Internship on the utilization of `KeOps`: it's use in solving definite positive systems and in the optimal transport field

Tanguy Lefort
With the help and under the supervision of Benjamin Charlier

May 15, 2024

Introduction

With the always increasing number of available data needing computations, the quest to handle them quickly, with a library that supports automatic differentiation is already on by now. A lot of research has been published, considering ways to improve the computation time. Especially in the cases where the operations needed are simple matrix-vector products and iterative system solving that are found in many fields. One, with a developer-like mindset can work his way with CUDA and C++ to use the GPU computation power, others tend to use approaches like sparse matrices or low-rank approximations. The `KeOps` library [2, Charlier Benjamin, Feydy Jean, Glaunès Joan Alexis, Collin, François-David and Durif Ghislain] uses both the power of the GPU and an idea of the sparse principle in order to get an easy accessible and high-performing library to solve kernel based problems. These are indeed used a lot, for example in the branch of optimal transport. So the computational power of `KeOps` was used to create the `GeomLoss` library [4]. A user-friendly library to manipulate OT solvers for large point clouds. Lots of options are available, including the customization of the cost function and the usage of the recent de-biased Sinkhorn's algorithm for the OT problem that allows freedom for the user as long as the required hypothesis are valid.

First, we will consider the conjugate gradient method available in the `PyKeOps` package and compare it the standard-to-use one in `Scipy` in the case of ill-determined systems. Then, after a recapitulation on the optimal transport theory, we will compute the loss with and without the newly available de-biased Sinkhorn's algorithm to see its effect and then take examples to demonstrate cases for which the cost function that the user wants might or not be used from hypothesis to check. And to finish, we will tackle the batch-mode feature of this package especially in the case of samples with different sizes.

1 Manipulating kernels with the `KeOps` library

When working with tensors-related problems, the two main library that are used are `PyTorch` and `TensorFlow`. Although very useful, they currently need the full tensor to be stored in order to compute the results. This can become an issue when using CUDA tensor types. Indeed, these will utilize the GPU for computation, at the cost of a limited storage. Depending on the hardware, only matrix of sizes up to 10 to 50 thousands can be stored on a GPU.

With large datasets, some problems become quickly unfeasible with these libraries. In particular, problems involving kernels or distance matrices product (which are quite common) fall

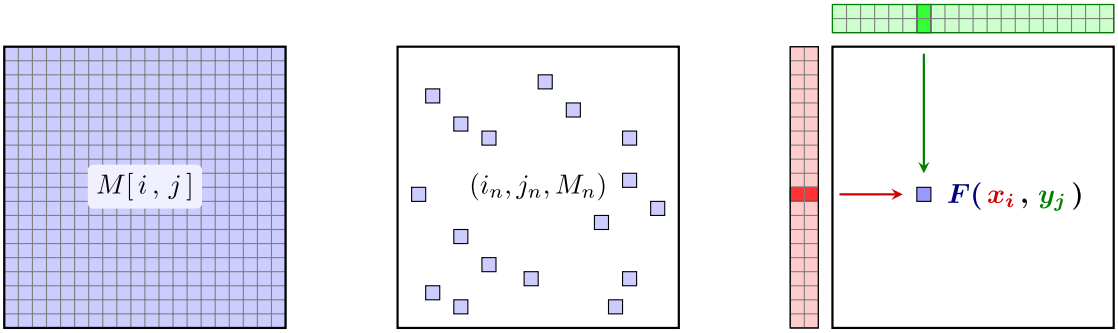
in this category. In dimension D, for a source $x = (x_i) \in \mathbb{R}^{N\times D}$, a target $y = (y_j) \in \mathbb{R}^{M\times D}$ and a signal $b = (b_j) \in \mathbb{R}^{M\times 1}$, a kernel-vector product is simply

$$a = K(x, y)b,$$

with $K(x, y)$ the kernel tensor of size $N \times M$. Instead of computing the whole kernel and then doing the product, `KeOps` idea is to only store the input data and the formula necessary, see Figure (1), to compute

$$a_i = \sum_j K(x_i, y_j)b_j,$$

and execute efficiently the reductions over the index j while parallelizing over the index i (actually blocks of indexes i). It may not be sparse in the traditional sense of the word ($ie$with a lot of zeros), but we can still take advantage of the structure of the data and the context in which it is used.



(a) Dense matrix (b) Sparse matrix (c) Symbolic matrix

Figure 1: Representations of the different types of matrix-storage: (a) is the traditional dense matrix (like a `numpy.ndarray`), (b) is a sparse matrix where we only need to store the indexes of the non-zero values and their value (see the `scipy.sparse` module) and (c) is `KeOps` symbolic storage using only the data and a formula.

This lets us use datasets over a hundred thousand points and compute a kernel-vector product operation compatible with automatic differentiation in seconds with very limited storage needed as explained in the first part of [3] and can be seen in the benchmarks in [2] or in the library documentation:

`https://www.kernel-operations.io/keops/_auto_benchmarks/index.html`

And this gain of time doesn't come with a cost in abstract programming syntax. For example, one can easily compute a Gaussian or Cauchy RBF kernel with the following code using LazyTensors (more user-friendly way to perform kernel products than the `kernel_product` method, faster and also more general because you can use any formula comes to mind from the large choice in the `PyKeOps` operators):

```
import pykeops
from pykeops.torch import Vi, Vj, Pm

# x,y are torch tensor of sizes (N,D) and (M,D) even if D=1
```

```
D_ij = Vi(x).sqdist(Vj(y)) # matrix of euclidean squared distances

# b and sigma are torch tensor of sizes (M,1) and (1)
Kg_ij = (- Pm(1 / sigma ** 2)  * D_ij ).exp() # Lazy Gaussian kernel
Kc_ij = (1 + Pm(1 / sigma ** 2) * D_ij ).power(-1) # Lazy Cauchy kernel

# K(x,y)b products -> (N,1) usual tensors
Kg_ij @ b
Kc_ij @ b
```

1.1 Regularized kernel and conjugate gradient with KeOps

In statistics, a common task is to fit a model to some available data noted $(x, y) = (x_i, y_i)_{i=1}^n$. This translates in finding a function f^* such that :

$$f^* = \arg\min_f \|y - f(x)\|_2^2 + \alpha \|f\|_2^2, \tag{1.1}$$

where $\alpha > 0$ is the ridge-Tikhonov regularization parameter and for any vector $u \in \mathbb{R}^n$, $\|u\|_2^2 = \sum_{i=1}^n u_i^2$.

For that, one can use a Gaussian kernel $K = [k(x_i, x_j)]_{i,j=1}^n$ defined by the equation

$$k(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right).$$

Using the representer theorem [7], it is known that the function we look for can be expressed as the following:

$$f(x) = \sum_{i=1}^n a_i k(x, x_i). \tag{1.2}$$

Substituting equation (1.2) into equation (1.1) and noting $a = (a_1, \ldots, a_n)^\top$, we obtain

$$a^* = \arg\min_{a \in \mathbb{R}^n} \|y - Ka\|^2 + \alpha \|Ka\|^2 = \arg\min_{a \in \mathbb{R}^n} J_\alpha(a).$$

Now, using the fact that K is a symmetric positive definite kernel and differentiating $J_\alpha(a)$ with respect to a to find the roots leads us to the following expression for the solution :

$$(K + \alpha \mathrm{Id}\,) a^* = y. \tag{1.3}$$

For large values of n, solving the system directly by inverting $(K + \alpha \mathrm{Id}\,)$ is not ideal because of both the stability of the numerical precision (related to the condition number) and the time complexity which is $\mathcal{O}(n^3)$.

The system (1.3) can be solved with an iterative method such as the conjugate gradient described in the algorithm (1) available at:

`https://github.com/getkeops/keops/tree/master/pykeops/common`

Algorithm 1 Conjugate gradient algorithm for the system $Ax = b$

Require: linear operator A, RHS of the system $b \in \mathbb{R}^n$, tolerance $\varepsilon > 0$
  $x_0 \leftarrow 0$ if no initial guess was given
  $r_0 \leftarrow b - Ax_0$
  while maximum number of iterations $i_{\max}$ is not reached **do**
    $\rho_{i-1} \leftarrow \|r_i\|^2$
    if it is the first iteration **then**
      $p_1 = r_0$
    else {find another direction}
      $p_i \leftarrow r_{i-1} + \dfrac{\rho_{i-1}}{\rho_{i-2}} p_{i-1}$
    end if
    $q_i = Ap_i$
    $\alpha_i = \dfrac{\rho_{i-1}}{\langle p_i \,|\, q_i \rangle}$
    $x_i = x_{i-1} + \alpha_i p_i$
    $r_i = r_{i-1} - \alpha_i q_i$
    if $\|r_i\|^2 \leq \varepsilon$ **then**
      stop
    else
      continue to the next iteration
    end if
  end while

Each iteration consists of a matrix-vector product so a complexity of $\mathcal{O}(n^2)$. The rest of the operations only consist of vector-vector or scalar-vector products.

The `KeOps` library provides an efficient way to compute the kernel-vector products [2]. A reference for the conjugate gradient algorithm is the `cg` method from the `Scipy` library [10] (based on [1]) implemented using both Python and Fortran and the reverse communication principle *ie*two functions call each other to separate the matrix-vector operations from the rest. Figure (2) represents the different steps accomplished by that algorithm with, in purple, the Fortran parts. The first step after the input calls the `make_system` function from the same package that does the sanity checks. In these, is implemented the fact that the right hand side (RHS) of the equation can not be multidimensional. This means that multiple systems must be solved one at a time.

We shall then compare it to the conjugate gradient that was already used in the `KeOps` library and a "Pythonized" version of `Scipy`'s using Python's dictionaries and `KeOps` power. All the kernel-vector operations will be computed using `KeOps` and GPU's operations were made using a standard gaming GPU (rtx2080). Residuals were calculated using `KeOps`.

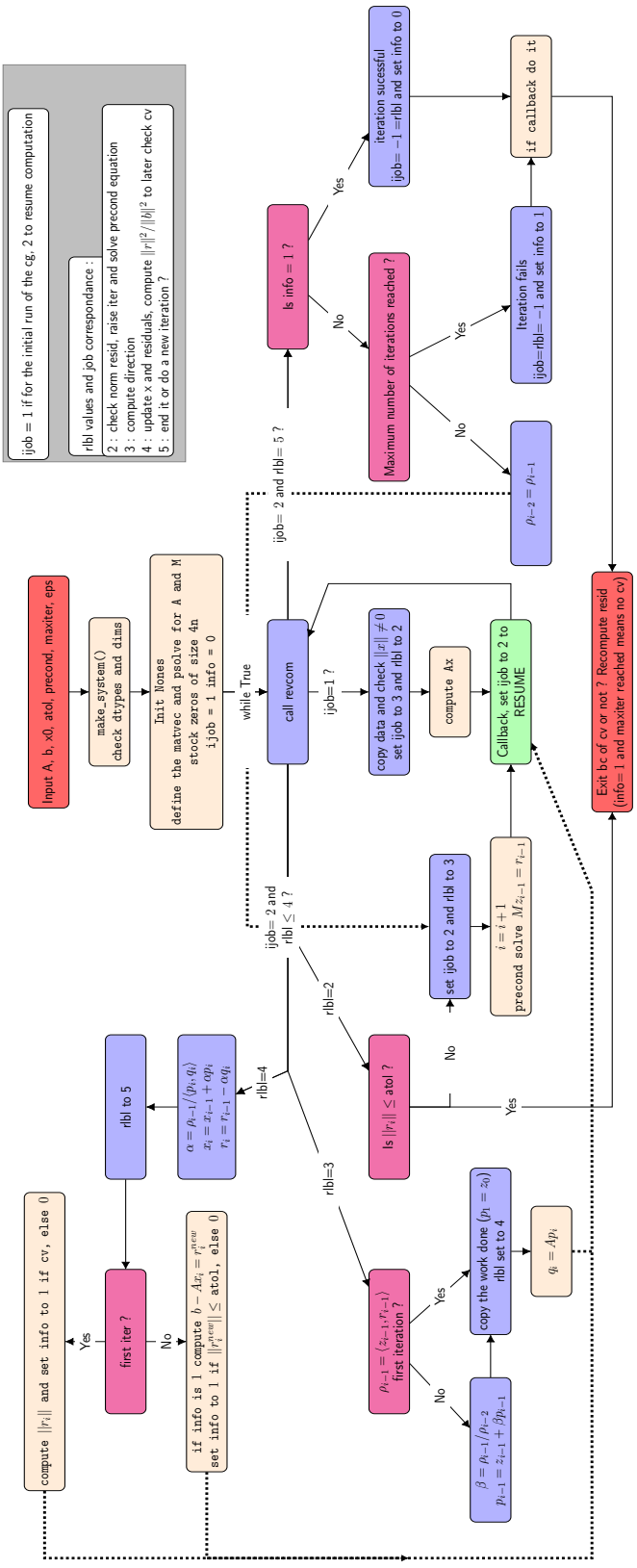
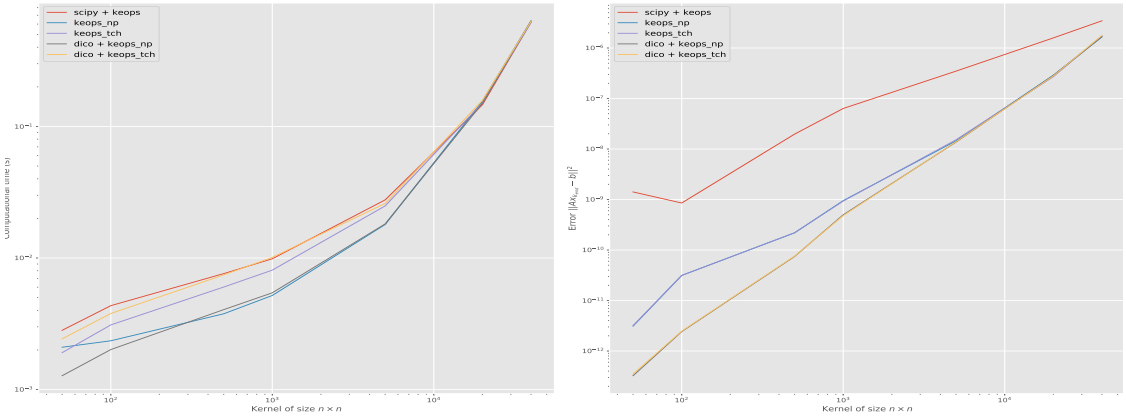


Figure 2: `Scipy`'s conjugate gradient algorithm using the REVCOM strategy with a main function in Python and a subroutine in Fortran.

`Scipy`'s limitations: The n points $x_i \in \mathbb{R}$ are defined such that $x_i = \frac{i}{n}$. The parameters α and γ were respectively set to 2 and $\frac{0.5}{0.01^2}$ hereinafter.



(a) Time comparison

(b) Residuals comparison

Figure 3: Time and residual benchmarking of `Scipy`'s conjugate gradient against the ones implemented in `KeOps` with an increasing size n of the Kernel.

As we can see on Figure (3b), `KeOps` implementations have a smaller error than `Scipy`'s with default parameters. Figure (3a) shows us that `Scipy` is also slower than the other versions. And for larger number of points, `KeOps` can still compute a result (as we shall see) but `Scipy`'s conjugate gradient simply burst as for its computation time.

Going further with `PyKeOps`: Because of `Scipy`'s limitations, we only use `PyKeOps`.

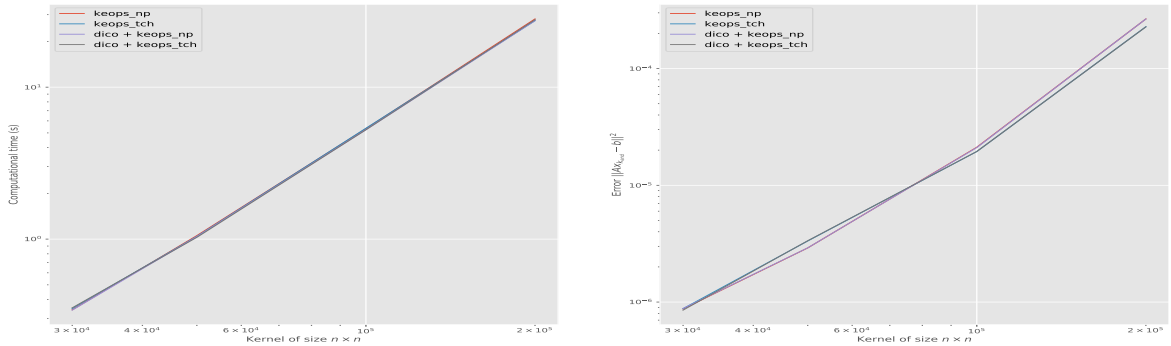
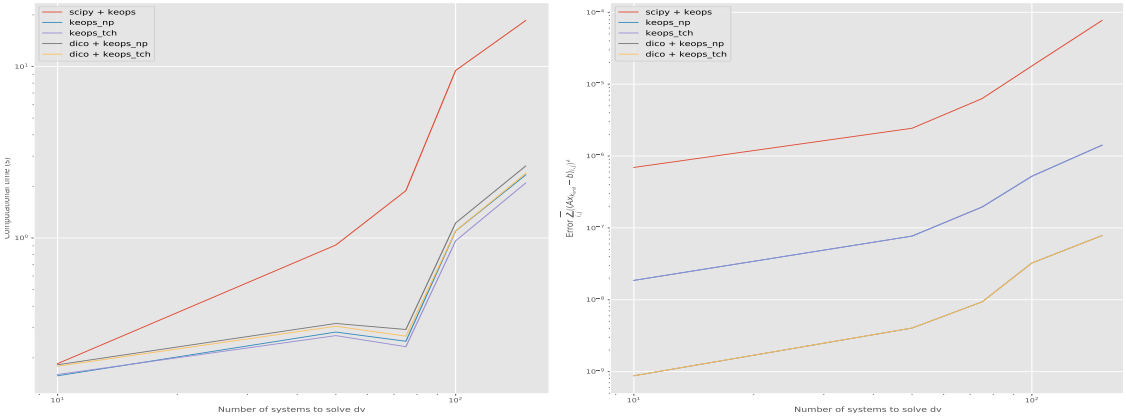


Figure 4: Time and residual benchmarking of `Scipy`'s conjugate gradient against the ones implemented in `KeOps` with an increasing size n of the Kernel.

As we can see on Figure (4), for large kernels, there is little to no difference between the newly implemented and the older algorithm for the conjugate gradient for both the time-consumption and the obtained precision.

Solving multiple systems at the same time: The $n = 50000$ points $x_i \in \mathbb{R}^4$ are uniformly distributed between 0 and 1. We change the RHS b such that $b \in \mathbb{R}^{n \times Dv}$ where Dv is the number of systems to solve. Because `Scipy`'s conjugate gradient only handles 1-dimensional RHS, and our solution needs to be a matrix of size (n, Dv), we need to create another matrix in which we store each solved system... which drastically increases the computation time.



(a) Time comparison (b) Residuals comparison

Figure 5: Time and residual benchmarking of `Scipy`'s conjugate gradient against the ones implemented in `KeOps` with an increasing size Dv of the number of systems to solve.

We can see on Figure (5) that for almost the same amount of time as the older version of PyKeOeps' algorithm, the Pythonized version using dictionaries has a better precision than all other algorithms.

1.2 Condition number

The `KeOps` new version of the conjugate gradient allows the user to check if the condition number of $A = (K + \alpha \mathrm{Id})$ is large. To do so, we use both the power iteration and the inverse power iteration methods. These algorithm can indeed be used knowing only the linear operator and not the dense matrix (which is useful for sparse matrix and in this case `KeOps` kernels).

The condition number of a symmetric matrix A of real and positive eigenvalues $(\lambda_i)_{i=1}^n$ (with the 2-norm) is defined as:

$$\mathrm{cond}(A) = \kappa = \|A\|\|A^{-1}\| = \frac{\lambda_{max}}{\lambda_{\min}}. \tag{1.4}$$

We consider κ to be large if it exceeds 500. Now, let's order the eigenvalues in increasing order from one to n with $\lambda_1 = \lambda_{min}$ and $\lambda_n = \lambda_{max}$. For $i, j \in [\![1, n]\!]$, we have $\lambda_{min} \leq \lambda_i \leq \lambda_j \leq \lambda_{max}$. From that we get:

$$\forall i, j \in \{1, \dots, n\}, \ \frac{\lambda_j}{\lambda_i} \leq \frac{\lambda_j}{\lambda_1} \leq \frac{\lambda_n}{\lambda_1}. \tag{1.5}$$

We obtain a lower bound for the condition number, we only need to estimate two eigenvalues and if their quotient is over 500, κ will be large. Let's consider the following algorithm.

Algorithm 2 Check if condition number is too large

Require: linear operator A, maximum number of iterations, conjugate gradient method CG
  Estimate λ_n using the power iterations
  $\delta \leftarrow \frac{\lambda_n}{500}$
  while maximum number of iterations not reached **do**
    Perform an iteration of the inverse power method using CG
    Use Rayleigh's quotient to get an iterated eigenvalue μ_k
    if $|\mu_k| \leq \delta$ **then**
      Condition number is over the limit
    else {keep estimating λ_1}
      continue
    end if
  end while

In practice, the numerical accuracy and a cluster of eigenvalues close to λ_1 can lead to difficulties for the inverse power method to reach λ_1 [11]. But thanks to equation (1.5), because we already estimated λ_n, we only need any λ_i for which the quotient $\frac{\lambda_n}{\lambda_i}$ is over 500 to say that the matrix is ill-conditioned.

However, can we be certain that if $\mu_k \leq \delta$, then there won't be an index $l > k$ such that $\mu_l > \delta$ *ie* does the sequence $(\mu_k)_k$ decreases using the inverse power method?

The answer to this question is yes, thanks to the convergence rate of the power method. Indeed, the power method states that for a random vector u_0, if we iterate the relations

$$u_k = \frac{Au_{k-1}}{\|Au_{k-1}\|} \text{ and } \lambda^{(k)} = \frac{u_k^\top Au_k}{u_k^\top u_k},$$

then $\lambda^{(k)} \to \lambda_n$ and $u_k \to u_{\lambda_n}$ the associated eigenvector. The rate of convergence is geometric of ratio $\frac{\lambda_{n-1}}{\lambda_n}$ [5], meaning that

$$\|u_{\lambda_n} - u_k\| = \mathcal{O}\left(\left|\frac{\lambda_{n-1}}{\lambda_n}\right|^k\right). \tag{1.6}$$

The inverse power method uses the same principle but with the relation:

$$u_k = \frac{(A - \mu \mathrm{Id}\,)^{-1} u_{k-1}}{\|(A - \mu \mathrm{Id}\,)^{-1} u_{k-1}\|}.$$

The inverse power method used converges to the eigenvector of the smallest magnitude choosing $\mu = 0$. It is the same as performing the power method on the matrix A^{-1} of eigenvalues $(\lambda_i^{-1})_{i=1}^n$. So using equation (1.6), the convergence rate is:

$$\|u_{\lambda_1} - u_k\| = \mathcal{O}\left(\left|\frac{\lambda_1}{\lambda_2}\right|^k\right).$$

Thus we can use algorithm (2) to check the condition number of our operator.

2 Optimal transport with the `GeomLoss` library

The `KeOps` library can be used in many fields. One of them: the optimal transport, has led to the creation of another package, `GeomLoss` [4]. Before going into this library, we first need to introduce some notions around the optimal transport.

2.1 Requirements about the Optimal Transport

Notations. Let $\mathcal{X}$ be a feature set with $\{x_i\}_i \subset \mathcal{X}$, $f, g \in \mathcal{X}$ measurable functions for which we note $f(x_i) = f_i$ and $g(x_j) = g_j$. Let's also take α, β two measures on $\mathcal{X}$. Then:

- the tensor sum $f \oplus g$ is defined as $f \oplus g = f + g^\top$ $ie (f \oplus g)_{ij} = f_i + g_j$,
- the tensor product $\alpha \otimes \beta$ is defined as $\alpha \otimes \beta = \alpha\beta^\top$ $ie (\alpha \otimes \beta)_{ij} = \alpha_i \beta_j$,
- if $\alpha = \sum_i \alpha_i \delta_{x_i}$ then $\langle \alpha, f \rangle = \sum_i \alpha_i f(x_i)$ is the expected value $\mathbb{E}_{X \sim \alpha}[f(X)]$.

The optimal transport problem has been studied for a long time. Monge (18th century) wanted to fill a hole with a pile a sand and minimize the loss when knowing the cost to transport each grain. Since then, multiple applications saw the day of light in economics, biology...

If we follow this idea, each grain of sand can be represented with its position within the ambient space $\mathcal{X} = \mathbb{R}^3$, and in a perfect world each grain has the exact same weight as its peers. Translating this situation in $\mathcal{X} = \mathbb{R}^1$ with three $1-$dimensional same grains of sand to fill three holes can be represented as in Figure (6) below.

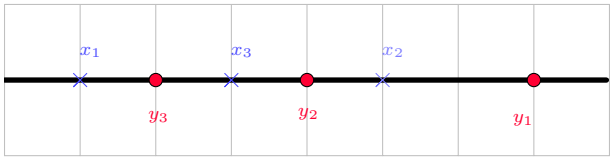


Figure 6: Two measures α (red) and β (blue): how can we minimize the cost to send α onto β?

The measure α is just a (uniformly-)weighted cloud of $n = 3$ points:

$$\alpha = \sum_{i=1}^{n} \alpha_i \delta_{x_i}, \quad \text{with} \quad \delta_a : A \subset \mathcal{X} \to \{0, 1\} \text{ such that } \delta_a = \mathbb{1}(a \in A).$$

And the same for β with the y_i's. Let's use the cost function

$$C(x, y) = \frac{1}{2n} \sum_{i=1}^{n} |x_i - y_i|^2 = \frac{1}{2} \|x - y\|^2. \tag{2.1}$$

If one chooses to trust the labels naively, $ie (x_1, x_2, x_3) \to (y_1, y_2, y_3)$:

$$C((x_1, x_2, x_3), (y_1, y_2, y_3)) = \frac{1}{2 \times 3} \left(6^2 + 1^2 + 1^2\right) = 6.33\ldots$$

Whereas someone who thinks about it see that we can chose to do $(x_1, x_2, x_3) \to (y_3, y_2, y_1)$. We can relabel (y_i) with the permutation σ to get $\sigma(y) = (y_{\sigma(i)})$. We finally obtain:

$$C(x, \sigma(y)) = \frac{1}{2 \times 3} \left(1^2 + 1^2 + 2^2\right) = 1 < C(x, y).$$

We define the optimal transport loss in this situation by:

$$\mathrm{OT}(\alpha, \beta) = \min_{\sigma : [\![1,n]\!] \to [\![1,n]\!]} C(x, \sigma(y)) = \min_{\sigma : [\![1,n]\!] \to [\![1,n]\!]} \frac{1}{2} \|x - \sigma(y)\|^2. \tag{2.2}$$

The loss function used at equation (2.2) is called the Wasserstein-2 function.

Soften the hypothesis. As of now, we worked with two point clouds with the same number of points and the same weights for each point. Almost two centuries later, Kantorovich generalized Monge's problem for two weighted point clouds with the same mass *ie*:

$$\alpha = \sum_{i=1}^{n} \alpha_i \delta_{x_i}, \ \beta = \sum_{i=1}^{m} \beta_i \delta_{y_i} \quad \text{with} \quad \sum_{i=1}^{n} \alpha_i = \sum_{i=1}^{m} \beta_i.$$

We can of course set the mass to 1 to work with probability distributions by normalizing each measure. Using the same cost function (2.1) as before, we can define the Wasserstein-2 distance as:

$$\mathrm{OT}(\alpha, \beta) = \min_{\pi_{ij} \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{m} \pi_{ij} C(x_i, y_j), \tag{2.3}$$

where $\Pi \subset \mathbb{R}^{n \times m}$ is such that the marginals of π_{ij} are α and β and $\pi_{ij} \geq 0$ for all i and j. The term $\pi_{i,j}$ can roughly be understood as "the portion of α_i that is sent to β_j". With this visualization, it is indeed logical to have such assumptions over the marginals as we can see in the Figure (7). This generalizes the combinatorial problem of Monge in higher dimensions with different number of points.

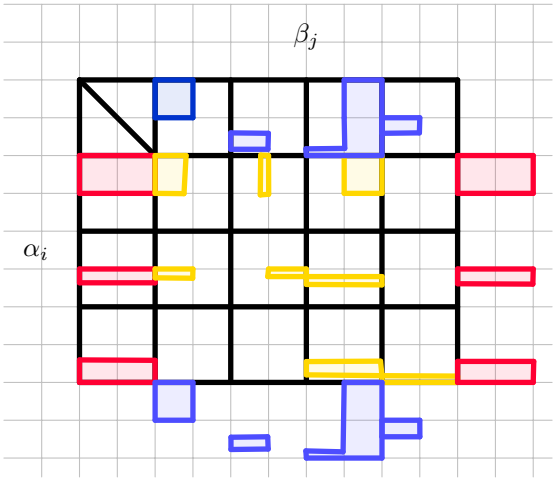


Figure 7: Visualization of Kantorovich problem where there isn't the same number of points in the two point clouds but the measures have the same mass.

We can even use the notations defined to consider an even larger problem:

$$\mathrm{OT}(\alpha, \beta) = \min_{\pi \in \mathcal{M}^+ \times \mathcal{M}^+} \langle \pi, C \rangle, \tag{2.4}$$

with $\pi \geq 0$ of marginals α and β. This indeed includes continuous measures. Fixing these marginals assures us that π has a density against $\alpha \otimes \beta$ in the discrete case.

Dual formulation of Kantorovich problem. The primal writing of the problem still needs us to compute large matrices and consider multi-dimensional sorting algorithm: and we all know how a simple combinatorial problem can require an enormous amount of time to be solved (if it can be solved in a lifetime).

The dual Kantorovich problem can be expressed as:

$$\mathrm{OT}(\alpha,\beta) = \max_{f_i,g_j} \langle \alpha, f\rangle + \langle \beta, g\rangle \quad \text{such that} \quad (f \oplus g)_{ij} \leq C(x_i, y_j). \tag{2.5}$$

The couple of optimal dual potentials is not unique (but it is up to an additive constant).

We can interpret the primal-dual formulas with two sides of a scenario going back to the sand issue with an economical point of view as it is usually done. Let's consider two people who need to transport a large mass of sand into a hole. They have (on the top of their head) two solutions:

- the primal: they can try to transport the sand strategically and **minimize** the cost needed for the task,
- the dual: they can hire a transporter.

But because they're a little stingy, the would only chose the dual solution if the cost of moving a portion of sand and the cost of delivering it to the hole is at most the equivalent cost they would have paid (their motivation may or may not be included). The transporter also has a motive, to be profitable they need to **maximize** their salary *ie*the cost to move a unit of sand and the cost to fill the hole (considering the whole mass at disposal).

Unfortunately (for a decision point of view), Kantorovich with the fundamental theorem of linear programming states that calling a transporter or doing it yourself will cost the same amount.

A little parenthesis for notions of linear programming. The fundamental theorem of linear programming states that solving

$$\min_{\substack{x\in\Sigma \\ \Sigma=\{x\in\mathcal{X},\ Ax\leq b\} \text{ bounded polyhedron}}} \langle c, x\rangle, \tag{2.6}$$

induces that the optimal solution is for $x^\star$ located on a vertex (uniqueness) or on a face (non-uniqueness) of Σ.

We can then use the relation between the primal and dual problem to solve it using the complementary slackness method [8]. Let's look at equations (2.4) and (2.5). We can see that the number of constraints in the dual equals the number of variables involved in the primal and reverse. So the variables are **complementary** to the constraints. The slackness added means that if a variable is positive in one problem, then the constraint in the other must be binding *ie*there is equality at the optimal point. On the other side, if the constraint is not binding, then the variable must be zero. The slackness is complementary in the sense that it can't be both in the primal variables (resp constraint) and dual constraint (resp variable).

Back to solving Kantorovich dual problem. The complementary slackness rules leads us to find an optimal pair (f, g) for the dual problem with $\mathcal{X} = \mathbb{R}^d$:

$$\begin{aligned} \forall x \in \mathcal{X},\ f(x) &= \min_{j=1}^{m} \left[C(x, y_j) - g(y_j)\right], \\ \forall y \in \mathcal{X},\ g(y) &= \min_{i=1}^{n} \left[C(x_i, y) - f(x_i)\right]. \end{aligned} \tag{2.7}$$

So instead of computing an $n \times m$ matrix, we only need to find one member of the optimal couple (f, g), let's take f, to deduce the other. So most of the information can be contained only within $f \in \mathbb{R}^n$.

The differentiability issue. The main problem of the equations (2.7) is that the translation into an algorithm won't necessarily return the optimal pairing, mainly because the min function is not differentiable everywhere. So we need to transform the min update in (2.7) into a smoothed update called SoftMin and a blur factor $\varepsilon > 0$ which will induce a relaxed problem that will lead to an ε-approximation of the optimal pairing.

We define the SoftMax of a continuous function φ for a measure α as:

$$\max_{\alpha}{}_{\varepsilon}[\varphi(x)] = \varepsilon \log\langle \alpha, \exp\left(\frac{\varphi}{\varepsilon}\right)\rangle.$$

And SoftMin is defined as $-\max_{\alpha}{}_{\varepsilon}[-\varphi(x)]$. When $\varepsilon \to 0$, we retrieve the min (or max).We can now replace it into (2.7) and get the Sinkhorn algorithm as below (in the *log*-domain for the numerical stability). Note that using `PyKeOps` LazyTensors, the `logsumexp` operation can be executed efficiently.

Algorithm 3 Sinkhorn (logsumexp) algorithm

Require: α, β discrete measures with same mass, $\varepsilon > 0$, cost function C.
  $f_i,\ g_j \leftarrow 0_n,\ 0_m$
  while convergence is not reached (up to a threshold) **do**
    $f_i \leftarrow -\varepsilon \log \sum_{j=1}^{m} \beta_j \exp\left(\varepsilon^{-1}\left[g_j - C(x_i, y_j)\right]\right)$
    $g_j \leftarrow -\varepsilon \log \sum_{i=1}^{n} \alpha_i \exp\left(\varepsilon^{-1}\left[f_i - C(x_i, y_j)\right]\right)$
  end while

Entropic regularization. The question that now comes up is: Is algorithm (3) really better than just the implementation of the equations (2.7)? And thankfully, it is. To see roughly why, we need to look at another optimal transport problem: Schrodinger's. But first, let's recap some information theory.

Let P be an $n \times m$ matrix with all positive coefficients and $\sum_{i,j} P_{ij} = 1$, $\sum_i P_{ij} = P_i$ and $\sum_j P_{ij} = P_j$, idem for Q, then:

- the entropy of P is $H(P) = -\sum_{i,j} P_{ij} \log P_{ij}$,
- the Kullback-Leibler divergence (or relative entropy) for P with respect to Q is the asymmetric expression $\mathrm{KL}(P, Q) = \sum_{i,j} P_{ij} \log \frac{P_{ij}}{Q_{ij}}$,
- the mutual information of a joint distribution, is $I(P_i, Q_j) = \mathrm{KL}(\,(P_i, Q_j),\, P_i \otimes Q_j)$.
- given two marginals P_i and P_j, the matrix with the maximal entropy is $(P_i \otimes P_j)$, in particular we have $\mathrm{KL}(P, (P_i \otimes P_j)) = H(P_i) + H(P_j) - H(P)$

So, using this information, if we would want to penalize the optimal transport entropy, one would simply add a $-\varepsilon H(\pi)$ on the primal. Yes, but remember that the marginals are predetermined. So $H(\alpha) + H(\beta)$ is just a constant. So up to an additive constant, we can consider the equivalent Schrodinger primal problem as follow:

$$\mathrm{OT}_{\varepsilon}(\alpha, \beta) = \min_{\substack{\pi \in \mathcal{M}^+ \times \mathcal{M}^+ \\ \pi_{ij} \geq 0 \\ \alpha, \beta \text{ marginals of } \pi}} \langle \pi, C\rangle + \varepsilon \mathrm{KL}(\pi, \alpha \otimes \beta). \tag{2.8}$$

Keep in mind for later that if $\alpha = \beta$, then π is the identity and $\mathrm{OT}_\varepsilon(\alpha, \alpha) \neq 0$. It's dual representation is

$$\mathrm{OT}\varepsilon(\alpha, \beta) = \max_{f,g} \langle \alpha, f \rangle + \langle \beta, g \rangle \quad \text{with} \quad \max_{\alpha \otimes \beta} {}_\varepsilon[f \oplus g - C] \leq 0. \tag{2.9}$$

Differentiating the dual problem with respect to f and g to find the optimum leads us to the exact equations used in Sinkhorn's algorithm (3) and it is known that Schrodinger dual problem converges to an ε-optimal pairing.

De-biased Sinkhorn Algorithm. Now, we have a working algorithm that approximates an optimal dual pairing, so a way to compute OT_ε, an approximation of the optimal transport loss. However, does it behave like we would it to ? Unfortunately, the answer is not always...

Let's take for example the quadratic cost $C(x, y) = \|x - y\|$, a blur of $\varepsilon = 0.05$ and observe thanks to GeomLoss the transport of a source point cloud onto a target one using a gradient descent to update the position of the source.

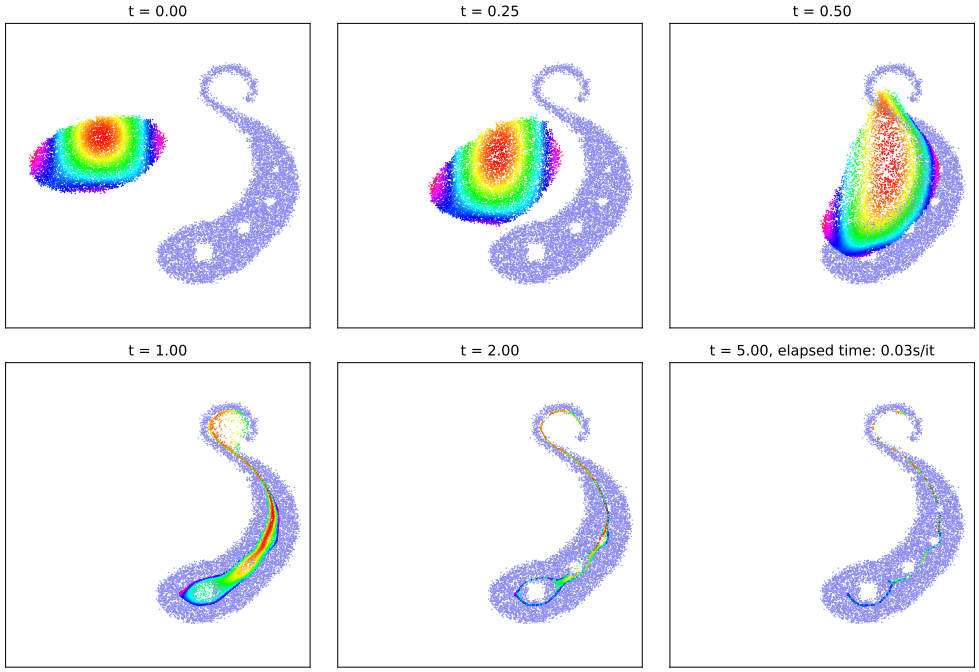


Figure 8: Steps for the transport using the loss OT_ε of a source shape onto a target using Sinkhorn's algorithm with the euclidean norm as cost function with a blur $\varepsilon = 0.05$ close to zero.

As we can see on Figure (8), our shape collapses. How can we retrieve a good looking transportation using the work done before? A first idea it to find an expression with OT_ε that actually is void when $\alpha = \beta$ for all $\varepsilon > 0$.

Introduced by Ramdas [6], the de-biased Sinkhorn divergence is defined by:

$$S_\varepsilon(\alpha, \beta) = \mathrm{OT}_\varepsilon(\alpha, \beta) - \frac{1}{2}\mathrm{OT}_\varepsilon(\alpha, \alpha) - \frac{1}{2}\mathrm{OT}_\varepsilon(\beta, \beta) \tag{2.10}$$

And this expression has (amongst others) two good properties:

$$\begin{aligned} S_\varepsilon(\alpha,\beta) &\overset{\varepsilon\to 0}{\longrightarrow} \mathrm{OT}(\alpha,\beta) \\ &\overset{\varepsilon\to\infty}{\longrightarrow} \langle \alpha\otimes\beta, C\rangle - 0.5\langle \alpha\otimes\alpha, C\rangle - 0.5\langle \beta\otimes\beta, C\rangle, \end{aligned}$$

and using the fact that $\alpha\otimes\beta = \alpha\beta^\top$, we can rewrite the latest limit as:

$$\begin{aligned} S_\varepsilon(\alpha,\beta) \overset{\varepsilon\to\infty}{\longrightarrow} \langle \alpha, C\star\beta\rangle - 0.5\langle \alpha, C\star\alpha\rangle - 0.5\langle \beta, C\star\beta\rangle &= \frac{1}{2}\langle \alpha-\beta, -C\star(\alpha-\beta)\rangle \\ &\text{"} = \frac{1}{2}\|\alpha-\beta\|^2_{-C}\text{"}. \end{aligned}$$

For the same value of the blur and distance as in Figure (8), we now get the following transportation in Figure (9) below, which is clearly better.

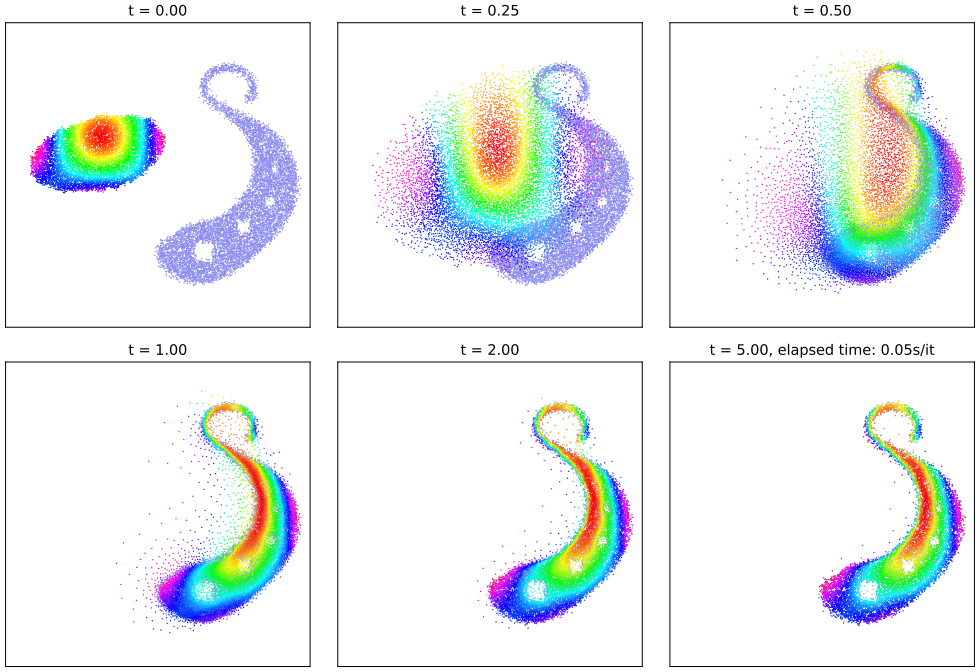


Figure 9: Steps for the transport using the loss S_ε of a source shape onto a target using Sinkhorn's algorithm with the euclidean norm as cost function with a blur $\varepsilon = 0.05$ close to zero.

2.2 Compute loss with another cost function.

Using GeomLoss to compute the loss is quite easy with the following code:

```
from geomloss import SamplesLoss
loss = SamplesLoss("sinkhorn", p=1, blur=.1, debias=True) # p=1 for Wasserstein-1
# let's not precise the weights to indicate a uniform distribution
loss(source, target)  # target of shape (M,D) and source (N,D), D>=1
```

Note that one can precise the argument `cost` and chose another cost function of its own and even define one with a `KeOps` formula. So, for which cost function can we use the sinkhorn's de-biased algorithm (or it's variants, see [3, p.119-123] for a symmetric-version, ε-scaling in order to increase speed and a multiscale one). An important theorem in link with the de-biased Sinkhorn's divergence is:

Theorem 2.1. Let $\mathcal{X}$ be a compact metric space with a Lispchitz cost function C such that the Gibbs kernel $k_\varepsilon(x,y) = \exp(-C(x,y)/\varepsilon)$ is positive and universal for all $\varepsilon > 0$. Then S_ε is a symmetric definite positive smooth loss function, convex, such that for two probability measures α, β we have

$$\begin{aligned} 0 = S_\varepsilon(\alpha,\alpha) \leq S_\varepsilon(\alpha,\beta), \\ \alpha = \beta \Longleftrightarrow S_\varepsilon(\alpha,\beta) = 0, \\ \forall f \in \mathcal{C}(\mathcal{X})\ \langle \alpha_n, f\rangle \to \langle \alpha, f\rangle \Longleftrightarrow S_\varepsilon(\alpha_n,\alpha) \to 0. \end{aligned}$$

Let's begin by defining each term. Let $(\mathcal{X}, d)$ be the feature space and $K_{ij} = k(x_i, x_j)$ be a kernel on $\mathcal{X}$, then

- $C(x,y)$ is Lipschitz with respect to both its variables if

$$\exists K > 0,\ \forall x, y, y' \in \mathcal{X}, |C(x,y) - C(x,y')| \leq K d(y,y'),$$

- K_{ij} is said (semi-)definite positive or valid if

$$\forall m \in \mathbb{N},\ \forall \{x_i\}_{i=1}^m \in \mathcal{X}^m,\ \forall c_i \in \mathbb{R},\ \sum_{i,j} c_i c_j k(x_i, x_j) \geq 0,$$

- k is said univeral if $k(x,\cdot)$ is continuous on $\mathcal{X}$ and the space of all functions spanned by k is dense in $\mathcal{X}$ for the norm $\|\cdot\|_\infty$.

Key propositions to use the theorem. Fortunately for us, researchers already paved the way and we have some other results [9] to prove that a kernel is universal and definite positive. Let's tackle the definite positive one first: It is easy to show that the kernel $k(x,y) = \langle x, y\rangle$ is valid. From there, with a few rules, we can create more complicated kernels.

- any positive linear combination of two valid kernels is still valid,
- the product of two valid kernels is still valid,
- if it exists, the limit of a sequence of valid kernels is still valid,
- the normalization of a valid kernel is still valid.

From there, we can show that a polynomial, exponential or Gaussian kernel is a valid kernel. Now let's consider ways to prove that a kernel is universal. There's three big results to do so:

- if k is universal then the normalized kernel $\tilde{k}(x,y) = \frac{k(x,y)}{\sqrt{k(x,x)k(y,y}}$ is also universal,
- if $\mathcal{X} = \mathbb{R}^d$ and $k(x,y) = k(\langle x, y\rangle)$ then k is universal if and only if

$$k(z) = \sum_{i \in \mathbb{N}} a_i z^i \quad \text{with } a_i > 0\ \forall i \in \mathbb{N},$$

- let $\mathcal{C}_0(\mathcal{X} = \mathbb{R}^d)$ be the space of functions that vanish at infinity, then k is said c_0-universal if and only if the support of its Fourier transform is $\mathbb{R}^d$.

Example with different cost functions. Now that we have some tools to work with, let's use them to see if a cost function $C(x, y)$ usable for theorem (2.1).

- Let's begin with a classic, $C(x, y) = \|x-y\|_2^2$. It's well known that it is a Lipschitz function. Let's check the properties for the Gibbs kernel $k(x, y) = \exp\left(-\frac{\|x-y\|^2}{\varepsilon}\right)$:

  i. $k(x, y)$ is a valid kernel. Let's note $\varepsilon^{-1} = \gamma$ and $k'(x, y) = e^{2\gamma^2\langle x,y\rangle}$. Then, k' is a valid kernel as the exponential (limit of a sequence) of the linear kernel multiplied by a scalar. Thus, k is a kernel because

  $$\frac{k'(x, y)}{\sqrt{k'(x, x)k'(y, y)}} = e^{-\|\gamma x\|^2} e^{-\|\gamma y\|^2} e^{2\gamma\langle x,y\rangle} = e^{-\gamma\|x-y\|^2} = k(x, y).$$

  ii. Using the same auxiliary kernel k', we can write

  $$k'(x, y) = \sum_{n\geq 0} \frac{(2\gamma^2)^n}{n!}\langle x, y\rangle^n,$$

  and then use the fact that the coefficient in this development are strictly positive for all $n \in \mathbb{N}$, thus k' is universal and its normalization k is too.

- Let's take one minus the cosine similarity (to have a positive cost) $ieC(x, y) = 1 - \frac{\langle x,y\rangle}{\|x\|\|y\|}$, then the Gibbs kernel induced is

  $$k(x, y) = \exp\left(-\left[1 - \frac{\langle x, y\rangle}{\|x\|\|y\|}\right]\frac{1}{\varepsilon}\right) = \exp\left(-\frac{1}{\varepsilon}\right)\exp\left(\frac{\langle x, y\rangle}{\|x\|\|y\|}\frac{1}{\varepsilon}\right)$$

  i. validity: Let's note $k_1(x, y) = -\frac{\langle x,y\rangle}{\|x\|\|y\|\varepsilon}$. Then k_1 is valid as the linear kernel normalized and multiplied by a positive constant. Then $e^{k_1(x,y)}$ is still valid as a defined limit of positive linear combination and multiplying it by a positive constant doesn't change it's validity.

  ii. universality: Let's rewrite $k(x, y)$,

  $$\begin{aligned} e^{-\left(1-\frac{\langle x,y\rangle}{\|x\|\|y\|}\right)/\varepsilon} &= e^{-\frac{1}{\varepsilon}} e^{\frac{\langle x,y\rangle}{\|x\|\|y\|}/\varepsilon} \\ &= e^{-\frac{1}{\varepsilon}} \sum_{n\geq 0} \frac{1}{n!\varepsilon^n}\left(\frac{\langle x, y\rangle}{\|x\|\|y\|}\right)^n \\ &= \sum_{n\geq 0} \underbrace{\frac{e^{-1/\varepsilon}}{n!\varepsilon^n}}_{a_n}\left(\frac{\langle x, y\rangle}{\|x\|\|y\|}\right)^n \end{aligned}$$

  Because $a_n > 0$ for all $n \in \mathbb{N}$, k is universal.

  iii. Not Lispschitz: To show the Lipschitzian behavior, we would need to prove that

  $$\exists\kappa > 0\ \forall t, x, y \in \mathcal{X}(= \mathbb{R}^2),\ |C(t, x) - C(t, y)| \leq \kappa\|x - y\|.$$

  Let's develop the left side:

  $$\begin{aligned} \left|1 - \frac{\langle x, t\rangle}{\|x\|\|t\|} - 1 + \frac{\langle y, t\rangle}{\|y\|\|t\|}\right| &= \left|\frac{\langle y, t\rangle}{\|y\|\|t\|} - \frac{\langle x, t\rangle}{\|x\|\|t\|}\right| \\ &= \left|\langle\frac{y}{\|y\|} - \frac{x}{\|x\|}, \frac{t}{\|t\|}\rangle\right|. \end{aligned}$$

We can represent the problematic situation with $\mathcal{X}_r = \{x \in \mathbb{R}^2,\ \|x\| \leq r\}$, on the following diagram.

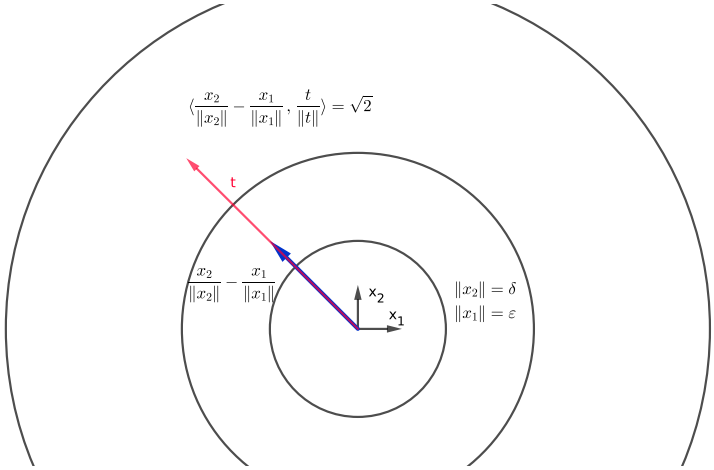


Figure 10: Visualization of an example around zero that indicates that $C(x, y)$ can't be Lispchitz.

$$\text{Thus } \left|\langle \frac{y}{\|y\|} - \frac{x}{\|x\|}, \frac{t}{\|t\|} \rangle\right| = \sqrt{2} \text{ for } x = (\varepsilon, 0), y = (0, \delta) \text{ and } t = (-2, 2),$$

$$\nexists \kappa > 0,\ \forall \varepsilon, \delta > 0\ \sqrt{2} \leq \kappa \sqrt{\varepsilon^2 + \delta^2}.$$

In the GeomLoss package documentation, different loss and cost are used to understand how to define each parameter to fit to the best the two point clouds. Amongst others loss functions there are the energy distance, Wasserstein, Gaussian loss,... We strongly suggest to check it and the other possibilities at the address below.

`www.kernel-operations.io/geomloss/_auto_examples/comparisons/plot_gradient_flows_2D.html#sphx-glr-auto-examples-comparisons-plot-gradient-flows-2d-py`

2.3 The batch-computation of the loss.

As of now, GeomLoss is compatible with batch computation at the cost that `PyKeOps` is the only backend that can be used if the data has labels. It's still a fast algorithm, but with a quadratic cost in memory in this case. Each batch is represented as a tile in a tensor as input.

One of the first thing that appears when looking for optimal transport problems that we talked about is the Kantorovich problem (2.4). One of its main features was that it removed the hypothesis on the same number of points needed to only keep the mass of the measures. However, considering that the batches are tiles of a same tensor, a dimension issue comes up as soon as we don't have the same sample sizes.

Let's take a simple situation: one source X of size $(N, 2)$ and two targets Y, Z of sizes $(M, 2)$ and $(P, 2)$ with $M < P$. To store the targets in a same tensor, we need $\tilde{Y}$, the padded version of Y such that $\tilde{y}_j = y_j$ for $j \in [\![1, M]\!]$ and $\tilde{y} = 0_{\mathbb{R}^2}$ for $j \in [\![M + 1, P]\!]$. However, doing so without

precising that the measure shouldn't be a uniform distribution (we don't want to transport the zeros!) will return a wrong loss. The answer to this problem is to pad the measure as in Figure (11). So the user needs to give as input X with its weights α, the padded tensor containing Y and Z and the padded weights for both of the targets (into one array).

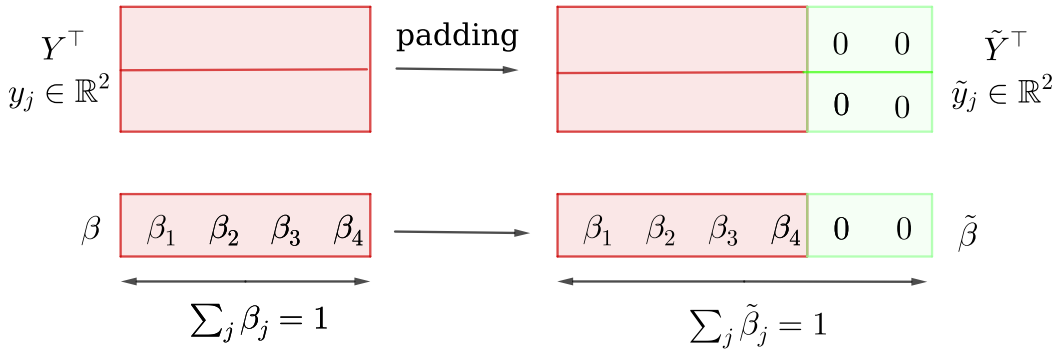


Figure 11: Visualization of the padding needed for both the sample and the measure to preserve the same mass

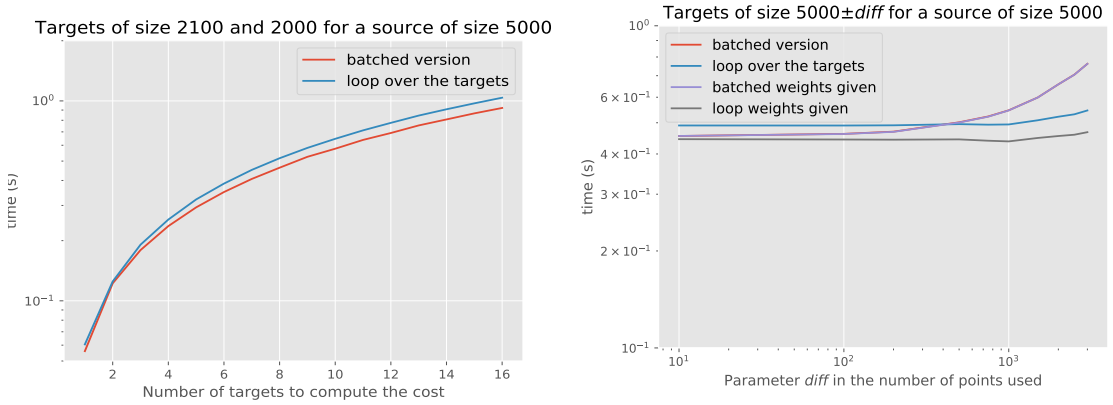
The fact is, it can become very tedious when working with more than two batches, considering the precautions to take in order to make all the paddings for the samples and the weights. For only two samples we need all the code below.

```
def get_weights(sample): # uniform distribution
    if sample.dim() == 2:
        N = sample.shape[0]
        return torch.ones(N).type_as(sample) / N
    elif sample.dim() == 3:
        B, N, _ = sample.shape
        return torch.ones(B,N).type_as(sample) / N

X_ = X.unsqueeze(0).repeat(2,1,1) # 2 targets means 2 tiles for the source

beta = get_weights(Y)
gamma = get_weights(Z)
padding = max(Y.shape[0], Z.shape[0]) # no need to pad too much

target_ = torch.stack([ # 2 targets as 1 tensor
    F.pad(Y, (0, 0, 0, padding - Y.shape[0])),
    F.pad(Z, (0, 0, 0, padding - Z.shape[0]))
    ])

alpha_ = get_weights(X_)
weights_ = torch.stack([ # padded weights
    F.pad(beta, (0, padding - Y.shape[0])),
    F.pad(gamma, (0, padding - Z.shape[0]))
])
```

```
loss(alpha_, X_, weights_, target_) # finally compute the loss
```

But we can wrap everything up into a black box for the users and let them use either the previous batch mode, or give the batch as a list of two dimensional tensors (or three as long as there is only one tile) with a list of weights, or without. In the latter case, we consider a uniformed distribution over the samples as usual. Now, no need to do all this, we can simply use this feature without ever having to consider the zeros-padding.

```
loss(X, [Y, Z]) # batch version without weights (dim Y = dim Z)
loss(alpha_, X, [beta, gamma], [y_j, z_j]) # with weights, we can use X or X_ here
```



(a) Time comparison when adding targets with a tensorized backend.

(b) Time comparison for two targets with an increasing difference in sample size

Figure 12: Time comparison for the loop against the batch strategy.

One could argue that the user could just loop over the targets one-by-one and simply abandon the idea of using batches because where there's a black-box, there's very often a catch. Here, the catch is quite logical: the padding is hidden, not totally removed. Of course if one batch is a single point and the other is made of thousand of points the computation time will be impacted. However, as we can see in Figure (12), as long as the padding doesn't exceed 3000, there isn't a real difference in time processing. In Figure (12a) we even see a slight gain. So considering batches over a loop have the advantage to be both user-friendly and not more time-consuming.

Conclusion

Using the `KeOps` library is a fast and efficient way to perform multiple types of calculations thanks to the GPU. It can be used to find the solution to a regularized system with the iterative method that is the conjugate gradient. This way, it outperforms the standard library used to solve systems that is `Scipy`'s in both time and precision but also in the size of the system itself. Using iterative methods like the power iterations and its inverse, we can also allow users to make some sanity checks without having to compute exactly the whole spectrum or store the entire matrix ever. All of these features are compatible with `PyKeOps` LazyTensors and thus be used on easily-created kernels and especially without the use of the `kernel_product` method.

One of its many fields of applications is the optimal transport. The package `GeomLoss` uses `KeOps`'s power to transport large weighted point clouds. The user can even customize its own cost function in order to compute the loss of the transport as long as they meet the required hypothesis for which we gave basics known techniques to verify. This part is then done thanks to the fully-implemented and user-friendly but also configurable modified Sinkhorn's algorithm. Finally, the fact that the loss computation is compatible with batches was taken into account to improve the case where samples don't have the same size. This case necessitated some work that could be performed internally and now only require the user to give the list of weights and/or samples ones. This simplifies the usage as should be any good OT solver [3, p.118] and there isn't any time lost when using `PyTorch` as backend.

References

[1] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994, p. 13.

[2] Benjamin Charlier, Jean Feydy, Joan Glaunès, and Ghislain Durif François-David Collin. "Kernel Operations on GPU, without memory overflows Kernel Operations on the GPU, with Autodiff, withoutMemory Overflows." In: *hal-02517462* (2020).

[3] Jean Feydy. "Geometric data analysis, beyond convolutions". 2020. URL: `http://www.jeanfeydy.com/geometric_data_analysis.pdf`.

[4] Jean Feydy, Thibault Séjourné, François-Xavier Vialard, Shun-ichi Amari, Alain Trouve, and Gabriel Peyré. "Interpolating between Optimal Transport and MMD using Sinkhorn Divergences". In: *The 22nd International Conference on Artificial Intelligence and Statistics*. 2019, pp. 2681–2690.

[5] Maysum Panju. "Iterative methods for computing eigenvalues and eigenvectors". In: *The Waterloo Mathematics Review* (2011).

[6] Aaditya Ramdas, Nicolás Trillos, and Marco Cuturi. "On Wasserstein Two-Sample Testing and Related Families of Nonparametric Tests". In: *Entropy* 19.2 (Jan. 2017), p. 47. ISSN: 1099-4300. DOI: `10.3390/e19020047`. URL: `http://dx.doi.org/10.3390/e19020047`.

[7] Bernhard Schölkopf, Ralf Herbrich, Alex J. Smola, and Robert Williamson. *A Generalized Representer Theorem*. 2000. DOI: `10.1007/3-540-44581-1_27`.

[8] Joel Sobel. *Duality and complementary slackness*. 2013. URL: `https://econweb.ucsd.edu/~jsobel/172aw02/notes6.pdf`.

[9] L. Song. "Learning via Hilbert Space Embedding of Distributions". 2008. URL: `https://www.cc.gatech.edu/~lsong/papers/lesong_thesis.pdf`.

[10] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: `https://doi.org/10.1038/s41592-019-0686-2`.

[11] Qiang Ye. "Accurate Inverses for Computing Eigenvalues of Extremely Ill-conditioned Matrices and DifferentialOperators." In: *hal-1512.05292* (2017).