Contents

1 Documentation

In this document, we describe the algorithms that we implemented to compute the Gromov-Wasserstein barycenters we talk about in theory.org.

1.1 MetricMeasureSpace.jl

In this file we implement the struct we talk about in implementation.org.

1.2 RepeatUntilConvergence

In this file, we defined a structure that can repeat a function until convergence. It will be more clear in the following.

1.2.1 abstract type BaseRepeatUntilConvergence{T}

It is a parametric abstract type that has, as subtypes, the concrete parametric types

- RepeatUntilConvergence
- LargeMemoryRepeatUntilConvergence
- SingleValueRepeatUntilConvergence

1.2.2 mutable struct RepeatUntilConvergence{T}

It is a parametric struct defined in the following way:

```
mutable struct RepeatUntilConvergence{T} <: BaseRepeatUntilConvergence{T}
    update_func::Function
    has_converged::Function
    history::CircularBuffer{T}
    init_vals::T
end
```

1. T T is the type on which a type RepeatUntilConvergence depends on: the idea is that we have an update function that takes some input of type T and returns something of type T, so that we can iterate this function.
2. $\text{update}_{\text{func}}$ The update function is the function that we need to iterate on something of type T.
3. $\text{has}_{\text{converged}}$ This function is a criterion for convergence that returns a Bool type. It checks if we have to stop the execution, possibly using the history field.

4. history The history contains the output that we need to store to check the stop criterion. It has been organized in a CircularBuffer{T} type because, as we said above, all the outputs of the update function are of type T and here we can store only that elements, and we didn't choose a Vector{T} because maybe is not necessary to store all the outputs: for example the stop criterion could involve just the last two output, so that we need to have a CircularBuffer{T} of length 2.

5. $\text{init}_{\text{vals}}$ It is the initial value of the iterative process, so it is the first value that is updated, which means that it must be of type T.

6. inner constructor Arguments: an update function, an $\text{has}_{\text{converged}}$ function and a Int $\text{memory}_{\text{size}}$. It returns an element of type RepeatUntilConvergence{T} initializing the memory size of the CircularBuffer{T} according to $\text{memory}_{\text{size}}$. Before doing that, it checks that the update function has a method for type T and returns a type T, and it check that the $\text{has}_{\text{converged}}$ function has a method for type Vector{T} and returns a Bool type, otherwise it raises an error. Achtung: it doesn't initialize $\text{init}_{\text{vals}}$.

1.2.3 mutable struct LargeMamoryRepeatUntilConvergence{T}

It is a parametric struct defined in the following way:

```
mutable struct RepeatUntilConvergence{T} <: BaseRepeatUntilConvergence{T}
    update_func::Function
    has_converged::Function
    history::Vector{T}
    init_vals::T
end
```

It is exactly as the other, except that the history is a Vector{T}. It could be used when we need to use all the outputs the execution produced. We didn't implemented it, because we never use it.

1.2.4 mutable struct SingleValueRepeatUntilConvergence{T}

It is a parametric struct defined in the following way:

```
mutable struct RepeatUntilConvergence{T} <: BaseRepeatUntilConvergence{T}
    update_func::Function
    has_converged::Function
```

```
    history::T
    init_vals::T
end
```

It is exactly as the others, except that the history is an element of type T. It could be used when we need to store only the last output (we could use also the classic RepeatUntilConvergence{T} with $\text{memory}_{\text{size}}$=1, but this is more efficient in this case). We didn't implemented it, because we never use it.

1.2.5 Function $\text{update}_{\text{history}}$!

Arguments (first method): an element of type either RepeatUntilConvergence{T} or LargeMemoryRepeatUntilConvergence{T}, and an element $\text{iter}_{\text{result}}$ of type T (usually this is the result of an execution of the update function). Output: the function pushes the $\text{iter}_{\text{result}}$ at the end of the history.

Arguments (second method): an element of type SingleValueRepeatUntilConvergence{T} and an element $\text{iter}_{\text{result}}$ of type T (usually this is the result of an execution of the update function). Output: it lets the history field to be $\text{iter}_{\text{result}}$.

1.2.6 Function $\text{has}_{\text{converged}}$wrapper

Arguments (first method): an element of type RepeatUntilConvergence{T}. Output: it converts the CircularBuffer{T} into a Vector{T} and then it returns true or false according to the $\text{has}_{\text{converged}}$ function applied on the converted history.

Arguments (second method): an element of type either LargeMemoryRepeatUntilConvergence{T} or SingleValueRepeatUntilConvergence{T}. Output: it returns true or false according to the $\text{has}_{\text{converged}}$ function applied on the converted history.

1.2.7 Function execute!

It just execute the process of a BaseRepeatUntilConvergence{T} element, given an initial value.

1.3 SinkhornKnopp.jl

The Sinkhorn-Knopp algorithm is an iterative algorithm which computes an (approximate) solution of the following minimum problem:

$$\min\ C \cdot + KL(T),$$

where the minimum is taken over all the transport plans between two fixed marginal distributions p and q, ε is a fixed (small) constant and KL is the Kullback-Leibler divergence.

1.3.1 struct $\mathbf{data_{SK}}$

The struct "data_{SK}" contains all the necessary to compute a single update of the Sinkhorn-Knopp algorithm.

```
struct data_SK:
    K::Matrix{Float64}
    p::Vector{Float64}
    q::Vector{Float64}
    T::Matrix{Float64}
    a::Vector{Float64}
    b::Vector{Float64}
    inner_constructor(K,p,q,T)
end
```

1. K The matrix K is the element-wise exponentiation of C/epsilon, so it must be used only with this setting.

2. p and q These two vectors are the marginal distributions, so they must be non-negative and with sum 1.

3. T T is a feasible transport plan between p and q.

4. a and b They are the vectors that are updated by the Sinkhorn algorithm.

5. $\text{inner}_{\text{constructor}}$ It takes just K, p, q and T. It just checks that the dimensions of this object are correct, and then it built an element of type data_{SK} with K, p, q, T, a \= constant vector with sum 1 (it actually could be any initialization, we just decided for this one) and b

1.3.2 Function update$_{\mathbf{SK}}$

Arguments: an element of type data$_{\mathrm{SK}}$. Output: it computes a single iteration of the Sinkhorn algorithm updating a, b and T in the following way:

$$a = \frac{p}{K * b}, \quad b = \frac{q}{K^T * a}, \quad T = diag(a) * K * diag(b)$$

1.3.3 Function compute$_{\mathbf{marginals}}$

Arguments: a squared matrix. Output: two vector, which are obtained summing all the rows and all the columns (one must think the matrix as the element T of a data$_{\mathrm{SK}}$ and the hope is that this two vectors are "similar" to p and q).

1.3.4 Function stop$_{\mathbf{SKT}}$

Arguments: a vector history, of size at least 2, of elements of type data$_{\mathrm{SK}}$ and a float tol, which is the tolerance. Output: it returns true if the 1-norm between history[end].T and history[end-1].T is less than tol, otherwise it return false.

1.3.5 Function stop$_{\mathbf{SKabold}}$

Arguments: a vector history, of size at least 1, of elements of type data$_{\mathrm{SK}}$ and a float tol, which is the tolerance. Output: it computes the marginals μ and ν of history[end].T and it returns true if both the 1-norm of history[end].p-μ and history[end-1].q-ν are less than tol, otherwise it return false.

1.3.6 Function stop$_{\mathbf{SKabnew}}$

Arguments: a vector history, of size at least 1, of elements of type data$_{\mathrm{SK}}$ and a float tol, which is the tolerance. Output: it does exactly the same of stop$_{\mathrm{SKabold}}$ without using directly the T filed of the struct, but recomputing it using a, b and K fields.

1.3.7 Function stop$_{\mathbf{SK}}$

Arguments: a vector history, of size at least 1, of elements of type data$_{\mathrm{SK}}$ and a float tol, which is the tolerance. Output: it returns true if both stop$_{\mathrm{SKT}}$ and stop$_{\mathrm{SKabnew}}$ return true on the same arguments, otherwise it returns false (this is the most precise stop criterion, since it checks both

the difference betwwen the updating of the transport and how much the marginal distributions are different from the ones we want).

1.4 loss.jl

In this file we built a simple struct "loss" to make more compact the syntax in the future algorithms concerning the chosen of the loss function.

The theory tells us that the Gromov-Wasserstein distance between two finite metric measure spaces (C, μ) and (D, ν) is given by

$$GW((C,\mu),(D,\nu)) = \min_T \sum_{i,j,k,l} L(C_{ik}, D_{jl}) T_{ij} T_{kl},$$

where the infimum is taken over all the transport plans T between the marginals μ and ν. In a more compact way, we will write the expression above as $\langle L \otimes T, T \rangle$, where the matrix $L(C, D) \otimes T$ is defined as

$$(L(C,D) \otimes T)_{kl} = L(L(C_{ik}, D_{jl}) T_{ij}).$$

So, to define the Gromov-Wasserstein distance, we need a function $L : \mathbb{R} \to \mathbb{R}$, called loss function. The only admissible functions for this work are the L2 loss and the KL loss, defined as

$$L2(a,b) = (a-b)^2, \quad KL(a,b) = a\log(a/b) - a + b.$$

In general, for this algorithm, one can consider loss functions that can be written as $L(a, b) = f_1(a) + f_2(b) - h_1(a)h_2(b)$ (note that L2 and KL can be written in this way). This form is important for the computation of the tensor product $L(C, D) \otimes T$, that can be computed using the following formula

$$L(C,D) \otimes T = f_1(C) * \mu * ones(n)^T + ones(m) * \nu * f_2(D)^T - h_1(C) * T * h_2(D)^T,$$

where n is the size of (C, μ), m is the size of (D, ν), the exponentiation to T is the transpose and the functions f1, f2, h1, h2 are applied element-wise.

1.4.1 struct loss

It contains all the informations we talked above regarding a loss function.

```
struct Loss:
    string::String
    f1::Function
```

```
    f2::Function
    h1::Function
    h2::Function
end
```

1. string It contains the name of the loss function. The only admissible strings are "L2" and "KL", to distinguish when we use the Euclidean loss or the Kullback-Leibler one.

2. f1, f2, h1, h2 They take a float and give another float. They are defined according to the structure above, depending if string=L2 or string=KL.

3. inner constructor Argument: a string Output: if the string is "L2" or "KL", it defines the function fields according to the decomposition above, otherwise it raises an error.

1.4.2 Function $\mathbf{GW_{cost}}$

Arguments: a Loss field, two metric measure spaces $M = (C, \mu)$ and $N = (D, \nu)$, a matrix of floats, a float ε. Output: it computes the tensor product $E = L(C, D) \otimes T$ (it raises an error if the size are not compatible) and then it returns the component-wise exponentiation of E/ε, so that the output is ready to be given in input as field K of a data_{SK}, so that it can be used for the Sinkhorn algorithm.

1.5 Barycenters.jl

In this file, we use all the code developed before to implement the main algorithm "$\text{GW}_{\text{barycenters}}$" that computes an approximatione of the Gromov Wasserstein barycenter.

1.5.1 Function $\mathbf{update_{transport}}$

Arguments: - a matrix Ts of Float (it is the transport plan to be updated)

- two MetricMeasureSpace data, between it has to compute the optimal transport plan
- a loss function (using a struct loss)
- a positive float epsilon
- a posotive tolerance to stop the Sinkhorn algorithm.

Output: building a struct RepeatUntilConvergence, it runs the Sinkhorn algorithm between the weights of the metric measure spaces with cost K, evaluated using the function $\mathrm{GW}_{\mathrm{cost}}$, so it returns an approximation of the optimal transport plan between the two spaces.

1.5.2 Function $\mathbf{stop_{tranport}}$

Arguments: a Vector of Matrix{Float64} and a Float64. Output: it checks if the last two elements of the Vector are close enough (considering if the ratio between the difference of the matrices and the last matrix is, w.r.t. the infinity norm, less than the error input).

1.5.3 Function $\mathbf{compute_C}$

Arguments: - a ConvexSum type λ

- a Vector of MetricMeasureSpace, that are the ones between, in the end, we want to compute the barycenters
- a Vector of Float64, that will be the weight of a MetricMeasureSpace
- a Vector of Matrix{Float64}, that is a collection of transport plans between p and the collection of MetricMeasurespaces
- a loss function

Output: if the loss function is the Euclidean loss, it calculates

$$C = \frac{1}{pp^T} \sum_s \lambda_s T_s^T C_s T_s,$$

otherwise the loss function is the Kullback-Leibler function and it calculates

$$C = exp(\frac{1}{pp^T} \sum_s \lambda_s T_s^T log(C_s) T_s).$$

Then the output is the MetricMeasureSpace given by the couple (C,p).

1.5.4 Function $\mathbf{init_{Ts}}$

Arguments: two MetricMeasureSpace data. Output: it returns the rank-one matrix obtained by multiplying the weights of the arguments, that is the trivial transport.

1.5.5 Function $\text{update}_{\text{barycenters}}$

Arguments: - a MetricMeasureSpace Cp

- a Vector of MetricMeasureSpace, that are the ones between, in the end, we want to compute the barycenters
- a ConvexSum type λ
- a loss function
- a Float ϵ
- a Float64 Ts_{tol}, that is the error that we can obtain updating the transport plans
- a Float64 SK_{tol}, that is the error that we admit on the iteration of the Sinkhorn algoritm.

Output: for any Cs in the Vector of MetricMeasureSpace, it updates the transport between Cp and Cs until the tolerance Ts_{tol} is satisfied. To do so we use a struct RepeatUntilConvergence, repeating the function $\text{update}_{\text{transport}}$ (keeping fixed all the data but the Ts) with stop criterion given by the function $\text{stop}_{\text{transport}}$, and with initial data given by init_{Ts}(Cp,Cs). In the end the function returns the output of the function $\text{compute}_{\text{C}}$ using our data and the Vector of transport plans that we obtained (so, in the end, it is updating the matrix of the MetricMeasureSpace Cp)

1.5.6 Function $\text{stop}_{\text{barycentersniter}}$

Arguments: a Vector, called history, of MetricMeasureSpace data and an Int64. Output: the algorithm returns true if the length of the history is greater than the Int64 data, otherwise it returns false.

1.5.7 Function init_{C}

Argument: a Vector p of Float64. Output: it initializes the barycenter to be a random square matrix C (with zeros on the diagonal) of size length(p), and then it returns the MetricMeasureSpace given by the couple (C,p).

1.5.8 Function $\text{GW}_{\text{barycenters}}$ (the main function)

Arguments: - a (collection) Vector $\text{Cs}_{\text{collection}}$ of MetricMeasureSpace data

- a ConvexSum type λ
- a Int64 n (the size of the barycenter as a MetricMeasureSpace)
- a Vector p of Float64 (the weight of the barycenter)
- a loss function
- a Float ϵ
- a Int64 Cp_{niter} (the number of iteration of the $\text{update}_{\text{barycenters}}$ function)
- a Float64 Ts_{tol}, that is the error that we can obtain updating the transport plans
- a Float64 SK_{tol}, that is the error that we admit on the iteration of the Sinkhorn algoritm.

Output: keeping fixed the parameters $\text{Cs}_{\text{collection}}$, λ, loss, ϵ, Ts_{tol}, SK_{tol}, it defines the function $\text{update}_{\text{barycentersrepeater}}$, that takes as input a MetricMeasureSpace and returns aMetricMEasureSpace. So, using a struct RepeatUntilConvergence, starting from the MetricMeasureSpace $\text{init}_{\text{C}}(\text{p})$, repeating the function $\text{update}_{\text{barycentersrepeater}}$, with stop function $\text{stop}_{\text{barycentersniter}}$, the functions returns the last MetricMeasureSpace obtained by the iteration, that is the approximation of the barycenters that we were looking for.