An efficient kernel product for automatic differentiation libraries, with applications to measure transport

Benjamin Charlier[1], Jean Feydy[2,3], Joan Alexis Glaunès[4] and Alain Trouvé[3]

[1]Institut Montpelliérain Alexander Grothendieck, CNRS, Univ. Montpellier
[2]DMA, École Normale Supérieure, Paris
[3]CMLA, CNRS, ENS Paris-Saclay, Cachan
[4]MAP5, Univ. Paris 5, Paris

November 12, 2017

Abstract

This paper presents a memory-efficient implementation of the kernel *matrix-vector* product (sparse convolution) and the way to link it with automatic differentiation libraries such as `PyTorch`. This piece of software alleviates the major bottleneck of autodiff libraries as far as diffeomorphic shape registration is concerned: memory consumption. As a result, symbolic `python` code can now scale up to large point clouds and shapes ($> 100,000$ vertices).
To showcase the value of automatic differentiation to the LDDMM community, we introduce the *normalized Hamiltonian* setting and show that it corresponds to a spatially regularized optimal transport of mass distributions: made tractable by autodiff libraries, the kernel normalization trick turns an *extrinsic* image deformation routine into an *intrinsic* measure transportation program.

Index terms— Automatic differentiation, convolution, CUDA, kernel, LDDMM, optimal transport, Sinkhorn algorithm

Contents

1 Introduction

[ToDo: Introduction classique sur LDDMM.]

This work is about the automatic differentiation (*autodiff*) libraries which have been recently developed by the Deep Learning community, allowing seamless computation of symbolic gradients. As of 2017, the most prominent frameworks are `Theano`, `TensorFlow` and `PyTorch`: in the following pages, we will explain how these tools work under the hood; how we can curb them to suit the needs of the LDDMM community; and what they can bring to us from a mathematical perspective.

As these libraries free the researchers from the burden of implementation and debugging, they are bound to allow mathematical ideas to be prototyped and tested at a much faster rate than what was previously the norm. We are thus convinced that the development of efficient autodiff libraries, a true engineering feat, can be the force that drives the study of shapes spaces beyond the current mainstream paradigms.

Plan of the paper To reflect this, and just for once, we invert the classical plan of applied mathematics papers and write *from the code to the maths*. In this introduction, we remind the reader of the *data flow* of a generic LDDMM pipeline. We shall write in detail the steps by which a shooting momentum "p_0" is turned into a descent direction and real-valued cost. In section 2, we proceed to explain the underlying principles of autodiff libraries without glancing over technicalities. This allows us to precisely identify the major bottleneck that is encountered when one tries to employ a Deep Learning framework for a shape registration task: the memory consumption in the computation of kernel products of the form

$$g \;=\; K_{x,y} b, \qquad \text{that is,} \qquad g_i \;=\; \sum_j k(x_i - y_j)\, b_j \tag{1}$$

where (x_i), (y_j) are families of points in $\mathbb{R}^D$, (b_j) is a family of vectors in $\mathbb{R}^E$ and $k : \mathbb{R}^D \to \mathbb{R}$ is a radial kernel function.**[ToDo: Speak about non-scalar kernels?]**

In section 3, we therefore propose a memory-efficient CUDA implementation of this "KernelProduct" operation, and explain how to link it with the most flexible autodiff framework to date: `PyTorch`. Empowered with a *convenient* and *scalable* development tool, we then propose in section 4 a new LDDMM-like algorithm suited to the transport of *measures*. Relying on an inner normalization loop, the *normalized kernel* setting will be shown to provide a link between the LDDMM and Optimal Transport theories, at a reasonable computational cost.

1.1 Data flow of a diffeomorphic registration pipeline

The LDDMM theory of shape analysis has already been covered extensively in the literature, and we refer the interested reader to introductory works such as !!! and !!!. Instead of recalling the major theoretical results, we dedicate this section to the actual *implementations* and detail the handful of variables and numerical routines that are needed to put the theory into practice.

Variables In this work, we leave *images* aside and focus on *segmented shapes* instead: triangulated surfaces, fiber bundles, bidimensional curves and proteins. All the shapes considered will be embedded in a Euclidean ambient space $\mathbb{R}^D$ of dimension 2 or 3, as each shape Q is represented by a *point cloud* $(q^i)_{i\in[\![1,N]\!]}$

and a *connectivity matrix* $(c_s)_{s\in[\![1,F]\!]}$, respectively encoded as an N-by-D float array and an F-by-G integer array. Here, N is the number of vertices of the shape, D is the dimension of the ambient space, F is the number of faces and G is the number of vertices per shape element. The latter is equal to 2 if Q is a segmented curve, and 3 if Q is a triangulated surface.

The Riemannian shape theory also relies on velocity fields "v" or momenta "p" associated to points clouds "q": those are encoded as N-by-D float arrays. Here, we list all the variable names used by a generic LDDMM matching algorithm:

1. A source shape A, modeled by a point cloud (a^i) where $i \in [\![1, N]\!]$.
2. A target shape B, modeled by a point cloud (b^j) where $j \in [\![1, M]\!]$.
3. A moving point cloud (q_t^i) and its associated momentum (p_t^i), where time $t \in [0, 1]$ and index $i \in [\![1, N]\!]$.

Functions We denote by $Q = \mathbb{R}^{N\times D}$ the space of point clouds of size N, and by $P = Q^* = \mathbb{R}^{N\times D}$ the dual space of momenta. Acting on the shape-momentum coordinates, a diffeomorphic shape registration program relies on the following routines:

1. The Hamiltonian – kinetic energy – function:
$$\begin{array}{rcccl} H & : & Q\times P & \to & \mathbb{R} \\ & & (q,p) & \mapsto & H(q,p) \;=\; \frac{1}{2}\sum_{i,j} k(\|q^i - q^j\|)\langle p^i, p^j\rangle_{\mathbb{R}^D} \end{array}, \qquad (2)$$
where $k : \mathbb{R}^D \to \mathbb{R}$ is a *kernel function* and $\langle\cdot\,,\cdot\rangle_{\mathbb{R}^D}$ is the standard L^2 dot product in the ambient space. The kernel formula written above is the most popular in the literature, as it is both tractable and theoretically principled. But as far as the Riemannian theory is concerned, this kinetic energy should simply be smooth with respect to the position q and quadratic, positive definite wrt. the momentum p. In section 4, we will introduce a new family of formulas which are suited to the transport of measures.

2. The Hamiltonian flow in the cotangent bundle
$$\begin{array}{rcccl} \text{HamFlow} & : & Q\times P & \to & P^*\times Q^* \simeq Q\times P \\ & & (q,p) & \mapsto & (+\partial_p H(q,p), -\partial_q H(q,p)) \end{array}, \qquad (3)$$
associated to the *geodesic equation*.

3. The geodesic shooting routine (aka. the Riemannian exponential map), which takes as input an initial shape-momentum state (q_0, p_0), flows along the geodesic vector field HamFlow from $t = 0$ to $t = 1$ and outputs the resulting shape q_1:
$$\begin{array}{rcccl} \text{Shoot} & : & Q\times P & \to & Q \\ & & (q_0,p_0) & \mapsto & q_1 \end{array} \quad \text{with} \quad \frac{\mathrm{d}}{\mathrm{d}t}(q_t, p_t) \;=\; \text{HamFlow}(q_t, p_t). \qquad (4)$$

Integration of the ODE is typically done using linear update rules from the Runge-Kutta family – say, from RK1 (Euler) to RK4.

4. The data attachment function, which penalizes the "dissimilarity" between a deformed shape q_1 and the target B:

$$\begin{array}{rcccl} \text{Att.} & : & Q & \to & \mathbb{R} \\ & & q & \mapsto & \text{Att.}\,(\,q \mid b) \end{array} . \tag{5}$$

A convenient way to get a parametrization-invariant formula is to work with measures, turning both q and b into sums of weighted diracs by using appropriate connectivity matrices. Still, the choice of a well-behaved data attachment cost is highly dependent on the properties of the dataset: popular choices include the non-local *currents* and *varifolds* kernel formulas, while global costs relying on the Optimal Transport theory have recently been shown to be computationally tractable.

Shape matching In the simplest LDDMM setting, matching a source shape A to a target B is about finding an optimal *shooting momentum* p_0 such that

$$\begin{array}{rcccl} \text{Cost} & : & P & \to & \mathbb{R} \\ & & p_0 & \mapsto & H(A, p_0) \;+\; \text{Att.}\,(\,\text{Shoot}(A, p_0) \mid B) \end{array} \tag{6}$$

is minimized – here, $H(A, p_0)$ acts as a regularizer. As Cost is a complicated numerical function, one is restricted to the use of *local descent* schemes such as gradient descent or quasi-Newton algorithms, which converge to local optimums. A typical L-BFGS run on these problems converges in 20-100 iterations: the numerical complexity of an LDDMM pipeline is therefore proportional to that of a single computation of the oracle

$$\begin{array}{rcccl} \text{Oracle} & : & P & \to & \mathbb{R} \times P^* \\ & & p_0 & \mapsto & (\text{Cost}(p_0), \partial_p \text{Cost}(p_0)) \end{array} . \tag{7}$$

1.2 Practical bottlenecks, motivations for this work

Structure of LDDMM codebases Now, assume that the data attachment formula has been chosen from the kernel or Optimal Transport families. As evidenced by Equations (2-5), an evaluation of Oracle(p_0) makes use of only three types of operations:

1. Pointwise additions, multiplications and L^2 dot products.

2. Kernel matrix-vector products, also known as *convolutions*, of the form

$$\begin{array}{rcccl} \text{KernelProd} & : & \mathbb{R}^{ND} \times \mathbb{R}^{MD} \times \mathbb{R}^{ME} & \to & \mathbb{R}^{NE} \\ & & ((x_i), (y_j), (b_j)) & \mapsto & \left(\sum_{j=1}^{M} k(x_i - y_j)\, b_j \right)_{i \in [\![1,N]\!]} \end{array} \tag{8}$$

Indeed, the Hamiltonian formula (2) can be written as

$$H(q,p) \;=\; \frac{1}{2}\langle\; p\;,\; \text{KernelProd}(q,q,p)\;\rangle_{\mathbb{R}^{ND}}\;, \tag{9}$$

and the various kernel or Optimal Transport fidelity terms have been precisely designed with this convolution operator in mind.

3. Gradient computations – that is, partial derivatives of H and Cost with respect to their input variables.

Operations of the first kind are both easy to code and quick to execute; unfortunately, this cannot be said of the rest of our codebases. As researchers, we collectively got used to dedicate *a lot* of time to the writing, testing and debugging of *gradient* routines. Meanwhile, the computation of *convolutions* makes up the bulk of the execution time of our pipelines.

The memory bottleneck Automatic differentiation libraries come with a promise: that of relieving us of these thankless gradient implementations. Unfortunately, out of the box, Deep Learning toolboxes allow us to implement kernel convolutions only in the most memory-intensive way: by computing and storing full N-by-M kernel matrices $(k(x_i - y_j))_{i,j}$. This method makes a lot of sense in a neural network setting, where data samples have a low memory footprint (mini batches of 2D images, etc.), because any intensive memory usage is directly correlated to the size of the neural model chosen by the researcher. As training speed is one of their primary concerns, data scientists never feel the need to trade time for memory and thus mostly content themselves with "the largest neural net" that fits on their GPUs.

Motivations for this work This approach is not sensible in our setting, as the time and memory complexity of an LDDMM matching comes directly from the large size of the point clouds considered: in a medical setting, N and M are typically of the order of $100,000$. Storing a handful of M-by-N matrices in the GPU memory is simply not possible.

In order to allow researchers of the shape analysis community to use automatic differentiation beyond toy examples, we propose a *memory efficient* CUDA implementation of the KernelProd operator of Eq. (8) and of its derivatives of order 1 and 2. Thanks to the flexibility of the `PyTorch` library, we were able to package those codes into a single operation, `KernelProd`, that can be used seamlessly within any symbolic `PyTorch` code.

First and foremost, this paper was written to explain and document what is happening beneath the convenient (black-box) `python` abstraction. Please note that our code is available online,

`https://plmlab.math.cnrs.fr/benjamin.charlier/libkp`

We hope that it will help to release the creativity of researchers in this field. As a showcase example of what can be achieved when one combines Riemannian geometry with automatic differentiation, we present in the last section of this paper the preliminary results of Jean Feydy and Alain Trouvé on a new class of *normalized* Hamiltonian formulas.

2 Automatic differentiation for shape analysis

The main contribution of this paper is a piece of software that can be plugged into any sufficiently flexible autodiff framework: the `KernelProd` operator and its underlying CUDA implementation. Relative to the standards of the shape analysis community, this work is definitely *low-level* software engineering. As the reader cannot be expected to be fluent with the internal design of recent Deep Learning libraries, we now dedicate a whole section to the step-by-step computation and differentiation of the classical *Hamiltonian* formula introduced in Eq. (2):

$$\begin{aligned} H(q,p) &= \frac{1}{2}\sum_{i,j=1}^{N} k(q^i - q^j)\,\langle p^i, p^j\rangle_{\mathbb{R}^D} && (10)\\ &= \frac{1}{2}\langle p, K_{q,q}p\rangle_{\mathbb{R}^{ND}} \qquad \text{with} \quad (K_{q,q})_{i,j} = k(q^i - q^j), && (11)\end{aligned}$$

where (q^i) and (p^i) are represented by N-by-D float arrays. By the end of this section, hopefully, the reader should have a clear understanding of what is (and what is *not*) possible within an automatic differentiation framework.

2.1 Backpropagation 101

Finite differences are not the solution Let $F : \mathbb{R}^n \to \mathbb{R}$ be a differentiable function defined as a symbolic computer program. Our problem of interest is: How does one *efficiently* compute the oracle value $(F(x_0), \partial_x F(x_0))$ at a given location $x_0 \in \mathbb{R}^n$?

A naive approach, the so-called *finite differences* scheme, would be to use a Taylor expansion of F around x_0 and write, for δt sufficiently small,

$$\begin{pmatrix} \partial_{x^1}F(x_0) \\ \partial_{x^2}F(x_0) \\ \vdots \\ \partial_{x^n}F(x_0) \end{pmatrix} \simeq \frac{1}{\delta t}\begin{pmatrix} F(x_0 + \delta t\cdot(1,0,0,\dots,0)) - F(x_0) \\ F(x_0 + \delta t\cdot(0,1,0,\dots,0)) - F(x_0) \\ \vdots \\ F(x_0 + \delta t\cdot(0,0,0,\dots,1)) - F(x_0) \end{pmatrix}. \qquad (12)$$

This idea is simple to implement. But it also requires $n+1$ evaluations of the function F to compute a *single* gradient vector! As soon as the dimension of the input space exceeds 10-100, this is not tractable: Just like inverting a full matrix A is not the sensible way to solve the linear system "$Ax = b$", one should not use finite differences – or any equivalent *forward* scheme – to compute a gradient.

Gradients between Hilbert spaces Thankfully, there exists an efficient way to compute gradients of real-valued functions: the reverse accumulation scheme. As it relies on a *backward* pass through the computational graph, this useful algorithm has recently been popularized under the name of "*backpropagation*" and lies at the core of every Deep Learning framework. To understand it, consider the following definition of (generalized) gradients between Hilbert spaces:

Definition 1. *Let $(X, \langle \cdot, \cdot \rangle_X)$ and $(Y, \langle \cdot, \cdot \rangle_Y)$ be two Hilbert spaces, and let $F : X \to Y$ be a continuously differentiable function between them.*
Let also $x_0 \in X$ be an input position and $\alpha \in Y^*$ be a linear form on Y, which we identify with a vector $a \in Y$ through the Riesz theorem.
Then, for all increment $\delta x \in X$, we have

$$\begin{aligned}
\langle \alpha, F(x_0 + \delta x) \rangle &= \langle \alpha, F(x_0) \rangle + \langle\ \alpha,\ \mathrm{d}_x F(x_0) \cdot \delta x\ \rangle + o(\delta x) && (13)\\
&= \langle \alpha, F(x_0) \rangle + \langle (\mathrm{d}_x F)^*(x_0) \cdot \alpha,\ \delta x\ \rangle + o(\delta x) && (14)\\
&= \langle a, F(x_0) \rangle_Y + \langle\ \partial_x F(x_0) \cdot a,\ \delta x\ \rangle_X + o(\delta x), && (15)
\end{aligned}$$

where we identify the adjoint of the differential $(\mathrm{d}_x F)^*(x_0) : Y^* \to X^*$ with a continuous linear ap $\mathbf{\partial_x F(x_0) : Y \to X}$, thanks to the Riesz theorem.
We say that the latter is the ***generalized gradient*** *of F at x_0, with respect to the Hilbertian structures of X and Y.*

If X and Y are respectively equal to $\mathbb{R}^n$ and $\mathbb{R}$ endowed with their canonical L^2-Euclidean structures, $\partial_x F(x_0)$ coincides when displayed in the canonical basis with the well-known vector $\nabla_x F(x_0)$ of directional derivatives.

Chain rule for gradients This Hilbertian definition of the gradient has two major advantages over the "vector of derivatives" one. First, it stresses the fact that a gradient is an object which is defined with respect to a *metric structure*, not a basis. As we frequently work with spaces of momenta on which the L^2 metric makes very little sense, this is important.

Second, it allows us to *compose* gradients without reserve. Indeed, if X, Y, Z are three Hilbert spaces, and if $F = H \circ G$ with $G : X \to Y$ and $H : Y \to Z$, then for all $x_0 \in X$, the composition rule asserts that

$$\mathrm{d}_x F(x_0) = \mathrm{d}_y H(G(x_0)) \circ \mathrm{d}_x G(x_0), \qquad (16)$$

so that

$$\begin{aligned}
& [\mathrm{d}_x F(x_0)]^* &= [\mathrm{d}_x G(x_0)]^* \circ [\mathrm{d}_y H(G(x_0))]^* && (17)\\
\text{i.e.}\quad & \partial_x F(x_0) &= \partial_x G(x_0) \circ \partial_y H(G(x_0)). && (18)
\end{aligned}$$

Backpropagation Suppose that the function of interest $F : \mathbb{R}^n \to \mathbb{R}$ is defined as a composition $F = F_p \circ \cdots \circ F_2 \circ F_1$ of elementary functions $F_i : \mathbb{R}^{N_{i-1}} \to \mathbb{R}^{N_i}$ where $N_0 = n$ and $N_p = 1$:

$$\mathbb{R}^n = \mathbb{R}^{N_0} \xrightarrow{F_1} \mathbb{R}^{N_1} \xrightarrow{F_2} \mathbb{R}^{N_2} \xrightarrow{\cdots} \cdots \xrightarrow{F_p} \mathbb{R}^{N_p} = \mathbb{R}$$

To keep the notations simple, we will assume that all the input and output spaces $\mathbb{R}^{N_i}$ are endowed with their canonical L^2-Euclidean metrics. Remember that we are interested in computing at an arbitrary location $x_0 \in \mathbb{R}^n$ the gradient

$$\partial_x F(x_0) : \mathbb{R} \to \mathbb{R}^n, \qquad (19)$$

a linear map which is entirely determined by the value of the "'gradient vector"

$$\begin{aligned}
\partial_x F(x_0)\cdot 1 &= \partial_x F_1(x_0)\circ \partial_x F_2(F_1(x_0))\circ\cdots\circ \partial_x F_p(\,F_{p-1}(\cdots(F_1(x_0)))\,)\cdot 1 \qquad (20)\\
&= \partial_x F_1(x_0)\circ \partial_x F_2(\quad x_1 \quad)\circ\cdots\circ \partial_x F_p(\qquad x_{p-1} \qquad)\cdot 1 \qquad (21)
\end{aligned}$$

where the $x_i = F_i \circ \cdots \circ F_1(x)$ are nothing but the intermediate results in the computation of $x_p = F(x_0)$.

Then, our major assumption is that the *forward* and *backward* operators

$$\begin{array}{rcccl}
F_i & : & \mathbb{R}^{N_{i-1}} & \to & \mathbb{R}^{N_i} \\
 & & x & \mapsto & F_i(x)
\end{array} \qquad (22)$$

and

$$\begin{array}{rcccl}
\partial_x F_i & : & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i} & \to & \mathbb{R}^{N_{i-1}} \\
 & & (x_0, a) & \mapsto & \partial_x F_i(x_0)\cdot a
\end{array} \qquad (23)$$

are all **encoded as computer programs**. According to Eq. (21), it is therefore possible to compute both $F(x_0)$ and $\partial_x F(x_0)$ by a forward-backward pass through the following diagram:

$$\begin{tikzcd}
\mathbb{R}^n & \mathbb{R}^{N_0} & \mathbb{R}^{N_1} & \mathbb{R}^{N_2} & \cdots & \mathbb{R}^{N_p} & \mathbb{R} \\
\cup & \cup & \cup & \cup & & \cup & \cup \\
x_0 \arrow[r, "\textbf{input}"] & x_0 \arrow[r, "F_1"] \arrow[dd, blue] & x_1 \arrow[r, "F_2"] \arrow[dd, blue] & x_2 \arrow[r, "\cdots"] \arrow[dd, blue] & \cdots \arrow[r, "F_p"] \arrow[dd, blue] & x_p \arrow[r, "\textbf{output}"] & F(x_0) \\
 & & & & & & \\
\partial_x F(x_0)\cdot 1 & x_0^* \arrow[l, red, "\textbf{output}"'] & x_1^* \arrow[l, "\partial_x F_1"'] & x_2^* \arrow[l, "\partial_x F_2"'] & \cdots \arrow[l, "\cdots"'] & x_p^* \arrow[l, "\partial_x F_p"'] & 1 \arrow[l, green, "\textbf{input}"'] \\
\cap & \cap & \cap & \cap & & \cap & \cap \\
\mathbb{R}^n & \mathbb{R}^{N_0} & \mathbb{R}^{N_1} & \mathbb{R}^{N_2} & \cdots & \mathbb{R}^{N_p} & \mathbb{R}
\end{tikzcd}$$

The *backpropagation* algorithm proceeds in two steps corresponding to the two lines of the above diagram:

1. Starting from $x_0 \in \mathbb{R}^n = \mathbb{R}^{N_0}$, compute and **store in memory** the successive vectors $x_i \in \mathbb{R}^{N_i}$. The last one, $x_p \in \mathbb{R}$, is equal to the value of the objective $F(x_0)$.

2. Starting from the canonical value of $x_p^* = 1 \in \mathbb{R}$, compute the successive *dual* vectors

$$x_i^* \;=\; \partial_x F_{i+1}(x_i)\cdot x_{i+1}^*. \qquad (24)$$

The last one, $x_0^* \in \mathbb{R}^n$, is equal to the gradient $\nabla F(x_0) = \partial_x F(x_0)\cdot 1$.

Implementation and performances The generalization of this procedure to any acyclic "forward" computational graph is straightforward. Hence, provided that the forward and backward operators of Eq. (22-23) are pre-implemented, one can compute *automatically* the gradient of any symbolic procedure that is written as a succession of elementary vector operations, the F_i's.

Consequently, Deep Learning libraries rely on three core modules: a set of low-level GPU routines; an exhaustive list of usual operations (forward and backward) provided to end-users; a high-level graph manipulation API.

Bottom line is: The *backwards* of the usual operations are seldom more costly than 4-5 applications of the corresponding *forward* operators. Ergo, if one has enough memory available to store the intermediate results during the forward pass, **the backpropatation algorithm is an automatic and time-effective way to compute arbitrary gradients**.

2.2 Memory usage in the computation of the Hamiltonian

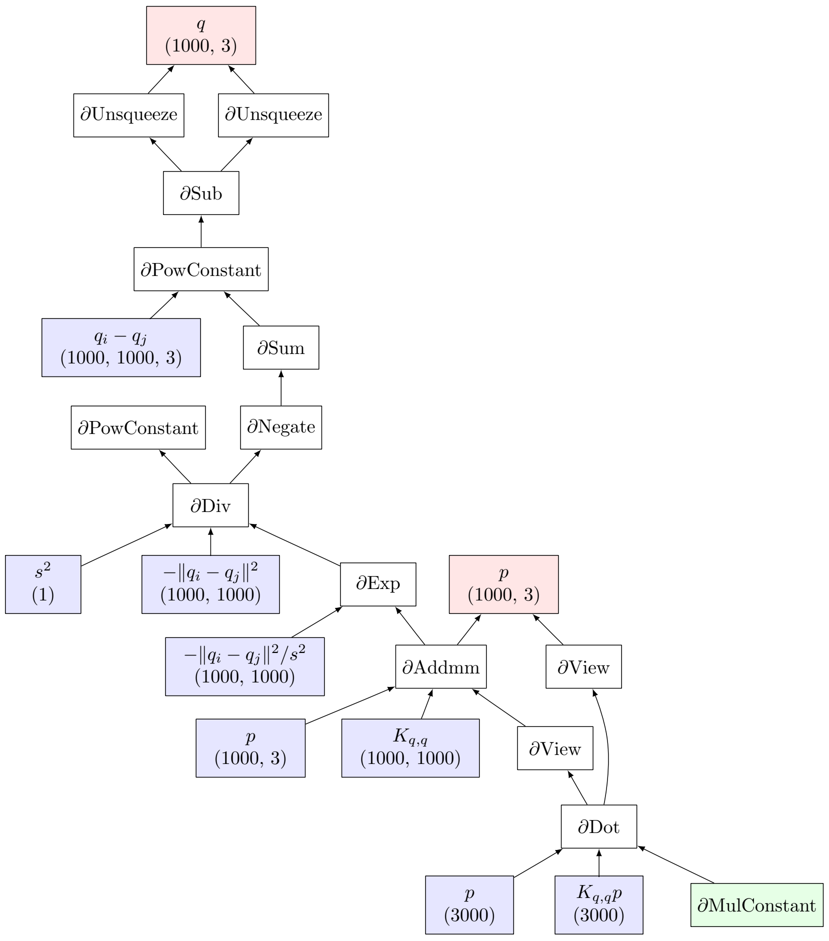
A minimal working example Let us illustrate the underlying mechanics of `PyTorch` – a Deep Learning library – in a simple case: the computation of the kernel Hamiltonian defined Eq. (10-11) when using a *gaussian kernel* of deviation $s > 0$:

$$k \quad : \quad \begin{array}{ccc} \mathbb{R}^D \times \mathbb{R}^D & \to & \mathbb{R} \\ (x,y) & \mapsto & e^{-\|x-y\|_2^2\,/\,s^2} \end{array} . \tag{25}$$

```python
import torch          # GPU + autodiff library
from visualize import make_dot # GraphViz tool to plot graphs
# See github.com/szagoruyko/functional-zoo/blob/master/visualize.py

# With PyTorch, using the GPU is that simple:
use_gpu   = torch.cuda.is_available()
dtype     = torch.cuda.FloatTensor if use_gpu else torch.FloatTensor
# Under the hood, this flag will determine the backend used for
# forward and backward operations, as they have all been
# implemented both in pure CPU and in GPU (CUDA) code.

N = 1000; D = 3 ; # Work with clouds of 10,000 points in 3D
# Generate arbitrary arrays on the host (CPU) or device (GPU):
q = torch.linspace( 0, 5, N*D ).type(dtype).view(N,D)
p = torch.linspace( 3, 6, N*D ).type(dtype).view(N,D)
s = torch.Tensor(      [2.5]    ).type(dtype)

# Wrap them into "autodiff" graph nodes. In this demo,
# we won't try to fine tune the deformation model, so
# we do not need any derivative with respect to s:
q = torch.autograd.Variable( q, requires_grad = True )
p = torch.autograd.Variable( p, requires_grad = True )
s = torch.autograd.Variable( s, requires_grad = False)

# Actual computations. Every PyTorch instruction is executed
# on-the-fly, but the graph API 'torch.autograd' keeps track of
# the order of the operations and stores in memory the intermediate
# results that are needed for the backward pass.
q_i   = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j   = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd   = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq  = torch.exp( - sqd / (s**2) )      # Gaussian kernel
v     = K_qq @ p # matrix multiplication. (N,N)@(N,D) = (N,D)
# Finally, compute the Hamiltonian H(q,p):
H     = .5 * torch.dot( p.view(-1), v.view(-1) ) # .5*<p,v>

# Display -- see next figure.
print(H); make_dot(H, {'q':q, 'p':p, 's':s}).render(view=True)
```

For the sake of completeness, we provide here a full, verbose working example: Needless to say, its header is bound to get deprecated sooner or later. But the core of the procedure, the lines related to the Hamiltonian formula, those are here to stay. From a mathematical point of view, these symbolic python instructions define a *computational graph* that can be used to differentiate $H(q, p)$ with respect to q and p.

Encoding a formula in the computer's memory In the figure below, we display this `torch.autograd.Variable` graph `'H'` as it is understood by `PyTorch`. This acyclic graph is the exact equivalent of the second "backward" line of the diagram presented page 9: Every white node stands for a backward operator $\partial_x F_i : (x_i, x^*_{i+1}) \mapsto x^*_i$. The green leave is the first covariable $x^*_p \in \mathbb{R}$, the "gradient with respect to the output" which is initialized to 1; the red leaves are the covariables x^*_0 in which the gradients are to be accumulated; and the blue ones are the *stored* values x_i computed during the forward pass.



Memory usage The precise meaning of the backpropagation graph will be made clear in section 2.3. Nevertheless, we can already point out the extravagant memory usage required for the computation of the velocity field `v = K(q,q) @ p`. To differentiate "PowConstant", "Div", "Exp" and "Addmm", the backpropagation algorithm has to store in memory full N-by-N intermediate results: $q_i - q_j$, $-\|q_i - q_j\|^2$, etc.

Therefore, the native `PyTorch` implementation of page 10 is intractable as soon as the number of points N exceeds the **square root of the GPU memory** – that is, about $50,000$ for a recent piece of hardware.

Our contribution To break this ceiling in a way which is most profitable for the shape analysis community, we propose to wrap the critical computation of `'v'` into a generic and memory efficient operator: the `KernelProduct` object, which implements the kernel convolution formula of Eq. (8). As `KernelProduct` takes as input two point clouds and one momentum field – of respective shapes `(N,D)`, `(M,D)` and `(M,E)` – to output a momentum field of shape `(N,E)`, the `torch.autograd` module will never need to store full `(N,M)` arrays in the GPU memory.

This way of doing bypasses the built-in `PyTorch` operators to rely on a finely crafted CUDA memory management scheme, explained in section 3. In section 4, we then showcase a typical example of use: the *normalized Hamiltonian* setting.

2.3 Linking custom CUDA routines with `PyTorch`

But first of all, we wish to document here the proper way of linking CUDA routines to `PyTorch` symbolic instructions. As we are about to implement a new `PyTorch` elementary operator, we start with a brief exposition of the framework's internal behaviors and paradigms.

Static autodiff Remember. The legacy `Theano` (2008-2017) library divided in *three steps* the translation of a `python` symbolic script into an efficient GPU routine. First, the `python` programmer declared a whole computational graph at once, without any actual *computation* taking place. Then, a graph optimizer pruned out unused nodes, merged subgraphs, etc., and automatically generated a C/CUDA program. The latter was then compiled using `gcc`, and the resulting executable was linked to a wrapper `python` function.

This way of doing made differentiation look easy: `'.grad'` was just another symbolic node in a purely abstract computational graph. Unfortunately, it also induced two adverse side effects: a lengthy compilation time at the start of every single script; the inability to implement dynamic flow control (`if-then-else` structures, etc.), which make the operations applied to arrays depend on their actual *values*.

The dynamic workflow The `PyTorch` library has recently been introduced to cover those deficiencies. Unlike the static Declaration-Optimization-Compilation frameworks, it implements a *dynamic* workflow which can be summarized as follows:

1. `Variables` are seen as graph objects, wrapped around int/float arrays.
2. Instructions are executed on-the-fly, using pre-compiled CPU or GPU routines in the back end. The result of any such operation is a new `Variable` wrapped around an "output" array, with a "graph history" attribute keeping track of all the operations that are needed to compute the output.
3. As differentiation upsets the whole "graph history" of input `Variables`, it is handled by a specific bunch of instructions. If a variable `H` was computed using two user-defined variables `q` and `p`, the most math-like way of applying a backpropagation pass on the graph history of `H` is to write:

   ```
   [dq,dp] = torch.autograd.grad( H, [q,p] , g, create_graph=True)
   ```

   where `g` is the "input" gradient x_p^* with respect to `H`, initialized by default to `1` if `H` is scalar. This special instruction outputs two variables `dq` and `dp` whose numerical values were computed as output of the *backpropagation* algorithm.

Computing second derivatives The default behaviour of `autograd.grad` is to output `Variable` objects with a blank history: if one simply needs to do gradient descent on `H`, keeping track of the computational history of the gradients is basically useless. However, as evidenced by the Shooting, Cost and Oracle routines of Eq. (4-7), differentiating the Hamiltonian *a second time* is crucial to our LDDMM shape analysis pipelines.

This is made possible by the `create_graph` flag. When it is set to the value `True` – as in the above instruction – `PyTorch` considers that the diagram displayed in page 9 is a new "*forward*" program which takes as input the vector x_0 and the covector x_p^*, to output the backpropagated gradient x_0^*.

This means that the `autograd.grad` instruction can be differentiated once again, at the condition that the order-0 and order-1 operators F_i and $\partial_x F_i$ defined Eq. (22-23) are available as computer programs... **As well as their own gradients**. In practice, this means that `PyTorch` must know how to compute the "gradients of gradients" operators:

$$\begin{array}{rcccl} \partial_{x_0}(\partial_x F_i(x_0)\cdot a) & : & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i}\times\mathbb{R}^{N_{i-1}} & \to & \mathbb{R}^{\mathbf{N_{i-1}}} \\ & & (\boldsymbol{x_0},\boldsymbol{a},\boldsymbol{e}) & \mapsto & \partial_{x_0}(\partial_x F_i(x_0)\cdot a)(\mathbf{x_0},\mathbf{a})\cdot\mathbf{e} \end{array} \tag{26}$$

$$\begin{array}{rcccl} \partial_a\ (\partial_x F_i(x_0)\cdot a) & : & \mathbb{R}^{N_{i-1}}\times\mathbb{R}^{N_i}\times\mathbb{R}^{N_{i-1}} & \to & \mathbb{R}^{\mathbf{N_i}} \\ & & (\mathbf{x_0},\mathbf{a},\mathbf{e}) & \mapsto & \partial_a(\partial_x F_i(x_0)\cdot a)(\mathbf{x_0},\mathbf{a})\cdot\mathbf{e} \end{array} \tag{27}$$

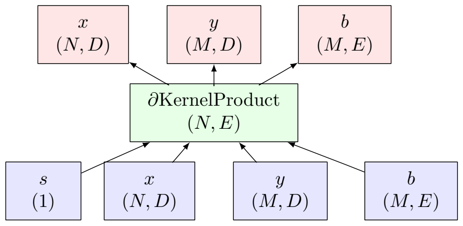
The convolution operator The simplest way to do this is to define operators of “order 1” such as `KernelProductGrad_x`, and explicitly use them in the `backward` method of our order 0 operator, `KernelProduct`.

As `PyTorch` is not yet fully documented, we provide below the meaningful elements of syntax that allowed us to implement a twice-differentiable CUDA-based operator. This may help other researchers to get their own non-standard ideas to work on real data. Note that for complete reference, our code is available on the CNRS gitlab:

`https://plmlab.math.cnrs.fr/benjamin.charlier/libkp`.

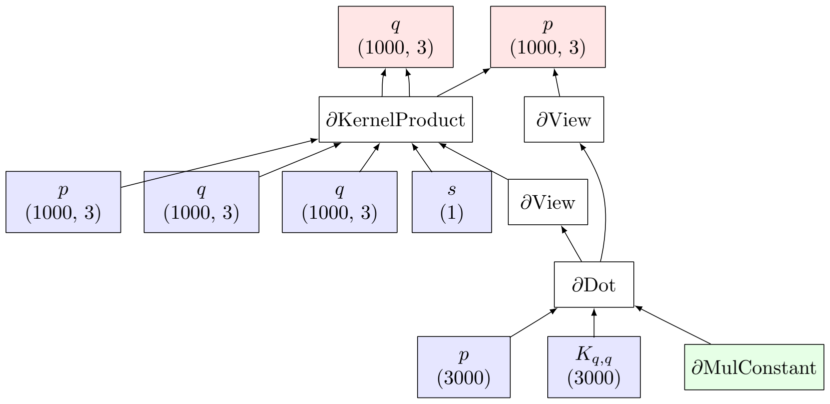
```
import torch

class KernelProduct(torch.autograd.Function):
    @staticmethod
    def forward(ctx, s, x, y, b, kernel_type):
        # save everything to compute the gradient
        ctx.save_for_backward( s, x, y, b )
        # init gamma, the output of the convolution K_xy @ b
        gamma  = torch.zeros( x.size()[0] * b.size()[1]
                                                    ).type(dtype)
        # Inplace CUDA routine on the raw float arrays,
        # loaded from .dll/.so files by the "ctypes" module
        cudaconv.cuda_conv(    x.numpy(), y.numpy(), b.numpy(),
                           gamma.numpy(), s.numpy(),
                           kernel = kernel_type)
        gamma  = gamma.view( x.size()[0], b.size()[1] )
        return gamma

    @staticmethod
    def backward(ctx, a):
        (ss, xx, yy, bb) = ctx.saved_variables
        # In order to get second derivatives, we encapsulated the
        # cudagradconv.cuda_gradconv routine in another
        # torch.autograd.Function object:
        kernelproductgrad_x = KernelProductGrad_x().apply

        # Call the CUDA routines
        # ...
        grad_x = kernelproductgrad_x( ... )
        # ...
        return (grad_s, grad_x, grad_y, grad_b, None)

class KernelProductGrad_x(torch.autograd.Function):
    @staticmethod
    def forward(ctx, s, a, x, y, b, kernel_type):
        # Save for Backward + Call the CUDA routines
        # ...
        return grad_x

    @staticmethod
    def backward(ctx, e):
        # Call the CUDA routines
        # ...
        return (grad_xs, grad_xa, grad_xx, grad_xy, grad_xb, None)
```

The resulting object can now be used seamlessly in a `PyTorch` computation, taking as input a kernel size, a kernel type (such as ``‘‘gaussian’’`` or ``‘‘energy’’``) and three tensors. `KernelProduct` is as easy to use as a built-in operator and stands for the following piece of graph:



Computing a Hamiltonian efficiently This `python` object can be used to compute “kernel”, “currents”, “varifolds” or “Optimal Transport” discrepancies between shapes. Crucially, it can also be used in the declaration of the Hamiltonian: in the `PyTorch` example showcased page 10, one simply has to replace the lines 25-35 with the code shown below. The resulting computational graph is then mathematically equivalent to that of page 11, but doesn’t store any large matrix in memory.

```
# Compute the kernel convolution
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p):
H = .5 * torch.dot( p.view(-1), v.view(-1) ) # .5*<p,v>
```



3 Computing a kernel product using Cuda

Cuda is a programming language similar to C/C++ allowing to run massively parallel programs on a Graphical Processor Unit. It is developed by Nvidia... and runs only on Nvidia hardware. This specificity allows a very fine tuning of memory and data transfer management yielding impressive performance. Moreover, the Cuda Toolkit now contains a version of most common standard linear algebra libraries (cuBlas, cuFft) and some relatively high level libraries (Thrust) helping the developer to use a Nvidia GPU at minor development cost.

The dark side of this efficiency, is a complete lack of portability: if you don't have an access to a Nvidia GPU you will not be able to run the code. Unfortunately, few alternatives exist (openCL or openACC may be used with GPU fom other brands) but they are not currently as competitive as the pair Nvidia/Cuda [?].

3.1 A crash course in Cuda

3.1.1 Data transfer: host to device

A Cuda function is sometime called a kernel (not to be confuse with the mathematical meaning of kernel). The CPU is often called the *host* whereas the GPU is called the *device*. Usually, the main part of a code runs on the host and only some specific operations are performed on the device.

The data are initially stored on the host and should be transfer to the device to be treated. This operation is often considered as the bottleneck of the GPU programming as the bandwidth of the PCIe port linking phisically the motherboard to the GPU card can be easily saturated. To overcome this limitation Nvidia has recently developed a proprietary port (with IBM) called NVLink.

The data transfer is coded in a gateway C or C++ function. Considering the convolution defined formula (1), we need to transfer the coordinates of the points $(x_i)_i$ and $(y_j)_j$, the vectors $(b_j)_j$, the kernel bandwidth σ and the address of $\gamma = (\gamma_i)_i$ where the results are stored. It reads :

Listing 1: Outline of the gateway function and data tranfer between host and device.

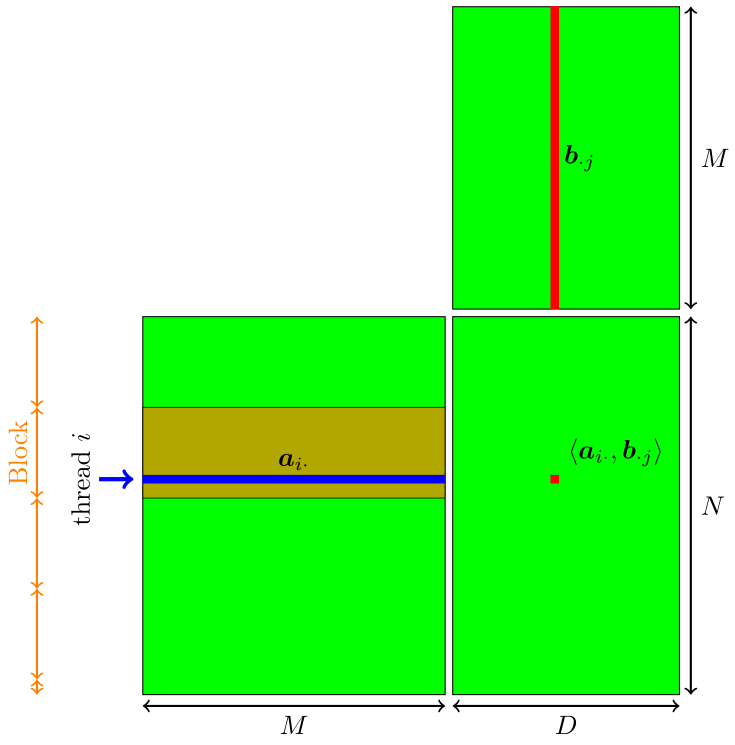
```
#include <cuda.h>
...
// x_h : matrix N x dimPoint
// y_h : matrix M x dimPoint
// b_h : matrix M x D
// gamma_h :

int KernelGpuEvalConv(float ooSigma2,float* x_h, float* y_h, float*
     beta_h, float* gamma_h,...){

    // Data on the device.
    float* x_d;
    float* y_d;
    float* beta_d;
    float* gamma_d;

    // Allocate arrays on device.
    cudaMalloc((void**)&x_d,sizeof(float)*(N*dimPoint));
    cudaMalloc((void**)&y_d,sizeof(float)*(M*dimPoint));
    cudaMalloc((void**)&beta_d,sizeof(float)*(M*D ));
    cudaMalloc((void**)&gamma_d, sizeof(float)*(N*D));

    ...

    // Upload data from host to device.
    cudaMemcpy(x_d,x_h,sizeof(float)*(N*dimPoint),
        cudaMemcpyHostToDevice);
    cudaMemcpy(y_d,y_h,sizeof(float)*(M*dimPoint),
        cudaMemcpyHostToDevice);
    cudaMemcpy(beta_d, beta_h, sizeof(float)*(M*D ),
        cudaMemcpyHostToDevice);


    ...
    // Call the cuda kernel which store the results in gamma_d
    // See next Sections
    ...

    // Download data from device to host.
    cudaMemcpy(gamma_h, gamma_d, sizeof(float)*(N*D),
        cudaMemcpyDeviceToHost);

    // Free memory (maybe not needed with last version of cuda??)
    cudaFree(x_d);
    cudaFree(y_d);
    cudaFree(beta_d);
    cudaFree(gamma_d);

    return 0;
```

3.1.2 Cuda kernel grid

A GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (*SM*s). In a SM, each single processor is called a *thread* and is able to execute an independent set of instructions. For example, a standard matrix multiplication between $A \in \mathbb{R}^{N\times M}$ and $B \in \mathbb{R}^{M\times D}$:

$$[AB]_{ij} = \sum_{k=1}^{M} a_{ik}b_{kj} = \langle \boldsymbol{a}_{i\cdot}, \boldsymbol{b}_{\cdot j}\rangle, \qquad i = 1,\cdots,N, j = 1,\cdots,D$$

may be though as the computation of ND scalar products between the column vector $\boldsymbol{a}_{i\cdot}$ (containing the entries in the i-th row of A) and $\boldsymbol{b}_{\cdot j}$ (being the column vector containing the j-th column of B). Now imagine that you have N processors (even if N is huge), so that each processor can be dedicated to the computation of D scalar products. The results of the matrix multiplication could then be returned in the same amount of time than D scalar product. In fact, this is roughly speaking what happen with a GPU as depicted in Figure 1. Note that in our setting we are mainly interrested in computing convolutions with $N \approx M \approx 10^6$ and $D = 2, 3$.



$\boldsymbol{b}_{\cdot j}$
M
Block
thread i
$\boldsymbol{a}_{i\cdot}$
$\langle \boldsymbol{a}_{i\cdot}, \boldsymbol{b}_{\cdot j} \rangle$
N
M
D

Figure 1: A matrix multiplication AB (where $A \in \mathbb{R}^{N\times M}$ and $B \in \mathbb{R}^{M\times D}$) is a set of ND scalar products. In a Cuda kernel with a 1d grid, D scalar products are computed by a single thread i. Source: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory`

In the Cuda paradigm, one start by specifying a computation plan (called a *grid*). A grid is composed by *blocks* and each block contains a certain number of thread. The maximum number of thread in each block is $2^{10} = 1024$. For historical reasons (remember that GPU were initially created to render 3d graphics), the grid is in one, two or three dimensional. The values of the three dimensions are stored in triple of `int` called a `dim3` For our convolution, we complete the piece of code above by illustrating (replace lines 30 to 33 in Listing 1) the call of a Cuda kernel with a 1d grid:

Listing 2: Definition of the grid size.

```
    // The blocksize is a triple of integer stored in a "dim3".
       Here we define a 1d kernel as the y and z coordinates are
       1. Each block then contains a 'vector' of 128 threads.
    dim3 blockSize(128,1,1);

    // We now have to defined the grid size (ie number of blocks in
         the grid). A dim3 is automatically initialize to (1,1,1).
        We just have to modify its first coordinates.
    dim3 gridSize;
    gridSize.x =  N / blockSize.x + (N%blockSize.x==0 ? 0 : 1);

    // Here, we call the cuda kernel executed on the device. The
        <<<nb_of_block_in_the_grid, nb_of_thread_in_each_block,
        size_of_shared_mem_per_block>>> syntax allows to pass the
        parameters of the grid to the compiler.
    KernelGpuConvOnDevice<<<gridSize,blockSize,blockSize.x*(D+
        dimPoint)*sizeof(float)>>>(ooSigma2, x_d, y_d, beta_d,
        gamma_d, N, M);
```

Note that we do not have written a single line of CUDA code yet. This is pure C/C++.

3.1.3 Memory management in Cuda

The memory architecture of a GPU is rather complicated and is evolving with each new generation of card. We stick here to the very basics and the interested reader may find many good introduction on this topic on Internet. We mention here only 3 different types of memory (pictured in the Figure [**?**]):

- Global memory: huge amount of space (typically several Giga Bytes) but access is slow. Every thread can red/write to this memory. This is the only memory that can be accessible from the CPU.

- Register and local memory: from Nvidia doc : "Local memory is so named because its scope is local to the thread, not because of its physical location. In fact, local memory is off-chip. Hence, access to local memory is as expensive as access to global memory. In other words, the term local in the name does not imply faster access. Local memory is used only to hold automatic variables. This is done by the nvcc compiler when it determines that there is insufficient register space to hold the variable. Automatic variables that are likely to be placed in local memory are large structures or arrays that would consume too much register space and arrays that the compiler determines may be indexed dynamically."

- Shared memory : very small amount of memory (48 KB per SM) but extremely fast. If N blocks runs at the same time on one SM, the max amount of shared mem avalaible per block is $48/N$ KB.

A smart use of the shared memory is often the key to provide an efficient code in term of computational time. This is classical for GPU developer and well documented in the Nvidia docs or any course in GPU programming. The interested reader may find all the characteristics of Nvidia Cards at the following website: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-spec`

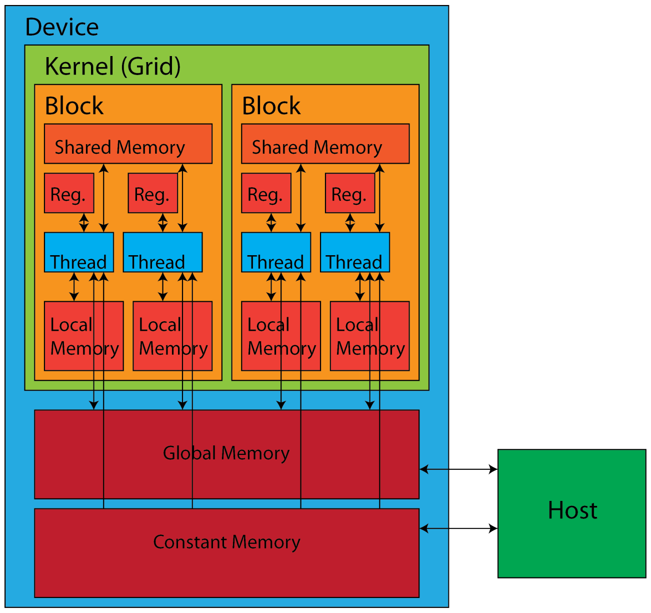


Figure 2: Sketch of the memory architecture of a GPU. A double heads arrow means read/write access and single head arrow means read only access. Source : `https://cvw.cac.cornell.edu/gpu/images/figure6.png`

To cast the memory architecture in a culinary metaphor: you have to cook a dish (perform a computation) with a complicated recipe (with a CUDA kernel). Imagine that you have at your disposal a huge restaurant (a GPU device) with many kitchens (blocks). Each kitchen has several chefs (threads). The spatial organization of the various kitchen in the restaurant should fit the particular need of the recipe (1d, 2d or 3d Grid). The ingredient are stored in the cellar (the global memory). To cook the dish, each chef of a kitchen first access to the cellar to fetch an ingredient (ie the access to global memory are paralyzed). If the ingredient is specific to each chef it goes to the personal kitchen counter of the chef (private local memory to each thread). If the ingredient is common to every chef of the kitchen it goes to the fridge of the kitchen (shared memory accessible by any thread in the block), so that every one in the kitchen may access quickly to the ingredient. Intuitively, the counter and the fridge of the kitchen are small but very efficient in term of time access.

3.2 The tiled kernel matrix-vector product

3.2.1 A first implementation with 1d grid

The implementation introduced in Section 3.1.2 uses a one dimensional grid meaning that each thread will compute D scalars products. We recall here that we are interested on convolution-like operations and in practice we have

$1 \leqslant D \leqslant 3$.

It is well known that the performances of a matrix multiplication implemented in Cuda without using shared memory are poor. It yields to many redundant memory access as described in `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#shared-memory-in-matrix-multiplication-c-ab__using-shared-memory-improve-global-`

In our example on convolution, we assume, for simplicity, that D is equal to 1. Given an index $i \in \{1, \cdots, N\}$, the Cuda kernel will contains the instruction for the thread i to compute $\gamma_i = \sum_j k(x_i, y_j)b_j$. The first lines of the codes are:

Listing 3: A Cuda kernel that computes a convolution. This function is executed on the device. First part: Data are loaded in private local memory.

```
__global__ void KernelGpuConvOnDevice(float ooSigma2, float *x, float *y, float *beta, float *gamma, int N, int M) {
    // The current thread (numbered i) computes : gamma_i = sum_j k(x_i,y_j)*beta_j. So we need to get this index i which depends of the grid:
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Declare the shared memory. The amount of shared mem is given at the function call with the <<< >>> syntax
    extern __shared__ float SharedData[];
    // size of shift to access the data in shared mem. See below
    int inc = DIMPOINT + D;

    // One thread = One line = One x_i + One gamma_i + a whole bunch of "y_j".
    float xi[DIMPOINT], gammai[DIMVECT];

    // we will compute gammai only if i is in the range (ie the last block may not be "full"
    if(i<nx){
        // Load xi from device global memory to the local register of the current thread
        for(int k=0; k<DIMPOINT; k++){xi[k] = x[i*DIMPOINT+k];}

        // Make sure to initialize the result to zero
        for(int k=0; k<DIMVECT; k++) {gammai[k] = 0.0f;}
    }

    ...
```

Now, let $T \in \mathbb{N}^*$ and we may write the Euclidean division $M = n_t T + r$ with $r \in \mathbb{N}$ and $0 \leqslant r < T$. In practice the tile size is $T =$ `Blocksize.x` the number of threads in a block and then $n_t =$ `gridSize.x` as written lines 6 of Listing 2. The thread i should then compute

$$\gamma_i = \langle \boldsymbol{a}_{i\cdot}, \boldsymbol{b}_{\cdot 1} \rangle = \sum_{j=1}^{M} a_{i,j} b_{j,1} \qquad \text{where } a_{ij} = k(x_i, y_j)$$

$$= \sum_{t=1}^{n_t} \sum_{\ell=1}^{T} a_{i,(t-1)T+\ell}\, b_{(t-1)T+\ell,1} + \sum_{\ell=1}^{r} a_{i,n_tT+\ell}\, b_{n_tT+\ell,1},$$

where the computation of the scalar product is divided in chunks of size T as Figure 3 illustrates. We have to be careful with the last chunk as it contains r terms with r possibly less than T.

`KernelGpuConvOnDevice`: common data is loaded in shared memory

```
    ...

    // We start the loop over the tile
    for(int jstart = 0, tile = 0; jstart < ny; jstart += blockDim.x
        , tile++){

         // get the current column
         int j = tile * blockDim.x + threadIdx.x;

        // We use all the threads to load the tiles in the Shared
            memory

        if(j<ny){

            // ... but we have to check that we are in the range j<
                ny (we may be in the last columns of the last tile)

            // Pretty uneasy to read : we store yj and betaj
                interleaved, for better performance : SharedData =
                "[ y0, b0, y1, b1, y2, b2, ... ]"

            for(int k=0; k<DIMPOINT; k++){SharedData[threadIdx.x*
                inc+k] = y[j*DIMPOINT+k];}
            for(int k=0; k<D; k++) {SharedData[threadIdx.x*inc+
                DIMPOINT+k] = beta[j*D +k];}
        }

        // Crucial to wait  for all the thread to be done. If not,
            we may access the shared memory with false values of
            y_j or beta_j
        __syncthreads();

    ...
```

All the data of the current tile is now loaded: x_i and γ_i are in the local memory (more exactly in the register) and y_j and b_j are in the shared memory as they are needed by all the threads of the block. We can now perform the computations :

`KernelGpuConvOnDevice`: the tiled matrix multiplication

```
        ...
        //We are still in the loop over the tile
        if(i<nx){ // we compute gammai only if needed

            // declare convenient shortcuts to access the shared
                mem
            float *yj, *betaj;
            yj    = SharedData;
            // As y_j and beta_j are interleaved...
            betaj = SharedData + DIMPOINT;

            // jrel is the iterator within the tile.
            for(int jrel = 0; jrel < blockDim.x && jrel<ny-jstart;
                jrel++, yj+=inc, betaj+=inc) {

            // Compute the squared norm of (x_i-y_j):
            float r2   = 0.0f;
            for(int k=0; k<DIMPOINT; k++) {
                float temp    =  xi[k]-yj[k];
                r2     +=   temp*temp;
            }

            // The kernel function
            float s = KernelF(r2,ooSigma2);

            // The matrix multiplication is here : add the vector s
                *beta_j to gamma_i
            for(int k=0; k<DIMVECT; k++) {gammai[k] += s * betaj[k
                ];}

        }

        // Once the loop is over, the current tiled matrix product
            has been reduced to gamma_i
        __syncthreads(); // So make sure that no one's left behind
            ...


    // End of the for loop on tile.
    }
    ...
```

The results γ_i sit in the local memory of each thread. We then have to copy in the global memory so that the results of the CUDA kernel can be brought back to the host.

`KernelGpuConvOnDevice`: tranfer the result from local to global memory (still on the device)

```
    ...
    // Save the result in global memory.
    if(i<nx){
        for(int k=0; k<DIMVECT; k++){gamma[i*DIMVECT+k] = gammai[k
            ];}
    }

// end of cuda function
}
```

The entire functions listed in the section may be found in `cudaconv.cu` and `cudaconv.cx`.

3.2.2 A second implementation with a 2d grid

The previous implementation of the convolution uses a 1D computing grid, each block being given a single index corresponding to a range of i indices. It is possible, and useful in some situations, to use also a different computing

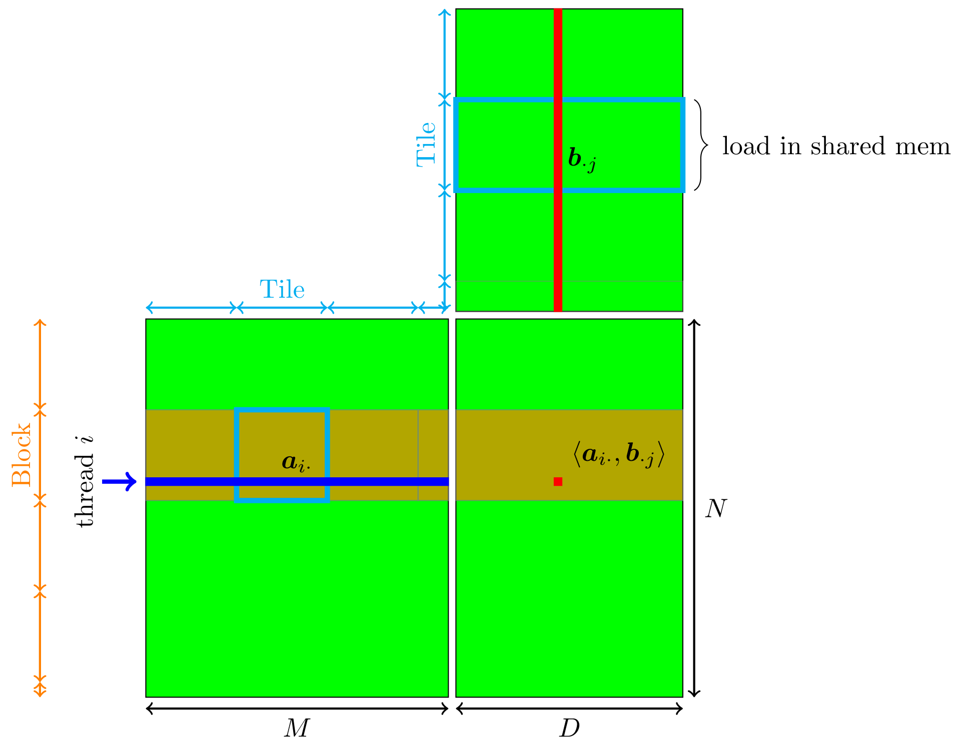

Tile
load in shared mem
$\boldsymbol{b}_{\cdot j}$
Tile
Block
thread i
$\boldsymbol{a}_{i\cdot}$
$\langle \boldsymbol{a}_{i\cdot}, \boldsymbol{b}_{\cdot j} \rangle$
N
M
D

Figure 3

scheme via a 2D grid, in which each block is given two indices corresponding to both i and j ranges of indices. In this new scheme, each block is devoted to compute, for its particular i index, only a partial sum for indices belonging to the corresponding range. This produces one output per i index and per range of j indices, and thus an additional reduction step is needed to sum up values.

The differences between the two methods is small because as we have seen previously the 1D grid method in fact already divides the j indices into sub-blocks due to shared memory limitations.

*** code description here : only present parts that are different from the 1D method : the gammaB output vector, the 2D grid specification, the reduction operation. ***

Figure 4 compares executions using both methods on a toy example. Each left and right diagram represents an ongoing computation of a kernel convolution with matrix size 9×11. Each color represents a particular running thread ; a total of 12 threads being available. The blocksize is fixed to 6. On the left, the 1D method. The first 6 threads are in Block 1, computing outputs for $i = 1$ to $i = 6$ and have currently loaded data vectors Y_7 to Y_{12} into shared memory. Threads 7 to 12 are in Block 2, have loaded data vectors Y_1 to Y_6, and threads 7 to 9 are computing outputs for $i = 7$ to $i = 9$. Here threads 8 and 9 are in

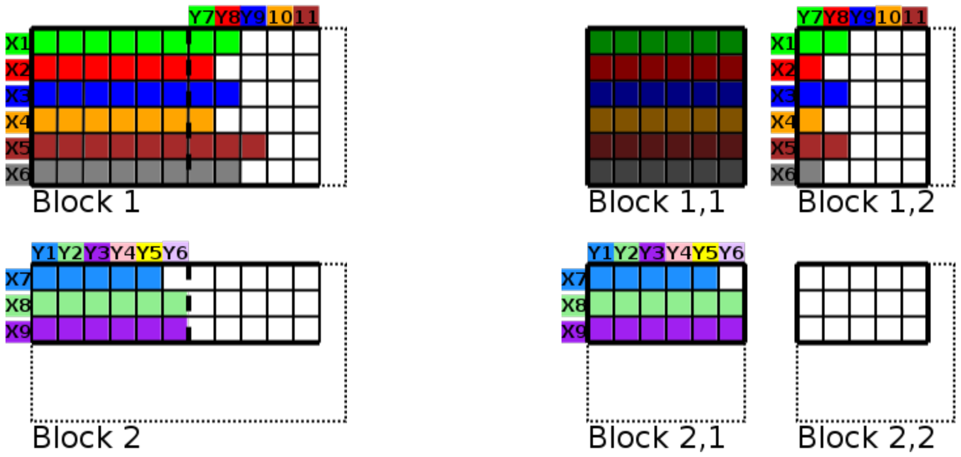


Figure 4: Comparison of executions of 1D and 2D grid methods.

fact waiting (at a ”syncthread” point) for thread 7 to finish the first sub-block before moving to the next. On the right, the 2D method. The process is similar, except that the grid of blocks is now 2D, and the computation of each sum is split into parts. Block 1,1 has terminated ; blocks 1,2 and 2,1 are active.

The advantage of the 2D method comes when the number of rows of the matrix is below the total number of threads, in which case remaining threads are available for populating new blocks in the 2D method, while they stay unused with the 1D method.

3.3 Implementation of a generic convolution, with order 2 derivatives

The kernel product operation

$$g_i = \sum_j k(x_i - y_j) b_j$$

may be implemented with various types of kernel functions k, including matrix-valued kernels, and various dimensions D and K. Moreover, since automatic differentiation is not available in CUDA , it is necessary to implement convolutions for first and second order derivatives of the kernel function. More precisely, the computation of the hamiltonian system and its adjoint differential necessitate to perform operations such as

$$g_i = \sum_j \nabla_x \left(a_i^T k(x_i - y_j) b_j \right)$$

and

$$g_i = \sum_j \nabla_x^2 \left(a_i^T k(x_i - y_j) b_j \right) c_j$$

Also, one may need to implement operations which have more complex formulation, such as products of kernels of the form

$$g_i = \sum_j k(x_i - y_j)h(a_i, b_j)$$

where h is another kernel function. In LDDMM applications, this is useful for deriving specific data attachment terms, such as the ones corresponding to the varifolds framework [].

All such operations can be implemented in CUDA following the same ideas presented in the previous section. From a programming viewpoint, one may then consider to write a generic code for convolutions of the type :

$$G_i = \sum_j F(X_i^1, \ldots, X_i^p, Y_j^1, \ldots, Y_j^q)$$

with any number of arguments p and q, and where each input and output have arbitrary dimensions $G_i \in \mathbb{R}^{d_g}$, $X_i^a \in \mathbb{R}^{d_{x,a}}$, $0 \leqslant a \leqslant p$, $Y_j^a \in \mathbb{R}^{d_{y,b}}$, $1 \leqslant b \leqslant q$. Using variadic templates, it is possible to factor out the implementation of function F from the GPU related implementation of the convolution. The F function itself can then be implemented in a separate file, and written as a member of a previously instantiated class if one uses a static member wrapper inside the class. This avoids having to pass extra parameters, such as the kernel size σ, through the device and host functions.

Host function. (inputs: pointers to data X, Y in CPU memory, and function F)

- Transfer input data X and Y from CPU memory to GPU global memory.
- define the computing grid
- call the device function
- transfer back result G from GPU global memory to CPU memory.

Device function. (inputs: pointers to data X, Y in GPU global memory, and function F)

- define indices i_t and j_t corresponding to thread id.
- set $G_{i_t} = 0$
- load variables $X_{i_t}^a$ into local memory, for all $1 \leqslant a \leqslant p$
- load variables $Y_{j_t}^b$ into shared memory, for all $1 \leqslant b \leqslant q$
- synchronize threads. This step is crucial to ensure all threads in the block have loaded the Y_j^b into shared memeory.
- loop through all indices j in the block : call the F function on $X_{i_t}^a$ and Y_j^b and add the result to G_{i_t}.

- synchronize threads
- transfer back result G_{i_t} into global memory

4 Normalizing kernels to gain mass awareness

The development framework presented in the last two sections is both *flexible* and *scalable*: Prototyping new ideas has never been so easy and we can only hope that researchers will make the most out of this newfound liberty. To give some food for thought to our readers, we now wish to showcase a simple model whose study was made accessible by autodiff libraries: the *normalized Hamiltonian* setting.

Our core idea is simple: Instead of using a kernel cometric

$$(K_q)_{i,j\in[\![1,N]\!]} \;=\; k(q^i - q^j) \tag{28}$$

in the computation of the Hamiltonian $\frac{1}{2}\langle p, K_q p\rangle$ of Eq. (2), couldn't we venture off the beaten track and use a normalized cometric

$$(\widetilde{K}_q)_{i,j\in[\![1,N]\!]} \;=\; \mathrm{diag}(\lambda_q)\cdot K_q \cdot \mathrm{diag}(\lambda_q), \tag{29}$$

where $\lambda_q \in \mathbb{R}_+^N$ is the unique nonnegative vector such that $\widetilde{K}_q$ is a bistochastic matrix? The existence, uniqueness and computability of such a vector λ_q is in fact guaranteed by the Sinkhorn Theorem [**?**], while the normalization of LDDMM metrics has been a niggling concern for a few years, more or less alleviated through the estimation of the global or local density of the point cloud representing the *source* shape "q_0".

Automatic differentiation is a game changer Prior to the development of automatic differentiation libraries, investing months of work into such a maverick line of thought would have been a risky long shot. Today, however, implementing and testing it from scratch is a matter of hours!

Preliminary numerical results are encouraging, and a genuine mathematical study reveals links between the normalized kernel $\widetilde{K}_q$ and the theory of Optimal Transport of measures. As it provides a neat example of what can be achieved with the numerical tools presented in sections 2 and 3, we expose in the next pages the preliminary results of Jean Feydy and Alain Trouvé on the subject. Note that we shall stick here to the practical tone of the previous sections: a thorough theoretical study of this new Hamiltonian formula is left for an ulterior, dedicated paper.

4.1 The LDDMM theory of kernel cometrics

The extrinsic point of view The *normalized Hamiltonian* algorithm can be seen as a natural "hack" of the classical LDDMM theory, which was historically developed as a *relaxation* of the affine registration setting: given a source shape X and a target shape Y, one is looking for an optimal matching φ in some fixed diffeomorphism group $G \subset \mathrm{Diff}(\mathbb{R}^D)$, such that the model $\varphi(X)$ is as close as possible to the target Y (according to a given data attachment formula).

If G is the group of rigid-body deformations, one falls back on Procustean analysis. If it is the group of linear changes of coordinates plus translations, we retrieve affine registrations. And if G is a group modeled on a Reproducible Kernel Hilbert Space of vector fields – endowed with the induced **right-invariant** metric – that is LDDMM theory.

The summary article [?] provides a clear introduction to this **extrinsinc** point of view, in which shape variability is explained through the action of diffeomorphisms of the *ambient space* with costs that do *not* depend on the shape being carried around. We stress that an LDDMM analysis is conducted as if the deformed shape $\varphi(X)$, instead of being a **measure** that carries a *mass*, was a mere **image** painted on an elastic canva – this is precisely what is implied by the geometers' term "right-invariant".

Purpose of this section In the next few pages, we intend to show that the normalization trick introduced in Eq. (29) breaks the right-invariance property of kernel cometrics, and replaces it with a behavior that mimics that of a *spatially regularized* transports of measures.

A slight change in the Hamiltonian computation is enough to turn an image registration routine into a measure transportation program. To make sense of this dramatic change of behavior, we now take the time needed to re-discover the LDDMM theory from an *intrinsic*, particular point of view. As we develop the theory underlying Eq. (2-7) in the limit setting of *landmarks*, we will argue that the LDDMM framework is popular because it is the theory of continuous deformations of point clouds with the *lowest computational cost.* We can then justify the Hamiltonian's normalization trick by the fact it provides, at a reasonable premium, a Wasserstein-like behavior that makes the theory relevant for the transport of general measures.

Optimal Transport of landmarks First, we introduce the classical problem of Monge-Wasserstein, concerned with the transport of *independent* landmarks. Let our shapes X and Y be represented as point clouds $(x^1, \dots, x^M)$ and $(y^1, \dots, y^M)$ in $\mathbb{R}^D$, and let us look for a permutation $\sigma : [\![1, M]\!] \rightarrow [\![1, M]\!]$, a collection of *transport* paths $\gamma^m : t \in [0, 1] \mapsto \gamma^m_t$ that minimizes the overall L^2-transport cost

$$\ell^2(\gamma) \;=\; \sum_{m=1}^{M} \int_{t=0}^{1} \|\dot\gamma^m_t\|^2 \, \mathrm{d}t \qquad (30)$$

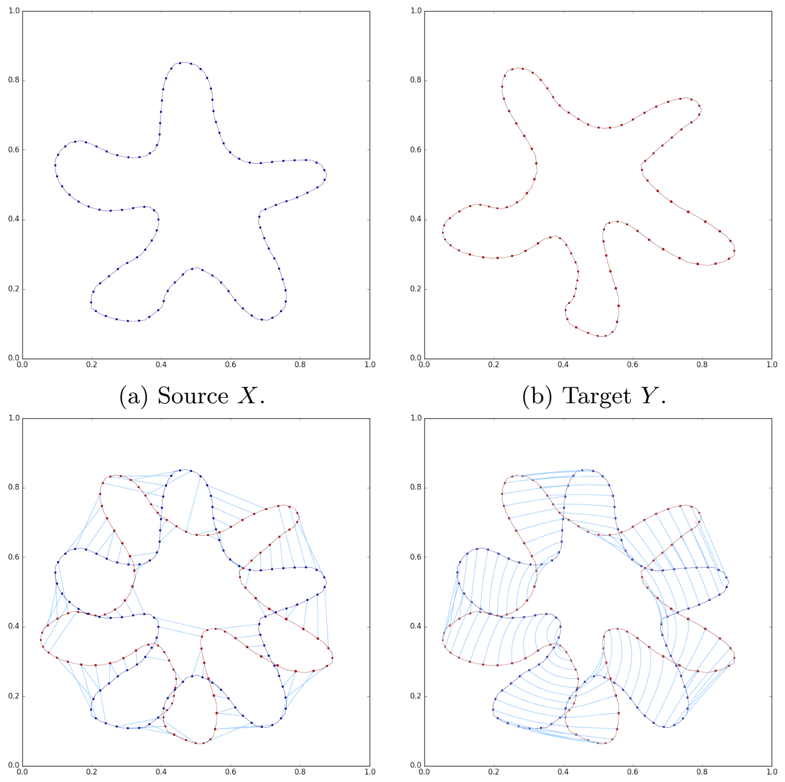
under the constraint that every point of X is matched to a point of Y:

$$\forall\ m, \qquad \gamma^m_0 \;=\; x^m \;\text{ and }\; \gamma^m_1 \;=\; y^{\sigma(m)}. \qquad (31)$$

In this decorrelated setting, particles γ^m_t are always better off traveling in straight lines: the problem can therefore be reduced to the combinatorial search of a permutation $\sigma : [\![1, M]\!] \rightarrow [\![1, M]\!]$ minimizing

$$C^{X,Y}(\sigma) \;=\; \sum_{m=1}^{M} \|x^m - y^{\sigma(m)}\|^2, \qquad (32)$$

and we call σ the optimal labeling between X and Y.



(a) Source X. (b) Target Y.

(c) OT matching . (d) LDDMM matching.

Figure 5: Difference between an Optimal Transport and an LDDMM registration between two curves. As the Monge-Wasserstein problem of Eq. (30) does not promote the correlation of neighboring particles, it completely discards the topology of shapes and breaks up the source curve into pieces (even if we relax the bijectivity constraint of Eq. (31), as shown here). The LDDMM theory induces a higher computational cost and the loss of convexity, but alleviates the topological issues by guaranteeing a *diffeomorphic* shape registration.

Regularizing Optimal Transport trajectories As evidenced by Figure 5, the complete independence of particles is not a reasonable modeling prior for shape analysis: discarding the topology of the shape matching problem allows one to get efficient, convex solvers... Which may very well *tear* shapes apart.

Hence, optimal transport should be *regularized spatially* as neighboring points get correlated to each other. A naive way to prevent tears would be to replace the cost $C^{X,Y}$ of Eq. (32) with a continuity-inducing formula :

$$C_k^{X,Y}(\sigma) = \underbrace{\sum_m \|x^m - y^{\sigma(m)}\|^2}_{\text{Displacement cost}} + \underbrace{\sum_{m,m'} k(x^m, x^{m'}) \cdot \| y^{\sigma(m)} - y^{\sigma(m')}\|^2}_{\text{Regularization cost}}, \qquad (33)$$

where $k(x, y)$ is a kernel **neighborhood** function – say, a Gaussian of given deviation σ. Even better, one could symmetrize the regularization cost and use

$$C^{X,Y}_{k,\text{sym}}(\sigma) = \underbrace{\sum_m \|x^m - y^{\sigma(m)}\|^2}_{\text{Displacement cost}} + \underbrace{\frac{1}{2}\sum_{m,m'} k(x^m, x^{m'}) \cdot \|y^{\sigma(m)} - y^{\sigma(m')}\|^2}_{X \to Y \text{ regularization cost}} + \underbrace{\frac{1}{2}\sum_{m,m'} k(y^m, y^{m'}) \cdot \|x^{\sigma^{-1}(m)} - x^{\sigma^{-1}(m')}\|^2}_{Y \to X \text{ regularization cost}}. \tag{34}$$

This "handcrafted" cost is **symmetric** and could be satisfying from a computational point of view. Unfortunately, it lacks a proper dynamical interpretation and puts too much emphasis on the source and target shapes, to the detriment of the interpolating trajectory.

Using Riemannian geometry to handle a continuous population of shapes This is a problem as most practical applications work with shapes that are sampled from *continuous* populations. For instance, X and Y may represent two snapshots of the same subject shape γ_t at different times t_0 and t_1: to prevent the introduction of an acquisition bias, we need to use a deformation model that weighs equivalently all the instants of the interval $[t_0, t_1]$.

We must therefore consider an infinitesimal, dynamic version of the model of Eq. (34) and look for a collection of paths γ^m from X to Y minimizing

$$C_k(\gamma) = \int_0^1 \Big[\underbrace{\sum_m \|\dot\gamma^m_t\|^2}_{\text{Displacement cost}} + \underbrace{\sum_{m,m'} k(\gamma^m_t, \gamma^{m'}_t) \cdot \|\dot\gamma^m_t - \dot\gamma^{m'}_t\|^2}_{\text{Regularization cost}} \Big] \mathrm{d}t, \tag{35}$$

under the transportation constraint of Eq. (31). According to this model, particles will move optimally if they are both *lazy* and *gregarious* with respect to their neighbors, as defined by the kernel function k.

A remarkable property of the infinitesimal cost integrated above is that it is *quadratic* with respect to the velocity $\dot\gamma_t$ and *smooth* with respect to the position γ_t. With $\gamma_t = (\gamma^1_t, \dots, \gamma^M_t) \in Q = \mathbb{R}^{M\times D}$, we can thus write the overall cost as

$$C_k(\gamma) \;=\; \int_0^1 \langle \dot\gamma_t \;,\; g_{\gamma_t}\dot\gamma_t\rangle_2 \,\mathrm{d}t \tag{36}$$

where $\langle \cdot\,, \cdot\rangle_2$ is the canonical euclidean product of $Q = \mathbb{R}^{M\times D}$ and $g : Q \to \mathbb{R}^{MD\times MD}$ is a smooth field of symmetric positive definite matrices.

Interestingly, **trying to regularize spatially the Monge-Wasserstein problem has led us to introduce Riemannian concepts**: the optimal deformations of the continuous model (36) are nothing but geodesics on the space of landmarks $\mathbb{R}^{M\times D}$ endowed with a Riemannian metric g_q such that, if

$\delta q = v\delta t$ is a small deformation of a point cloud q,

$$\frac{\big(\mathrm{d}_g(q \to q + v\cdot\delta t)\big)^2}{(\delta t)^2} + o(1) = \sum_m \|v^m\|^2 + \sum_{m,m'} k(q^m, q^{m'}) \cdot \|v^m - v^{m'}\|^2 \tag{37}$$

$$= \langle v\,,\, g_q v\rangle_2 \;=\; \|v\|^2_{g_q}\,. \tag{38}$$

Shooting geodesics Riemannian geometry is a convenient theoretical setting, which appears to be relevant for shape analysis as the natural framework for regularized OT. With practical applications, however, comes the question of tractability: what are the metrics g_q on the space of landmarks whose *geodesics* are easy to compute? At first glance, geodesics are locally "straight" curves governed by a non-trivial second order ODE (involving Christoffel symbols) on the *tangent bundle* TQ, with coordinates

$$(q_t, v_t) \;=\; (\gamma_t, \dot\gamma_t). \tag{39}$$

This could be pretty damning from a practical point of view, as computing Christoffel symbols involves the computation of both the metric tensor g_q *and* its inverse K_q. Fortunately though, physicists involved with classical mechanics have noted more than a century ago that the equation of geodesics can be considerably simplified by a locally linear *change of variables.* Instead of using the coordinates (q_t, v_t) of Eq. (39), one should work on the **cotangent bundle** and use :

$$(q_t, p_t) \;=\; (q_t, g_{q_t} v_t). \tag{40}$$

We define the *cometric* tensor $K_q = g_q^{-1}$, and introduce the general Hamiltonian function $H(q,p) = \frac{1}{2}\langle p\,, K_q p\rangle$ so that

$$\frac{1}{2}\langle v_t\,,\, g_{q_t} v_t\rangle \;=\; \underbrace{\frac{1}{2}\|\dot\gamma_t\|^2_{\gamma_t}}_{\text{Kinetic energy}} \;=\; \frac{1}{2}\langle p_t\,,\, K_{q_t} p_t\rangle \;=\; H(q_t, p_t). \tag{41}$$

The following theorem, which links geodesic on (Q, g_q) to the sole Hamiltonian function H, is of primary importance:

Theorem 1 (Hamilton, 1833)**.** *Assume that g_q endows the space of landmarks $Q = \mathbb{R}^{MD}$ with a Riemannian structure. Then, with the notations described above, a smooth curve γ_t is a geodesic if and only if the lifted cotangent trajectory (q_t, p_t) follows the Hamiltonian equation :*

$$\begin{cases} \dot q_t &= +\frac{\partial H}{\partial p}(q_t, p_t) &= +K_{q_t} p_t \\ \dot p_t &= -\frac{\partial H}{\partial q}(q_t, p_t) &= -\frac{1}{2}\partial_q (p_t, K_q p_t)(q_t) \end{cases}. \tag{42}$$

In the cotangent **phase space**, the geodesic equation is therefore given by the flow along the *symplectic gradient* of H which can (very informally) be written as :

$$X(q,p) = \begin{pmatrix} +\frac{\partial H}{\partial p}(q,p) \\ -\frac{\partial H}{\partial q}(q,p) \end{pmatrix} = \text{“Rot}_{-90^\circ}\text{”}\big(\nabla H(q,p)\big). \tag{43}$$

Summary This theorem is the theoretical basis on which relies the LDDMM shooting routine "Shoot" presented in Eq. (4), which implements the Riemannian exponential. Now, if we recapitulate the exposition of the last paragraphs, we must recall that our *prime motivation* is to find a principled and practical way of regularizing Optimal Transport. As shown by Eq. (36), this amounts to finding a Riemannian metric on the space of landmarks which is *tearing-adverse*, and for which the *computation of geodesics* is as cheap as possible.

The important lesson given by the Theorem 1 (Hamilton) is that the computational cost of shooting a geodesic on a Riemannian manifold is directly related to that of computing the **cometric** $K_q = g_q^{-1}$ and its derivatives. **From a practical point of view, the tensor K_q is more important than its inverse, the metric g_q.** This is a major change of perspective, as it discourages us from using the "naive" regularizing metric of Eq. (35).

GPUs and kernel cometrics In today's landscape, scientific computing is heavily biased towards operations that can be *parallelized* efficiently. On a GPU, given a point cloud $q = (q^1, \dots, q^M) \in \mathbb{R}^{M\times D}$, computing a kernel matrix

$$k_q = \begin{pmatrix} k(q^1,q^1) & k(q^1,q^2) & \cdots & k(q^1,q^M) \\ k(q^2,q^1) & k(q^2,q^2) & \cdots & k(q^2,q^M) \\ \vdots & \vdots & \ddots & \vdots \\ k(q^M,q^1) & k(q^M,q^2) & \cdots & k(q^M,q^M) \end{pmatrix} \tag{44}$$

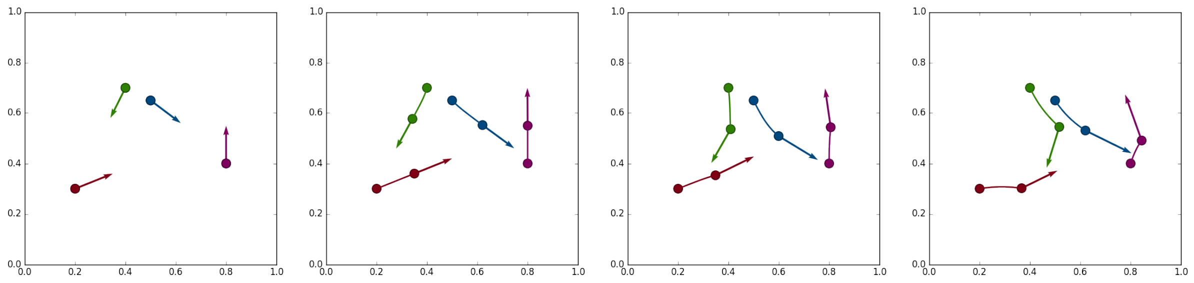
has nearly become an *atomic operation*. The simple yet efficient idea behind the LDDMM theory is then to use this very matrix k_q, appropriately expanded into an `(MD,MD)` shape by using a Kronecker product

$$K_q \;=\; k_q \otimes \mathrm{I}_D \tag{45}$$

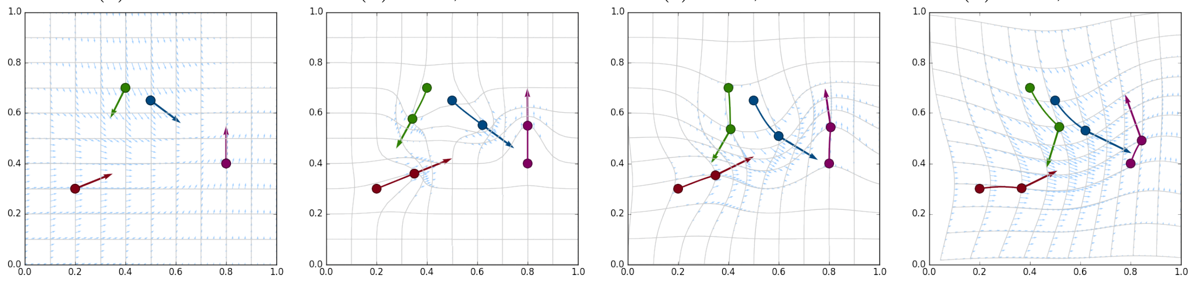
so that

$$H(q,p) \;=\; \frac{1}{2}\langle p\,, K_q p\rangle \;=\; \frac{1}{2}\sum_{i,j=1}^{M} k(q^i,q^j)\cdot\langle p^i\,,\, p^j\rangle_{\mathbb{R}^D}\,. \tag{46}$$

Using a translation and rotation invariant kernel $k(x,y) = k(\|x-y\|)$, we retrieve the expression presented Eq. (2). **In a computational sense, this formula defines the simplest family of cometrics and geodesics on the space of point clouds.**



(a) $t = 0$. (b) $t = 1$, `s=.25`. (c) $t = 1$, `s=.35`. (d) $t = 1$, `s=.50`.



(e) $t = 0$, `s=.50`. (f) $t = 1$, `s=.25`. (g) $t = 1$, `s=.35`. (h) $t = 1$, `s=.50`.

Figure 6: First line: in the unit square, results (q_1, p_1) of the "Shoot" LDDMM routine on the initial situation (q_0, p_0) depicted in (a), for various choices of Gaussian kernels $k : x \mapsto \exp(-x^2/2s^2)$. Second line: the Theorem 2 (Reduction principle) allows us to lift the landmarks trajectory in a manifold of diffeomorphisms of the plane. Starting from an initial state (e), the moving (landmark, momentum) pairs (q_t, p_t) generate a flow field $v_t = k \star p_t$ (in blue) that deforms the ambiant space, carrying around the Identity grid.

Understanding the kernel cometrics We could be content with this algorithmic reasoning. Since the landmarks q^i only interact with each other through the kernel function k, the latter's smoothness should prevent tears in the matchings. The geodesics shown in the first line of Figure 6 tend to confirm this intuition: as the radius of the kernel function k increases, neighboring points behave more and more like a cohesive unit.

Surprisingly, there is more to this than a simple coincidence. The little miracle of the LDDMM theory is that the "kernel cometric setting" – which is the most favorable one from a computational perspective – is also gifted with a very strong theoretical structure: the landmarks trajectories defined by the shooting routine of Eq. (4) can be canonically lifted to a manifold G_k of diffeomorphisms of the ambient space, whose smoothness is directly controlled by the kernel function.

Kernel metrics on vector fields Indeed, let $k : \mathbb{R}^D \to \mathbb{R}$ be a *kernel* function with a positive Fourier transform, that is, such that $\widehat{k}(\omega) \in \mathbb{R}_+^\star$ for all frequencies

ω. If $v : \mathbb{R}^D \to \mathbb{R}^D$ is a vector field on the ambient space, we define the k-norm

$$\|v\|_k^2 \;=\; \int_{\omega \in \mathbb{R}^D} \frac{1}{\widehat{k}(\omega)}\, |\widehat{v}(\omega)|^2 \,\mathrm{d}\omega \quad \in \quad \mathbb{R} \cup \{+\infty\}. \tag{47}$$

We can then consider the set of "k-smooth" vector fields

$$V_k = \{ v \mid \|v\|_k < \infty \}\,, \tag{48}$$

endowed with the Hilbertian norm $\|\cdot\|$. **[ToDo: quelles sont les conditions sur k pour que V_k soit complet ?]** One technical assumption should be made: we will assume that k is smooth enough to let the pointwise evaluation $\delta_x : v \mapsto v(x)$ be a continuous mapping from $(V, \|\cdot\|_V)$ to $\mathbb{R}$. This allows us to link with the theory of *Reproducing Kernel Hilbert Spaces* and guarantee the integrability results to come (all of which can be found, with demonstrations, in [**?**]).

Integrating trajectories of finite energy We now remark that if $(v_t) \in L^2([0,1], V_k)$ is a time-varying vector field such that

$$\ell_k^2(v) \;=\; \int_0^1 \|v_t\|_k^2 \,\mathrm{d}t \;<\; \infty, \tag{49}$$

then, according to Picard-Lindelöf theorem, we can *integrate its flow*, and find a unique trajectory φ_t of *diffeomorphisms* such that for every point $x \in \mathbb{R}^D$ and time $t \in [0,1]$:

$$\varphi_0(x) = x \qquad \text{and} \qquad \frac{\mathrm{d}}{\mathrm{d}t}\left[\varphi_t(x)\right] = v_t \circ \varphi_t(x), \tag{50}$$

$$\text{i.e.} \qquad \varphi_0 \;=\; \mathrm{Id}_{\mathbb{R}^D} \qquad \text{and} \qquad \varphi_t \;=\; \mathrm{Id}_{\mathbb{R}^D} + \int_{s=0}^{t} v_s \circ \varphi_s \,\mathrm{d}s. \tag{51}$$

The reduction principle As diffeomorphisms carry around *images*, *measures* and landmarks, we could try to minimize over $L^2([0,1], V_k)$ the cost

$$C^2(\varphi_1) \;=\; \ell_k^2(v) \;=\; \int_0^1 \|v_t\|_k^2 \,\mathrm{d}t, \tag{52}$$

under the constraint that $\varphi_1(X) = Y$. This is an infinite-dimensional minimization problem, related to k through the k-norm of Eq. (47) which coerces the integrable flows v_t into having Fourier spectrums comparable to that of k. Surprisingly, the critical points of this functional coincide with the Riemannian geodesics in the space of landmarks Q – endowed with the kernel cometric K_q of Eq. (45) – as stated by the following theorem:

Theorem 2 (Reduction Principle, **[ToDo: ref.]**)**.** *Let q_t be a time-dependent point cloud, and k be a kernel function which is smooth enough. Then, the two propositions below are equivalent :*

1. *q_t is a geodesic for the kernel cometric K_q, with momentum p_t associated to the Hamiltonian*
$$H(q,p) \;=\; \frac{1}{2}\langle p, K_q p\rangle\,. \tag{53}$$

2. *q_t is carried around by a locally optimal diffeomorphic trajectory φ_t = Flow(v_t), and we have*
$$v_t \;=\; k \star p_t \qquad \textit{i.e.} \qquad v_t(x) \;=\; \sum_{m=1}^{M} k(q_t^m, x)\, p_t^m. \tag{54}$$

The three pillars of LDDMM Figure 6 provides an illustration of the reduction principle. The latter allows one to make sense of the kernel cometric, and completes our brief re-exposition of the LDDMM theory – which was originally developed from the diffeomorphic point of view, towards the case of landmarks [**?**]. As we have shown in the previous pages, the relevance of the LDDMM framework relies on three pillars :

Hamilton's theorem, which shows that geodesics on a Riemannian manifold can be computed as soon as the *cometric* tensor is simple enough.

The current availability of GPUs, which promotes the fully parallelizable *kernel* cometric in applications.

The reduction principle, which links the kernel cometric to the action of k-smooth diffeomorphisms on the ambient space.

4.2 Normalized kernels for landmarks transport

Now that the theory underlying kernel cometrics has been briefly recalled, we introduce the *normalized* Hamiltonian in the very simple case of landmarks. But first, we have to point out some of the shortcomings of the classical LDDMM setting as far as the transport of measures is concerned.

Did we bridge the gap between Procustes and Optimal Transport? No, we didn't. At first glance, it looks as though the LDDMM infinitesimal metric $\|v\|_k$ on vector fields (Eq. (47)), defined for Gaussian kernels $k : x \mapsto \exp(-\|x\|_2^2/s^2)$ interpolates between the translation-only metric $\|\cdot\|_{s=\infty}$ and a Wasserstein-like norm $\|\cdot\|_{s=0}$, respectively defined as

$$\|\,v\,\|_{s=\infty} \;=\; \begin{cases} \|w\|_2^2 & \text{if } \forall\; x \in \mathbb{R}^D, v(x) = w \\ +\infty & \text{otherwise} \end{cases}, \tag{55}$$

$$\|\,v\,\|_{s=0} \;=\; \int_{\mathbb{R}^D} \|v(x)\|_2^2 \,\mathrm{d}x. \tag{56}$$

Therefore, one could be tempted to assume that as the radius s of the kernel k tends to zero, LDDMM geodesics degenerate to Optimal Transport solutions... Unfortunately, this is completely untrue. Problem is, the hand-waving argument presented above does not take into account the fact that the action of LDDMM diffeomorphisms is fully extrinsic, whereas the Monge-Wasserstein cost of Eq. (30) keeps track of the *mass*, the number of landmarks that are currently being carried aroud.

Right-invariant metrics and cigar-shaped geodesics The most blatant

A toy example We provide here a minimal generalization of the above example to the case of

On considère un état $q \in \mathcal{L}_6^2$, donné par six points $(q^1, q^2, q^3, q^4, q^5, q^6)$ du plan sur lesquels on fait les hypothèses simplificatrices suivantes :

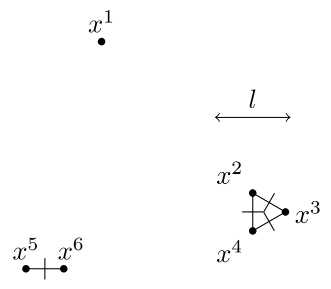
- Les points se répartissent en trois groupes de taille variable : (q^1), (q^2, q^3, q^4), et (q^5, q^6), qui sont éloignés les uns des autres à une distance très grande devant l'échelle l du noyau.
- q^5 et q^6 sont à une distance d donnée l'un de l'autre.
- Le groupe (q^2, q^3, q^4) est un triangle équilatéral de côté d.

On peut voir Figure **??** une telle situation, qui modélise de manière simple la présence d'amas de masses variées dans l'image.

Sous ces hypothèses, on peut écrire très simplement la matrice de noyau réduite :

$$k_q = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & 1 & a & a & \cdot & \cdot \\ \cdot & a & 1 & a & \cdot & \cdot \\ \cdot & a & a & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & a \\ \cdot & \cdot & \cdot & \cdot & a & 1 \end{pmatrix}, \tag{57}$$

où $a = \exp(-d^2/2\,l^2) \in [0,1]$, et où l'on a remplacé les termes négligeables par des points.



À la limite, la matrice de noyau réduite k_q puis son homologue vectoriel K_q

sont des matrices diagonales par blocs, remplies de blocs élémentaires

$$B_n(a) = (1-a)\cdot I_n + a\cdot(1)(1)^{\mathsf{T}} = \begin{pmatrix} 1 & & & & \\ & 1 & & (a) & \\ & & \ddots & & \\ & (a) & & 1 & \\ & & & & 1 \end{pmatrix}. \tag{58}$$

Pour trouver la métrique, inverser k_q, la clé est donc de savoir inverser les blocs $B_n(a)$ associés aux amas de n points.

Lemma 1 (Pertinence des métriques à noyaux, version discrète)**.** *On note $e = (1)/\|(1)\|_2$ le vecteur unitaire constant de taille n, rempli de $1/\sqrt{n}$. L'inverse de $B_n(a)$ est alors donné par :*

$$(B_n(a))^{-1} = \frac{1}{1+(n-1)\,a}ee^{\mathsf{T}} + \frac{1}{1-a}(I_n - ee^{\mathsf{T}}). \tag{59}$$

Autrement dit, pour tout vecteur $v = (v^1,\dots,v^n)$ que l'on décompose en

$$\begin{aligned} v &= e\,(e^{\mathsf{T}}v) \ + (v - e\,(e^{\mathsf{T}}v)) \quad (60)\\ &= \quad v_{moy} \quad + \quad v_{var} \quad, \quad (61)\end{aligned}$$

une partie "moyenne" constante et une partie de somme nulle, la variance. On a

$$v^{\mathsf{T}}(B_n(a))^{-1}v \quad = \quad \frac{1}{1+(n-1)\,a}\ \|v_{moy}\|_2^2 \ + \ \frac{1}{1-a}\ \|v_{var}\|_2^2 \tag{62}$$

Proof. Il suffit d'écrire la décomposition spectrale de $B_n(a)$, i.e. trouver les axes de l'ellipsoïde associé :

$$B_n(a) = (1+(n-1)\,a)\,ee^{\mathsf{T}} + (1-a)\,(I_n - ee^{\mathsf{T}}). \tag{63}$$

$B_n(a)$ possède donc une valeur propre $1+(n-1)\,a$ selon la direction e, et agit comme $(1-a)$ fois l'identité sur l'orthogonal. Pour trouver l'inverse, il suffit alors d'inverser les valeurs propres – qui correspondent ici aux valeurs singulières, il n'y a vraiment aucun piège. □

Interprétation Rappelons que $a = \exp(-(d/l)^2/2)$, où d est le diamètre de l'amas et l l'échelle du noyau utilisé par la cométrique. Il vaut donc 1 si $d << l$, et décroît jusqu'à 0 lorsque $d >> l$. L'équation (62) est extrêmement précieuse, car elle contient en germe tout la dynamique associée à la cométrique K_q.

D'abord, elle met en évidence un fait rassurant : le rôle particulier joué par les champs de vitesses constants, "colinéaires". Le coût associé à un champ de vitesses sur l'amas est donc la somme d'un terme de *translation*, proportionnel à $\|v_{\text{col}}\|_2^2$, et d'un terme de *régularisation* pénalisant la non-uniformité en $\|v_{\text{ncol}}\|_2^2$.

Lorsque le diamètre d de l'amas est bien supérieur à l'échelle du noyau, a est petit devant 1. On a alors

$$\frac{1}{1+(n-1)\,a} \ \simeq\ 1\ \simeq\ \frac{1}{1-a}, \tag{64}$$

l'équilibre entre les deux pénalisations. Le coût $v^\intercal (B_n(a))^{-1} v$ est simplement égal au coût Wasserstein $v^\intercal v = \|v\|_2^2$ du transport décorrélé. On dira que les particules n'*interagissent* pas ensemble.

À l'inverse, si le diamètre d de l'amas est petit devant l, si le noyau voit les points de l'amas comme quasiment confondus, on aura $a \simeq 1^-$ et par suite

$$v^\intercal (B_n(a))^{-1} v \ \ \simeq\ \ \frac{1}{n}\,\|v_{\text{moy}}\|_2^2 \ +\ \infty\ \|v_{\text{var}}\|_2^2\,. \tag{65}$$

Lorsque les points sont l-proches les uns des autres, qu'ils interagissent entre eux au sens de k, on a donc combinaison de deux effets : la sur-pénalisation des non-uniformités, des déchirures, avec le poids quasi-infini devant v_{var}; la mutualisation des coûts de translation, avec une atténuation en $1/n$ du coût quadratique sur v_{moy}. **Tout se passe donc comme si notre amas de n particules se réduisait à un seul atome, très difficile à éclater mais aussi facile à transporter qu'une particule seule.**

Retour sur la forme globale, combinaison de plusieurs amas Si l'on revient au nuage de la Figure **??**, on peut maintenant exprimer simplement la métrique g_q associée par k_q aux déformations infinitésimales de sa géométrie :

$$g_q = (k_q)^{-1} = \left(\begin{array}{c|c|c} 1 & \cdot\ \ \cdot\ \ \cdot & \cdot\ \ \cdot \\ \hline \cdot & & \cdot\ \ \cdot \\ \cdot & B_3(a)^{-1} & \cdot\ \ \cdot \\ \cdot & & \cdot\ \ \cdot \\ \hline \cdot & \cdot\ \ \cdot\ \ \cdot & \\ \cdot & \cdot\ \ \cdot\ \ \cdot & B_2(a)^{-1} \end{array}\right) \tag{66}$$

(on se dispense ici d'écrire le produit de Kronecker avec I_d, qui impose simplement de sommer les coûts sur les dimensions). Les trois amas sont donc complètement indépendants, ce qui n'est pas une surprise puisqu'ils sont décorrélés au sens de k.

Interprétation Étant donné un champ de vitesses $v = (v^1, v^2, v^3, v^4, v^5, v^6)$, comment se trouve-t-il pénalisé par g_q ? Les comportements limites se retrouvent aussi facilement que pour un amas simple. Aussi, lorsque $d >> l$ et donc $a \simeq 0$, on a :

$$v^\intercal g_q v \ \simeq\ \|v^1\|_2^2 + \|v^2\|_2^2 + \|v^3\|_2^2 + \|v^4\|_2^2 + \|v^5\|_2^2 + \|v^6\|_2^2. \tag{67}$$

Par contre, si $d << l$, alors un champ de coût fini s'écrit nécessairement :

$$(v^1, v^2, v^3, v^4, v^5, v^6) = (w^1, w^2, w^2, w^2, w^3, w^3), \tag{68}$$

avec w^i la “vitesse de groupe” de l’amas i, et :

$$\begin{aligned} v^\top g_q v &= \|v^1\|_2^2 + \frac{1}{3}\big(\|v^2\|_2^2 + \|v^3\|_2^2 + \|v^4\|_2^2\big) + \frac{1}{2}\big(\|v^5\|_2^2 + \|v^6\|_2^2\big) \quad (69) \\ &= \|w^1\|_2^2 + \|w^2\|_2^2 + \|w^3\|_2^2. \quad (70) \end{aligned}$$

Cahin-caha, on peut donc se forger une certaine intuition des trajectoires géodésiques “typiques”, qui tiennent groupés les k-amas.

Normalizing kernels to straighten geodesics

A modified Sinkhorn algorithm

4.3 A new diffeomorphic setting for measures

Measure-dependent RKHS

Including translations in our model

Computing the local density

Computing the velocity field

`PyTorch` implementation

Examples

5 Conclusion