BlockTensorFactorization.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 BlockTensorFactorization.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 stepsize 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: BlockTensorFactorization.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.

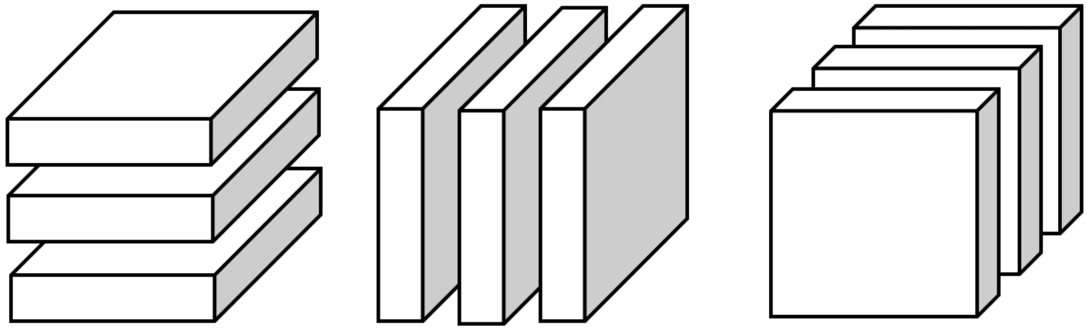
2.1.b Vectors, Matrices, and Tensors

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)`.

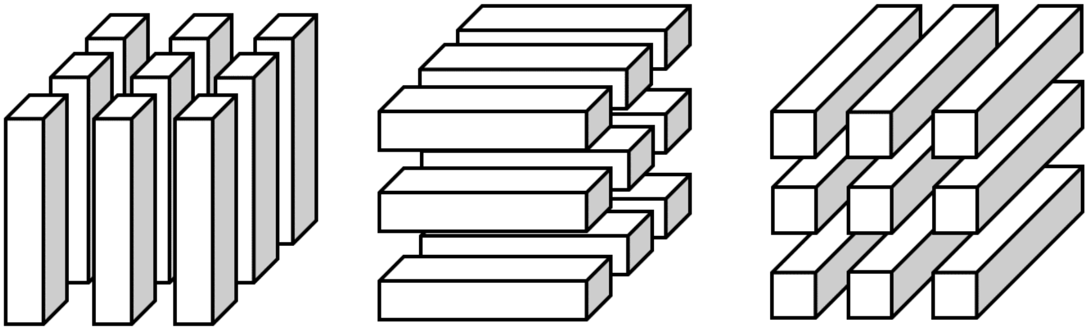


(a) horizontal slices *A[i, :, :]* (b) lateral slices *A[:, j, :]* (c) frontal slices *A[:, ;, k]*

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))`. BlockTensorFactorization.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 $\underbrace{I \times \cdots \times I}_{N \text{ times}}$ tensor where $\mathrm{id}_I[i_1, ..., i_N] = 1$ when $i_1 = ... = i_N \in [I]$, and is zero otherwise.

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

For a vector, matrix, or tensor filled with ones, we use $\mathbb{1} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. 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 $A \in \mathbb{R}^{I_1 \times \cdots \times I_M}$ and $B \in \mathbb{R}^{J_1 \times \cdots \times J_N}$ yields an order $M + N$ tensor $A \otimes B \in \mathbb{R}^{I_1 \times \cdots \times I_M \times J_1 \times \cdots \times J_N}$ 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 BlockTensorFactorization.jl

The Frobenius inner product between two tensors $A, B \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ 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

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

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 BlockTensorFactorization.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
```

BlockTensorFactorization.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 $A \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ and matrix $B \in \mathbb{R}^{I_n \times J}$, returns a tensor $(A \times_n B) \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$ with entries

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

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

```
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
```

```julia
    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 $\times_{1,\ldots,n} = \times_{1:n} = \times_{[n]}$ 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

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

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 $\times_{N-n+1,\ldots,N} = \times_{(N-n+1):N} = \times_{[-n]}$,

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

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 $2N$

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

we have

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

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 $2N$. This contrasts the Hessian *matrix* constructed by vectorizing the function f's input [13 (Ch. 10)].

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

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

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

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

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

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

⚠ 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

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

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.

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

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

```julia
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

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

The *Tucker Product* defined by Equation 5 is implemented in BlockTensorFactorization.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

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

```
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.

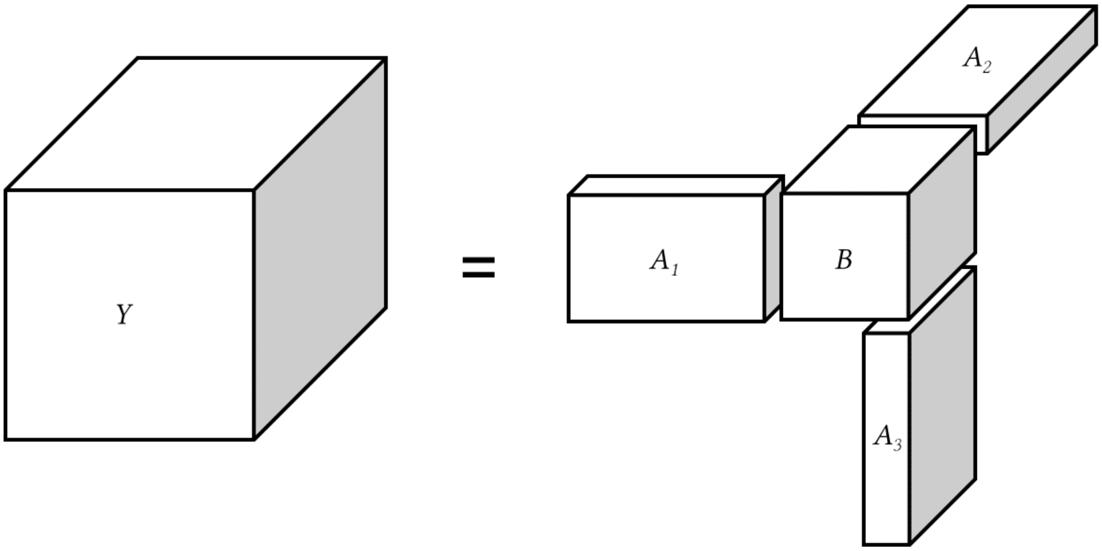


Figure 3: Tucker factorization of a 3rd order tensor Y.

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

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

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

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

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 [14], are specialized for order 3 tensors [15, 16], or provide alternate decomposition models entirely like tensor-trains [17]. 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 [14, 18, 19].

2.2.a Representing Tucker Decompositions

There are implemented in BlockTensorFactorization.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.

```julia
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

```julia
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
```

```julia
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 $Y = [\![A_1, \ldots, A_N]\!]$,

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

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 $Y = AB$

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

By convention, we say the zero tensor $Y = 0$ has rank 0.

For the Tucker and Tucker-n decompositions, we instead call a particular factorization **a** rank-$(R_1, \ldots, R_N)$ Tucker factorization or **a** rank-$(R_1, \ldots, R_n)$ 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

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

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 [20]. 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. See section Section 3.2.d for an implementation of this strategy.

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

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

for $t = 1, 2, ...$ 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 $\mathcal{C}_n$,

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

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 stepsizes can be used in theory [21 (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 BlockTensorFactorization.jl, `factorize`, is defined at a very high level.

```
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

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

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 BlockTensorFactorization.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.

```
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
```

[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.

```
        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

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

```
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:

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

where

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

```
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 [22].

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

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

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

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

```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.

```
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
```

3.2.d Estimating Tensor Rank

In many applications, the rank of the input tensor Y may not be known. In the case of CP and Tucker-1 decompositions, there are known bounds on the rank.

Lemma 3.1: Given a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, we have the following bounds on the Tucker-1 and CP rank [14 (Sec. 3.1)].

$$0 \leq \operatorname{rank}_{\text{CP}}(Y) \leq \min_n \prod_{m \neq n} I_m = \min(I_2 \cdots I_N, I_1 I_3 \cdots I_N, ..., I_1 \cdots I_{N-1}),$$

and

$$0 \leq \operatorname{rank}_{\text{Tucker-1}}(Y) \leq \min\left(I_1, \prod_{n=2}^{N} I_n\right) = \min(I_1, I_2 \cdots I_N).$$

We have a rank of 0 if and only if $Y = 0$ is the zero tensor.

Both of these bounds are natural extensions of the typical matrix rank bounds. For a matrix $Y \in \mathbb{R}^{I \times J}$, we know

$$0 \leq \operatorname{rank}\,(Y) \leq \min(I, J).$$

In the case that we try to factorize a matrix Y at a rank $R \geq \operatorname{rank}\,(Y)$, we should still be able to achieve a final objective of zero $0 = \frac{1}{2}\|A^* B^* - Y\|_F^2$, and at any rank $R < \operatorname{rank}\,(Y)$, the objective is positive and decreasing. This fact extends to CP and Tucker-1 decompositions. So a naive strategy for finding the rank of a tensor Y, would be first check if it is rank zero, and then try factorizing Y at incrementally larger ranks these two types of decompositions until the final objective hits zero.

This approach has two problems. With real work data, the input tensor is often a noisy version of a low rank tensor $Y = Y_{\text{clean}} + Z$. Noise is often full rank and would cause the input tensor Y to be the maximum rank [23]. Secondly, even if we had a perfectly low rank tensor Y, our algorithm at best converges to the optimal solution in the limit. So any finite stopping point has a positive objective.

A simple fix would be to extend the concept of numerical rank (also called ϵ-rank) to tensors.[4] This serves as an approximation for the rank where taking $\epsilon \to 0^+$ would return the usual definition of the rank.

Definition 3.2: The ϵ-rank [24] of a matrix A is the smallest rank obtainable by an ϵ-perturbation of the matrix $A + E$:

$$\operatorname{rank}_\epsilon(A) = \min_{\|E\| \leq \epsilon} \operatorname{rank}(A + E). \tag{13}$$

[4] See https://mpf-optimization-laboratory.github.io/opt-blog/posts/epsilon-rank/ for an in-depth discussion on this topic. TODO move some proofs to this document.

Here we use $\|E\| = \max_{\|v\|_2=1} \|Ev\|_2$ to denote the the operator norm.

This definition is *stable* in the sense that adding a small amount of noise to a matrix does not effect the ϵ-rank. This is unlike the traditional rank where adding noise to a low rank matrix can turn it into a full rank matrix, even for an "ϵ" amount of noise. This is clear from Proposition 3.1 which also offers a practical method for computing $\text{rank}_\epsilon(A)$.

Proposition 3.1 (Singular Value Characterization): The ϵ-rank of A is equal to the number of singular values of A strictly bigger than ϵ.

The following new result, Theorem 3.1, shows that we can use the ϵ-rank to compute the true rank of a matrix under some additive noise B with the right choice of ϵ.

Theorem 3.1 (Rank recovery): Let $A, B \in \mathbb{R}^{m\times n}$ where $\|B\|_2 < \sigma_r/2$. Then,

$$\text{rank}_\epsilon(A + B) = \text{rank}(A) = r$$

for all ϵ between $\|B\|_2 \leq \epsilon < \sigma_r/2$. Here, we list the singular values of A in non-increasing order: $\sigma_1 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_{\min(m,n)} = 0$.

Note there are no assumptions on the noise B. It could be fixed, or come from any distribution. In the special case that you know the distribution B comes from, you could use that information to estimate an appropriate choice of ϵ. Corollary 3.3 gives the case where $B = Z$ is standard Gaussian noise.

Corollary 3.3 (Rank Recovery with Gaussian Noise): Let $Z \in \mathbb{R}^{m\times n}$ be a Gaussian matrix with standard normal entries $Z_{ij} \sim \mathcal{N}(0, 1)$ and σ_r be the largest nonzero singular value of $A \in \mathbb{R}^{m\times n}$. If $\sigma_r > 2(\sqrt{m} + \sqrt{n})$, then for $(\sqrt{m} + \sqrt{n} + t) \leq \epsilon < \sigma_r/2$,

$$\text{rank}_\epsilon(A + Z) = \text{rank}(A)$$

with high probability at least $1 - 2\exp(-ct^2)$ for some constant $0 < c \in \mathbb{R}$.

Extending these results to constrained tensors factorization should be possible, but there presents multiple ways we could approach the problem. In the case of identifying rank for a signal decomposition, we are interested the smallest sum of sources under constraints on the allowed sources. For nonnegative sources, the well studied nonnegative rank [25] is most appropriate:

$$\text{rank}_+(X) = \min\left\{ R \in \mathbb{Z}_+ \;\middle|\; \exists A, B \geq 0 \;\text{ s.t. }\; X[i, j] = \sum_{r=1}^{R} A[i, r]B[r, j] \right\}. \tag{14}$$

Unfortunately, computing this is NP hard in general [20, 25, 26]. This means extending the ϵ-rank to something like an ϵ-nonnegative-rank may not be practically computable.

Even in the unconstrained case, we have only kicked the problem down the road and must estimate a suitable ϵ to extract the rank.

There are a number of proposed solutions to this problem that all follow the general principle of Occam's razor of selecting the model with the smallest rank that still achieves a reasonable fit with the data. For automated rank selection, there are many criteria used to balance simplicity with accuracy. The following is a non exhaustive list of methods used in practice in roughly chronological order:

- Akaike [27, 28] and Bayesian [29, 30] Information Criterion
- Numerical rank [24] and in combination with polynomial filtering [31]
- Matrix perturbation theory [32]
- Consensus clustering and cophenetic correlation coefficient [33, 34]
- Point of maximum curvature of the approximation error as a function of the rank [35]
- Cross validation [36]
- "SCREE" plot and segmented linear regression [37 (Sec. 3.2)]
- Minimum description length [38]
- Concordance [39]
- And hypothesis testing [40].

The number of ways to estimate rank—and lack of consensus on when to use each method—is best summarized by Fu et. al. [38]:

> "Even though the selection of rank is very important, in most studies on [non-negative tensor factorization] the value of R is simply determined by trial and error or specialists' insights, and there does not exist a good way to determine R automatically."

We take an approach similar to Satopaa [35], Saylor [37 (Sec. 3.2)], and Graham [41]. We first factorize the tensor at every possible rank (as given by the bounds in Lemma 3.1), and then look at the function that takes as input a rank r, and output the final relative error between the reconstructed decomposition at that rank X_r^* and the input tensor Y,

$$f(r) = \left\|X_r^* - Y\right\|_F / \|Y\|_F.$$

We select the "knee" or "elbow" of this function by finding the point of maximum curvature $\kappa_f(r)$ with finite differences,

$$\hat{R} = \arg\max_r \kappa_f(r) := \frac{f''(r)}{\left(1 + (f'(x))^2\right)^{3/2}}.$$

Since scaling the function f would influence the curvature, we standardize the function by scaling it to the $[0, 1]^2$ box. The following is the specific implementation of the rank finding algorithm, and method for calculating the standard curvature.

Since the maximum rank could be very large, we also include an option `online_rank_estimation` which stops looking are larger ranks if the standard curvature has noticeably dropped, implying a local maximum.

```julia
function rank_detect_factorize(Y; online_rank_estimation=false, rank=nothing, model=Tucker1, kwargs...)
    if isnothing(rank)
        # Initialize output and final error lists
        all_outputs = []
        final_rel_errors = Float64[]

        # Make sure RelativeError is part of the stats keyword argument
        kwargs = isempty(kwargs) ? Dict{Symbol,Any}() : Dict{Symbol,Any}(kwargs)
        get!(kwargs, :stats) do # If stats is not given, populate stats with RelativeError
            [Iteration, RelativeError, ObjectiveValue, isnonnegative(Y) ? GradientNNCone : GradientNorm]
        end
        if RelativeError ∉ kwargs[:stats] # If stats was given, make sure RelativeError is in the list stats
            kwargs[:stats] = [RelativeError, kwargs[:stats]...] # not using pushfirst! since kwargs[:stats] could be a Tuple
        end
        kwargs[:model] = model # add the model back into kwargs

        for rank in possible_ranks(Y, model)
            @info "Trying rank=$rank..."

            kwargs[:rank] = rank # add the rank into kwargs

            output = factorize(Y; kwargs...) # safe to call factorize (rather than _factorize) since both factorize and rank_detect_factorize have checks to see if the keyword `rank` is provided
            push!(all_outputs, output)
            _, stats, _ = output

            final_rel_error = stats[end, :RelativeError]
            push!(final_rel_errors, final_rel_error)
            @info "Final relative error = $final_rel_error"

            if (online_rank_estimation == true) && length(final_rel_errors) >= 3 # Need at least 3 points to evaluate curvature
                curvatures = standard_curvature(final_rel_errors)
                if curvatures[end] ≈ maximum(curvatures) # want the last curvature to be significantly smaller than the max
                    continue
                else
```

```
                    # we must have curvature[end] < maximum(curvature) so we
can now return
                    R = argmax(curvatures)
                    @info "Optimal rank found: $R"
                    return ((all_outputs[R])..., final_rel_errors)
                end
            end
        end

        # Return if online_rank_estimation == false, or a clear rank was not
found
        R = argmax(standard_curvature(final_rel_errors))
        @info "Optimal rank found: $R"
        return ((all_outputs[R])..., final_rel_errors)
    else
        return factorize(Y; rank, model, kwargs...)
    end
end
```

```
"""
Approximate first derivative with finite elements. Assumes y[i] = y(x_i) are
samples with unit spaced inputs x_{i+1} - x_i = 1.
"""
function d_dx(y::AbstractVector{<:Real})
    d = similar(y)
    each_i = eachindex(y)

    # centred estimate
    for i in each_i[begin+1:end-1]
        d[i] = (-y[i-1] + y[i+1])/2
    end

    # three point forward/backward estimate
    i = each_i[begin+1]
    d[begin] = (-3*y[i-1] + 4*y[i] - y[i+1])/2

    i = each_i[end-1]
    d[end] = (y[i-1] - 4*y[i] + 3*y[i+1])/2
    return d
end

function d2_dx2(y::AbstractVector{<:Real})
    d = similar(y)

    for i in eachindex(y)[begin+1:end-1]
        d[i] = y[i-1] - 2*y[i] + y[i+1]
    end
    # Assume the same second derivative at the end points
    d[begin] = d[begin+1]
    d[end] = d[end-1]
    return d
end

function standard_curvature(y::AbstractVector{<:Real}; kwargs...)
    # An interval 0:10 has length(0:10) = 11, but measure 10-0 = 10
    # hence the length(y) - 1
    Δx = 1 / (length(y) - 1)
    y_max = maximum(y)
    dy_dx = d_dx(y; kwargs...) / (Δx * y_max)
    dy2_dx2 = d2_dx2(y; kwargs...) / (Δx^2 * y_max)
    return @. dy2_dx2 / (1 + dy_dx^2)^1.5
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 BlockTensorFactorization.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:

$$a_{n,r}^{t+1} \leftarrow P_{\mathcal{C}_{n,r}}\left(a_{n,r}^t - \frac{1}{L_{n,r}^t}\nabla f_{n,r}^t\left(a_{n,r}^t\right)\right), \tag{15}$$

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

$$A_{n,r}^t(a) = \begin{bmatrix} \uparrow & & \uparrow & \uparrow & \uparrow & & \uparrow \\ a_{n,1}^{t+1} & \cdots & a_{n,r-1}^{t+1} & a & a_{n,r+1}^t & \cdots & a_{n,R_n}^t \\ \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \\ & & & & & & \end{bmatrix}. \tag{16}$$

In theory, the block update shown in Equation 15 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 15 by first calculating the gradient ∇f_n^t at

$$\hat{A}_n^t = \begin{bmatrix} \uparrow & & \uparrow & \uparrow & \uparrow & & \uparrow \\ a_{n,1}^{t} & \cdots & a_{n,r-1}^{t} & a_{n,r}^t & a_{n,r+1}^t & \cdots & a_{n,R_n}^t \\ \downarrow & & \downarrow & \downarrow & \downarrow & & \downarrow \\ & & & & & & \end{bmatrix}, \tag{17}$$

and then updating each sub-block r according to

$$a_{n,r}^{t+1} \leftarrow P_{\mathcal{C}_{n,r}}\left(a_{n,r}^t - \frac{1}{L_{n,r}^t}\nabla f_n^t\left(\hat{A}_n^t\right)\right). \tag{18}$$

Note the difference between Equation 16 and Equation 17 is that we don't use the most recent columns $a_{n,j}$ for $j < r$ in Equation 17.

The update given in Equation 18 can be merged back to an update on the whole block A_n

$$A_n^{t+1} \leftarrow P_{\mathcal{C}_n}\left(A_n^t - \nabla f_n^t\left(A_n^t\right)\left(\hat{L}_n^t\right)^{-1}\right) \tag{19}$$

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[5] 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 19 to the standard projected gradient descent update shown in Equation 10. The difference is that we calculate a “matrix step-size” $\hat{L}_n^t \in \mathbb{R}^{R_n \times R_n}$ 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 $L_n^t = \left\| \tilde{X}_n \cdot_n \tilde{X}_n \right\|_{\text{op}}$. 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

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

[5] 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.

in the merged sub-block update shown in Equation 19. 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

$$B^{t+1} \leftarrow P_{\mathcal{C}_B}\left(B^t - \nabla f_0^t(B^t) \times_B \left(\hat{L}_B^t\right)^{-1}\right) \tag{20}$$

with the multiplication

$$\begin{aligned} \nabla f_0^t(B^t) \times_B \left(\hat{L}_B^t\right)^{-1} &= \nabla f_0^t(B^t) \bigtimes_n \left(\hat{L}_{B,n}^t\right)^{-1} \\ &= \left\llbracket \nabla f_0^t(B^t); \left(\hat{L}_{B,1}^t\right)^{-1}, \ldots, \left(\hat{L}_{B,n}^t\right)^{-1} \right\rrbracket. \end{aligned} \tag{21}$$

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 [42–45]

```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

$$\hat{A}_n^t \leftarrow A_n^t + \omega_n^t \left(A_n^t - A_n^{t-1}\right) \tag{22}$$

for some amount of extrapolation $\omega_n^t \geq 0$ before applying the update Equation 19 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 [46].

Given a parameter[6] $\delta \in [0, 1)$, we define the momentum parameters and τ^t and ω_n^t according to the following updates

$$\begin{aligned} \tau^0 &= 1 \\ \tau^{t+1} &\leftarrow \frac{1}{2}\left(1 + \sqrt{1 + 4(\tau^t)^2}\right) \\ \hat{\omega}^t &\leftarrow \frac{\tau^t - 1}{\tau^{t+1}} \\ \omega_n^t &\leftarrow \min\left(\hat{\omega}^t, \delta\sqrt{\hat{L}_n^{t-1}\left(\hat{L}_n^t\right)^{-1}}\right). \end{aligned} \tag{23}$$

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 23 as operating element-wise. This also means the momentum parameter ω_n^t is a matrix or tensor and takes the same shape as $\hat{L}_n^t$.

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

In order to perform Equation 22, 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 21.

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 BlockTensorFactorization

fact = BlockTensorFactorization.factorize

options = (
    :rank => 3,
    :tolerance => (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 $(\text{old} - \text{new}) /$ 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.[7] 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

[7] 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.

- 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 BlockTensorFactorization.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. BlockTensorFactorization.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.

```
# 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 BlockTensorFactorization.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, BlockTensorFactorization.jl defines

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

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

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

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

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

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

```julia
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,

```julia
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 $\mathcal{C}_{\|\cdot\|_a} = \{v \mid \|v\|_a = 1\}$ 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 BlockTensorFactorization.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)))
```

Basic linear constrains AX=B are also implemented in the following manner. This constraint projects a factor X onto the affine space defined by AX=B for a linear operator or matrix A and bias B.

```
struct LinearConstraint{T <: Union{Function, AbstractArray}} <:
AbstractConstraint
    linear_operator::T
    bias::AbstractArray
end

check(C::LinearConstraint{Function}, X::AbstractArray) = C.linear_operator(X)
≈ C.bias
check(C::LinearConstraint{<:AbstractArray}, X::AbstractArray) =
C.linear_operator * X ≈ C.bias

# TODO implement linear constraint given an operator
function (C::LinearConstraint{Function})(X::AbstractArray)
    error("Linear Constraints defined in terms of an operator are not
implemented (YET!)")
end

function (C::LinearConstraint{<:AbstractArray})(X::AbstractArray)
    error("Linear Constraints defined in terms of a general array are not
implemented (YET!)")
end

function (C::LinearConstraint{<:AbstractMatrix})(X::AbstractArray)
    A = C.linear_operator
    b = C.bias
    X .-= A' * ( (A*A') \ (A*X .- b) ) # Projects X onto the subspace AX=b
end
```

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.

```julia
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 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).

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

```
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.

```
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.

```
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`.

```
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.

```
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.

```
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 $\llbracket A, B, C \rrbracket$ 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`.

```
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 `AbstractUpdates` 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 `AbstractUpdates` 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 `AbstractVectors` like `Views`.

We forward many standard methods so that `BlockedUpdates` 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 `BlockedUpdates` 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 automatically 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.

```
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.

```
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
```

```
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.

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 recent applications of tensor decomposition to geology [41] 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, the demixing of R probability densities $b_1, \ldots, b_R$ from I mixtures $y_1, \ldots, y_I$ for $I > R$ can be accomplished with a nonnegative matrix factorization (cite). We have the system of equations

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

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

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

$$Y = AB,$$

for matrices 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[8]

$$b_r \in \Delta_J = \left\{ v \in \mathbb{R}_+^J \;\middle|\; \sum_{j=1}^{J} v[j] = 1 \right\},$$

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

$$B \in \Delta_J^R = \left\{ B \in \mathbb{R}_+^{R \times J} \;\middle|\; \forall r \in [R], \sum_{j=1}^{J} B[r,j] = 1 \right\}.$$

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

$$A \in \Delta_R^I = \left\{ A \in \mathbb{R}_+^{I \times R} \;\middle|\; \forall i \in [I], \sum_{r=1}^{R} A[i,r] = 1 \right\}.$$

The question we investigate in this section is the following: how can we best constrain the factors A and B to their respective simplexes, while performing block gradient decent to minimize the least squared error $\frac{1}{2}\|AB - Y\|_F^2$?

5.1 The two approaches for simplex constraints

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

$$\Delta_J = \left\{ v \in \mathbb{R}_+^J \;\middle|\; \sum_{j=1}^{J} v[j] = 1 \right\},$$

we could apply a Euclidean projection

[8] This assumes the entries $b_r[j]$ in the discretization of the density b_r represent probabilities or areas under some continuous 1D density function, and not the sample values of the density $b_r(x_j)$. One possible discretization is to take J sample points x_j of a grid on the interval $[x_0, x_J]$ where b_r is supported, and define the entries of the discretization to be $B[r,j] = b_r[j] = b_r(x_j)(x_j - x_{j-1})$. For normalized densities $\int_{x_0}^{x_J} b_r(x)dx = 1$, the sum of the entries $\sum_{j=1}^{J} b_r[j] \approx 1$ when large enough number of samples J are taken.

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

or a generalized Kullback-Leibler (KL) divergence projection

$$v \leftarrow \arg\min_{u \in \Delta_J} \sum_j u[j] \log\left(\frac{u[j]}{v[j]}\right) - u[j] + v[j] \tag{24}$$

among other reasonable maps onto Δ_J.

The Euclidean simplex projection can be done with the following implementation of Chen and Ye's algorithm [47]. The essence of the algorithm is to efficiently compute the special $t \in \mathbb{R}$ so that

$$v \leftarrow \max(0, v - t\mathbb{1}) \in \Delta_J. \tag{25}$$

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

```julia
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,

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

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

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

The generalized Kullback-Leibler divergence projection as stated in Equation 24 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. [48 (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.[9] All together, this looks like

$$v \leftarrow \frac{\max(0, v)}{\sum_j \max(0, v[j])}. \tag{26}$$

We will refer to Equation 26 as nonnegative projection and rescaling (NNPR). NNPR has the following implementation in BlockTensorFactorization.jl.

```julia
l1scale! ∘ nonnegative!
```

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

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

5.2 The Rescaling Trick for Matrix Factorization

- Explain that we can move the weight from one matrix to another

Comparing the two methods of constraining a vector to the simplex 1) by Euclidean projection (Equation 25) or 2) nonnegative projection and rescaling (NNPR, Equation 26), the latter offers

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

a few advantages. NNPR is cheaper and conceptually easier to compute. Another advantage of NNPR to tensor factorizations, is its ability to constrain a factor without loosing progress while performing gradient descent.

For example, consider the low rank factorization problem of finding matrices A, B such that $Y = AB$ where you would like the sum of entries in B to be one

$$\min_{A\in\mathbb{R}^{I\times R}, B\in\mathbb{R}^{R\times J}} \frac{1}{2}\|AB - Y\|_F^2 \quad \text{s.t.} \quad B \in \Delta_{RJ} = \left\{ B \in \mathbb{R}_+^{R\times J} \;\middle|\; \sum_{r,j} B[r,j] = 1 \right\}. \quad (27)$$

The basic alternating projected gradient descent algorithm using a Euclidean projection would be

$$A \leftarrow A - \frac{1}{L_A}\nabla_A f(A,B)$$

$$B \leftarrow EP_{\Delta_{RJ}}\left(B - \frac{1}{L_B}\nabla_B f(A,B)\right)$$

where $f(A,B) = \frac{1}{2}\|AB - Y\|_F^2$ and $EP_{\Delta_{RJ}}$ is the Euclidean projection onto the simplex Δ_{RJ}. In the event the updated value for B,

$$B - \frac{1}{L_B}\nabla_B f(A,B) := \hat{B} \in \mathbb{R}_+^{R\times J}$$

is already nonnegative, the objective f at the new point $\left(A, EP_{\Delta_{RJ}}\left(\hat{B}\right)\right)$ could be bigger or smaller than the objective before the Euclidean projection $f\left(A, \hat{B}\right)$.

If we use the nonnegative projection and rescaling $\text{NNPR}_{\Delta_{RJ}}$ instead of the Euclidean projection $EP_{\Delta_{RJ}}$ when $\hat{B}$ is already nonnegative, then

$$\text{NNPR}_{\Delta_{RJ}}\left(\hat{B}\right) = \frac{1}{\sum_{rj} B[r,j]}\hat{B} := c^{-1}\hat{B}.$$

This means the objective value f at the point $\left(c^{-1}A, c\hat{B}\right)$ will be the same as the objective value before the KL divergence projection

$$f\left(cA, \text{NNPR}_{\Delta_{RJ}}\left(\hat{B}\right)\right) = f\left(cA, c^{-1}\hat{B}\right) = \frac{1}{2}\left\|Acc^{-1}B - Y\right\|_F^2 = \frac{1}{2}\|AB - Y\|_F^2 = f\left(A, \hat{B}\right).$$

This suggests the following update may be a useful alternative to the standard projected gradient descent

$$\begin{aligned}
A &\leftarrow A - \frac{1}{L_A}\nabla_A f(A,B) \\
B &\leftarrow B - \frac{1}{L_B}\nabla_B f(A,B) \\
c &\leftarrow \sum_{r,j} B[r,j] \\
A &\leftarrow cA \\
B &\leftarrow c^{-1}B.
\end{aligned}$$

Of course, it is possible that $\hat{B} = B - \frac{1}{L_B}\nabla_B f(A,B)$ has negative entries. So we use both the nonnegative projection and rescaling part of NNPR in the alternating gradient descent with rescaling update

$$\begin{aligned}
A &\leftarrow A - \frac{1}{L_A}\nabla_A f(A,B) \\
B &\leftarrow \max\left(0, B - \frac{1}{L_B}\nabla_B f(A,B)\right) \\
c &\leftarrow \sum_{r,j} B[r,j] \\
A &\leftarrow cA \\
B &\leftarrow c^{-1}B.
\end{aligned} \tag{28}$$

This algorithm can be called in BlockTensorFactorization.jl with the following code.

```
options = (
    model=Tucker1,
    constraints=[l1scale! ∘ nonnegative!, noconstraint],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

5.3 Generalizing the Matrix Rescaling Trick

The rescaling trick discussed in Section 5.2 applies more generally to other tensor factorization, other simplex-type constraints, and other `ScaledNormalizations`.

5.3.a Simplex-type constraints

Instead of the matrix factorization problem where $B \in \Delta_{RJ}$ is constrained to the full simplex (Equation 27), we can apply the rescaling trick the problem where the rows of B are constrained to the simplex

$$\min_{\substack{A\in\mathbb{R}^{I\times R}\\ B\in\mathbb{R}^{R\times J}}} \frac{1}{2}\|AB - Y\|_F^2 \quad \text{s.t.} \quad B \in \Delta_J^R = \left\{ B \in \mathbb{R}_+^{R\times J} \;\middle|\; \forall r \in [R], \sum_{j=1}^{J} B[r,j] = 1 \right\}. \tag{29}$$

This could make sense in applications where rows of B represent probability densities such as the demixing problem discussed at the start of this section (Section 5).

We can adjust the alternating gradient descent update with NNPR (Equation 28) to the following update

$$\begin{aligned} A &\leftarrow A - \frac{1}{L_A}\nabla_A f(A, B) \\ B &\leftarrow \max\left(0, B - \frac{1}{L_B}\nabla_B f(A, B)\right) \\ C[r, r] &\leftarrow \sum_j B[r, j] \quad (C \in \mathbb{R}^{R\times R} \text{ is diagonal}) \\ A &\leftarrow AC \\ B &\leftarrow C^{-1}B. \end{aligned} \tag{30}$$

It is clear that the objective value would be maintained for any invertible matrix C

$$f(AC, C^{-1}B) = \frac{1}{2}\left\|ACC^{-1}B - Y\right\|_F^2 = \frac{1}{2}\|AB - Y\|_F^2 = f(A, B).$$

This algorithm can be called in BlockTensorFactorization.jl with the following code.

```
options = (
    model=Tucker1,
    constraints=[l1scale_rows! ∘ nonnegative!, noconstraint],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

⚠ Warning

This trick would also work if we wanted the columns of A normalized to the simplex. But it does *not* work when we would like each column of B to be constrained to the simplex. The normalizing matrix C would have to be multiplied to the right of B rather than between A amd B. A similar story can be said with the rows of A.

5.3.b Other `ScaledNormalization`’s

This trick can also apply to other normalization constraints. For example, we may want the maximum magnitude of each row of B to one. This could make sense in applications where each row represents a waveform audio file (WAV) which has an audio format that takes values between -1 and 1. Instead of the Euclidean projection[10]

$$v \leftarrow \max(-1, \min(1, v))$$

[10] In audio processing, this is commonly called “clipping”.

onto the infinity ball

$$\mathcal{B}_J(\infty) = \left\{ v \in \mathbb{R}^J \;\middle|\; \max_{j\in[J]} |v[j]| \leq 1 \right\},$$

we can apply the rescaling[11]

$$v \leftarrow \frac{v}{\max_{j\in[J]} |v[j]|}.$$

Applying this principle to the factorization problem

$$\min_{\substack{A\in\mathbb{R}^{I\times R} \\ B\in\mathbb{R}^{R\times J}}} \frac{1}{2}\|AB - Y\|_F^2 \quad \text{s.t.} \quad B \in \mathcal{B}_J^R(\infty) = \left\{ B \in \mathbb{R}^{R\times J} \;\middle|\; \forall r \in [R], \max_{j\in[J]} |B[r,j]| \leq 1 \right\} \tag{31}$$

gives us the update

$$\begin{aligned} A &\leftarrow A - \frac{1}{L_A}\nabla_A f(A,B) \\ B &\leftarrow B - \frac{1}{L_B}\nabla_B f(A,B) \\ C[r,r] &\leftarrow \max_{j\in[J]} |B[r,j]| \quad (C \in \mathbb{R}^{R\times R} \ \text{ is diagonal}) \\ A &\leftarrow AC \\ B &\leftarrow C^{-1}B. \end{aligned} \tag{32}$$

This algorithm can be called in BlockTensorFactorization.jl with the following code.

```
options = (
    model=Tucker1,
    constraints=[linftyscale_rows! ∘ nonnegative!, noconstraint],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

A similar story can be made about other p-norm balls $\mathcal{B}(p)$ or spheres.

5.3.c Other Tensor Factorizations

The rescaling trick is applicable to other tensor factorizations, but is dependent on the exact model and constraints.

The simplest extension of matrix factorization is the Tucker-1 factorization of an order N tensor $Y = B \times_1 A$.

[11] In audio processing, this is commonly called “normalizing”. Normalizing is often preferred to clipping since it maintains the perceived audio, but just at a different volume than the original signal. Clipping often introduces undesirable distortion (think of sound from a megaphone).

If we would like the first order slices of B to be constrained to the simplex

$$\begin{aligned} B[r,:,:] \in \Delta_{JK} &= \left\{ B[r,:,:] \in \mathbb{R}_+^{J\times K} \;\middle|\; \sum_{j,k} B[r,j,k] = 1 \right\} \\ B \in \Delta_{JK}^R &= \left\{ B \in \mathbb{R}_+^{R\times J\times K} \;\middle|\; \forall r \in [R], \sum_{j,k} B[r,j,k] = 1 \right\}, \end{aligned}$$

we can solve the problem

$$\min_{\substack{A\in\mathbb{R}^{I\times R} \\ B\in\mathbb{R}^{R\times J\times K}}} \frac{1}{2}\|B \times_1 A - Y\|_F^2 \quad \text{s.t.} \quad B \in \Delta_{JK}^R. \tag{33}$$

by iterating the update

$$\begin{aligned} A &\leftarrow A - \frac{1}{L_A}\nabla_A f(A,B) \\ B &\leftarrow B - \frac{1}{L_B}\nabla_B f(A,B) \\ C[r,r] &\leftarrow \sum_{j\in[J],k\in[K]} B[r,j,k] \quad (C \in \mathbb{R}^{R\times R} \text{ is diagonal}) \\ A &\leftarrow AC \\ B &\leftarrow B \times_1 C^{-1}. \end{aligned} \tag{34}$$

This algorithm can be called in BlockTensorFactorization.jl with the following code.

```
options = (
    model=Tucker1,
    constraints=[l1scale_1slices! ∘ nonnegative!, noconstraint],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

A setting where this model applies would be if the first order slices of B represent 2-dimensional probability densities.[12]

For constraints that constrain an entire factor to some scale, the weight of that factor can be distributed to one or multiple other factors. For example, we may wish to find a CP-decomposition of an order 4-tensor $Y = [\![A, B, C, D]\!]$ where the Frobenius norm of A is 1. In the rescaling step, we could move the norm of A to just the matrix B

[12] If sampling a 2-dimensional probability density $p_r(x,y)$ on a rectangular grid, entries of B could be interpreted as $B[r,j,k] = p_r(x_j, y_k)(x_j - x_{j-1})(y_k - y_{k-1})$.

$$
\begin{aligned}
c &\leftarrow \|A\|_F \\
A &\leftarrow c^{-1}A \\
B &\leftarrow cB
\end{aligned}
$$

or all three other factors equally

$$
\begin{aligned}
c &\leftarrow \|A\|_F \\
A &\leftarrow c^{-1}A \\
B &\leftarrow c^{1/3}B \\
C &\leftarrow c^{1/3}C \\
D &\leftarrow c^{1/3}D.
\end{aligned}
$$

In either case, the recombined tensor remains unchanged

$$\llbracket c^{-1}A, cB, C, D\rrbracket = \llbracket c^{-1}A, c^{1/3}B, c^{1/3}C, c^{1/3}D\rrbracket = \llbracket A, B, C, D\rrbracket.$$

The two algorithms can be called in BlockTensorFactorization.jl with the following code.

```
options = (
    model=CPDecomposition,
    constraints=ConstraintUpdate(1, l2scaled!;
        whats_rescaled=(Y -> factor(Y, 2))), # B is the second factor
)

decomposition, stats, kwargs = factorize(Y; options...);
```

```
options = (
    model=CPDecomposition,
    constraints=ConstraintUpdate(1, l2scaled!),
    # assumes all other factors are rescaled
)

decomposition, stats, kwargs = factorize(Y; options...);
```

5.4 Constraining Multiple Factors

When constraining multiple factors with the rescaling approach, there must be at least one factor that is not constrained with rescaling.

Consider the following matrix factorization $Y = AB$ problem where we want both the columns of A and rows of B to be constrained to the simplex.[13]

[13] We define the constraint on A in terms of $A^\top$ to be consistent with the simplex constraint defined on B.

$$
\begin{aligned}
&\min_{\substack{A\in\mathbb{R}^{I\times R}\\ B\in\mathbb{R}^{R\times J}}} \frac{1}{2}\|AB-Y\|_F^2 \\
&\text{s.t.} \\
A^\top \in \Delta_I^R &= \left\{ A^\top \in \mathbb{R}_+^{R\times I} \,\middle|\, \forall r\in[R], \sum_i A^\top[r,i] = 1 \right\} \\
B \in \Delta_J^R &= \left\{ B \in \mathbb{R}_+^{R\times J} \,\middle|\, \forall r\in[R], \sum_j B[r,j] = 1 \right\}
\end{aligned}
\tag{35}
$$

If we try to rescale both A and B while moving the weights to the other factor, we often observe numerical instability or blow up. To be clear, the following update does not seem to work in practice.

$$
\begin{aligned}
A &\leftarrow \max\left(0, A - \frac{1}{L_A}\nabla_A f(A,B)\right) \\
C_A[r,r] &\leftarrow \sum_{i\in[I]} A[i,r] \\
A &\leftarrow AC_A^{-1} \\
B &\leftarrow C_A B \\
B &\leftarrow \max\left(0, B - \frac{1}{L_B}\nabla_B f(A,B)\right) \\
C_B[r,r] &\leftarrow \sum_{j\in[J]} B[r,j] \\
A &\leftarrow AC_B \\
B &\leftarrow C_B^{-1} B
\end{aligned}
$$

The relevant call in BlockTensorFactorization would be the following.

```
options = (
    model=Tucker1,
    constraints=[ # B is the 0th factor, A is the 1st factor
        ConstraintUpdate(1, l1scale_cols! ∘ nonnegative!;
            whats_rescaled=(x -> eachrow(factor(x, 0)))),
        ConstraintUpdate(0, l1scale_rows! ∘ nonnegative!;
            whats_rescaled=(x -> eachcol(factor(x, 1)))),
    ],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

Instead, one of the factors can be scaled without moving the weight to the other factor. For example, if we wanted to remove the 4th line $B \leftarrow C_A B$, we can call the following.

```
options = (
    model=Tucker1,
    constraints=[ # B is the 0th factor, A is the 1st factor
        ConstraintUpdate(1, l1scale_cols! ∘ nonnegative!;
            whats_rescaled=nothing),
        ConstraintUpdate(0, l1scale_rows! ∘ nonnegative!;
            whats_rescaled=(x -> eachcol(factor(x, 1)))),
    ],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

This approach seems to work better in practice. The same principle of relaxing how constraints are enforced can be applied to the very similar problem

$$\begin{aligned}
&\min_{\substack{A\in\mathbb{R}^{I\times R}\\ B\in\mathbb{R}^{R\times J}}} \frac{1}{2}\|AB-Y\|_F^2\\
&\text{s.t.}\\
A\in\Delta_R^I &= \left\{A\in\mathbb{R}_+^{I\times R}\;\middle|\;\forall i\in[I], \sum_r A[i,r]=1\right\}\\
B\in\Delta_J^R &= \left\{B\in\mathbb{R}_+^{R\times J}\;\middle|\;\forall r\in[R], \sum_j B[r,j]=1\right\},
\end{aligned}\tag{36}$$

where we want both the rows of A and B to be constrained to the simplex. Here, we cannot move the weights from A to B since there are I rows of A but only R rows of B. Instead, we relax the problem to

$$\begin{aligned}
&\min_{\substack{A\in\mathbb{R}^{I\times R}\\ B\in\mathbb{R}^{R\times J}}} \frac{1}{2}\|AB-Y\|_F^2\\
&\text{s.t.}\\
&\quad A\in\mathbb{R}_+^{I\times R}\\
B\in\Delta_J^R &= \left\{B\in\mathbb{R}_+^{R\times J}\;\middle|\;\forall r\in[R], \sum_j B[r,j]=1\right\}.
\end{aligned}\tag{37}$$

The relevant update for this relaxed problem would be

$$\begin{aligned}
A &\leftarrow \max\left(0, A - \frac{1}{L_A}\nabla_A f(A, B)\right) \\
B &\leftarrow \max\left(0, B - \frac{1}{L_B}\nabla_B f(A, B)\right) \\
C_B[r, r] &\leftarrow \sum_{j\in[J]} B[r, j] \\
A &\leftarrow AC_B \\
B &\leftarrow C_B^{-1}B
\end{aligned}$$

and is called in BlockTensorFactorization.jl with the following code.

```julia
options = (
    model=Tucker1,
    constraints=[ # B is the 0th factor, A is the 1st factor
        ConstraintUpdate(1, nonnegative!),
        ConstraintUpdate(0, l1scale_rows! ∘ nonnegative!;
            whats_rescaled=(x -> eachcol(factor(x, 1)))),
    ],
)

decomposition, stats, kwargs = factorize(Y; options...);
```

We justify this relaxation with the following argument. When the rows of Y are in the simplex, we can bound how close the rows of A are to summing to one with Theorem 5.1.

Theorem 5.1 (Closeness of A's rows summing to one): Let $Y \in \Delta_J^I$, $A \in \mathbb{R}_+^{I \times R}$, and $B \in \Delta_J^R$ where

$$\|Y - AB\|_F \leq \epsilon.$$

Then for any row $i \in [I]$, the sum of the row $A[i, :]$ is $\epsilon\sqrt{J}$ close to 1

$$\left|1 - \sum_{r \in [R]} A[i, r]\right| \leq \epsilon\sqrt{J}.$$

Proof. We have the following inequalities.

$$\begin{aligned} \|Y - AB\|_F &\leq \epsilon \\ \frac{1}{\sqrt{J}}\|Y - AB\|_\infty &\leq \epsilon \quad \left(\frac{1}{\sqrt{J}}\|X\|_\infty \leq \|X\|_2 \leq \|X\|_F \text{ for } X \in \mathbb{R}^{I \times J}\right) \\ \frac{1}{\sqrt{J}} \max_{i \in [I]} \left(\sum_{j \in [J]} |(Y - AB)[i, j]|\right) &\leq \epsilon \\ \max_{i \in [I]} \left(\sum_{j \in [J]} |(Y - AB)[i, j]|\right) &\leq \epsilon\sqrt{J} \end{aligned}$$

So for all rows $i \in [I]$,

$$\begin{aligned} \epsilon\sqrt{J} &\geq \sum_{j \in [J]} |(Y - AB)[i, j]| \\ &\geq \left|\sum_{j \in [J]} (Y - AB)[i, j]\right| \\ &= \left|\sum_{j \in [J]} Y[i, j] - \sum_{j \in [J]} (AB)[i, j]\right| \\ &= \left|1 - \sum_{j \in [J]} \sum_{r \in [R]} A[i, r]B[r, j]\right| \quad \left(\text{since } Y \in \Delta_J^I\right) \\ &= \left|1 - \sum_{r \in [R]} A[i, r] \left(\sum_{j \in [J]} B[r, j]\right)\right| \\ &= \left|1 - \sum_{r \in [R]} A[i, r]\right| \quad \left(\text{since } B \in \Delta_J^R\right). \end{aligned}$$

Theorem 5.1 implies that solutions to the relaxed problem (Equation 37) are approximate solutions to the problem shown in Equation 36. Moreover, if there exists an exact factorization $Y = AB$ for the relaxed problem, then it is also a solution the original problem.

5.5 Experiment

To illustrate the advantage of this rescaling trick over Euclidean projection, we will consider solving the problem shown in Equation 36 where $Y \in \Delta_J^I$ and there exists an exact factorization $Y = AB$ with $A \in \Delta_R^I$ and $B \in \Delta_J^R$ using eight different algorithms.

6 Multi-scale

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

In many applications [37] [TODO add spatial transcriptomics], the data tensor Y represents a discretization of continuous data. In these cases, we can decide how finely or coarsely to sample the data and effectively have control over the dimension of tensor needing to be factored.

For high quality results, we would like as fine of a discretization as possible. Or if the data is already collected in a discretized form, we would like to incorporate all the data. But smaller tensors are faster to decompose because there are fewer parameters to fit and learn. Most matrix operations like addition, multiplication, and finding their norm are also faster for smaller tensors.

We propose a multiresolution approach inspired wavelets [49] and multigrid [50] that factorizes the data at progressively finer scales. This can greatly speed up how fast large scale tensors can be factorized.

6.1 Basic Approach

We will start with same example from the beginning of Section 5. Give a tensor $Y \in \mathbb{R}_+^{I \times J}$ representing mixtures of discretized densities, we would like to demix the densities according to the model

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

or

$$Y = AB.$$

Notice that the rows of Y and B represent samples of continuous densities, so the size of their second dimension J is arbitrary. Suppose each of the 1-dimensional densities are uniformly discretized on an interval $[a, b]$ with J_s number of points. We use s to represent the scale or spacing of the number of points. For example, $J_1 = J$ would be the finest scale using every point in a discretization $x_1, x_2, x_3..., x_{J_1}$ with $x_1 = a$ and $x_{J_1} = b$, $J_2 = J_1/2$ would be coarser and use every other point $x_1, x_3, x_5, ...x_{J_1-1}$.[14]

The basic approach is to factorize $Y_2 \in \mathbb{R}^{I \times J_2}$ with entries $Y_2[i, j] = y_i(x_{2j-1})$ to obtain $A_2^{T_2} \in \mathbb{R}^{I \times R}$ and $B_{T_2} \in \mathbb{R}^{I \times J_2}$ after T_2 many iterations. We use the factors $A_2^{T_2}$ and $B_2^{T_2}$ to initialize the factorization of $Y_1 = Y \in \mathbb{R}^{I \times J_1}$. We can initialize $A_1^0 = A_2^{T_2}$ since the size of A is the same at both scales, and repeat every entry of B to initialize B_1^0 with entries $B_1^0[i, j] = B_2^{T_2}[i, \lceil j/2 \rceil]$.

Factorizing Y_2 is faster than factorizing Y_1 since 1) there are fewer parameters to learn[15] and 2) most arithmetic like addition and multiplication, as well as calling other operators like `norm` are faster to compute. This gives us a better initialization for A and B in the factorization of Y_1 than some other random initialization so that fewer iterations are needed at the more expensive finer scale.

6.2 General Approach

The basic approach can be generalized in two ways: the data could be continuous in multiple dimensions, and we can recursively apply this multi-scale approach to progressively refine a very coarse factorization.

Suppose we are given a tensor $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ where the dimensions $I_{n_1}, ..., I_{n_M}$ represent a grided discretization of M-dimensional continuous data.

An example of this setting would be an extension of the example shown in Section 6.1 to higher dimensional distributions. We could consider an order-3 tensor where the horizontal slices correspond to a 2-dimensional discretization of a bivariate density. Entries of the input tensor Y would be given by $Y[i, j, k] = f_i(x_j, y_k)$ for continuous probability density functions $f_i : \mathbb{R}^2 \to \mathbb{R}_+$ for a 2D grid of points (x_j, y_k). In this example, the second and third dimensions would be continuous.

If we want to perform a rank-$(R_1, ..., R_N)$ Tucker decomposition of $Y = [\![A_0; A_1, ..., A_N]\!]$, we can initialize the factors with a very corse factorization $A_{n_m}^s \in \mathbb{R}^{J_{n_m}^s \times R_{n_m}}$ where J^s would be a discretization that uses every 2^s points, or more accurately

$$J_{n_m}^s = 2^{\max(S_m - s, 0)} + 1$$

points in total. We select S_m so that at the finest scale $s = 0$, we have $J_{n_m}^0 = I_{n_m}$. The dimensions that do not represent continuous values will use the full sized factors $A_n^s \in \mathbb{R}^{J_n \times R_n}$ with $J_n = I_n$, and the core will remain $A_0^s \in \mathbb{R}^{R_1 \times \cdots \times R_N}$ at all scales s.[16]

The aim is to fit the product of the factors $[\![A_0^s; A_1^s, ..., A_N^s]\!]$ to a lower resolution version of $Y \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, namely $Y^s \in \mathbb{R}^{J_1^s \times \cdots \times J_N^s}$, and use the result to initialize a finer version of the factors A. The code for this looks like the following.

[14] We assume J is even here, but we could define $J_2 = (J_1 - 1)/2$ if $J_1 = J$ is odd.

[15] Only $(I + J_2)R$ at the coarse scale which is less than $(I + J_1)R$ for the fine scale.

[16] This assumes the dimensions of Y where Y is continuous have been discretized as one more than a power of two. The same idea holds with some other discretization plan, but becomes more complicated to express notationally and keep track of the number of points at each scale.

```
function multiscale_factorize(Y; kwargs...)
    continuous_dims, kwargs = initialize_continuous_dims(Y; kwargs...)
    scales, kwargs = initialize_scales(Y; kwargs...)
    coarsest_scale, finer_scales... = scales

    # Factorize Y at the coarsest scale
    Yₛ = coarsen(Y, coarsest_scale; dims=continuous_dims, kwargs...)

    constraints, kwargs = scale_constraints(Yₛ, coarsest_scale; kwargs...)
    decomposition, stats, _ = factorize(Yₛ; kwargs...)

    # Factorize Y at progressively finer scales
    for scale in finer_scales
        # Use an interpolated version of the coarse factorization
        # as the initialization.
        decomposition = interpolate(decomposition, 2; dims=continuous_dims,
kwargs...)
        kwargs[:decomposition] = decomposition

        Yₛ = coarsen(Y, scale; dims=continuous_dims, kwargs...)

        constraints, kwargs = scale_constraints(Yₛ, scale; kwargs...)

        decomposition, stats, _ = factorize(Yₛ; kwargs...)
    end
    return decomposition, stats, kwargs
end
```

6.2.a Coarsening and Interpolating

Straightforward subsampled coarsening and constant interpolating can be used for `coarsen` and `interpolate`, but more sophisticated methods can be used in principle. Since the final solve of `factorize` is on the original sized problem, the choice of coarsening and interpolating only influences the initialization used at this finest scale. Bellow are examples of the basic coarsening and interpolation methods.

```
coarsen(Y::AbstractArray, scale::Integer; dims=1:ndims(Y), kwargs...) =
    Y[(d in dims ? axis[begin:scale:end] : axis for (d, axis) in
enumerate(axes(Y)))...]

function interpolate(Y, scale; dims=1:ndims(Y), kwargs...)
    Y = repeat(Y; inner=(d in dims ? scale : 1 for d in 1:ndims(Y)))

    # Chop the last slice of repeated dimensions
    # since we only interpolate between the values
    return Y[(d in dims ? axis[begin:end-scale+1] : axis for (d, axis) in
enumerate(axes(Y)))...]
end
```

To apply a linear interpolation, we can first perform the constant interpolation and smooth out the result. The following code averages neighbouring values along the continuous dimensions specified. Since this is applied after the constant interpolation, every even indexed value becomes the average of its neighbours, and every odd indexed value remains fixed.

```
function linear_smooth!(Y, dims)
    all_dims = 1:ndims(Y)
    for d in dims
        axis = axes(Y, d)
        Y1 = @view Y[(i==d ? axis[begin+1:end-1] : (:) for i in all_dims)...]
        Y2 = @view Y[(i==d ? axis[begin+2:end] : (:) for i in all_dims)...]

        @. Y1 = 0.5 * (Y1 + Y2)
    end
    return Y
end
```

When interpolating an array that is an `AbstractDecomposition`, we can interpolate the factors directly instead of the combined array.

```
function interpolate(CPD::CPDecomposition, scale; dims=1:ndims(CPD),
kwargs...)
    interpolated_matrix_factors = (d in dims ? interpolate(A, scale; dims=1,
kwargs...) : A for (d, A) in enumerate(matrix_factors(CPD)))
    return CPDecomposition(Tuple(interpolated_matrix_factors))
end

function interpolate(T::Tucker1, scale; dims=1:ndims(T), kwargs...)
    core_dims = setdiff(dims, 1) # Want all dimensions except possibly the
first
    interpolated_core = interpolate(core(T), scale; dims=core_dims, kwargs...)

    matrix = matrix_factor(T, 1)

    interpolated_matrix = 1 in dims ? interpolate(matrix, scale; dims=1,
kwargs) : matrix
    return Tucker1((interpolated_core, interpolated_matrix))
end

function interpolate(T::Tucker, scale; dims=1:ndims(T), kwargs...)
    interpolated_matrix_factors = (d in dims ? interpolate(A, scale; dims=1,
kwargs...) : A for (d, A) in enumerate(matrix_factors(T)))
    # Core is not interpolated
    return Tucker(Tuple(core(T), interpolated_matrix_factors...))
end
```

6.3 Constraints with Multi-scale

Some constraints like `Entrywise` constrains can be used as-is at any scale, but other constraints like normalizations and linear constraints require some more thought. The main strategy is to modify constraints along continuous dimensions. Given a constraint on a factor, and the number of continuous dimensions it constrains, we can construct a modified constraint by scaling the full sized constraint appropriately.

For a p-norm constraint where some part of a factor X_i needs to be normalized to $\|X_i\|_p = C$, we require the coarsened parts of the factor $\overline{X}_i$ to have norm $\|X_i\|_p = (C/s)^{1/p}$ where s is the scale of the coarsening. For example, a scale of 3 would mean we only take every 3 entries of X_i.

For linear constraints `AX=B`, we construct a new constraint that coarsens A, and scales the bias B by the scale. For example, if x is a vector that represents a continuous function that is constrained to the affine space $Ax = b$, we treat the rows of the matrix A as also being continuous functions that are inner product-ed with x and can be coarsened in a similar manner to x. This means a `LinearConstraint(A, b)` gets scaled to the constraint `LinearConstraint(A[:, begin:scale:end], b ./ scale)`.

In all the constraints, the scale will need to be applied in each continuous dimension, so we implement this as `scale^n_continuous_dims`. The full implementation of `scale_constraint` is shown below.

```
function scale_constraint(constraint::AbstractConstraint, scale,
n_continuous_dims)
    @warn "Unsure how to scale constraints of type $(typeof(constraint)).
Leaving the constraint $constraint alone."
    return constraint
end

# Constraints that are fixed like entrywise constraints do not need to be
scaled
function scale_constraint(constraint::Union{FIXED_CONSTRAINTS...}, scale,
n_continuous_dims)
    return constraint
end

# TODO handle LinearConstraint{<:AbstractArray} or LinearConstraint{Function}
function scale_constraint(constraint::LinearConstraint{<:AbstractMatrix},
scale, n_continuous_dims)
    A = constraint.linear_operator
    b = constraint.bias
    return LinearConstraint(A[:, begin:scale:end], b ./
scale^n_continuous_dims)
end

function
scale_constraint(constraint::ScaledNormalization{<:Union{Real,AbstractArray{<:Real}}},
scale, n_continuous_dims)
    norm = constraint.norm
    F = constraint.whats_normalized
    S = constraint.scale
    return ScaledNormalization(norm, F, S ./ scale^n_continuous_dims)
end

function scale_constraint(constraint::ScaledNormalization{<:Function}, scale,
n_continuous_dims)
    norm = constraint.norm
    F = constraint.whats_normalized
    S = constraint.scale
    return ScaledNormalization(norm, F, (x -> x ./ scale^n_continuous_dims) ∘
S)
end

# TODO scale a projected normalization
function scale_constraint(constraint::ProjectedNormalization, scale,
n_continuous_dims)
    @warn "Scaling ProjectedNormalization constraints is not implemented
(YET!) Leaving the constraint $constraint alone."
    return constraint
end
```

We justify scaling constraints in this way with the following propositions and observations.

Theorem 6.1 (Constraint Proposition Assumptions): In the following proposition, we assume $x \in \mathbb{R}^I$ represents an evenly-spaced discretization of an L_f-Lipschitz function $f : [l, u] \to \mathbb{R}$ on a finite interval $t \in [l, u]$ with entries

$$x[i] = f(t[i]),$$

where

$$t[i] = l + (i - 1)\Delta t = l + (i - 1)\frac{u - l}{I - 1}.$$

We will use $\overline{x} \in \mathbb{R}^{\lfloor (I+1)/2 \rfloor}$ to represent the subvector made by removing every other entry of $x \in \mathbb{R}^I$ where

$$\overline{x} = (x[1], x[3], x[5], ..., x[\lfloor I \rfloor_o]).$$

We use the shorthand

$$\lfloor I \rfloor_o = 2\left\lfloor \frac{I + 1}{2} \right\rfloor - 1$$

to round I down to the nearest odd integer.

Proposition 6.1 (Linear Constraint Scaling): Let the assumption in Theorem 6.1 hold. Let $a \in \mathbb{R}^I$ be a discretization of an L_g-Lipschitz function $g : [l, u] \to \mathbb{R}$.

If $\langle a, x\rangle = b$, then for even I,

$$|\langle \overline{a}, \overline{x}\rangle - b/2| \leq \max(\|f\|_\infty, \|g\|_\infty)\frac{I}{I-1}\frac{\left(L_f + L_g\right)(u-l)}{4}$$

and for odd I,

TODO check formatting

$$\begin{aligned}
\left|2\|\overline{x}\|_1 - \frac{I+1}{I}\|x\|_1\right| &= \left|2\|\overline{x}\|_1 - \|x\|_1 - \frac{\|x\|_1}{I}\right| \\
&= \left|2\sum_{i \text{ odd}}|x_i - \sum_{i=1}^{I}|x_i| - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{i \text{ odd}}|x_i + \sum_{i \text{ odd}}|x_i| - \left(\sum_{i \text{ odd}}|x_i| + \sum_{i \text{ even}}|x_i|\right) - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{i \text{ odd}}|x_i - \sum_{i \text{ even}}|x_i| - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{j=1}^{(I-1)/2}|x_{2j-1} + |x_I| - \sum_{j=1}^{(I-1)/2}|x_{2j}| - \frac{\|x\|_1}{I}|\right| \\
&\leq \sum_{j=1}^{(I-1)/2}||x_{2j-1} - |x_{2j}||| + \left||x_I - \frac{\|x\|_1}{I}|\right| \\
&\leq \sum_{j=1}^{(I-1)/2}\left|x_{2j-1} - x_{2j}\right| + \frac{1}{I}|I|x_I - \|x\|_1|| \\
&\leq \sum_{j=1}^{(I-1)/2} C + \frac{1}{I}\left|\sum_{i=1}^{I}(|x_I - |x_i|)|\right| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}||x_I - |x_i||| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}|x_I - x_i| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}C(I-i) \text{ (Apply Lipschitz recursively)} \\
&= C\frac{I-1}{2} + \frac{1}{I}\frac{CI(I-1)}{2} \\
&= \frac{L(u-l)}{2} + \frac{L(u-l)}{2} \\
&= L(u-l).
\end{aligned}$$

Proof. See Section 8.3.a.

Proposition 6.1 can be extended to a bound on the L_1 distance between $\overline{A}\overline{x}$ and $b/2$ where $Ax = b$ and $\overline{A} = A[:, begin : 2 : end]$ is the submatrix with every other column removed. TODO add corollary.

We also have a similar proposition for a 1-norm constraint.

Proposition 6.2 (\$L_1\$-norm Constraint Scaling): Let the assumption in Theorem 6.1 hold.

If $\|x\|_1 = b$, then for even I,

$$|\|\overline{x}\|_1 - b/2| \leq \frac{I}{I-1}\frac{L(u-l)}{4}$$

and for odd I,

$$\left|\|\overline{x}\|_1 - \frac{I+1}{I}\frac{b}{2}\right| \leq \frac{L(u-l)}{2}.$$

Proof. See Section 8.3.b.

Importantly, for normalized vectors $z = x/\|x\|_1$, the error bound in Proposition 6.2 (for even I) becomes

$$|\|\overline{z}\|_1 - 1/2| \leq \frac{I}{I-1}\frac{L(u-l)}{4b}.$$

As the number of points get large, $I \to \infty$, a finer discretization will have more points and have a 1-norm becoming unbounded $b \to \infty$. This means the error bound goes to zero and $\|\overline{z}\|_1 \to 1/2$.[17]

6.4 Convergence of a Multi-scale Method

- show the multi-scale method has tighter bounds than regular gradient descent for lipschitz data
- this assumes no constraints

As implemented, both the regular and multi-scale approaches execute many iterations at the finest scale. The multi-scale method uses its coarsened iterations to warm start the fine iteration. So we should expect the multi-scale method to converge. We make this explicit with the following lemmas and theorems. The first subsection (Section 6.4.b) shows general functional and convex analysis facts about Lipschitz, smooth, and strongly convex functions, and what happens when you linearly interpolate Lipschitz functions. The next subsection (Section 6.4.c) shows convergence for the multi-scale method when we resolve the problem along the entire new discretization at each scale. And the final subsection (Section 6.4.d) looks at what happens when you freeze previously computed points so you only need to solve the problem at the interpolated points for each scale.

[17] This also holds for odd I.

6.4.a Definitions

Definition 6.1: A differentiable function $f : \mathbb{R}^I \to \mathbb{R}$ is S-smooth when ∇f is S-Lipschitz,

$$\|\nabla f(x) - \nabla f(y)\|_2 \leq S\|x - y\|_2.$$

Definition 6.2: A function $f : \mathbb{R}^I \to \mathbb{R}$ is μ-strongly convex when $g(x) = f(x) - \frac{\mu}{2}\|x\|_2^2$ is convex.

When f is differentiable, f is μ-strongly convex when

$$f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\mu}{2}\|x - y\|_2^2$$

for all $x, y \in \mathbb{R}^I$.

Definition 6.3: One step of projected gradient descent for an S-smooth and μ-strongly convex function $\mathcal{L} : \mathbb{R}^I \to \mathbb{R}$, a constraint set $\mathcal{C} \subseteq V$, at an iterate x^k is the update,

$$x^{k+1} \leftarrow P_{\mathcal{C}}\left(x^k - \frac{1}{S}\nabla \mathcal{L}\left(x^k\right)\right)$$

where $P_{\mathcal{C}} : \mathbb{R}^I \to \mathcal{C}$ is the (Euclidean) projection operator onto $\mathcal{C}$.

Proposition 6.3 (Descent Lemma): In the projected gradient descent setting with an S-smooth and μ-strongly convex function (Definition 6.3), we can bound the distance to the unique minimizer $x^* \in \mathcal{C}$ in terms of the initial point $x^0 \in \mathbb{R}^I$,

$$\left\|x^t - x^*\right\|_2 \leq (1 - c)^t \left\|x^0 - x^*\right\|_2$$

with *condition number* $c = \mu / S$.

TODO cite this.

The vector space $\mathbb{R}^I$ in Definition 6.2, Definition 6.1, and Definition 6.3 can be extended to tensor spaces $\mathbb{R}^{I_1 \times \cdots \times I_N}$ with the Frobenius inner product $\langle X, Y \rangle_F$ and norm $\|\cdot\|_F$ more generally.

6.4.b Functional Analysis Lemmas

Before we can analyze the convergence of a multi-scaled method, we first need a handle on how linear interpolations play with discretized Lipschitz functions. To simplify the analysis of multi-scaled methods, we look at solving the general problem

$$\min_{x\in\mathcal{C}} \mathcal{L}(x) := \sum_{i\in[I]} \ell_i(x[i])$$

using projected gradient descent

$$x \leftarrow P_{\mathcal{C}}\left(x - \frac{1}{L}\nabla\mathcal{L}(x)\right)$$

where the samples $x[i]$ come from some L_f-Lipschitz function $f : \mathbb{R} \to \mathbb{R}$

$$x[i] = f(t_i)$$

on an interval $t \in [a, b]$ that has been evenly discretized $t_{i+1} - t_i = \Delta t$ for all $i \in [I]$.

We assume the loss functions ℓ_i over each entry $x[i]$ are S-smooth and μ-strongly convex. A natural example of such a function $\mathcal{L}$ would be a least-squares loss

$$\mathcal{L}(x) = \frac{1}{2}\|x - y\|_2^2 = \sum_{i\in[I]} \frac{1}{2}(x[i] - y[i])^2 = \sum_{i\in[I]} \ell_i(x[i]).$$

The following lemmas explain that the smoothness and strong convexity of the functions ℓ_i carry over to the full loss $\mathcal{L}$.

Lemma 6.1: If $\ell_i : \mathbb{R} \to \mathbb{R}$ are each S-smooth, then $\mathcal{L} : \mathbb{R}^I \to \mathbb{R}$ is S-smooth.

Proof. First note that $(\nabla\mathcal{L}(x))_i = \frac{\partial}{\partial x[i]}\mathcal{L}(x) = \frac{\partial}{\partial x[i]}\sum_{j=1}^{I} \ell_i(x[j]) = \sum_{j=1}^{I} \frac{\partial}{\partial x[i]}\ell_i(x[j]) = \frac{\partial}{\partial x[i]}\ell_i(x[i]) = \ell_i'(x[i])$ since $\mathcal{L}$ is separable in each coordinate. This gives us,

$$\begin{aligned}
\|\nabla\mathcal{L}(x) - \nabla\mathcal{L}(y)\|_2^2 &= \sum_{i=1}^{I} (\nabla\mathcal{L}(x) - \nabla\mathcal{L}(y))[i]^2 \\
&= \sum_{i=1}^{I} \left(\ell_i'(x[i]) - \ell_i'(x[i])\right)^2 \\
&\leq \sum_{i=1}^{I} S^2(x[i] - y[i])^2 \\
&= S^2\|x - y\|_2^2.
\end{aligned}$$

Taking square roots completes the proof.

Lemma 6.2: If $\ell_i : \mathbb{R} \to \mathbb{R}$ are all m strongly convex, then $\mathcal{L} : \mathbb{R}^I \to \mathbb{R}$ is m strongly convex (under the 2-norm in $\mathbb{R}^I$).

Proof. Consider

$$\begin{aligned} \mathcal{L}(x) - \frac{m}{2}\|x\|_2^2 &= \sum_{i=1}^{I} \ell_i(x[i]) - \frac{m}{2}\sum_{i=1}^{I} x[i]^2 \\ &= \sum_{i=1}^{I}\Big(\ell_i(x[i]) - \frac{m}{2}x[i]^2\Big). \end{aligned}$$

The function $g(x[i]) = \ell_i(x[i]) - \frac{m}{2}x[i]^2$ is convex because each function $\ell_i(x[i])$ is m strongly convex. The sum of convex functions is also convex, so $\mathcal{L}(x) - \frac{m}{2}\|x\|_2^2$ is convex.

These conditions can be relaxed to include larger sets of function like quasi-strongly convex functions [51], but the general proof approach remains the same. We do require the separability of $\mathcal{L}$ so it can make sense to solve the problem over coarser discretizations.

Now we discuss how Lipschitz functions play with discretizations and linear interpolations.

Lemma 6.3 (Lipschitz Function Interpolation): Let $f : \mathbb{R}^n \to \mathbb{R}$, $a, b \in \mathbb{R}^n$, $t \in [0, 1]$. Suppose f is L-Lipshtiz. Then the error between the function at a point on the line segment between a and b, and the linear interpolation is

$$|(tf(a) + (1-t)f(b)) - f(ta + (1-t)b)| \leq 2Lt(1-t)\|a - b\|_2.$$

Using the bound on the linear interpolation of a Lipschitz function repeatedly, for centre point interpolation ($t = 1/2$ in Lemma 6.3), we can bound the error between an exact discretization of a function at a fine scale ($Y[j] = f(X[j])$) and a linear interpolation ($\hat{Y}$) coming from a coarser discretization ($y[k] = f(x[k])$).

Lemma 6.4 (Exact Interpolation): Given K uniformly space points $x[k]$ on $[a, b]$, (nearly) double them to get $J = 2K - 1$ uniformly spaced points $X[j]$ on $[a, b]$. Let $y[k] = f(x[k])$ for some L-Lipshitz function $f : \mathbb{R} \to \mathbb{R}$, and linearly interpolate the function values

$$\hat{Y}[j] = \begin{cases} y\left[\frac{j+1}{2}\right] & \text{if } \ j \ \text{ is odd} \\ \frac{1}{2}\left(y\left[\frac{j}{2}\right] + y\left[\frac{j}{2} + 1\right]\right) & \text{if } \ j \ \text{ is even} \end{cases},$$

where the true values are given by $Y[j] = f(X[j])$. Then the difference between the interpolated $\hat{Y}$ and exact values Y is bounded by

$$\left\|\hat{Y} - Y\right\|_2 \leq \frac{L}{2\sqrt{K-1}}\|a - b\|_2.$$

We also have an inexact version when we interpolate not from an exact coarse discretization ($y[k] = f(x[k])$), but from an approximate coarse discretization ($\tilde{y}[k] = f(x[k]) + \delta_k$).

Lemma 6.5 (Inexact Interpolation): Given a linear interpolation $\hat{\hat{Y}}$ of an inexact discretization $\tilde{y}[k] = f(x[k]) + \delta_k$ of a function f as described in Lemma 6.4, where the interpolation is defined as

$$\hat{\hat{Y}}[j] = \begin{cases} \tilde{y}\left[\frac{j+1}{2}\right] & \text{if } \ j \ \text{ is odd} \\ \frac{1}{2}\left(\tilde{y}\left[\frac{j}{2}\right] + \tilde{y}\left[\frac{j}{2} + 1\right]\right) & \text{if } \ j \ \text{ is even} \end{cases},$$

we have the error bound between the interpolated inexact discretization $\hat{\hat{Y}}$ and the exact discretization of the function Y,

$$\begin{aligned} \left\|\hat{\hat{Y}} - Y\right\|_2 &\leq \left\|\hat{\hat{Y}} - \hat{Y}\right\|_2 + \left\|\hat{Y} - Y\right\|_2 \\ &\leq \sqrt{2}\|\tilde{y} - y\| + \frac{L}{2\sqrt{K-1}}\|a - b\|_2. \end{aligned}$$

Proof. See Section 8.4.a.

This makes sense in the following way: the error in our interpolation of approximate points, is bounded by two errors. The first comes from the fact that we interpolated using approximated values $\tilde{y}$ in place of the true values y, and the second comes from using a linear interpolation $\hat{Y}$ in place of the exact values Y.

6.4.c Re-solved Multi-scale

LEFT OFF HERE

We use the following general approach to show convergence.

Given some initial guess x_S^0 at the coarsest scale, we use the decent lemma (Proposition 6.3) to bound the error between our iterate $x_S^{K_S}$ after K_S iterations, and the solution x_S^* at the scale S. We can linearly interpolate our point $\hat{x}_S^{K_S}$ and use it to initialize another round of projected gradient descent $x_{S-1}^0 = \hat{x}_S^{K_S}$ at the slightly finer scale. We can link the error between our iterate $x_S^{K_S}$ and the solution x_S^* at scale S with the initial error between an iterate at the finer scale x_{S-1}^0 and the solution x_{S-1}^* at this scale using the inexact interpolation lemma (Lemma 6.5). We perform K_s iterations at each scale $s = S, S-1, ..., 2, 1$ to get an error bound between our iterate at the finest scale $x_1^K{}_1$ the solution at this scale x_1^* in terms of the initial error at the coarsest scale x_S^0.

We present the main descent theorem here, and leave the rest of the details in the appendix.

Theorem 6.2 (Re-solved Multi-scale Descent Error): Let c be the condition number of the loss function $\mathcal{L}$, L_f be the Lipschitz constant for the underlying continuous function f, and S be the coarsest scale. For the setting described in Section 6.4.b, we have the following error bound on the fixed multi-scaled method.

$$\begin{aligned}&\left\|x_1^0 - x_1^*\right\|_2 \\ &\leq \sqrt{2^{S-1}}(1-c)^{\sum_{s=1}^{S} K_s}\left\|x_S^0 - x_S^*\right\|_2 + \frac{L_f|a-b|}{2\sqrt{2^{S+1}}}\sum_{s=1}^{S-1} 2^s (1-c)^{\sum_{t=1}^{s} K_t}\end{aligned}$$

Proof. See Section 8.4.b

6.4.d Freezed Multi-scale

We take a nearly identical approach to the re-solved multi-scale method as described in Section 6.4.c, but we instead freeze the entries of our interpolation $\hat{x}_s^{K_s}$ that correspond to the same points as the prior iteration $x_s^{K_s}$. The projected gradient update will only act on the interpolated and unfrozen values which we will call $x_{(s)}^k$.

TODO can I just introduce this notation in the proofs so that the main paper is cleaner?

To be clear let us look at an example where the finest scale has $2^3 + 1 = 9$ points. After performing K_3 iterations at the coarsest scale $S = 3$ with a total of three points, and slightly finer scale $S = 2$ with five points (but only two unfrozen points), the full vector at the finest scale would be

$$\begin{aligned}&x_1^k = \\ &\left(x_{(3)}^{K_3}[1], x_{(1)}^k[1], x_{(2)}^{K_2}[1], x_{(1)}^k[2], x_{(3)}^{K_3}[2], x_{(1)}^k[3], x_{(2)}^{K_2}[2], x_{(1)}^k[4], x_{(3)}^{K_3}[3]\right)\end{aligned}$$

where the vector of free variables is

$$x_{(1)}^k = \left(x_{(1)}^k[1], x_{(1)}^k[2], x_{(1)}^k[3], x_{(1)}^k[4]\right).$$

Summary of notation

- $\hat{x}$ is some approximation of x
- $x_{(s)}$ is a vector of just the free variables at scale s
- x_s is a vector of the free and fixed variables
- x^k is the kth iteration
- K_s is the number of iterations performed at scale s
- $e = \hat{x} - x^*$ is the error
- have $e_{(s)}, e_s, e^k$ similarly.
  - Example, $e_1^2 = x_1^2 - x_1^*$ is the error between our iterates at the finest scale $s = 1$ and the true values x_1^* after 2 iterations

We present the main descent theorem here, and leave the rest of the details in the appendix.

Theorem 6.3 (Freezed Multi-scale Descent Error): Let c be the condition number of the loss function $\mathcal{L}$. For the setting described in Section 6.4.b, we have the following error bound on the freezed multi-scaled method.

$$\left\|e_1^{K_1}\right\| \leq \left\|e_S^0\right\|(1-c)^{K_S} \prod_{s=1}^{S-1}(1+(1-c)^{K_s}) \\ + \frac{L_f}{2}|b-a| \sum_{s=1}^{S-1} \frac{1}{\sqrt{2^{S-s}}}(1-c)^{K_s} \prod_{j=1}^{s-1}\left(1+(1-c)^{K_j}\right).$$

If we use the same number of iterations $K_s = K$ at each scale, this reduces to the closed form upper bound

$$\left\|e_1^{K_1}\right\| \leq \left\|e_S^0\right\| d(K)(1+d(K))^{S-1} \\ + \frac{L_f}{2}|b-a| \frac{d(K)\left(\sqrt{2^S}(d(K)+1)^S - \sqrt{2}(d(K)+1)\right)}{\sqrt{2^S}(d(K)+1)\left(\sqrt{2}d(K)+\sqrt{2}-1\right)}$$

where $d(K) = (1-c)^K$ is a small number between $0 < d(K) < 1$.

Proof. See Section 8.4.c.

We summarize the convergence with Corollary 6.1 that sends the number of iterations to infinity.

Corollary 6.1 (Multiscale Convergence): As the total number of iterations grows $\sum_{s=1}^{S} K_s \to \infty$ (in the case of re-solve multi-scale in Theorem 6.2) or the number of iterations at each scale $K_s \to \infty$ (in the case of freezed multi-scale in Theorem 6.3), the final error goes to 0, $\left\|e_1^{K_1}\right\|_2 \to 0$, and we converge to the solution $x_1^{K_1} \to x_1^*$ at the finest scale.

6.5 Comparison With Projected Gradient Descent

From Proposition 6.3, projected gradient descent with a generic initialization on the fine grid gives us the following iterate convergence

$$\left\|x_1^K - x_1^*\right\| \leq (1-c)^K \left\|x_1^0 - x_1^*\right\|.$$

We can use this in combination with an expected initial error to get an expected number of iterations needed until we have converged to a desired tolerance. This is make precise by Theorem 6.4.

Theorem 6.4 (Expected Projected Gradience Descent Convergence): Assume the problem is either scaled or shifted so that the solution is normalized $\left\|x_1^*\right\|_2 = 1$ or centred $\left\|x_1^*\right\|_2 = 0$ where $x_1^* \in \mathbb{R}^I$. Choose an initialization $x_1^0 \in \mathbb{R}^I$ with i.i.d. standard normal entries $\left(x_1^0\right)_i \sim \mathcal{N}(0,1)$. Then the expected initial error is

$$\mathbb{E}\left\|x_1^0 - x_1^*\right\|_2 = \sqrt{I+1}.$$

And, as the number of points grows $I \to \infty$, if we iterate projected gradient descent K times where

$$K \geq \frac{\log(1/\epsilon) + \log(I-1)/2}{-\log(1-c)},$$

the expected error is less than ϵ,

$$\mathbb{E}\left\|x_1^0 - x_1^*\right\|_2 \leq \epsilon.$$

Proof. See Section 8.4.d.

This is contrasted with the expected initial and final error for re-solved multi-scaled descent (Theorem 6.5) and freezed multi-scaled descent (Theorem 6.6).

Theorem 6.5 (Expected Re-solved Multi-scale Convergence): Assume the same setting as in Theorem 6.4, but instead using re-solved multi-scaled descent method starting with a scale s_0. Let the number of points at the finest scale be $I = 2^S + 1$. Then we have the expected initial error

$$\mathbb{E}\left\|x_{s_0}^0 - x_{s_0}^*\right\|_2 = \sqrt{2^{S-s_0+1} + 2}.$$

Performing K_s iterations at each scale gives us the expected final error

$$\begin{aligned} &\mathbb{E}\left\|x_1^{K_1} - x_1^*\right\|_2 \\ &\leq (1-c)^{K_1}\sqrt{2^{S+1}}\left((1-c)^{\sum_{s=2}^{S} K_s} + \frac{L|a-b|}{2 \cdot 2^S} + \frac{L|a-b|}{2 \cdot 2}\sum_{s=2}^{S-1}\frac{(1-c)^{\sum_{t=2}^{s} K_t}}{2^{S-s}}\right). \end{aligned}$$

Proof. See Section 8.4.e.

It is not as straightforward to get the required number of iterations to achieve a desired level of accuracy. Additionally, this would not be a fair comparison with projected gradient descent since we expect an iteration at a coarse scale S to be cheaper (less time and fewer floating point operations) than an iteration at the finest scale $s = 1$. For this reason, we need to cost an iteration of projected gradient descent at a scale s in terms of the size of the problem at that scale.

Lemma 6.6 (Cost of Projected Gradient Descent vs Multi-scale): Suppose the total cost of regular descent is given by

$$C_{\text{GD}} = C_1 K$$

where C_1 is the cost of performing one iteration at the finest scale $s = 1$.

The total cost of multi-scale descent is

$$C_{\text{MS}} = \sum_{s=1}^{S} C_s K_s$$

similarly.

If we assume the cost of projected gradient scales at least in the size of the problem ($C_1 = \Omega(I)$, i.e. $C_1 \geq CI$ for some $C \geq 0$), then the cost of projected gradient descent at scale s is

$$C_s \geq \frac{2^{S-s+1} + 1}{2^{S-s} + 1} C_{s+1} \geq \frac{3}{2} C_{s+1},$$

which gives the total cost of multi-scale descent at most

$$C_{\text{MS}} \leq C_1 \sum_{s=1}^{S} \left(\frac{2}{3}\right)^{s-1} K_s.$$

Proof. See Section 8.4.f.

We can use this to select a plan for the number of iterations at each scale K_s that will be cheaper than projected gradient descent, yet still give the same upper bound on the expected final error.

Corollary 6.2 (Sufficient Conditions for Re-solved Multi-scale to be Cheaper): Assume the problem is well conditioned with a condition number at least $c \geq 0.3$, and the finest scale problem is discretized with at least $I \geq 33 = 2^{4+1} + 1$ points. Then performing re-solved multi-scale with one iteration $K_s = 1$ at each scale except the finest scale where we iterate $K_1 = K - 3$ times, yields a tighter upper bound on the expected final error with a cheaper cost, than performing projected gradient descent with K iterations.

Proof. See Section 8.4.g.

We can play a similar game for the freezed multi-scale approach.

Theorem 6.6 (Expected Freezed Multi-scale Convergence): Assumed the same setting as Theorem 6.4, but instead using the freezed multi-scale approach starting at the coarsest scale S, where the finest scale has $I = 2^S + 1$ many points. Then the expected initial error is two,

$$\mathbb{E}\left\|x_{s_0}^0 - x_{s_0}^*\right\|_2 = 2.$$

Performing the same number of iterations $K_s = K$ at each scale gives us the expected final error

$$\begin{aligned}&\mathbb{E}\left\|x_1^K - x_1^*\right\|_2 \\ &\leq d(K)\left(2(1 + d(K))^{S-1} + \frac{L_f}{2}|b - a|\frac{\left(\sqrt{2^S}(d(K) + 1)^S - \sqrt{2}(d(K) + 1)\right)}{\sqrt{2^S}(d(K) + 1)\left(\sqrt{2}d(K) + \sqrt{2} - 1\right)}\right)\end{aligned}$$

where $d(K) = (1 - c)^K$, c is the condition number for the problem, and the underlying continuous function f is L_f Lipschitz on the interval $[a, b]$.

Proof. See Section 8.4.h.

To better analyze when the bound for freezed multi-scale descent (Theorem 6.6) is small than for regular projected gradient descent (Theorem 6.4), we will look at the case when we have many iterations. This lets us approximate $d(K) \approx 0$ since we know $d(K) \to 0$ as $K \to \infty$.

Corollary 6.3 (Sufficient Conditions for Freezed Multi-scale to be Cheaper): Assume the finest scale has at least $I \geq 2^S + 1$ many points where S is at least

$$S > \log_2\left(\left(\frac{\frac{L_f}{2}|b - a| + 1/2}{\left(\sqrt{2} - 1\right)} + 2\right)^2 - 2\right).$$

Then performing freezed multi-scale with $K_s = \lceil K/3 \rceil - 1$ iterations at each scale starting with a scale $s = S$, yields a tighter upper bound on the expected final error with a cheaper cost, than performing projected gradient descent with K iterations.

Proof. See Section 8.4.i.

6.6 Benchmarks

6.6.a Synthetic Data

TODO see multiscalesynthetic3d.jl

For a synthetic test, we generate 3 source distributions. Each distribution is a 3 dimensional product distribution of some standard continuous distributions.

```julia
using Distributions

source1a = Normal(4, 1)
source1b = Uniform(-7, 2)
source1c = Uniform(-1, 1)

source2a = Normal(0, 3)
source2b = Uniform(-2, 2)
source2c = Exponential(2)

source3a = Exponential(1)
source3b = Normal(0, 1)
source3c = Normal(0, 3)

source1 = product_distribution([source1a, source1b, source1c])
source2 = product_distribution([source2a, source2b, source2c])
source3 = product_distribution([source3a, source3b, source3c])

sources = (source1, source2, source3)
```

We generate the following 5×3 mixing matrix

```julia
p1 = [0, 0.4, 0.6]
p2 = [0.3, 0.3, 0.4]
p3 = [0.8, 0.2, 0]
p4 = [0.2, 0.7, 0.1]
p5 = [0.6, 0.1, 0.3]

C_true = hcat(p1,p2,p3,p4,p5)'
```

and use it to construct 5 mixture distributions.

```julia
distribution1 = MixtureModel([sources...], p1)
distribution2 = MixtureModel([sources...], p2)
distribution3 = MixtureModel([sources...], p3)
distribution4 = MixtureModel([sources...], p4)
distribution5 = MixtureModel([sources...], p5)
distributions = [distribution1, distribution2, distribution3, distribution4,
distribution5]
```

These are discretized into $65 \times 65 \times 65$ sample tensors, and stacked into a $5 \times 65 \times 65 \times 65$. We normalize the 1-slices so that they sum to one.

```
sinks = [pdf.((d,), xyz) for d in distributions]
Y = cat(sinks...; dims=4)
# reorder so the first dimension indexes mixtures rather than the final
dimension
Y = permutedims(Y, (4,1,2,3))
Y_slices = eachslice(Y, dims=1)
correction = sum.(Y_slices) # normalize slices to 1
Y_slices ./= correction
```

Because these are large tensors, we only test a single shot decomposition (after they have been compiled). This gives us the following.

```
# Run once to compile functions
factorize(Y; options...);
multiscale_factorize(Y; continuous_dims=[2, 3, 4], options...);
```

```
# Time the functions
@time decomposition, stats_data, kwargs = factorize(Y; options...);
```

```
11.828295 seconds (214.97 k allocations: 15.197 GiB, 32.42% gc time, 0.00%
compilation time)
```

```
@time decomposition, stats_data, kwargs = multiscale_factorize(Y;
continuous_dims=[2, 3, 4], options...);
```

```
2.335374 seconds (201.33 k allocations: 2.905 GiB, 25.26% gc time, 0.00%
compilation time)
```

We can see that `multiscale_factorize` is roughly five times as fast and uses about a fifth of the memory in this example.

6.6.b Real Data

We use the same sedimentary data and Tucker-1 model as described in [41] to factorize a tensor containing mixtures of estimated densities. We discretize the densities with $K = 2^{10} + 1 = 1025$ points to obtain an input tensor $Y \in \mathbb{R}_{+}^{20\times7\times1025}$ and normalize the depth fibres so that $\sum_{k\in[K]} Y[i, j, k] = 1$ for all $i \in [20]$ and $j \in [7]$.

We run the multi-scale factorization algorithm

```
multiscale_factorize(Y; continuous_dims=3, options...)
```

with the following options.

```
options = (
    rank=3,
    momentum=false,
    do_subblock_updates=false,
    model=Tucker1,
    tolerance=(0.12),
    converged=(RelativeError), # relative error ≤ 12%
    constrain_init=true,
    constraints=[l1scale_average12slices! ∘ nonnegative!, nnonnegative!],
    stats=[Iteration, ObjectiveValue, GradientNNCone, RelativeError],
    maxiter=200
)
```

We use the same convergence criteria at each scale and iterate until the relative error between the input Y and our model X is at most 12%, or until 200 iterations have passed. We use 12% because this is roughly the error Graham et. al. observe in their final factorization [41]. The third dimension is specified as continuous since each depth fibre $Y[i, j, :]$ is a discretized continuous probability density function.

This is compared to the regular factorization algorithm

```
factorize(Y; options...)
```

using the Julia package `BenchmarkingTools`. This runs the algorithm as many times as it can within a default time window. Note that a new random initialization is generated for each run. After running the following,

```
using BenchmarkingTools

# Run each function once so they compile
factorize(Y; options...);
multiscale_factorize(Y; continuous_dims=3,options...);

benchmark1 = @benchmark factorize(Y; options...)
display(benchmark1)

benchmark2 = @benchmark multiscale_factorize(Y; continuous_dims=3,options...)
display(benchmark2)
```

we observe the two benchmarks. For `factorize`, we have

```
BenchmarkTools.Trial: 22 samples with 1 evaluation per sample.
Range (min … max):  103.648 ms … 541.438 ms  ┊ GC (min … max): 13.87% … 14.20%
Time  (median):     191.769 ms               ┊ GC (median):    15.00%
Time  (mean ± σ):   227.315 ms ± 130.666 ms  ┊ GC (mean ± σ):  15.04% ±  1.79%
```

```
▃▃     █    ▃    ▃
██▆▁▆▁▁█▁▁▆▆█▆▁▁▁█▁▁▆▁▁▆▁▁▁▁▁▁▁▁▁▁▁▆▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▁▆▁▆▁▁▁▁▆ ▁
104 ms           Histogram: frequency by time          541 ms <

Memory estimate: 201.53 MiB, allocs estimate: 78879.
```

and for `multiscale_factorize`, we observe the following.

```
BenchmarkTools.Trial: 59 samples with 1 evaluation per sample.
Range (min … max):   63.974 ms … 147.756 ms  ┊ GC (min … max):  0.00% … 22.20%
Time  (median):      81.719 ms               ┊ GC (median):    13.13%
Time  (mean ± σ):    85.094 ms ±  13.163 ms  ┊ GC (mean ± σ):  12.13% ±  5.42%

            ▁█  ▁ ▁      ▁
▆▁▁▁▁▁▁▁▆▁▃▆▇██▆▇█▇█▃▆▃▃▃▆█▁▁▁▃▁▃▃▁▃▁▁▃▃▁▁▁▁▁▁▃▁▁▁▁▁▁▁▁▁▁▁▁▃ ▁
64 ms            Histogram: frequency by time          125 ms <

Memory estimate: 84.93 MiB, allocs estimate: 634376.
```

By every metric, the multi-scale approach is faster and uses less memory than the regular factorize approach. Looking at the median time and memory estimate, it is roughly twice as fast, and uses about half as much memory in the case of this problem. These wall clock times include a nontrivial amount of garbage collection (GC) so it means there could be design improvements such as using more preallocated arrays to reduce memory usage. But these improvements would speed up both `factorize` and `multiscale_factorize`. Without these improvements, we can see that `multiscale_factorize` is less likely to use garbage collection since more computations are occurring on smaller arrays.

7 Conclusion

The BlockTensorFactorization.jl Julia provides a new all-in-one package for performing constrained tensor factorizations, and a playground for designing new decompositions and custom constraints. Careful design elements were engineered to balance flexibility and efficiency. By creating this package, new advancements like enforcing constraints through scaling rather than projection, and performing optimization over multiple scales were mathematically examined and numerically tested. These novel ideas are worth investigating further to see their applicability to continuous optimization beyond tensor factorization.

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

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

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

$$\nabla F(X)[i_1, \dots, i_M, j_1, \dots, j_N] = \frac{\partial f_{i_1 \dots i_M}}{\partial X[j_1, \dots, j_N]}(X). \tag{38}$$

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 38.

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

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

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

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

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

8.3 Constraint Rescaling Proofs

In the following proofs, we have an L_f-Lipschitz function $f : [l, u] \to \mathbb{R}$ that is discretized according to

$$x[i] = f(t[i]),$$

where

$$t[i] = l + (i-1)\Delta t = l + (i-1)\frac{u-l}{I-1}.$$

See Theorem 6.1.

This ensures neighboring entries of x are close together. Specifically, we have

$$|x_i - x_{i+1}| = |f(t_i) - f(t_{i+1})| \leq L_f |t_i - t_{i+1}| = L_f \frac{u-l}{I-1} := C_x.$$

8.3.a Linear Constraint Scaling

Proof. First, assume an even number of points I.

$$\begin{aligned}
|2\langle\overline{a},\overline{x}\rangle - b| &= \left|2\sum_{i\text{ odd}} a_i x_i - \sum_{i=1}^{I} a_i x_i\right| \\
&= \left|\sum_{i\text{ odd}} a_i x_i - \sum_{i\text{ even}} a_i x_i\right| \\
&= \left|\sum_{i\text{ odd}} a_i x_i - \sum_{i\text{ odd}} a_{i+1} x_{i+1}\right| \\
&\leq \sum_{i\text{ odd}} \left|a_i x_i - a_{i+1} x_{i+1}\right| \\
&= \sum_{i\text{ odd}} \left|a_i x_i - a_{i+1} x_i + a_{i+1} x_i - a_{i+1} x_{i+1}\right| \\
&\leq \sum_{i\text{ odd}} \left(\left|a_i x_i - a_{i+1} x_i\right| + \left|a_{i+1} x_i - a_{i+1} x_{i+1}\right|\right) \\
&= \sum_{i\text{ odd}} \left(|x_i|\left|a_i - a_{i+1}\right| + \left|a_{i+1}\right|\left|x_i - x_{i+1}\right|\right) \\
&\leq \max(\|a\|_\infty, \|x\|_\infty) \sum_{i\text{ odd}} \left(\left|a_i - a_{i+1}\right| + \left|x_i - x_{i+1}\right|\right) \\
&\leq \max(\|a\|_\infty, \|x\|_\infty) \sum_{i\text{ odd}} (C_a + C_x) \\
&= \max(\|a\|_\infty, \|x\|_\infty)(C_a + C_x)\frac{I}{2} \\
&= \max(\|a\|_\infty, \|x\|_\infty)\frac{I\left(L_g + L_f\right)(u - l)}{2(I - 1)} \\
&\leq \max(\|g\|_\infty, \|f\|_\infty)\frac{I\left(L_g + L_f\right)(u - l)}{2(I - 1)}
\end{aligned}$$

8.3.b L_1 norm Constraint

Proof. For I even,

$$
\begin{aligned}
|2\|\overline{x}\| - \|x\|_1| &= \left|2\sum_{i \text{ odd}} |x_i - \sum_{i=1}^{I} |x_i||\right| \\
&= \left|\sum_{i \text{ odd}} |x_i + \sum_{i \text{ odd}} |x_i| - \left(\sum_{i \text{ odd}} |x_i| + \sum_{i \text{ even}} |x_i|\right)|\right| \\
&= \left|\sum_{i \text{ odd}} |x_i - \sum_{i \text{ even}} |x_i||\right| \\
&= \left|\sum_{i \text{ odd}} |x_i - \sum_{i \text{ odd}} |x_{i+1}||\right| \\
&\leq \sum_{i \text{ odd}} ||x_i - |x_{i+1}||| \\
&\leq \sum_{i \text{ odd}} |x_i - x_{i+1}| \\
&\leq \sum_{i \text{ odd}} C \\
&= C\frac{I}{2} \\
&= \frac{IL(u-l)}{2(I-1)}.
\end{aligned}
$$

For I odd, we have

$$
\begin{aligned}
\left|2\|\overline{x}\| - \frac{I+1}{I}\|x\|_1\right| &= \left|2\|\overline{x}\| - \|x\|_1 - \frac{\|x\|_1}{I}\right| \\
&= \left|2\sum_{i \text{ odd}}|x_i - \sum_{i=1}^{I}|x_i| - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{i \text{ odd}}|x_i + \sum_{i \text{ odd}}|x_i| - \left(\sum_{i \text{ odd}}|x_i| + \sum_{i \text{ even}}|x_i|\right) - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{i \text{ odd}}|x_i - \sum_{i \text{ even}}|x_i| - \frac{\|x\|_1}{I}|\right| \\
&= \left|\sum_{j=1}^{(I-1)/2}|x_{2j-1} + |x_I| - \sum_{j=1}^{(I-1)/2}|x_{2j}| - \frac{\|x\|_1}{I}|\right| \\
&\leq \sum_{j=1}^{(I-1)/2}||x_{2j-1} - |x_{2j}||| + \left||x_I - \frac{\|x\|_1}{I}|\right| \\
&\leq \sum_{j=1}^{(I-1)/2}|x_{2j-1} - x_{2j}| + \frac{1}{I}|I|x_I - \|x\|_1|| \\
&\leq \sum_{j=1}^{(I-1)/2} C + \frac{1}{I}\left|\sum_{i=1}^{I}(|x_I - |x_i|)|\right| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}||x_I - |x_i||| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}|x_I - x_i| \\
&\leq C\frac{I-1}{2} + \frac{1}{I}\sum_{i=1}^{I}C(I-i) \text{ (Apply Lipschitz recursively)} \\
&= C\frac{I-1}{2} + \frac{1}{I}\frac{CI(I-1)}{2} \\
&= \frac{L(u-l)}{2} + \frac{L(u-l)}{2} \\
&= L(u-l).
\end{aligned}
$$

8.4 Multi-scale Convergence Proofs

8.4.a Inexact Interpolation Error Proof

Proof of Lemma 6.5.

Proof. We let the inexact values be $\tilde{y}[k] = y[k] + \delta_k$, and we interpolate the inexact values to get $\hat{\tilde{Y}}$:

$$\begin{aligned}\hat{\hat{Y}}[j] &= \begin{cases} \tilde{y}\left[\frac{j+1}{2}\right] & \text{if } \ j \ \text{ is odd} \\ \frac{1}{2}\left(\tilde{y}\left[\frac{j}{2}\right] + \tilde{y}\left[\frac{j}{2}+1\right]\right) & \text{if } \ j \ \text{ is even} \end{cases} \\ &= \begin{cases} y\left[\frac{j+1}{2}\right] + \delta_{\frac{j+1}{2}} & \text{if } \ j \ \text{ is odd} \\ \frac{1}{2}\left(y\left[\frac{j}{2}\right] + y\left[\frac{j}{2}+1\right]\right) + \frac{1}{2}\left(\delta_{\frac{j}{2}} + \delta_{\frac{j}{2}+1}\right) & \text{if } \ j \ \text{ is even} \end{cases}.\end{aligned}$$

So for odd j,

$$\left|\hat{\hat{Y}}[j] - \hat{Y}[j]\right| = \left|\delta_{\frac{j+1}{2}}\right| := e[j],$$

and even j,

$$\left|\hat{\hat{Y}}[j] - \hat{Y}[j]\right| = \frac{1}{2}\left|\delta_{\frac{j}{2}} + \delta_{\frac{j}{2}+1}\right| := e[j].$$

So we have

$$\left\|\hat{\hat{Y}} - \hat{Y}\right\| = \sqrt{\sum_{j\in[J]} \left|\hat{\hat{Y}}[j] - \hat{Y}[j]\right|^2} = \sqrt{\sum_{j\in[J]} |e[j]|^2} = \|e\|$$

We would like some bound on $\|e\|$ in terms of the closeness of y and $\tilde{y}$ (that is, in terms of δ).

$$\begin{aligned}
\|e\|^2 &= \sum_{j\text{ odd}} |e[j]|^2 + \sum_{j\text{ even}} |e[j]|^2 \\
&= \sum_{j\text{ odd}} \left|\delta_{\frac{j+1}{2}}\right|^2 + \sum_{j\text{ even}} \left|\frac{1}{2}\left(\delta_{\frac{j}{2}} + \delta_{\frac{j}{2}+1}\right)\right|^2 \\
&= \sum_{k=1}^{K} \left|\delta_k\right|^2 + \frac{1}{4}\sum_{k=1}^{K-1} \left(\delta_k + \delta_{k+1}\right)^2 \\
&= \|\delta\|^2 + \frac{1}{4}\sum_{k=1}^{K-1} \left(\delta_k^2 + \delta_{k+1}^2 + 2\delta_k\delta_{k+1}\right) \\
&= \|\delta\|^2 + \frac{1}{4}\left(\sum_{k=1}^{K-1} \delta_k^2 + \sum_{k=1}^{K-1} \delta_{k+1}^2 + 2\sum_{k=1}^{K-1} \delta_k\delta_{k+1}\right) \\
&\leq \|\delta\|^2 + \frac{1}{4}\left(\sum_{k=1}^{K} \delta_k^2 + \sum_{k=0}^{K-1} \delta_{k+1}^2 + 2\sum_{k=1}^{K-1} \delta_k\delta_{k+1}\right) \\
&= \|\delta\|^2 + \frac{1}{4}\left(\|\delta\|^2 + \|\delta\|^2 + 2\sum_{k=1}^{K-1} \delta_k\delta_{k+1}\right) \\
&= \frac{3}{2}\|\delta\|^2 + \frac{1}{2}\sum_{k=1}^{K-1} \delta_k\delta_{k+1} \\
&\leq \frac{3}{2}\|\delta\|^2 + \frac{1}{2}\sqrt{\sum_{k=1}^{K-1} \delta_k^2}\sqrt{\sum_{k=1}^{K-1} \delta_{k+1}^2} \quad \text{(Cauchy–Schwarz)} \\
&\leq \frac{3}{2}\|\delta\|^2 + \frac{1}{2}\sqrt{\sum_{k=1}^{K} \delta_k^2}\sqrt{\sum_{k=0}^{K-1} \delta_{k+1}^2} \\
&= \frac{3}{2}\|\delta\|^2 + \frac{1}{2}\|\delta\|\|\delta\| \\
&= 2\|\delta\|^2
\end{aligned}$$

Therefore,

$$\|e\| \leq \sqrt{2}\|\delta\|$$

or substituting our notation,

$$\left\|\hat{\hat{Y}} - \hat{Y}\right\| \leq \sqrt{2}\|\tilde{y} - y\|.$$

This is saying that the error in our fine grid (Y) is bounded by a factor of $\sqrt{2}$ of the error in the coarse grid (y) when we double the number of points.

Now we can use the triangle inequality to bound the difference between the interpolated approximate values $\hat{\hat{Y}}$ and the true values on the finer grid Y:

$$\begin{aligned}\left\|\hat{\hat{Y}} - Y\right\| &\leq \left\|\hat{\hat{Y}} - \hat{Y}\right\| + \left\|\hat{Y} - Y\right\| \\ &\leq \sqrt{2}\|\tilde{y} - y\| + \frac{L}{2\sqrt{K-1}}\|a - b\|_2.\end{aligned}$$

END OF PROOF

8.4.b Re-solved Multi-scale Descent Error Proof

Proof of Theorem 6.2.

Using the lemmas in Section 6.4.b, we showed

$$\left\|\hat{\hat{Y}} - Y\right\| \leq \sqrt{2}\|\tilde{y} - y\| + \frac{L|a-b|}{2\sqrt{K-1}}.$$

Note the L is the Lipchitz constant of the continuous function f, not the smoothness of the objective $\mathcal{L}$ above, and the K is the number of points in the discretization, not the number of iterations. Translating this into our notation for the multi-scaled descent, we have

$$\left\|x^0_{\frac{s}{2}} - x^*_{\frac{s}{2}}\right\| = \left\|\hat{x}^{K_s}_s - x^*_{\frac{s}{2}}\right\| \leq \sqrt{2}\left\|x^{K_s}_s - x^*_s\right\| + \frac{L|a-b|}{2\sqrt{2^{S-s+1}}}.$$

Note that at scale s, we have $2^{S-s+1} + 1$ many points in our discretization, where we have $2^S + 1$ many points in our finest scale, and the discretization is over the interval $a \leq t \leq b$ for an L Lipschitz function f.

Combining this with our convergence for gradient descent gives us the inequality

$$\begin{aligned}\left\|x^0_{\frac{s}{2}} - x^*_{\frac{s}{2}}\right\| &\leq \sqrt{2}\left\|x^{K_s}_s - x^*_s\right\| + \frac{L|a-b|}{2\sqrt{2^{S-s+1}}} \\ &\leq \sqrt{2}(1-c)^{K_s}\left\|x^0_s - x^*_s\right\| + \frac{L|a-b|}{2\sqrt{2^{S-s+1}}}\end{aligned}$$

I noticed I am using s in two different ways here. Originally, s was supposed to be the number of points we skip, as in "only keep every s points in our discretization". This is meant to start at some large power of 2 and keep halving until we hit 1. But the s in 2^{S-s+1} counts what scale we are at and starts at S and decreasing by 1 until we hit 1. The second definition makes more sense so we will go with that from now on. This means our descent lemma looks like

$$\left\|x^0_{s-1} - x^*_{s-1}\right\| \leq \sqrt{2}(1-c)^{K_s}\left\|x^0_s - x^*_s\right\| + \frac{L|a-b|}{2\sqrt{2^{S-s+1}}}.$$

Or writing it in terms of $s+1$:

$$\left\|x^0_s - x^*_s\right\| \leq \sqrt{2}(1-c)^{K_{s+1}}\left\|x^0_{s+1} - x^*_{s+1}\right\| + \sqrt{2^s}\frac{L|a-b|}{2\sqrt{2^S}}.$$

Let $e_s = \left\|x^0_s - x^*_s\right\|$ and $C = \frac{L|a-b|}{2\sqrt{2^{S+1}}}$ to clean up notation

$$e_s \leq \sqrt{2}(1-c)^{K_{s+1}} e_{s+1} + \sqrt{2^{s+1}}C.$$

Now we recurse! Starting from the last initial error at the finest scale e_1, we want to write this in terms of the first initial error at the largest scale e_S.

$$\begin{aligned}
e_1 &\leq \sqrt{2}(1-c)^{K_2} e_2 + \sqrt{2^2}C \\
&\leq \sqrt{2}(1-c)^{K_2}\left(\sqrt{2}(1-c)^{K_3} e_3 + \sqrt{2^3}C\right) + \sqrt{2^2}C \\
&= \sqrt{2}(1-c)^{K_2}\sqrt{2}(1-c)^{K_3} e_3 + \sqrt{2}(1-c)^{K_2}\sqrt{2^3}C + \sqrt{2^2}C \\
&= \sqrt{2}^2(1-c)^{K_2+K_3} e_3 + \sqrt{2}(1-c)^{K_2}\sqrt{2^3}C + \sqrt{2^2}C
\end{aligned}$$

Before going further, we actually want to run gradient descent with the starting point x_1^0 (the x inside e_1), so we should really be writing

$$\begin{aligned}
&\left\|x_1^{K_1} - x_1^*\right\| \\
&\leq (1-c)^{K_1}\left\|x_1^0 - x_1^*\right\| \\
&= (1-c)^{K_1} e_1 \\
&\leq (1-c)^{K_1}\left(\sqrt{2}(1-c)^{K_2} e_2 + \sqrt{2^2}C\right) \\
&= \sqrt{2}(1-c)^{K_1+K_2} e_2 + (1-c)^{K_1}\sqrt{2^2}C \\
&\leq \sqrt{2}(1-c)^{K_1+K_2}\left(\sqrt{2}(1-c)^{K_3} e_3 + \sqrt{2^3}C\right) + (1-c)^{K_1}\sqrt{2^2}C \\
&= \sqrt{2}^2(1-c)^{K_1+K_2+K_3} e_3 + \sqrt{2}(1-c)^{K_1+K_2}\sqrt{2^3}C + (1-c)^{K_1}\sqrt{2^2}C \\
&\leq \sqrt{2}^2(1-c)^{K_1+K_2+K_3}\left(\sqrt{2}(1-c)^{K_4} e_4 + \sqrt{2^4}C\right) + \sqrt{2}(1-c)^{K_1+K_2}\sqrt{2^3}C + (1-c)^{K_1}\sqrt{2^2}C \\
&= \sqrt{2}^3(1-c)^{K_1+K_2+K_3+K_4} e_4 + \sqrt{2}^2(1-c)^{K_1+K_2+K_3}\sqrt{2^4}C + \sqrt{2}(1-c)^{K_1+K_2}\sqrt{2^3}C + (1-c)^{K_1}\sqrt{2^2}C \\
&\vdots \\
&\leq \sqrt{2}^{S-1}(1-c)^{\sum_{s=1}^{S} K_s} e_S + \sum_{s=1}^{S-1}\sqrt{2}^{s-1}(1-c)^{\sum_{t=1}^{s} K_t}\sqrt{2^{s+1}}C \\
&= \sqrt{2^{S-1}}(1-c)^{\sum_{s=1}^{S} K_s} e_S + C\sum_{s=1}^{S-1} 2^s(1-c)^{\sum_{t=1}^{s} K_t}.
\end{aligned}$$

8.4.c Freezed Multi-scale Descent Error Proof

Proof of Theorem 6.3.

Using the work below, we have

$$\left\|e_{(s)}^0\right\| \leq \frac{L|b-a|}{2\sqrt{2^{S-s}}} + \left\|e_{s+1}^{K_{s+1}}\right\|.$$

Putting this back into the GD inequality gets us

$$\left\| e_{(s)}^{K_s} \right\| \leq (1-c)^{K_s} \left(\frac{L|b-a|}{2\sqrt{2^{S-s}}} + \left\| e_{s+1}^{K_{s+1}} \right\| \right).$$

And finally, we get the recursion relation

$$\left\| e_s^{K_s} \right\|^2 \leq (1-c)^{2K_s} \left(\frac{L|b-a|}{2\sqrt{2^{S-s}}} + \left\| e_{s+1}^{K_{s+1}} \right\| \right)^2 + \left\| e_{s+1}^{K_{s+1}} \right\|^2$$

We will use big $C = \frac{L}{2}|b-a|$ to clean up a bit. Note little $c = \frac{\mu}{L_{\nabla \ell}}$ is the ratio of strongly convex to smoothness of the loss function ℓ.

$$\left\| e_s^{K_s} \right\|^2 \leq (1-c)^{2K_s} \left(\frac{C}{\sqrt{2^{S-s}}} + \left\| e_{s+1}^{K_{s+1}} \right\| \right)^2 + \left\| e_{s+1}^{K_{s+1}} \right\|^2$$

Or the looser bound by removing the squares

$$\begin{aligned} \left\| e_s^{K_s} \right\| &\leq (1-c)^{K_s} \left(\frac{C}{\sqrt{2^{S-s}}} + \left\| e_{s+1}^{K_{s+1}} \right\| \right) + \left\| e_{s+1}^{K_{s+1}} \right\| \\ &= \left(1 + (1-c)^{K_s}\right) \left\| e_{s+1}^{K_{s+1}} \right\| + C(1-c)^{K_s} \frac{1}{\sqrt{2^{S-s}}} \end{aligned}$$

- we have true points x which are true sample values of an L Lipschitz function
- we have an approximation of x, given by $\hat{x}$
- then we have a linear interpolation of $\hat{x}$ given by $\hat{X}$. These are ONLY the in-between points
- the true values at the in-between points are Y
- the linear interpolation of the true points is X
- the respective errors are e and E
- want to bound E by e
- suppose we have $M = 2^{S-s} + 1$ points at the coarse scale for x, for some positive integer s (this is the scale $s+1$, the previous scale)
- we will have $N = 2^{S-s} = M - 1$ in-between points at the finer grid for X
  - Note, this gives $2^{S-s+1} + 1$ may points at the current scale s

We have

$$\hat{x}_i - x_i = e_i$$

and the linear interpolation (only the in-between points) is

$$\hat{X}_j = \frac{1}{2}\left(\hat{x}_j + \hat{x}_{j+1}\right)$$

Note there is one fewer point. Similarly, with the interpolation of the true values

$$X_j = \frac{1}{2}\left(x_j + x_{j+1}\right)$$

The error is defined as

$$E_j = \hat{X}_j - Y_j.$$

This is what we want to bound. We also have the difference between the interpolation of the approximate and interpolation of the true values

$$\hat{X}_j - X_j = \frac{1}{2}\left(e_j + e_{j+1}\right) := \delta_j$$

There is also the difference between the true values at the in-between points Y and the interpolation of the true points X.

$$X_j - Y_j$$

We have

$$\|E\| = \left\|\hat{X} - Y\right\| \leq \|X - Y\| + \left\|\hat{X} - X\right\| = \|X - Y\| + \|\delta\|$$

Now

$$\begin{aligned} \|X - Y\|^2 &= \sum_{j=1}^{N} \left(X_j - Y_j\right)^2 \\ &\leq \sum_{j=1}^{N} \left(\frac{L}{2}\Delta_j\right)^2 \\ &\leq N\left(\frac{L}{2}\Delta_j\right)^2 \end{aligned}$$

where Δ_i is the (input) space between the i and $i+1$ points. For M equally spaced points on (and including the boundary) the interval $[a, b]$,

$$\Delta_j = \frac{b-a}{M-1} = \frac{b-a}{N}$$

So we have

$$\|X - Y\| \leq \sqrt{N}\frac{L}{2}\frac{b-a}{N} = \frac{L|b-a|}{2\sqrt{N}}.$$

Now we look at the second term δ.

$$\begin{aligned}
\|\delta\|^2 &= \sum_{j=1}^{N} \delta_j^2 \\
&= \sum_{j=1}^{N} \frac{1}{2^2}\left(e_j + e_{j+1}\right)^2 \\
&= \frac{1}{4}\sum_{j=1}^{N}\left(e_j^2 + e_{j+1}^2 + 2e_j e_{j+1}\right) \\
&= \frac{1}{4}\left(\sum_{j=1}^{N} e_j^2 + \sum_{j=1}^{N} e_{j+1}^2 + 2\sum_{j=1}^{N} e_j e_{j+1}\right) \\
&\leq \frac{1}{4}\left(\sum_{j=1}^{N} e_j^2 + \sum_{j=1}^{N} e_{j+1}^2 + 2\sqrt{\sum_{j=1}^{N} e_j^2}\sqrt{\sum_{j=1}^{N} e_{j+1}^2}\right) \\
&\leq \frac{1}{4}\left(\sum_{i=1}^{M} e_i^2 + \sum_{i=1}^{M} e_j^2 + 2\sqrt{\sum_{i=1}^{M} e_i^2}\sqrt{\sum_{i=1}^{M} e_i^2}\right) \\
&= \frac{1}{4}(\|e\|^2 + \|e\|^2 + 2\|e\|\|e\|) \\
&= \frac{1}{4}(4\|e\|^2) \\
&= \|e\|^2
\end{aligned}$$

So finally, we have

$$\|E\| \leq \frac{L|b-a|}{2\sqrt{N}} + \|e\|.$$

This is the same thing as we got in the previous attempt, but without the factor of $\sqrt{2}$.

We have the relation

$$\left\|e_s^{K_s}\right\| \leq \left(1 + (1-c)^{K_s}\right)\left\|e_{s+1}^{K_{s+1}}\right\| + C(1-c)^{K_s}\frac{1}{\sqrt{2^{S-s}}}$$

for every $s = 1, ..., S$. So let's expand this.

$$\begin{aligned}
\left\|e_1^{K_1}\right\| &\leq (1+(1-c)^{K_1})\left\|e_2^{K_2}\right\| + C(1-c)^{K_1}\frac{1}{\sqrt{2^{S-1}}} \\
&\leq (1+(1-c)^{K_1})\left((1+(1-c)^{K_2})\left\|e_3^{K_3}\right\| + C(1-c)^{K_2}\frac{1}{\sqrt{2^{S-2}}}\right) + C(1-c)^{K_1}\frac{1}{\sqrt{2^{S-1}}} \\
&= (1+(1-c)^{K_1})(1+(1-c)^{K_2})\left\|e_3^{K_3}\right\| \\
&\qquad + C(1+(1-c)^{K_1})(1-c)^{K_2}\frac{1}{\sqrt{2^{S-2}}} + C(1-c)^{K_1}\frac{1}{\sqrt{2^{S-1}}} \\
&\leq (1+(1-c)^{K_1})(1+(1-c)^{K_2})\left((1+(1-c)^{K_3})\left\|e_4^{K_4}\right\| + C(1-c)^{K_3}\frac{1}{\sqrt{2^{S-3}}}\right) \\
&\qquad + C(1+(1-c)^{K_1})(1-c)^{K_2}\frac{1}{\sqrt{2^{S-2}}} + C(1-c)^{K_1}\frac{1}{\sqrt{2^{S-1}}} \\
&= (1+(1-c)^{K_1})(1+(1-c)^{K_2})(1+(1-c)^{K_3})\left\|e_4^{K_4}\right\| \\
&\qquad + C(1+(1-c)^{K_1})(1+(1-c)^{K_2})(1-c)^{K_3}\frac{1}{\sqrt{2^{S-3}}} \\
&\qquad + C(1+(1-c)^{K_1})(1-c)^{K_2}\frac{1}{\sqrt{2^{S-2}}} + C(1-c)^{K_1}\frac{1}{\sqrt{2^{S-1}}}
\end{aligned}$$

So the general formula goes to

$$\left\|e_1^{K_1}\right\| \leq \left\|e_S^{K_S}\right\| \prod_{s=1}^{S-1}(1+(1-c)^{K_s}) + C\sum_{s=1}^{S-1}\frac{1}{\sqrt{2^{S-s}}}(1-c)^{K_s}\prod_{j=1}^{s-1}\left(1+(1-c)^{K_j}\right).$$

After adding the GD on the coarsest scale, we get

$$\left\|e_1^{K_1}\right\| \leq \left\|e_S^0\right\|(1-c)^{K_S}\prod_{s=1}^{S-1}(1+(1-c)^{K_s}) + C\sum_{s=1}^{S-1}\frac{1}{\sqrt{2^{S-s}}}(1-c)^{K_s}\prod_{j=1}^{s-1}\left(1+(1-c)^{K_j}\right).$$

This gives us the first upper bound for any plan of iterations K_s. If we now assume each scale uses the name number of iterations $K_s = K$ we get the following.

$$\left\|e_1^{K_1}\right\| \leq \left\|e_S^0\right\|(1-c)^K\left(1+(1-c)^K\right)^{S-1} + C\sum_{s=1}^{S-1}\frac{1}{\sqrt{2^{S-s}}}(1-c)^K\left(1+(1-c)^K\right)^{s-1}.$$

Via wolfram alpha, we have

$$\begin{aligned}
&\sum_{s=1}^{S-1}\left((1-c)^K\left((1-c)^K+1\right)^{s-1}\right)\frac{1}{\sqrt{2^{S-s}}} \\
&= \frac{(1-c)^K\left(\sqrt{2^S}\left((1-c)^K+1\right)^S - \sqrt{2}\left((1-c)^K+1\right)\right)}{\sqrt{2^S}\left((1-c)^K+1\right)\left(\sqrt{2}(1-c)^K+\sqrt{2}-1\right)}
\end{aligned}$$

Setting $d(K) = (1-c)^K$ gives the final upper bound in the theorem.

8.4.d Expected Projected Gradient Descent Convergence Proof

Proof of Theorem 6.4.

Case 1: $\left\|x_1^*\right\|_2 = 1$.

Without loss of generality, assume (by symmetry) that $\left(x_1^*\right)_I = 1$ and $\left(x_1^*\right)_i = 0$ (fix a point on the sphere at the pole) so that $x_1^* = e_I$ (the unit vector! Not the error!). Note that this is not a realistic solution since we already assume the solution comes from a continuous function but we'll use this to illustrate how a warm start from these interpolations can do better than a random start.

$$\begin{aligned}
\mathbb{E}_{g_i \sim \mathcal{N}}\|g - e_I\|_2^2 &= \mathbb{E}_{g_i \sim \mathcal{N}}\left[\sum_{i=1}^{I}\left(g_i - \left(e_I\right)_i\right)^2\right] \\
&= \mathbb{E}_{g_i \sim \mathcal{N}}\left[\sum_{i=1}^{I-1}\left(g_i - 0\right)^2 + \left(g_i - 1\right)^2\right] \\
&= \sum_{i=1}^{I-1}\mathbb{E}_{g_i \sim \mathcal{N}}[g_i^2] + \mathbb{E}_{g_i \sim \mathcal{N}}\left[\left(g_i - 1\right)^2\right] \\
&= \sum_{i=1}^{I-1} 1 + \mathbb{E}_{g_i \sim \mathcal{N}}[g_i^2 - 2g_i + 1] \\
&= I - 1 + (1) - 2(0) + 1 \\
&= I + 1.
\end{aligned}$$

With some Gaussian concentration, we can square root both sides.

Case 2: $\left\|x_1^*\right\|_2 = 0$.

Assume the solution is centred so that $x_1^* = 0 \in \mathbb{R}^I$. Here, we have the well-known result

$$\mathbb{E}\left[\left\|x_1^0 - x_1^*\right\|\right] = \mathbb{E}[\|g - 0\|] = \sqrt{I}.$$

So either way, our initial error for a scaled or centred problem goes like $\sim \sqrt{I}$. Since the number of points we have is I is one plus a power of two $I = 2^S + 1$, we *expect* (that is, with high probability) the following convergence

$$\left\|x_1^K - x_1^*\right\| \leq (1 - c)^K \sqrt{2^S + 2}.$$

So to ensure $\left\|x_1^K - x_1^*\right\| \leq \epsilon$, we need

$$\begin{aligned}
(1-c)^K\sqrt{2^S+2} &\leq \epsilon \\
\frac{\sqrt{2^S+2}}{\epsilon} &\leq (1-c)^{-K} \\
\log\left(\frac{\sqrt{2^S+2}}{\epsilon}\right) &\leq K\log\left((1-c)^{-1}\right) \\
\frac{\log\left(\frac{\sqrt{2^S+2}}{\epsilon}\right)}{-\log(1-c)} &\leq K \\
\frac{\frac{1}{2}\log\left(2^S+2\right)+\log(1/\epsilon)}{-\log(1-c)} &\leq K.
\end{aligned}$$

For large S, this is approximately

$$\frac{\log(1/\epsilon)+\frac{S}{2}\log(2)}{-\log(1-c)} \leq K.$$

8.4.e Expected Resolved Multi-scale Descent Convergence Proof

Proof of Theorem 6.5.

First we need to have a handle on how close we are to our coarsest discretization at scale $s = s_0$. The coarsest we could go would be when $s_0 = S$ which would result in $2^{S-s_0+1}+1 = 3$ points total at $t = a$, $\frac{1}{2}(a+b)$, and b.

For the centred problem, $x_{s_0}^* = 0 \in \mathbb{R}^{2^{S-s_0+1}+1}$, so we would expect a Gaussian initialization to have an initial error of

$$\mathbb{E}\left[\left\|x_{s_0}^0 - x_{s_0}^*\right\|\right] = \sqrt{2^{S-s_0+1}+1}.$$

If we used the scaled problem, it is a bit tricker to consider what happens when we only include $2^{S-s_0+1}+1$ many points, out of the total 2^S+1 possible points.

We will actually work out a bound of $\sqrt{2^{S-s_0+1}+2}$ by considering the “smoothest” case where $\left(x_1^*\right)_i = \frac{1}{\sqrt{2^S+1}}$ (normalized perfectly to a diagonal), and the “roughest” case where $x_1^* = e_{2^S+1} \in \mathbb{R}^{2^S+1}$ was aligned to the pole. After discretization, the smooth case still has all the same entries, but there are just less of them ($2^{S-s_0+1}+1$ many). In the rough case, we stay aligned to a pole $x_{s_0}^* = e_{2^{S-s_0+1}+1} \in \mathbb{R}^{2^{S-s_0+1}+1}$ since the last entry stays at a one, and the rest are still zero. We use a standard normal initialization of $g \in \mathbb{R}^{2^{S-s_0+1}+1}$ in the smaller space. This is identical to choosing the same initialization on the finer grid, and dropping our the coordinates that we skip. So this really is a fair comparison to the regular gradient descent! In the smooth case,

$$
\begin{aligned}
\mathbb{E}\left[\left\|x_{s_0}^0 - x_{s_0}^*\right\|^2\right] &= \sum_{i=1}^{2^{S-s_0+1}+1} \mathbb{E}\left(g_i - \frac{1}{\sqrt{2^S+1}}\right)^2 \\
&= \sum_{i=1}^{2^{S-s_0+1}+1} \left(\mathbb{E}[g_i^2] - 2\frac{1}{\sqrt{2^S+1}}\mathbb{E}[g_i] + \frac{1}{2^S+1}\mathbb{E}[1]\right) \\
&= \sum_{i=1}^{2^{S-s_0+1}+1} \left(1 - 0 + \frac{1}{2^S+1}\right) \\
&= \left(2^{S-s_0+1}+1\right)\left(1 + \frac{1}{2^S+1}\right) \\
&= \left(2^{S-s_0+1}+1\right)\left(\frac{2^S+2}{2^S+1}\right).
\end{aligned}
$$

In the rough case

$$
\begin{aligned}
\mathbb{E}\left[\left\|x_{s_0}^0 - x_{s_0}^*\right\|^2\right] &= \sum_{i=1}^{2^{S-s_0+1}+1-1} \mathbb{E}(g_i - 0)^2 + \mathbb{E}\left(g_{2^{S-s_0+1}+1} - 1\right)^2 \\
&= 2^{S-s_0+1} + 2.
\end{aligned}
$$

For $2^{S-s_0+1} \geq 3$ (which is the case), $2^{S-s_0+1} + 2 > \left(2^{S-s_0+1}+1\right)\left(\frac{2^S+2}{2^S+1}\right)$ (in fact, it is bigger by 1 asymptotically as S grows large).

So we are justified in using $\sqrt{2^{S-s_0+1}+2}$ as our *expected* bound on $\left\|x_{s_0}^0 - x_{s_0}^*\right\|$.

This means our descent is

$$
\begin{aligned}
&\left\|x_1^{K_1} - x_1^*\right\| \\
&\leq \sqrt{2^{S-1}}(1-c)^{\sum_{s=1}^{S} K_s} e_S + C\sum_{s=1}^{S-1} 2^s (1-c)^{\sum_{t=1}^{s} K_t} \\
&\leq \sqrt{2^{S-1}}(1-c)^{\sum_{s=1}^{S} K_s} \sqrt{2^{S-S+1}+2} + C\sum_{s=1}^{S-1} 2^s (1-c)^{\sum_{t=1}^{s} K_t} \\
&= \sqrt{2^{S+1}}(1-c)^{\sum_{s=1}^{S} K_s} + C\sum_{s=1}^{S-1} 2^s (1-c)^{\sum_{t=1}^{s} K_t} \\
&= (1-c)^{K_1}\left(\sqrt{2^{S+1}}(1-c)^{\sum_{s=2}^{S} K_s} + 2C + C\sum_{s=2}^{S-1} 2^s (1-c)^{\sum_{t=2}^{s} K_t}\right) \\
&= (1-c)^{K_1}\left(\sqrt{2^{S+1}}(1-c)^{\sum_{s=2}^{S} K_s} + 2\frac{L|a-b|}{2\sqrt{2^{S+1}}} + \frac{L|a-b|}{2\sqrt{2^{S+1}}}\sum_{s=2}^{S-1} 2^s (1-c)^{\sum_{t=2}^{s} K_t}\right) \\
&= (1-c)^{K_1}\sqrt{2^{S+1}}\left((1-c)^{\sum_{s=2}^{S} K_s} + \frac{L|a-b|}{2\cdot 2^S} + \frac{L|a-b|}{2\cdot 2}\sum_{s=2}^{S-1} \frac{(1-c)^{\sum_{t=2}^{s} K_t}}{2^{S-s}}\right).
\end{aligned}
$$

8.4.f Cost of Projected Gradient Descent vs Multi-scale Proof

Proof of Lemma 6.6.

The total cost of regular descent is given by

$$C_{\mathrm{GD}} = C_1 K.$$

For multiscale, it is

$$C_{\mathrm{MS}} = \sum_{s=1}^{S} C_s K_s.$$

It is reasonable to assume that

$$C_s \geq \frac{3}{2} C_{s+1}$$

since one has to compute $2^{S-s+1} + 1$ entries of x_s at scale s. If it is a fixed cost C to compute an entry of x, then we have

$$C_s = C\left(2^{S-s+1} + 1\right).$$

So

$$\frac{C_s}{C_{s+1}} = \frac{2^{S-s+1} + 1}{2^{S-(s+1)+1} + 1} = \frac{2^{S-s+1} + 1}{2^{S-s} + 1} \geq \frac{3}{2}$$

since the function $\frac{2^{x+1}+1}{2^x+1}$ (for nonnegative x) is minimized at $x = 0$, where $x = S - s \geq 0$. This gives us the chain,

$$C_1 \geq \frac{3}{2} C_2 \geq \left(\frac{3}{2}\right)^2 C_3 \geq \ldots \geq \left(\frac{3}{2}\right)^{S-1} C_S$$

or

$$C_S \leq \left(\frac{2}{3}\right) C_{S-1} \leq \ldots \leq \left(\frac{2}{3}\right)^{S-2} C_2 \leq \left(\frac{2}{3}\right)^{S-1} C_1.$$

This means

$$C_{\mathrm{MS}} = \sum_{s=1}^{S} C_s K_s \leq C_1 \sum_{s=1}^{S} \left(\frac{2}{3}\right)^{s-1} K_s.$$

8.4.g Re-solved Multi-scale Descent Cost Proof

Proof of Corollary 6.2.

What we need to ensure, for multi-scale to be cheaper, with our mild assumption on cost at each scale, is

$$C_{\text{MS}} = \sum_{s=1}^{S} C_s K_s \leq C_1 \sum_{s=1}^{S} \left(\frac{2}{3}\right)^{s-1} K_s \leq C_1 K.$$

To leave the most budget left for K_1, we can make all other $K_s = 1$ which gives

$$C_1 \sum_{s=1}^{S} \left(\frac{2}{3}\right)^{s-1} K_s = C_1 \left(K_1 + \sum_{s=2}^{S} \left(\frac{2}{3}\right)^{s-1} \right) = C_1 \left(K_1 + 3 \left(1 - \left(\frac{2}{3}\right)^{S} \right) \right)$$

And we can upper bound by considering what happens when we increase the number of scales $S \to \infty$ (as in the total number of points $I \to \infty$).

$$C_{\text{MS}} < C_1 (K_1 + 3)$$

So interestingly, we can set $K_1 = K - 3$, and the rest of the scales $K_s = 1$ and still be cheaper!

TODO By hand I've shown that for this setup to give a better accuracy, it does not matter what K is (as long at it is 4 or bigger). There is just a minimum scale S that is needed. And for well conditioned problems $0.3 \leq c \leq 1$, a scale bigger than 4 will do the trick. (Note it is not quite 0.3, it can be made slightly lower)

8.4.h Expected Freezed Multi-scale Descent Convergence Proof

Proof of Theorem 6.6.

For multi-scale, we start out with only $I = 2^{S - s_0 + 1} + 1$ points. Taking the largest scale we can $s_0 = S$, we have an initial error bound of $\sqrt{2+2} = 2$ giving us the expected error bound

$$\left\| e_1^{K_1} \right\| \leq 2 d(K) (1 + d(K))^{S-1} + C \frac{d(K) \left(\sqrt{2}^S (d(K)+1)^S - \sqrt{2} (d(K)+1) \right)}{\sqrt{2}^S (d(K)+1) \left(\sqrt{2} d(K) + \sqrt{2} - 1 \right)}$$

for $C = \frac{L}{2} |b - a|$ (see Theorem 6.4).

If we factor out a $d(K)$, we get

$$\left\| e_1^{K_1} \right\| \leq d(K) \left(2 (1 + d(K))^{S-1} + C \frac{\left(\sqrt{2}^S (d(K)+1)^S - \sqrt{2} (d(K)+1) \right)}{\sqrt{2}^S (d(K)+1) \left(\sqrt{2} d(K) + \sqrt{2} - 1 \right)} \right).$$

8.4.i Freezed Multi-scale Descent Cost Proof

Proof of Corollary 6.3.

Recall our expected regular descent error have the bound

$$\left\| x_1^K - x_1^* \right\| \leq (1 - c)^K \sqrt{2^S + 2}.$$

From Theorem 6.6, we have the expected error bound for freezed multi-scale

$$\left\| e_1^{K_1} \right\| \leq d(K) \left(2 (1 + d(K))^{S-1} + C \frac{\left(\sqrt{2}^S (d(K)+1)^S - \sqrt{2} (d(K)+1) \right)}{\sqrt{2}^S (d(K)+1) \left(\sqrt{2} d(K) + \sqrt{2} - 1 \right)} \right),$$

so need the thing in the bracket to be less than $\sqrt{2^S + 2}$.

In the limit as $K \to \infty$, we have that $d(K) \to 0$. So for a very small $d(K)$, we get the bracket expression to be

$$2 + C\frac{\left(\sqrt{2^S} - \sqrt{2}\right)}{\sqrt{2^S}\left(\sqrt{2} - 1\right)}.$$

After simplifying we get

$$2 + C\frac{\left(\sqrt{2}^{S-1} - 1\right)}{\sqrt{2}^{S-1}\left(\sqrt{2} - 1\right)}.$$

What S do we need for multi scale to be more accurate?

$$2 + C\frac{\left(\sqrt{2}^{S-1} - 1\right)}{\sqrt{2}^{S-1}\left(\sqrt{2} - 1\right)} < \sqrt{2^S + 2}$$

$$C < \left(\sqrt{2^S + 2} - 2\right)\frac{\sqrt{2}^{S-1}\left(\sqrt{2} - 1\right)}{\left(\sqrt{2}^{S-1} - 1\right)}$$

$$C < \left(\sqrt{2} - 1\right)\left(\sqrt{2^S + 2} - 2\right)\frac{\sqrt{2}^{S-1}}{\sqrt{2}^{S-1} - 1}$$

The first factor is less than one $\sqrt{2} - 1 < 1$, and the last factor is basically 1 when $S > 6$ so let's use that

$$C < \left(\sqrt{2^S + 2} - 2\right)$$

$$\left(\frac{C}{\sqrt{2} - 1} + 2\right)^2 - 2 < 2^S$$

$$\log_2\left(\left(\frac{C}{\sqrt{2} - 1} + 2\right)^2 - 2\right) < S$$

note because of our approximation, if the above is *not* satisfied, *then* GD is more accurate. This is not an iff condition.

There are a number of ways to modify it so that it becomes "if this condition holds, then multiscale is better". You could replace C with $C + 0.5$ as a simple one.

As a simple case, if $C = \frac{1}{2}$ (the function is 1-Lipschitz on $[0, 1]$), then a scale of $S = 2$ or higher is already better for multi scale.

The above work shows that multi-scale will give a tighter expected final error bound, but we need to make sure multi-scale is cheaper.

Reusing the work shown in Section 8.4.f, if we assume each scale has the same number of iterations $K_s = K'$, then we get

$$C_{\mathrm{MS}} \leq C_1 K' \sum_{s=1}^{S} \left(\frac{2}{3}\right)^{s-1} = C_1 K' 3 \left(1 - \left(\frac{2}{3}\right)^{S}\right) \leq 3 C_1 K'.$$

So if we make sure $K' < \frac{K}{3}$, where K is the number of iterations we perform with regular projected gradient descent, then

$$C_{\mathrm{MS}} < C_{GD}.$$

The largest integer less than $\frac{K}{3}$ would be $\lceil \frac{K}{3} \rceil - 1$.

TODO : Note I think the argument can be tightened to get something like $K' < \frac{K}{2+\delta}$ for a small delta, but this idea should be fine.

The way the expected error upper bound works for the freezed multi-scale, the condition on the number of scales needed S is enough to ensure that multi-scale gives a better error bound, assuming we take both methods to large enough iterations K.

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] J. R. Magnus and H. Neudecker, *Matrix differential calculus with applications in statistics and econometrics*, 3rd ed. Hoboken (N.J.): Wiley, 2019.

[14] 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.

[15] 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).

[16] 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.

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

[18] 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.

[19] 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

[20] 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.

[21] 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.

[22] 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

[23] R. Vershynin, *High-Dimensional Probability*. University of California, Irvine: Cambridge University Press, 2018. Accessed: Apr. 11, 2023. [Online]. Available: https://www.math.uci.edu/~rvershyn/papers/HDP-book/HDP-book.html#

[24] G. H. Golub, V. Klema, and G. W. Stewart, “Rosetak Document 4: Rank Degeneracies and Least Square Problems,” no. 260355. Rochester, NY, Feb. 1977.

[25] N. Gillis, *Nonnegative Matrix Factorization*. Philadelphia, PA: Society for Industrial, Applied Mathematics, 2020. doi: 10.1137/1.9781611976410.

[26] N. Gillis and F. Glineur, “On the geometric interpretation of the nonnegative rank,” *Linear Algebra and its Applications*, vol. 437, no. 11, pp. 2685–2712, Dec. 2012, doi: 10.1016/j.laa.2012.06.038.

[27] H. Akaike, “A new look at the statistical model identification,” *IEEE Transactions on Automatic Control*, vol. 19, no. 6, pp. 716–723, Dec. 1974, doi: 10.1109/TAC.1974.1100705.

[28] K. P. Burnham and D. R. Anderson, *Model Selection and Inference*. New York, NY: Springer, 1998. doi: 10.1007/978-1-4757-2917-7.

[29] G. Schwarz, “Estimating the Dimension of a Model,” *The Annals of Statistics*, vol. 6, no. 2, pp. 461–464, Mar. 1978, doi: 10.1214/aos/1176344136.

[30] A. A. Neath and J. E. Cavanaugh, “The Bayesian information criterion: background, derivation, and applications,” *WIREs Computational Statistics*, vol. 4, no. 2, pp. 199–203, 2012, doi: 10.1002/wics.199.

[31] S. Ubaru and Y. Saad, “Fast methods for estimating the Numerical rank of large matrices,” in *Proceedings of The 33rd International Conference on Machine Learning*, Jun. 2016, pp. 468–477. Accessed: Aug. 19, 2024. [Online]. Available: https://proceedings.mlr.press/v48/ubaru16.html

[32] Z. Ratsimalahelo, “Rank Test Based On Matrix Perturbation Theory,” 2001. Accessed: Feb. 13, 2024. [Online]. Available: https://www.econstor.eu/handle/10419/142479

[33] S. Monti, P. Tamayo, J. Mesirov, and T. Golub, “Consensus Clustering: A Resampling-Based Method for Class Discovery and Visualization of Gene Expression Microarray Data,” *Machine Learning*, vol. 52, no. 1, pp. 91–118, Jul. 2003, doi: 10.1023/A:1023949509487.

[34] J. P. Brunet, P. Tamayo, T. R. Golub, and J. P. Mesirov, “Metagenes and molecular pattern discovery using matrix factorization,” *Proc. Natl. Acad. Sci.*, no. 101, pp. 4164–4169, 2004.

[35] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, “Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior,” in *2011 31st International Conference on Distributed Computing Systems Workshops*, Jun. 2011, pp. 166–171. doi: 10.1109/ICDCSW.2011.20.

[36] W. Austin, T. G. Kolda, and T. Plantenga, “Tensor Rank Prediction via Cross Validation.” Sandia National Labs, Livermore, CA, Aug. 2014.

[37] J. Saylor, K. Sundell, and G. Sharman, “Characterizing Sediment Sources by Non-Negative Matrix Factorization of Detrital Geochronological Data,” *Earth and Planetary Science Letters*, vol. 512, pp. 46–58, Apr. 2019, doi: 10.1016/j.epsl.2019.01.044.

[38] Y. Fu, S. Matsushima, and K. Yamanishi, “Model Selection for Non-Negative Tensor Factorization with Minimum Description Length,” *Entropy*, vol. 21, no. 7, p. 632, Jun. 2019, doi: 10.3390/e21070632.

[39] P. Fogel, C. Geissler, N. Morizet, and G. Luta, “On Rank Selection in Non-Negative Matrix Factorization Using Concordance,” *Mathematics*, vol. 11, no. 22, p. 4611, Jan. 2023, doi: 10.3390/math11224611.

[40] Y. Cai, H. Gu, and T. Kenney, “Rank selection for non-negative matrix factorization,” *Statistics in Medicine*, vol. 42, no. 30, pp. 5676–5693, 2023, doi: 10.1002/sim.9934.

[41] N. Graham, N. Richardson, M. P. Friedlander, and J. Saylor, “Tracing Sedimentary Origins in Multivariate Geochronology via Constrained Tensor Factorization,” *Mathematical Geosciences*, Feb. 2025, doi: 10.1007/s11004-024-10175-0.

[42] 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

[43] 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).

[44] 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).

[45] 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.

[46] 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.

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

[48] 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).

[49] J. J. Benedetto and M. W. Frazier, *Wavelets: Mathematics and Applications*, 1st ed. Boca Raton: CRC Press, 1993. Accessed: Aug. 15, 2024. [Online]. Available: https://doi.org/10.1201/9781003210450

[50] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid Methods*. Academic Press, 2001.

[51] I. Necoara, Y. Nesterov, and F. Glineur, “Linear convergence of first order methods for non-strongly convex optimization,” *Mathematical Programming*, vol. 175, no. 1, pp. 69–107, May 2019, doi: 10.1007/s10107-018-1232-1.