_{R_n}, A_{n+1}, ..., A_N \right]\!\right].$$

Proof. The result follows from Theorem 2.1 and Theorem 2.2.

Corollary 3.2: Let $A_n \in \mathbb{R}^{I_n \times R_n}$, and $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. The function

$$f(B) = \frac{1}{2}\|[B; A_1, ..., A_N] - Y\|_F^2$$

is quadratic, and L-smooth with constant

$$L_B = \prod_{n=1}^{N} \left\|A_n^\top A_n\right\|_{\mathrm{op}}.$$

Proof. The result follows from Theorem 2.1 and Theorem 2.2.

This yields the following efficient implementations.

! Note

It is tempting to use the identity $\left\|A^\top A\right\|_{\mathrm{op}} = \|A\|_{\mathrm{op}}^2$ to calculate the Lipschitz constant without forming $A^\top A$. For tall dense matrices, using this identity is slower and more memory intensive as of Julia 1.11.2. See LinearAlgebra.jl issue 1185 on Github.

```julia
function make_lipschitz(T::Tucker, n::Integer, Y::AbstractArray; objective::L2,
kwargs...)
    N = ndims(T)
    if n==0 # the core is the zeroth factor
        function lipschitz_core(X::AbstractTucker; kwargs...)
            return prod(A -> opnorm(A'A), matrix_factors(X))
        end
        return lipschitz_core

    elseif n in 1:N # the matrix is the zeroth factor
        function lipschitz_matrix(X::AbstractTucker; kwargs...)
            B = core(X)
            matrices = matrix_factors(X)
            X̃ₙ = tuckerproduct(B, matrices; exclude=n)
            return opnorm(slicewise_dot(X̃ₙ, X̃ₙ; dims=n))
        end
        return lipschitz_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

In the case of Tucker decomposition, the Lipschitz constants simplify to

$$L_B = \left\| A^\top A \right\|_{\text{op}}, \quad L_B = \| B \cdot_1 B \|_{\text{op}}.$$

```julia
function      make_lipschitz(T::Tucker1,      n::Integer,      Y::AbstractArray;
objective::L2, kwargs...)
    if n==0 # the core is the zeroth factor
        function lipschitz0(X::Tucker1; kwargs...)
            A = matrix_factor(X, 1)
            return opnorm(A'A)
        end
        return lipschitz0

    elseif n==1 # the matrix is the zeroth factor
        function lipschitz1(X::Tucker1; kwargs...)
            B = core(X)
            return opnorm(slicewise_dot(B, B))
        end
        return lipschitz1

    else
        error("No $(n)th factor in Tucker1")
    end
end
```

Lastly, for CP decomposition, the Lipschitz constants for the matrices can be calculated similarly to the Tucker decomposition.

```julia
function   make_lipschitz(T::CPDecomposition,   n::Integer,   Y::AbstractArray;
objective::L2, kwargs...)
    N = ndims(T)
    if n in 1:N
        function lipschitz_matrix(X::CPDecomposition; kwargs...)
            id = core(X)
            matrices = matrix_factors(X)
            X̃ₙ = tuckerproduct(id, matrices; exclude=n)
            return opnorm(slicewise_dot(X̃ₙ, X̃ₙ; dims=n))
        end
        return lipschitz_matrix

    else
        error("No $(n)th factor in CPDecomposition")
    end
end
```

4 Computational Techniques

- As stated, algorithm works
- But can be slow, especially for constrained or large problems

As stated, the algorithm described in Section 3.2 works. It will converge to a solution to our optimization problem and factorize the input tensor. It is worth discussing how the algorithm can be modified to improve convergence to maintain quick convergence for large problems, and what sort of architectural methods are used to allow for maximum flexibility, without over engineering the package.

4.1 For Improving Convergence Speed

There are a few techniques used to assist convergence. Two ideas that are well studied are discussed in this section. They are 1) breaking up the updates into smaller blocks, and 2) using momentum or acceleration. What is perhaps novel is considering the synergy between these two ideas.

Two more techniques are implemented in BlockTensorDecomposition.jl to improve convergence. To the authors knowledge, these are new to tensor factorization, but may or may not be applicable depending on the exact factorization problem or data being studied. For these reasons, these other techniques are discussed separately in Section 5 and Section 6.

4.1.a Sub-block Descent

- Use smaller blocks, but descent in parallel (sub-blocks don't wait for other sub-blocks)
- Can perform this efficiently with a "matrix step-size"

When using block coordinate descent as in Section 3.2, it is natural to treat each factor as its own block. This requires the fewest blocks while ensuring the objective is still convex with respect to each block. We could just as easily use smaller blocks.

In the case of Tucker decomposition, one modification of the update shown in Equation 10 would be to update each column $a_{n,r}$, $r = 1, ..., R_n$ of the matrix A_n separately. This would be suitable if the constraint that $A_N \in \mathcal{C}_n$ can be broken up further into the constraints $a_{n,r} \in \mathcal{C}_{n,r}$. This is shown in the following update scheme:

$$a_{n,r}^{t+1} \leftarrow P_{\mathcal{C}_{n,r}}\left(a_{n,r}^{t} - \frac{1}{L_{n,r}^{t}} \nabla f_{n,r}^{t}\left(a_{n,r}^{t}\right)\right), \tag{13}$$

where $f_{n,r}^{t}(a) = \frac{1}{2}\left\| \left[B; A_1^{t+1}, ..., A_{n-1}^{t+1}, A_{n,r}^{t}(a), A_{n+1}^{t}, ..., A_N^{t}\right] - Y \right\|_F^2$ and

$$A_{n,r}^{t}(a) = \begin{bmatrix} \uparrow & & \uparrow & \uparrow & \uparrow & & \uparrow \\ a_{n,1}^{t+1} & \cdots & a_{n,r-1}^{t+1} & a & a_{n,r+1}^{t} & \cdots & a_{n,R_n}^{t} \\ \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \end{bmatrix}. \tag{14}$$

In theory, the block update shown in Equation 13 should be a bit more expensive than using the larger blocks on the matrices A shown in Equation 10, since the gradient needs to be recomputed R_n times for each matrix block n, rather than only computing the gradient once per block n. To get around this, we use the fact that $\nabla f_{n,r}^{t}(a)$ is the rth column from the gradient $\nabla f_n^t(A)$ where $f_n^t(A) = \frac{1}{2}\left\| \left[B; A_1^{t+1}, ..., A_{n-1}^{t+1}, A, A_{n+1}^{t}, ..., A_N^{t}\right] - Y \right\|_F^2$. So we can approximate Equation 13 by first calculating the gradient ∇f_n^t at

$$\hat{A}_n^t = \begin{bmatrix} \uparrow & & \uparrow & \uparrow & \uparrow & & \uparrow \\ a_{n,1}^t & \cdots & a_{n,r-1}^t & a_{n,r}^t & a_{n,r+1}^t & \cdots & a_{n,R_n}^t \\ \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \end{bmatrix}, \tag{15}$$

and then updating each sub-block r according to

$$a_{n,r}^{t+1} \leftarrow P_{\mathcal{C}_{n,r}}\left(a_{n,r}^t - \frac{1}{L_{n,r}^t}\nabla f_n^t\left(\hat{A}_n^t\right)\right). \tag{16}$$

Note the difference between Equation 14 and Equation 15 is that we don't use the most recent columns $a_{n,j}$ for $j < r$ in Equation 15.

The update given in Equation 16 can be merged back to an update on the whole block A_n

$$A_n^{t+1} \leftarrow P_{\mathcal{C}_n}\left(A_n^t - \nabla f_n^t\left(A_n^t\right)\left(\hat{L}_n^t\right)^{-1}\right) \tag{17}$$

where we have the $R_n \times R_n$ diagonal "Lipschitz Matrix"

$$\hat{L}_n^t = \begin{bmatrix} L_{n,1}^t & 0 & & 0 \\ 0 & L_{n,2}^t & & 0 \\ & & \ddots & \vdots \\ 0 & 0 & \cdots & L_{n,R_n}^t \end{bmatrix}.$$

It is not too hard to show that the Lipschitz $L_{n,r}^t$ for $\nabla f_{n,r}^t$ is the Euclidean norm of the rth column[4] of the matrix $\tilde{X}_n \cdot_n \tilde{X}_n$ from Corollary 3.1,

$$L_{n,r}^t = \left\|\left(\tilde{X}_n \cdot_n \tilde{X}_n\right)[:, r]\right\|_2.$$

This leads to the following efficient calculation of the Lipschitz matrix in Julia.

```julia
function diagonal_lipschitz_matrix(T::Tucker, n::Int; kwargs...)
    B = core(T)
    matrices = matrix_factors(T)
    X̃ₙ = tuckerproduct(B, matrices; exclude=n)
    return Diagonal_col_norm(slicewise_dot(X̃ₙ, X̃ₙ; dims=n))
end

Diagonal_col_norm(X) = Diagonal(norm.(eachcol(X)))
```

We can now compare the merged sub-block update Equation 17 to the standard projected gradient descent update shown in Equation 10. The difference is that we calculate a "matrix step-size"

[4]We could have used the rth row of $\tilde{X}_n \cdot_n \tilde{X}_n$ since this matrix is symmetric. Since Julia store matrices in column-major order, many operations that perform column-wise are more efficient than their equivalent row-wise operation.

$\hat{L}_n^t \in \mathbb{R}^{R_n \times R_n}$ rather than a scalar $L_n^t \in \mathbb{R}$. In practice, this leads to an improvement in convergence speed for two reasons.

First, computing the matrix $\hat{L}_n^t$ often faster than the scalar $L_n^t = \left\| \tilde{X}_n \cdot_n \tilde{X}_n \right\|_{\mathrm{op}}$. The former only requires calculating the Euclidean norm of R vectors for a total cost of $2R_n^2$ floating point operations (FLOPs), whereas the latter requires the top eigenvalue of $\tilde{X}_n \cdot_n \tilde{X}_n$. This is usually done with a power method or truncated SVD which can be costlier than the flat rate of $2R_n^2$ FLOPs.

Secondly, using the matrix $\hat{L}_n^t$ means columns where $L_{n,r}^t$ is small can take larger descent steps. This is because the largest singular value of $\tilde{X}_n \cdot_n \tilde{X}_n$ is an upper bound on the Euclidean norm of each column: $L_{n,r}^t \leq L_n^t$. Using the scaler Lipschitz L_n^t is equivalent to the diagonal matrix

$$D = \begin{bmatrix} L_n^t & 0 & & 0 \\ 0 & L_n^t & & 0 \\ & & \ddots & \vdots \\ 0 & 0 & \cdots & L_n^t \end{bmatrix}$$

in the merged sub-block update shown in Equation 17. So each column of A_n is forced to use the worst case (largest) singular value of $\tilde{X}_n \cdot_n \tilde{X}_n$. In this way, the matrix $\hat{L}_n^t$ acts like a cheap approximate Hessian as if we were doing a quasi-Newton update with step-size 1.

For completeness, we can perform the same merged sub-block update to update the core B. In this case, we obtain the more complicated "Lipschitz tensor" $\hat{L}_B^t \in \mathbb{R}^{(R_1 \times \cdots \times R_N)^2}$ defined by

$$\hat{L}_B^t = \hat{L}_{B,1}^t \otimes \cdots \otimes \hat{L}_{B,N}^t$$

where each matrix $\hat{L}_{B,n}^t \in \mathbb{R}^{R_n \times R_n}$ is diagonal with non-zero entries

$$L_{B,n}^t[r,r] = \left\| \left(\left(A_n^t\right)^\top A_n^t \right)[:,r] \right\|_2 .$$

The merged sub-block update for the core becomes

$$B^{t+1} \leftarrow P_{\mathcal{C}_B}\left(B^t - \nabla f_0^t(B^t) \times_B \left(\hat{L}_B^t \right)^{-1} \right) \tag{18}$$

with the multiplication

$$\begin{aligned} \nabla f_0^t(B^t) \times_B \left(\hat{L}_B^t\right)^{-1} &= \nabla f_0^t(B^t) \bigtimes_n \left(\hat{L}_{B,n}^t\right)^{-1} \\ &= \left[\!\left[\nabla f_0^t(B^t); \left(\hat{L}_{B,1}^t\right)^{-1}, \ldots, \left(\hat{L}_{B,n}^t\right)^{-1} \right]\!\right]. \end{aligned} \tag{19}$$

This should be thought of as normalizing each dimension of the tensor $\nabla f_0^t(B^t)$ so that we can take a unit step-size.

TODO use the multi-mode product in stead of defining a new multiplication $\times_B$ here. I think I'll have the same tensor product issue as before where the indices.

Putting the core and matrices Lipschitz calculations together gives us the following Julia code. Note we store $\hat{L}_B^t$ in factored form as a tuple of diagonal matrices to save space and computation.

TODO Compare to preconditioned decent. See [22–25]

```julia
function make_block_lipschitz(T::AbstractTucker, n::Integer, Y::AbstractArray;
objective::L2, kwargs...)
    N = ndims(T)
    if n==0 # the core is the zeroth factor
        function lipschitz_core(X::AbstractTucker; kwargs...)
            return map(A -> Diagonal_col_norm(A'A), matrix_factors(X))
        end # Returns a tuple of diagonal matrices
        return lipschitz_core

    elseif n in 1:N
        function lipschitz_matrix(X::AbstractTucker; kwargs...)
            matrices = matrix_factors(X)
            X̃ₙ = tuckerproduct(core(X), matrices; exclude=n)
            return Diagonal_col_norm(slicewise_dot(X̃ₙ, X̃ₙ; dims=n))
        end
        return lipschitz_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

4.1.b Momentum

- This one is standard
- Use something similar to [10]
- This is compatible with sub-block descent with appropriately defined matrix operations

In practice, we find that extrapolating the iterate based on the prior iterate

$$\hat{A}_n^t \leftarrow A_n^t + \omega_n^t (A_n^t - A_n^{t-1}) \tag{20}$$

for some amount of extrapolation $\omega_n^t \geq 0$ before applying the update Equation 17 greatly improves the speed of descent. This can be thought of as a type of momentum where we continue to move in directions that showed a lot of improvement during the last iteration.

Our selection for ω_n^t follows Xu and Yin's method for block coordinate descent [10], which is itself inspired by Tseng and Yun's coordinate gradient descent method [26].

Given a parameter[5] $\delta \in [0, 1)$, we define the momentum parameters and τ^t and ω_n^t according to the following updates

[5]Usually we pick a number close to 1. For example, we use the default $\delta = 0.9999$.

$$
\begin{aligned}
\tau^0 &= 1 \\
\tau^{t+1} &\leftarrow \frac{1}{2}\left(1+\sqrt{1+4(\tau^t)^2}\right) \\
\hat{\omega}^t &\leftarrow \frac{\tau^t - 1}{\tau^{t+1}} \\
\omega_n^t &\leftarrow \min\left(\hat{\omega}^t, \delta\sqrt{\hat{L}_n^{t-1}\left(\hat{L}_n^t\right)^{-1}}\right).
\end{aligned}
\tag{21}
$$

TODO notation is going to get confusing. We use hat/not hat L for the scaler vs matrix/tensor version. But we use hat/not hat ω for the ideal vs clamped momentum.

What is novel with our approach is that we perform this momentum on the Lipschitz matrices and tensors $\hat{L}_n^t$ rather than scaler Lipschitz constant L_n^t. In this way, we should interpret the operations shown in Equation 21 as operating element-wise. This also means the momentum parameter ω_n^t is a matrix or tensor and takes the same shape as $\hat{L}_n^t$.

In order to perform Equation 20, we use the equivalent but more efficient formulation

$$\hat{A}_n^t \leftarrow A_n^t\left(\mathrm{id}_{R_n} + \omega_n^t\right) - A_n^{t-1}\omega_n^t.$$

```
function (U::MomentumUpdate)(X::T; X_last::T, ω, δ, kwargs...) where T
    n = U.n

    L = U.lipschitz(X; kwargs...)
    L_last = U.lipschitz(X_last; kwargs...)
    ω = min.(ω, δ .* .√(L_last/L))

    A, A_last = factor(X, n), factor(X_last, n)

    A .= U.combine(A, id + ω)
    A .-= U.combine(A_last, ω)
end
```

In the code above, the momentum stores the factor n in acts on, how to compute the Lipschitz constant, matrix, or tensor, and how to combine (multiply) the constant with the factor. In the case of matrix factors A_n in a Tucker decomposition, this is simply right matrix-matrix multiplication. The core factor B uses $\times_B$ as described in Equation 19.

The parameters of the momentum update are handled separately. This is to treat the momentum update as "apply this update with these parameters". The parameters τ^t and $\hat{\omega}^t$ are updated by the following function that keeps track of all parameters needed to perform the iteration. In this case, we keep track of what iteration t we are at, the previous iterate, and a few options for the order in which to cycle through and update the blocks.

```
update_τ(τ) = 0.5*(1 + sqrt(1 + 4*τ^2))

function  initialize_parameters(decomposition,  Y,  previous;  momentum::Bool,
random_order, recursive_random_order, kwargs...)
    # parameters for the update step are symbol => value pairs
    # held in a dictionary since we may mutate these, e.g. the step-size
    parameters = Dict{Symbol, Any}()

    # General Looping
    parameters[:iteration] = 0
    parameters[:X_last] = previous[begin] # Last iterate
    parameters[:random_order] = random_order
    parameters[:recursive_random_order] = recursive_random_order

    # Momentum
    if momentum
         parameters[:τ_last] = float(1) # need this field to hold Floats, not
Ints
        parameters[:τ] = update_τ(float(1))
        parameters[:ω] = (parameters[:τ_last] - 1) / parameters[:τ]
        parameters[:δ] = kwargs[:δ]
    end

    function updateparameters!(parameters, decomposition, previous)
        parameters[:iteration] += 1
        # parameters[:x_last] = previous[begin]
# This is commented since parameters[:x_last] already points to previous[begin]

        if momentum
            parameters[:τ_last] = parameters[:τ]
            parameters[:τ] = update_τ(parameters[:τ_last])
            parameters[:ω] = (parameters[:τ_last] - 1) / parameters[:τ]
        end
    end

    return parameters, updateparameters!
end
```

4.1.c Empirical Evidence for Sub-Block Descent and Momentum

To showcase that the combination of these two tricks can speed up convergence, we will benchmark them by factorizing a random 10×10 tensor (a matrix) with rank 3. The Julia code is shown below, and the results are shown in Table 2.

```
using BenchmarkTools
using BlockTensorDecomposition

fact = BlockTensorDecomposition.factorize
```

```
options = (
    :rank => 3,
    :tolerence => (1, 0.03),
    :converged => (GradientNNCone, RelativeError),
    :δ => 0.9,
)

n_subblock_n_momentum(Y) = fact(Y;
    do_subblock_updates=false,
    momentum=false,
    options...
)

y_subblock_n_momentum(Y) = fact(Y;
    do_subblock_updates=true,
    momentum=false,
    options...
)

n_subblock_y_momentum(Y) = fact(Y;
    do_subblock_updates=false,
    momentum=true,
    options...
)

y_subblock_y_momentum(Y) = fact(Y;
    do_subblock_updates=true,
    momentum=true,
    options...
)

I, J = 10, 10
R = 3

@benchmark n_subblock_n_momentum(Y) setup=(Y=Tucker1((I, J), R))
@benchmark n_subblock_y_momentum(Y) setup=(Y=Tucker1((I, J), R))
@benchmark y_subblock_n_momentum(Y) setup=(Y=Tucker1((I, J), R))
@benchmark y_subblock_y_momentum(Y) setup=(Y=Tucker1((I, J), R))

performance_increase(old, new) = (old - new) / new * 100
```

The code `Tucker1((I, J), R)` produces a random $I \times J$ rank-R matrix by generating two matrices $A \in \mathbb{R}^{I \times R}$ and $B \in \mathbb{R}^{R \times J}$ with standard normal entries, and multiplies them together.

Table 2: Summary of median times to factorize a random 10×10 rank-3 matrix under different methods. The performance increase is given by the formula $(\text{old} - \text{new}) / \text{new}$.

	No Momentum	**Yes Momentum**
No Sub-Block	48.843 ms	45.738 ms (6.7887% faster)
Yes Sub-Block	27.473 ms (77.785% faster)	**24.350 ms (100.59% faster)**

In Table 2, you can see that having both sub-block descent and momentum yields the fastest factorization. Moreover, the performance increase is *more* than simply the performance increases obtained by exclusively sub-block or momentum alone.[6] This suggests that there is synergy with these two methods and are best used together.

TODO Repeat this experiment on a less trivial factorization. What I've done above can be done with an SVD in less time. Ideally use a 10 x 10 x 10 Tucker decomposition with rank 2 x 3 x 4.

4.2 For Flexibility

- there are a number of software engineering techniques used
- these help flexibility for hot swapping and a language for making custom…
  - convergence criterion (and having multiple stopping conditions)
  - probing info during the iterations (stats collected at the end)
  - having multiple constraints and ways to enforce them
  - cyclically or partially randomly or fully randomly update factors
- smart enough to apply these in a reasonable order

There are a number of software engineering techniques used to ensure BlockTensorDecomposition.jl is flexible and applicable to a wide range of problems. These enable key algorithmic choices to be hot-swapped and easily compared with each other.

4.2.a Convergence Criteria and Stats

- Can request info about any factor at each outer iteration
- any subset of stats can be the convergence criteria

Some iterative algorithms produce the exact solution of a problem after a finite number of iterations. Generalized minimal residual method (GMRES) is a good example of this [TODO cite!]. Our algorithm, like many others, only converges to the exact solution in the limit as the number of iterations grow. Since we would like a solution in finite time, we must halt the algorithm early.

In finite precision, we can halt the algorithm if we can guarantee the solution is accurate to machine precision. This can often be too strict if convergence is not at a fast enough rate. Furthermore, depending on *why* we are decomposing a tensor, we may want different stats to be within a given a tolerance. BlockTensorDecomposition.jl attempts to solve this issue by defining some standard criteria that can be used to halt the algorithm. These are subtypes of the abstract type `AbstractStat` and are listed below.

[6]The expected performance increase if sub-block descent and momentum where independence would be $(1 + 0.067887)(1 + 0.77785) = 1.89854$ or only 89.854% faster.

```
# X is the decomposition model e.g. Tucker((B, A1, A2, A2)))
# Y is the input tensor we want to decompose
# norm(X) is the Frobenius norm of X

GradientNorm # norm(∇f(X))
GradientNNCone # sqrt(sum(norm(∇f(Ai)[Ai .> 0 .| ∇f(Ai) .< 0])^2 for Ai in factors(X)))
ObjectiveValue # 1/2 norm(X - Y)^2
ObjectiveRatio # norm(X_last - Y)^2 / norm(X_current - Y)^2
RelativeError # norm(X - Y) / norm(Y)
IterateNormDiff # norm(X_current - X_last)
IterateRelativeDiff # norm(X_current - X_last) / norm(X_last)
```

Most of these are self explanatory, expect perhaps `GradientNNCone`. When performing first order unconstrained optimization, we usually slow down progress when the norm of the gradient is small. In the limit, we expect to converge to a stationary point where the gradient is zero. When performing optimization under nonnegative constraints, and even further restrictions like simplex constraints, is makes more sense to ignore entries of the gradient where `X` is negative and the gradient is positive since those coordinates of `X` will not change after they are projected back to the nonnegative constraint.

TODO add theory of negative gradient is in the normal cone of the constraint?

As many or as few of these stats can be used in the call to `factorize`. Their tolerances can also be set independently of each other. The algorithm stops iterating when at least one of the criteria has a value less than its tolerance. As a fail safe, we also define one more type, `Iteration`, which is always active and halts the algorithm when that many iterations have past.

The `AbstractStat` type can also be subtyped to create custom stats that may be used to probe the iterates and diagnose issues.

```
EuclidianStepSize
EuclidianLipschitz
FactorNorms
```

These stats are recorded every iteration in a `DataFrame` and are one of the returns of `factorize`.

Finally, there are two auxiliary stats `PrintStats` and `DisplayDecomposition` which can be used to print all stats or the current decomposition each iteration.

4.2.b Constraints

- one type of update (other than the typical GD update)
- can combine them with composition
  - which is different than projecting onto their intersection!
- Constraint updates combine the constraint with how they are enforced
  - need to go together since there are multiple ways to enforce them e.g. simplex (see next section)

One of the main motivations for developing BlockTensorDecomposition.jl is to solve constrained tensor problems. Other code did not have the expressivity to handle constraints beyond the most common constraints on the factors: nonnegativity and Euclidean normalized columns. To enable flexibility, BlockTensorDecomposition.jl defines

```
abstract type AbstractConstraint <: Function end
```

which has two interfaces. The first treats the constraint like a function

```
(C::AbstractConstraint)(A::AbstractArray)
```

and applies the constraint to an abstract array, and the other interface

```
check(C::AbstractConstraint, A::AbstractArray)
```

checks if the array `A` satisfies the constraint `C`. A generic constrain only needs to define these two functions.

```
struct GenericConstraint <: AbstractConstraint
    apply::Function # input a AbstractArray -> mutate it so that `check` would
return true
    check::Function
end

function (C::GenericConstraint)(D::AbstractArray)
    (C.apply)(D)
end

check(C::GenericConstraint, A::AbstractArray) = (C.check)(A)
```

In general, the function `check` could be defined as the following,

```
function check(C, A)
    A_copy = copy(A)
    C(A)
    return A ≈ A_copy
end
```

but there is often an easier way to check if a tensor is constrained than to apply the constraint.

Although our basic block projected gradient descent algorithm Equation 10 relies on Euclidean projections to the relevant constraint set, we want to remain flexible and allow for other maps that move an iterate to the constraint set. So each constraint needs to store more than just the constraint itself (the *what*), but also the map from an iterate to the constraint set (the *how*). When prototyping algorithms, it is worth comparing alternate approaches to see if there is a more efficient method to enforce a given constraint (See Section 5 for an example of constraining to a set without Euclidean projections).

The first class of constraints are entry-wise constraints. This is convenient for defining a scalar function that gets applied to all entries of a tensor.

```julia
struct Entrywise <: AbstractConstraint
    apply::Function
    check::Function
end

function (C::Entrywise)(A::AbstractArray)
    A .= (C.apply).(A)
end

check(C::Entrywise, A::AbstractArray) = all((C.check).(A))
```

Some common examples would be a nonnegative constraint, constraining entries to an interval, or constraining entries to be one or zero.

```julia
nonnegative! = Entrywise(x -> max(0, x), x ≥ 0)

IntervalConstraint(a, b) = Entrywise(x -> clamp(x, a, b), x -> a ≤ x ≤ b)

binary! = Entrywise(x > 0.5 ? one(x) : zero(x), x -> x in (0, 1))
```

Another class of constraints are normalizations $\mathcal{C}_{\|\cdot\|_a} = \left\{ v \mid \|v\|_a = 1 \right\}$ for some norm $\|\cdot\|_a$. These can be enforced by a (Euclidean) projection

$$\hat{v} \in \arg\min_{u \in \mathcal{C}_{\|\cdot\|_a}} \|u - v\|_2^2,$$

or a scaling (assuming $v \neq 0$)

$$\hat{v} \leftarrow \frac{v}{\|v\|_a}.$$

These operations agree for the Frobenius norm (entry-wise 2-norm), but are different operations in general. In BlockTensorDecomposition.jl, we define these classes as the following.

TODO simplify the following code?

```julia
abstract type AbstractNormalization <: AbstractConstraint end
```

```julia
struct ProjectedNormalization <: AbstractNormalization
    norm::Function # calculate the norm of an AbstractArray
    projection::Function # mutate an AbstractArray so it has norm == 1
    whats_normalized::Function # what part of the array is normalized
                               # e.g. eachrow, or omit for the entire array
end

ProjectedNormalization(norm, projection; whats_normalized=identityslice) =
    ProjectedNormalization(norm, projection, whats_normalized)

function (P::ProjectedNormalization)(A::AbstractArray)
    whats_normalized_A = P.whats_normalized(A)
    (P.projection).(whats_normalized_A)
end

check(P::ProjectedNormalization, A::AbstractArray) =
    all((P.norm).(P.whats_normalized(A)) .≈ 1)
```

```julia
struct  ScaledNormalization{T<:Union{Real,AbstractArray{<:Real},Function}}  <:
AbstractNormalization
    norm::Function # calculate the norm of an AbstractArray
    whats_normalized::Function # what part of the array is normalized
    scale::T # the scale that `whats_normalized` should be normalized to
             # Can be a real, array of reals, or function of the input that
             # that returns a real or array of reals
end

ScaledNormalization(norm;whats_normalized=identityslice,scale=1) =
    ScaledNormalization{typeof(scale)}(norm, whats_normalized, scale)

function          (S::ScaledNormalization{T})(A::AbstractArray)          where
{T<:Union{Real,AbstractArray{<:Real}}}
    whats_normalized_A = S.whats_normalized(A)
    A_norm = (S.norm).(whats_normalized_A) ./ S.scale
    whats_normalized_A ./= A_norm
    return A_norm
end

function (S::ScaledNormalization{T})(A::AbstractArray) where {T<:Function}
    whats_normalized_A = S.whats_normalized(A)
    A_norm = (S.norm).(whats_normalized_A) ./ S.scale(A)
    whats_normalized_A ./= A_norm
    return A_norm
end

check(S::ScaledNormalization{<:Union{Real,AbstractArray{<:Real}}},
A::AbstractArray) =
    all((S.norm).(S.whats_normalized(A)) .≈ S.scale)
```

```
check(S::ScaledNormalization{<:Function}, A::AbstractArray) =
    all((S.norm).(S.whats_normalized(A)) .≈ S.scale(A))
```

We can define some common constraints like L_1 normalized through a projection

```
l1normalize! =
    ProjectedNormalization(l1norm, l1project!)
l1normalize_rows! =
    ProjectedNormalization(l1norm, l1project!; whats_normalized=eachrow)
l1normalize_1slices! =
    ProjectedNormalization(l1norm, l1project!;
        whats_normalized=(x -> eachslice(x; dims=1)))
```

or L_∞ normalized by scaling

```
linftyscale_cols! = ScaledNormalization(linftynorm; whats_normalized=eachcol)
```

or even normalize the 3-fibres of a third order tensor on average!

```
l2scale_average12slices! = ScaledNormalization(l2norm;
    whats_normalized=(x -> eachslice(x; dims=1)),
    scale=(A -> size(A, 2)))
```

Constraints can also be composed. This is *not* the same as the intersection of constraints. This is only a handy way to apply multiple constraints in series, but there is no clever logic that interprets the constraints and combines them into a single constraint.

```
struct  ComposedConstraint{T<:AbstractConstraint,  U<:AbstractConstraint}  <:
AbstractConstraint
    outer::T
    inner::U
end

function (C::ComposedConstraint)(A::AbstractArray)
    C.inner(A)
    C.outer(A)
end

check(C::ComposedConstraint, A::AbstractArray) =
    check(C.outer, A) & check(C.inner, A)

Base.:∘(f::AbstractConstraint, g::AbstractConstraint) =
    ComposedConstraint(f, g)
```

This means the following three constraints are all different!

```
l1normalize! ∘ nonnegative!
nonnegative! ∘ l1normalize!
simplex!
```

See Section 5 for a discussion on why it may be advantages to use one of these constraints over the other.

4.2.c `BlockUpdate` Language

- construct the updates as a list of updates
- very functional programming
- can apply them in sequence or in a random order (or partially random)

To put all these pieces together, we define an abstract type that can be subtyped for the various types of update. This will include gradient descent steps, momentum updates, and constraint enforcing updates.

```
abstract type AbstractUpdate <: Function end
```

A generic update is a function that can take some keyword arguments and mutate an iterate.

```
struct GenericUpdate <: AbstractUpdate
    f::Function
end

(U::GenericUpdate)(x; kwargs...) = U.f(x; kwargs...)
```

The first types of updates are gradient descent updates.

```
abstract type AbstractGradientDescent <: AbstractUpdate end
```

This includes the regular gradient descent update, and also the sub-block descent updates.

```
struct GradientDescent <: AbstractGradientDescent
    n::Integer
    gradient::Function
    step::AbstractStep
end

function (U::GradientDescent)(x; x_last, kwargs...)
    n = U.n
    if checkfrozen(x, n)
        return x
    end
    grad = U.gradient(x; kwargs...)
    # Note we pass a function for grad_last (lazy) so that we only compute it
```

```julia
if needed for the step
     s = U.step(x; n, x_last, grad, grad_last=(x -> U.gradient(x; kwargs...)),
kwargs...)
    a = factor(x, n)
    @. a -= s*grad
end
```

The only addition to `BlockGradientDescent` is a function that combines the step with the gradient.

```julia
struct BlockGradientDescent <: AbstractGradientDescent
    n::Integer
    gradient::Function
    step::AbstractStep
     combine::Function # takes a step (number, matrix, or tensor) and combines
it with a gradient
end

function (U::BlockGradientDescent)(x; x_last, kwargs...)
    n = U.n
    if checkfrozen(x, n)
        return x
    end
    grad = U.gradient(x; kwargs...)
     # Note we pass a function for grad_last (lazy) so that we only compute it
if needed for the step
     s = U.step(x; n, x_last, grad, grad_last=(x -> U.gradient(x; kwargs...)),
kwargs...)
    a = factor(x, n)
    a .-= U.combine(grad, s)
end
```

A separate step type is defined to allow for different types of steps like constant step, Lipschitz, and spectral projected gradient (SPG).

```julia
abstract type AbstractStep <: Function end
```

```julia
struct LipschitzStep <: AbstractStep
    lipschitz::Function
end

function (step::LipschitzStep)(x; kwargs...)
    L = step.lipschitz(x)
    return L^(-1) # allow for Lipschitz to be a diagonal matrix
end
```

```
function (step::LipschitzStep)(x::Tucker; kwargs...)
    L = step.lipschitz(x)
    if typeof(L) <: Tuple # Currently the only case is when we are updating the
core of a Tucker factorization
                          # Using this condition as a way to tell if it is the
core we are calculating the constant for
        return map(X -> X^(-1), L)
    else
        return L^(-1) # allow for Lipschitz to be a diagonal matrix
    end
end
```

```
struct ConstantStep <: AbstractStep
    stepsize::Real
end

(step::ConstantStep)(x; kwargs...) = step.stepsize
```

```
struct SPGStep <: AbstractStep
    min::Real
    max::Real
end

SPGStep(;min=1e-10, max=1e10) = SPGStep(min, max)

# Convert an input of the full decomposition, to a calculation on the nth factor
(step::SPGStep)(x::T; n, x_last::T, grad_last::Function, kwargs...) where {T <:
AbstractDecomposition} =
      step(factor(x,n);  x_last=factor(x_last,n),  grad_last=grad_last(x_last),
kwargs...)

function   (step::SPGStep)(x;   grad,   x_last,   grad_last,   stepmin=step.min,
stepmax=step.max, kwargs...)
    s = x - x_last
    y = grad - grad_last
    sy = (s ⋅ y)
    if sy <=0
        return stepmax
    else
        suggested_step = (s ⋅ s) / sy
        return clamp(suggested_step, stepmin, stepmax)
    end
end
```

The gradient and Lipschitz stepsize is calculated by a separate function that gets make on initialization of the `factorization` algorithm (See Section 3.2.b and Section 3.2.c). This de-couples applying the gradient descent (the *what*), and the computation of the gradient (the *how*) so that the gradient can be calculated manually (if an efficient method is known), or with automatic differentiation. This also allows other updates to use the same computation code. For example, momentum updates also use the same Lipschitz calculation function. Momentum updates also use the same `combine` function as the sub-block gradient descent updates.

```julia
struct MomentumUpdate <: AbstractUpdate
    n::Integer
    lipschitz::Function
    combine::Function # How to combine the momentum variable `ω` with a factor
`a`
end

MomentumUpdate(n, lipschitz) = MomentumUpdate(n, lipschitz, (ω, a) -> ω * a)

function MomentumUpdate(GD::AbstractGradientDescent)
    n, step = GD.n, GD.step
    @assert typeof(step) <: LipschitzStep

    return MomentumUpdate(n, step.lipschitz)
end

function MomentumUpdate(GD::BlockGradientDescent)
    n, step, combine = GD.n, GD.step, GD.combine
    @assert typeof(step) <: LipschitzStep

    return MomentumUpdate(n, step.lipschitz, combine)
end
```

Constraints also get their own abstract subtype of updates.

```julia
abstract type ConstraintUpdate <: AbstractUpdate end
```

We define a constructor for applying a constraint to a particular factor n. This turns an `AbstractConstraint` into a `ConstraintUpdate`.

```julia
ConstraintUpdate(n, constraint::GenericConstraint; kwargs...) =
    GenericConstraintUpdate(n, constraint)
ConstraintUpdate(n, constraint::ProjectedNormalization; kwargs...) =
    Projection(n, constraint)
ConstraintUpdate(n, constraint::Entrywise; kwargs...) =
    Projection(n, constraint)

function          ConstraintUpdate(n,         constraint::ScaledNormalization;
skip_rescale=false, whats_rescaled=missing, kwargs...)
    if skip_rescale
        ismissing(whats_rescaled) ||
        isnothing(whats_rescaled) ||
        @warn "skip_rescale=true but whats_rescaled=$whats_rescaled was given.
Overriding to whats_rescaled=nothing"
        return Rescale(n, constraint, nothing)
    else
        return Rescale(n, constraint, whats_rescaled)
    end
end
```

A generic constraint update extracts the factor it needs to constrain, and applies the constraint to that factor.

```julia
struct GenericConstraintUpdate <: ConstraintUpdate
    n::Integer
    constraint::GenericConstraint
end

check(U::GenericConstraintUpdate,          D::AbstractDecomposition)          =
check(U.constraint, factor(D, U.n))

function (U::GenericConstraintUpdate)(x::T; kwargs...) where T
    n = U.n
    A = factor(x, n)
    U.constraint(A)
    check(U, A) ||
                 error("Something  went  wrong  with  GenericConstraintUpdate:
$GenericConstraintUpdate")
end
```

Projections follow this pattern closely.

```julia
struct Projection <: ConstraintUpdate
    n::Integer
    proj::Union{ProjectedNormalization, Entrywise}
end

check(P::Projection, D::AbstractDecomposition) = check(P.proj, factor(D, P.n))

function (U::Projection)(x::T; kwargs...) where T
    n = U.n
    U.proj(factor(x, n))
end
```

A more involved type of constraint update is a rescaled update. This uses a scaled normalization, but moves the weight of the factor to other factors.

For example, if we have three matrix factors A, B, C in a CP decomposition where we want the sum of the entries in A to sum to 1, we can divide A by its sum, and multiple factor B by this amount. That way the recombined tensor $[\![A, B, C]\!]$ remains unchanged, but now A satisfies the desired constraint. We could instead multiply both B and C by the square root of the sum to achieve a similar outcome. When it is not specified what is rescaled (`whats_rescaled = missing`), we assume we should multiply every other factor by the geometric mean of the scaling. If we just want to scale the factor but skip any rescaling, we can use `whats_rescaled = nothing`.

```julia
struct Rescale{T<:Union{Nothing,Missing,Function}} <: ConstraintUpdate
    n::Integer
    scale::ScaledNormalization
    whats_rescaled::T
end

check(S::Rescale, D::AbstractDecomposition) = check(S.scale, factor(D, S.n))

function (U::Rescale{<:Function})(x; kwargs...)
    Fn_scale = U.scale(factor(x, U.n))
    to_scale = U.whats_rescaled(x)
    to_scale .*= Fn_scale
end

(U::Rescale{Nothing})(x; kwargs...) =
    U.scale(factor(x, U.n))

function (U::Rescale{Missing})(x; skip_rescale=false, kwargs...)
    Fn_scale = U.scale(factor(x, U.n))
    x_factors = factors(x)
    N = length(x_factors) - 1

    # Nothing to rescale, so return here
    if N == 0 || skip_rescale
        return nothing
    end

     # Assume we want to evenly rescale all other factors by the Nth root of
Fn_scale
    scale = geomean(Fn_scale)^(1/N)
    for (i, A) in zip(eachfactorindex(x), x_factors)
        # skip over the factor we just updated
        if i == U.n
            continue
        end
        A .*= scale
    end
end
```

See Section 5 for a more in-depth discussion on when it may be beneficial to use this type of constraint update over a simple projection.

We would like to combine all these updates to execute them in serial. We use a `BlockedUpdate` type to do this. This should be thought of as a list of `AbstractUpdate`s that get applied one after another.

```
struct BlockedUpdate <: AbstractUpdate
    updates::Vector{AbstractUpdate}
end
```

! Technical Julia Note

We need the updates to be exactly something of the form `AbstractUpdate[]` since we want to push any type of `AbstractUpdate`s such as a `MomentumUpdate` or another `BlockedUpdate`, even if not already present. This means it cannot be `Vector{<:AbstractUpdate}` since a `BlockedUpdate` constructed with only `GradientDescent` would give a `GradientDescent[]` vector and we couldn't push a `MomentumUpdate`. And it cannot be `AbstractVector{AbstractUpdate}` since we may not be able to `insert!` or `push!` into other `AbstractVector`s like `View`s.

We forward many standard methods so that `BlockedUpdate`s can behave like usual Julia vectors.

TODO should I actually show all these details?

```
Base.getindex(U::BlockedUpdate, i::Int) = getindex(updates(U), i)
Base.getindex(U::BlockedUpdate, I::Vararg{Int}) = getindex(updates(U), I...)
Base.getindex(U::BlockedUpdate, I) = getindex(updates(U), I) # catch all
Base.firstindex(U::BlockedUpdate) = firstindex(updates(U))
Base.lastindex(U::BlockedUpdate) = lastindex(updates(U))
Base.keys(U::BlockedUpdate) = keys(updates(U))
Base.length(U::BlockedUpdate) = length(updates(U))
Base.iterate(U::BlockedUpdate,  state=1)  =  state  >  length(U)  ?  nothing  :
(U[state], state+1)
Base.filter(f, U::BlockedUpdate) = BlockedUpdate(filter(f, updates(U)))
```

We define how `BlockedUpdate`s get applied to a tensor with the following function.

```
function      (U::BlockedUpdate)(x::T;      recursive_random_order::Bool=false,
random_order::Bool=recursive_random_order, kwargs...) where T
    U_updates = updates(U)
    if random_order
        order = shuffle(eachindex(U_updates))
        U_updates = U_updates[order]
    end

    for update! in U_updates
        update!(x; recursive_random_order, kwargs...)
        # note random_order does not get passed down
    end
end
```

The default order the blocks are updated is cyclically through each factor of the decomposition `D::AbstractDecomposition`, in the order of `factors(D)`. For `AbstractTucker` decompositions like Tucker, Tucker-1, and CP, this means starting with the core, followed by the matrix factor for the first dimension, second dimension, and so on.

As an example, this would be the default order of updates for nonnegative CP decomposition on an order 3 tensor.

```
BlockedUpdate(
    MomentumUpdate(1, lipschitz)
    GradientStep(1, gradient, LipschitzStep)
    Projection(1, Entrywise(ReLU, isnonnegative))
    MomentumUpdate(2, lipschitz)
    GradientStep(2, gradient, LipschitzStep)
    Projection(2, Entrywise(ReLU, isnonnegative))
    MomentumUpdate(3, lipschitz)
    GradientStep(3, gradient, LipschitzStep)
    Projection(3, Entrywise(ReLU, isnonnegative))
)
```

The order of updates can be randomized with the `random_order` keyword.

```
X, stats, kwargs = factorize(Y; random_order=true)
```

By default, this will keep momentum steps, gradient steps, and constraint steps for each factor together as a block, in this order.

A possible order of updates could be the following. Note that the updates for each factor are grouped together, but each factor is updated in a random order.

```
BlockedUpdate(
    BlockedUpdate(
        MomentumUpdate(2, lipschitz)
        GradientStep(2, gradient, LipschitzStep)
        Projection(2, Entrywise(ReLU, isnonnegative))
    )
    BlockedUpdate(
        MomentumUpdate(1, lipschitz)
        GradientStep(1, gradient, LipschitzStep)
        Projection(1, Entrywise(ReLU, isnonnegative))
    )
    BlockedUpdate(
        MomentumUpdate(3, lipschitz)
        GradientStep(3, gradient, LipschitzStep)
        Projection(3, Entrywise(ReLU, isnonnegative))
    )
)
```

For more randomization, use the `recursive_random_order` keyword which will also randomize the order in which the momentum steps, gradient steps, and constraint steps are performed.

```
X, stats, kwargs = factorize(Y; recursive_random_order=true)
```

A possible order of updates could now be the following. The updates for each factor are still grouped together, but the updates within each block appear in a random order.

```
BlockedUpdate(
    BlockedUpdate(
        Projection(2, Entrywise(ReLU, isnonnegative))
        MomentumUpdate(2, lipschitz)
        GradientStep(2, gradient, LipschitzStep)
    )
    BlockedUpdate(
        MomentumUpdate(1, lipschitz)
        Projection(1, Entrywise(ReLU, isnonnegative))
        GradientStep(1, gradient, LipschitzStep)
    )
    BlockedUpdate(
        GradientStep(3, gradient, LipschitzStep)
        Projection(3, Entrywise(ReLU, isnonnegative))
        MomentumUpdate(3, lipschitz)
    )
)
```

The opposite of this would be to keep the outer order of blocks as given, but randomize the order which the updates for each factor gets applied, use the following code.

```
X, stats, kwargs = factorize(Y; recursive_random_order=true, random_order=false,
group_by_factor=true)
```

A possible order of updates could now be the following. Note the order of factors is preserved (1, 2, 3) but the inner `BlockedUpdates` have a random order.

```
BlockedUpdate(
    BlockedUpdate(
        Projection(1, Entrywise(ReLU, isnonnegative))
        MomentumUpdate(1, lipschitz)
        GradientStep(1, gradient, LipschitzStep)
    )
    BlockedUpdate(
        MomentumUpdate(2, lipschitz)
        Projection(2, Entrywise(ReLU, isnonnegative))
        GradientStep(2, gradient, LipschitzStep)
    )
    BlockedUpdate(
        GradientStep(3, gradient, LipschitzStep)
        MomentumUpdate(3, lipschitz)
        Projection(3, Entrywise(ReLU, isnonnegative))
    )
)
```

Note all the previously mentioned options still keeps the various updates for each factor together. For full randomization, use the following code.

```
X,     stats,     kwargs     =     factorize(Y;     recursive_random_order=true,
group_by_factor=false)
```

A possible order of updates could now be the following. Note that every update can appear anywhere in the order.

```
BlockedUpdate(
    Projection(3, Entrywise(ReLU, isnonnegative))
    MomentumUpdate(2, lipschitz)
    GradientStep(2, gradient, LipschitzStep)
    MomentumUpdate(1, lipschitz)
    GradientStep(1, gradient, LipschitzStep)
    Projection(2, Entrywise(ReLU, isnonnegative))
    MomentumUpdate(3, lipschitz)
    MomentumUpdate(2, lipschitz)
    Projection(1, Entrywise(ReLU, isnonnegative))
    GradientStep(3, gradient, LipschitzStep)
)
```

The complete behaviour is summarized in Table 3.

We also use `BlockedUpdate` to handle a composition of constraints.

```
ConstraintUpdate(n, constraint::ComposedConstraint; kwargs...) =
    BlockedUpdate(ConstraintUpdate(n, constraint.inner; kwargs...),
                  ConstraintUpdate(n, constraint.outer; kwargs...))
end
```

The `BlockUpdate` language becomes especially helpful for automaically inserting additional updates as they are requested. For example, if we want to add momentum to a list of updates, we can call the following function.

```julia
function add_momentum!(U::BlockedUpdate)
    # Find all the GradientDescent updates
    U_updates = updates(U)
    indexes = findall(u -> typeof(u) <: AbstractGradientDescent, U_updates)

    # insert MomentumUpdates before each GradientDescent
    # do this in reverse order so "i" correctly indexes a GradientDescent
    # as we mutate updates
    for i in reverse(indexes)
        insert!(U_updates, i, MomentumUpdate(U_updates[i]))
    end
end
```

We can also interlace two lists of updates to put updates on the same factor next to each other, or group all updates by the factor they act on.

```julia
function smart_interlace!(U::BlockedUpdate, other_updates)
    for V in other_updates
        smart_insert!(U::BlockedUpdate, V::AbstractUpdate)
    end
end

smart_interlace!(U::BlockedUpdate,    V::BlockedUpdate)    =    smart_interlace!
(U::BlockedUpdate, updates(V))

function smart_insert!(U::BlockedUpdate, V::AbstractUpdate)
    U_updates = updates(U)
    i = findlast(u -> u.n == V.n, U_updates)

    # insert the other update immediately after
    # or if there is no update, push it to the end
    isnothing(i) ? push!(U_updates, V) : insert!(U_updates, i+1, V)
end
```

```julia
function group_by_factor(blockedupdate::BlockedUpdate)
    factor_labels = unique(getproperty(U, :n) for U in blockedupdate)
      updates_by_factor  =  [filter(U  ->  U.n  ==  n,  blockedupdate)  for  n  in
factor_labels]
    return BlockedUpdate(updates_by_factor)
end
```

5 Rescaling to Constrain Tensor Factorization

- for bounded linear constraints
  - first project
  - then rescale to enforce linear constraints
- faster to execute then a projection
- often does not loose progress because of the rescaling (decomposition dependent)

Constraints on factors in a tensor decomposition can arise naturally when modeling physical problems. A common class of constraints is normalizations which restrict a factor or slices of a factor to have unit norm. These are sometimes intersected with interval constraints such as requiring entries to be nonnegative.

For example, the demixing of R probibility densities $b_1, ..., b_R$ from I mixtures $y_1, ..., y_I$ for $I > R$ can be accomplished with a nonneagive matrix factorization (cite). We have the system of equations

$$\begin{aligned} y_1 &= a_{11}\, b_1 + a_{12}\, b_2 + ... + a_{1R}\, b_R \\ y_2 &= a_{21}\, b_1 + a_{22}\, b_2 + ... + a_{2R}\, b_R \\ &\vdots \\ y_I &= a_{I1}\, b_1 + a_{I2}\, b_2 + ... + a_{IR}\, b_R \end{aligned}$$

with unknown mixing coefficients $a_{i,r}$ and densities b_r. If we can discritized the mixtures y_i, we can rewrite this system as a rank R factorization of the matrix Y where $Y[i, :] = y_i$ and

$$\begin{bmatrix} \leftarrow & y_1^\top & \rightarrow \\ \leftarrow & y_2^\top & \rightarrow \\ \leftarrow & \vdots & \rightarrow \\ \leftarrow & y_I^\top & \rightarrow \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1R} \\ a_{21} & a_{22} & ... & a_{2R} \\ \vdots & & & \vdots \\ a_{I1} & a_{I2} & ... & a_{IR} \end{bmatrix} \begin{bmatrix} \leftarrow & b_1^\top & \rightarrow \\ \leftarrow & b_2^\top & \rightarrow \\ & \vdots & \\ \leftarrow & b_R^\top & \rightarrow \end{bmatrix}$$

$$Y = AB,$$

for matricies A and B.

For the factorization to remain interpretable, we need to ensure each row b_r of B is a density. This means we would like to constrain each row $B[r, :] = b_r$ to the simplex[7]

$$b_r \in \Delta_J = \left\{ v \in \mathbb{R}^J \,\middle|\, \sum_{j=1}^{J} v[j] = 1 \quad \text{and} \quad \forall j \in [J],\ v[j] \geq 0 \right\},$$

which we can write as constraining the matrix B to the simplex

$$B \in \Delta_{R,J} = \left\{ B \in \mathbb{R}^{R\times J} \,\middle|\, \forall r \in [R], \sum_{j=1}^{J} B[r, j] = 1 \quad \text{and} \quad \forall (r, j) \in [R] \times [J], B[r, j] \geq 0 \right\}.$$

[7]This assumes the entries $b_r[j]$ in the discretization of the density b_r represent probibilities or areas under some continuous 1D density function, not the sample values of the density $b_r(x_j)$. Assuming J sample points x_j of a grid on the interval $[x_0, x_J]$, the entries of the discretization vector can be defined as $b_r[j] = b_r(x_j)(x_j - x_{j-1})$.

Given the rows of B represent densities, to ensure the rows of the reconstructed matrix $\hat{Y} = AB$ are still densities, we need the mixing coefficients to be nonnegative $a_{ir} \geq 0$ and rows to sum to one $\sum_r a_{ir} = 1$. This constrains the matrix A to the simplex

$$A \in \Delta_{I,R} = \left\{ A \in \mathbb{R}^{I \times R} \;\middle|\; \forall i \in [I], \sum_{r=1}^{R} A[i,r] = 1 \quad \text{and} \quad \forall (i,r) \in [I] \times [R], A[i,r] \geq 0 \right\}.$$

5.1 The two approaches for simplex constraints

With constraints being defined in a flexible manner (see Section 4.2.b), we decided to test conventional wisdom that Euclidean projections are the right kind of map to use when enforcing a constraint. A constraint that came up in applications was enforcing the 1st mode slices of a tensor (e.g. rows in a matrix) or 3rd mode fibres of a third order tensor to lie in their respective simplex.

For example, to constrain a vector $v \in \mathbb{R}^J$ to the simplex

$$\Delta_J = \left\{ v \in \mathbb{R}^J \;\middle|\; \sum_{j=1}^{J} v[j] = 1, \quad \text{and} \quad \forall j \in [J],\ v[j] \geq 0 \right\},$$

we could apply a Euclidean projection

$$v \leftarrow \arg\min_{u \in \Delta_J} \|u - v\|_2^2,$$

or a generalized Kullback-Leibler divergence projection

$$v \leftarrow \arg\min_{u \in \Delta_J} \sum_j u[j] \log\left(\frac{u[j]}{v[j]}\right) - u[j] + v[j] \tag{22}$$

amoung other reasonable maps onto Δ_J.

The Eucledian simplex projection can be done with the following implimentation of Chen and Ye's algorithm [27]. The essence of the algorithm is to efficiently compute the special $t \in \mathbb{R}$ so that

$$v \leftarrow \max(0, v - t\mathbb{1}) \in \Delta_J. \tag{23}$$

The $\max(0, x)$ function should be understood as operating entrywise on x. In BlockTensorDecomposition, we use the helper `ReLU(x) = max(0, x)` for this function to assist with broadcasting.

```
function projsplx(v)
    J = length(v)

    if J==1 # quick exit for trivial length-1 "vectors" (i.e. scalars)
        return [one(eltype(v))]
    end

    v_sorted = sort(v[:]) # Vectorize/extract input and sort all entries
    j = J - 1
    t = 0 # need to ensure t has scope outside the while loop
    while true
        t = (sum(@view v_sorted[j+1:end]) - 1) / (J-j)
        if t >= v_sorted[j]
            break
        else
            j -= 1
        end

        if j >= 1
            continue
        else # j == 0
            t = (sum(v_sorted) - 1) / J
            break
        end
    end
    return ReLU.(v .- t)
end
```

This is turned into an operation that can mutated `v` with the following definition,

```
function projsplx!(y)
    y .= projsplx(y)
end
```

and can be turned into a `ProjectedNormalization` (see Section 4.2.b) with the following code.

```
simplex! = ProjectedNormalization(isnonnegative_sumtoone, projsplx!)
isnonnegative_sumtoone(x) = all(isnonnegative, x) && sum(x) ≈ 1
```

The generalized Kullback-Leibler divergence projection as stated in Equation 22 is only well-defined when $v[j] > 0$ for all $j \in [J]$. In this case the solution is given by

$$v \leftarrow \frac{v}{\sum_j v[j]}$$

which is well described by Ducellier et. al. [28 (Sec. 2.1)].

To extend the applicability of this map to any v (when there is at least one positive entry $v[j] > 0$), we can first (Euclidean) project onto the nonnegative orthant $\mathbb{R}_+^J$,

$$v \leftarrow \arg\min_{u \in \mathbb{R}_+^J} \|u - v\|_2^2 = \max(0, v),$$

and then apply the divergence projection.[8] All together, this looks like

[8]In the unfortunate case where every entry of v is nonpositive, we can fallback to the Euclean simplex projection.

$$v \leftarrow \frac{\max(0, v)}{\sum_j \max(0, v[j])}. \tag{24}$$

We will refer to Equation 24 as nonnegative projection and rescaling (NNPR). NNPR has the following implimentation in BlockTensorDecomposition.jl.

```
nnpr! = l1scale! ∘ nonnegative!
```

We define the two constraints as the following, using the constraint language from Section 4.2.b.

```
nonnegative! = Entrywise(ReLU, isnonnegative)
l1scale! = ScaledNormalization(l1norm)
l1norm(x) = mapreduce(abs, +, x)
```

5.2 Constraining Tensor Factorizations to Simplexes

6 Multi-scale

- use a coarse discretization along continuous dimensions
- factorize
- linearly interpolate decomposition to warm start larger decompositions

7 Conclusion

- all-in-one package
- provide a playground to invent new decompositions
- like auto-diff for factorizations

8 Appendix

8.1 Building the Hessian from two gradients

To build the Hessian from the definition of the gradient, we first extend the gradient to tensor-valued functions. For a function $F : \mathbb{R}^{J_1 \times \cdots \times J_N} \to \mathbb{R}^{I_1 \times \cdots \times I_M}$ where

$$F(X) = \left[f_{i_1 \ldots i_M}(X)\right]$$

is a tensor of scalar functions $f_{i_1 \ldots i_M} : T \to \mathbb{R}$, the gradient of F at X is defined entry-wise as

$$\begin{aligned} \nabla F(X)[i_1, \ldots, i_M][j_1, \ldots, j_N] &= \nabla f_{i_1 \ldots i_M}(X)[j_1, \ldots, j_N] \\ &= \frac{\partial f_{i_1 \ldots i_M}}{\partial X[j_1, \ldots, j_N]}(X). \end{aligned}$$

This treats the gradient at X as a tensor of tensors $\nabla F(X) \in \left(\mathbb{R}^{I_1 \times \cdots \times I_M}\right)^{J_1 \times \cdots \times J_N}$. This is naturally isomorphic to a tensor of order $M + N$ with entries

$$\nabla F(X)[i_1, ..., i_M, j_1, ..., j_N] = \frac{\partial f_{i_1 \dots i_M}}{\partial X[j_1, ..., j_N]}(X). \tag{25}$$

So we conclude that the gradient at X of a tensor-valued function F is $\nabla F : \mathbb{R}^{J_1 \times \dots \times J_N} \to \mathbb{R}^{I_1 \times \dots \times I_M \times J_1 \times \dots \times J_N}$ is given by Equation 25.

We can define the Hessian of a scalar function $f : \mathbb{R}^{I_1 \times \dots \times I_N} \to \mathbb{R}$ at X as $\nabla^2 f(X) = \nabla(\nabla f)(X) : \mathbb{R}^{I_1 \times \dots \times I_N} \to \mathbb{R}^{(I_1 \times \dots \times I_N)^2}$. The inner nabla ∇ is the gradient of the scalar function f, and the outer nabla ∇ is the gradient of the tensor-valued function ∇f.

This means

$$\nabla^2 f(A)[i_1, ..., i_N, j_1, ..., j_N] = \frac{\partial^2 f}{\partial A[j_1, ..., j_N] \partial A[i_1, ..., i_N]}(A),$$

but if the function has continuous second derivatives, we can perform the partial derivatives in either order

$$\frac{\partial^2 f}{\partial A[j_1, ..., j_N] \partial A[i_1, ..., i_N]}(A) = \frac{\partial^2 f}{\partial A[i_1, ..., i_N] \partial A[j_1, ..., j_N]}(A).$$

8.2 Randomizing the order of updates

Table 3: Full description of randomizing the order of updates within a `BlockUpdate`.

`group_by_factor`	`random_order`	`recursive_random_order`	Description
`false`	`false`	`false`	In the order given
`false`	`false`	`true`	In order given, but randomize how existing blocks are ordered (recursively)
`false`	`true`	`false`	Randomize updates, but keep existing blocks in order
`false`	`true`	`true`	Fully random
`true`	`false`	`false`	In the order given
`true`	`false`	`true`	In order of factors, but updates for each factor a random order
`true`	`true`	`false`	Random order of factors, preserve order of updates within each factor
`true`	`true`	`true`	Almost fully random, but updates for each factor are done together

Bibliography

[1] Martín Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems,” 2015. https://www.tensorflow.org/

[2] J. Ansel *et al.*, “PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, Apr. 2024, vol. 2, pp. 929–947. doi: 10.1145/3620665.3640366.

[3] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, “TensorLy: Tensor Learning in Python,” *Journal of Machine Learning Research*, vol. 20, no. 26, pp. 1–6, 2019, Accessed: Aug. 16, 2024. [Online]. Available: http://jmlr.org/papers/v20/18-277.html

[4] B. W. Bader and T. G. Kolda, “Tensor Toolbox for MATLAB.” Sep. 2023.

[5] J. Li, J. Bien, and M. T. Wells, “rTensor: An R Package for Multidimensional Array (Tensor) Unfolding, Multiplication, and Decomposition,” *Journal of Statistical Software*, vol. 87, pp. 1–31, Nov. 2018, doi: 10.18637/jss.v087.i10.

[6] Jutho, “Jutho/TensorKit.jl,” Aug. 2024. https://github.com/Jutho/TensorKit.jl (accessed Aug. 15, 2024).

[7] M. Abbott *et al.*, “mcabbott/Tullio.jl: v0.3.7,” Oct. 2023. https://doi.org/10.5281/zenodo.10035615

[8] A. Peter, “under-Peter/OMEinsum.jl,” Aug. 2024. https://github.com/under-Peter/OMEinsum.jl (accessed Aug. 16, 2024).

[9] Y.-J. Wu, “yunjhongwu/TensorDecompositions.jl,” Feb. 2024. https://github.com/yunjhongwu/TensorDecompositions.jl (accessed Aug. 16, 2024).

[10] Y. Xu and W. Yin, “A Block Coordinate Descent Method for Regularized Multiconvex Optimization with Applications to Nonnegative Tensor Factorization and Completion,” *SIAM Journal on Imaging Sciences*, vol. 6, no. 3, pp. 1758–1789, Jan. 2013, doi: 10.1137/120887795.

[11] J. Kim, Y. He, and H. Park, “Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework,” *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, Feb. 2014, doi: 10.1007/s10898-013-0035-4.

[12] Z. Yang and E. Oja, “Unified Development of Multiplicative Algorithms for Linear and Quadratic Nonnegative Matrix Factorization,” *IEEE Transactions on Neural Networks*, vol. 22, no. 12, pp. 1878–1891, Dec. 2011, doi: 10.1109/TNN.2011.2170094.

[13] T. G. Kolda and B. W. Bader, “Tensor Decompositions and Applications,” *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009, doi: 10.1137/07070111X.

[14] L. Qi, Y. Chen, M. Bakshi, and X. Zhang, “Triple Decomposition and Tensor Recovery of Third Order Tensors,” Mar. 01, 2020. http://arxiv.org/abs/2002.02259 (accessed Aug. 01, 2023).

[15] F. Wu, C. Li, and Y. Li, “Manifold Regularization Nonnegative Triple Decomposition of Tensor Sets for Image Compression and Representation,” *Journal of Optimization Theory and Applications*, vol. 192, no. 3, pp. 979–1000, Mar. 2022, doi: 10.1007/s10957-022-02001-6.

[16] I. V. Oseledets, “Tensor-Train Decomposition,” *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, Jan. 2011, doi: 10.1137/090752286.

[17] J. B. Kruskal, “Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics,” *Linear Algebra and its Applications*, vol. 18, no. 2, pp. 95–138, Jan. 1977, doi: 10.1016/0024-3795(77)90069-6.

[18] A. Bhaskara, M. Charikar, and A. Vijayaraghavan, “Uniqueness of Tensor Decompositions with Applications to Polynomial Identifiability,” in *Proceedings of The 27th Conference on Learning Theory*, May 2014, pp. 742–778. Accessed: Jan. 08, 2025. [Online]. Available: https://proceedings.mlr.press/v35/bhaskara14a.html

[19] S. A. Vavasis, “On the Complexity of Nonnegative Matrix Factorization,” *SIAM Journal on Optimization*, vol. 20, no. 3, pp. 1364–1377, Jan. 2010, doi: 10.1137/070709967.

[20] Y. Nesterov, “Nonlinear Optimization,” *Lectures on Convex Optimization.* Springer International Publishing, Cham, pp. 3–58, 2018. doi: 10.1007/978-3-319-91578-4_1.

[21] A. Virmaux and K. Scaman, “Lipschitz regularity of deep neural networks: analysis and efficient estimation,” in *Advances in Neural Information Processing Systems*, 2018, vol. 31. Accessed: Jan. 11, 2025. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/d54e99a6c03704e95e6965532dec148b-Abstract.html

[22] F. Kunstner, V. Sanches Portella, M. Schmidt, and N. Harvey, “Searching for Optimal Per-Coordinate Step-sizes with Multidimensional Backtracking,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 2725–2767, Dec. 2023, Accessed: Feb. 12, 2025. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/hash/07e436cdeb48e2a67618274f5d5eff85-Abstract-Conference.html

[23] W. Gao, Y.-C. Chu, Y. Ye, and M. Udell, “Gradient Methods with Online Scaling,” Nov. 2024. http://arxiv.org/abs/2411.01803 (accessed Feb. 07, 2025).

[24] W. Gao, Z. Qu, M. Udell, and Y. Ye, “Scalable Approximate Optimal Diagonal Preconditioning,” Nov. 2024. http://arxiv.org/abs/2312.15594 (accessed Feb. 11, 2025).

[25] Z. Qu, W. Gao, O. Hinder, Y. Ye, and Z. Zhou, “Optimal Diagonal Preconditioning,” *Operations Research*, p. opre.2022.0592, Mar. 2024, doi: 10.1287/opre.2022.0592.

[26] P. Tseng and S. Yun, “A coordinate gradient descent method for nonsmooth separable minimization,” *Mathematical Programming*, vol. 117, no. 1, pp. 387–423, Mar. 2009, doi: 10.1007/s10107-007-0170-0.

[27] Y. Chen and X. Ye, “Projection Onto A Simplex,” Feb. 2011. http://arxiv.org/abs/1101.6081 (accessed Aug. 18, 2023).

[28] A. Ducellier *et al.*, “Uncertainty Quantification under Noisy Constraints, with Applications to Raking,” Sep. 2024. http://arxiv.org/abs/2407.20520 (accessed Feb. 26, 2025).