

KeOps library: Scalable Kernel Methods [for Geometric Data]

Benjamin Charlier (MIAT, INRAE), Jean Feydy (INRIA), Joan Glaunès (Univ. Paris Cité)
CESBIO, Toulouse — June, 2025.

The word *kernel* may have different meanings

- Mathematical Analysis: a function $X \times X \rightarrow Y$ interaction between points
- Linear algebra: the null set of a linear operator
- Programming: a short piece of code, a function. Usually intend to be run on GPU. A “cuda kernel”, “openCL kernel”, ...

Motivations: shapes analysis using kernels

KeOps and symbolic matrices

Autodiff engine

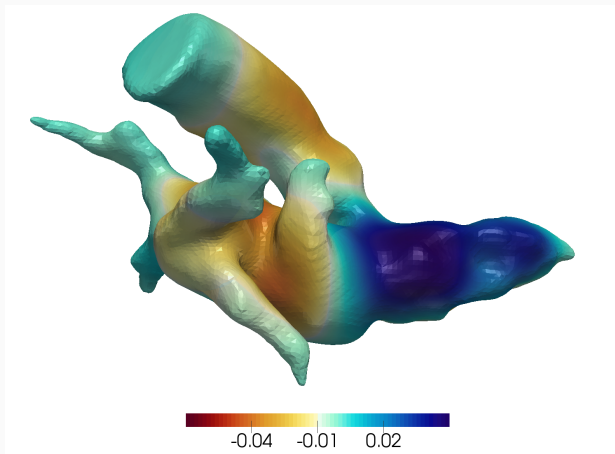
- Introduction

- KeOps autograd

Advanced linear algebra operations

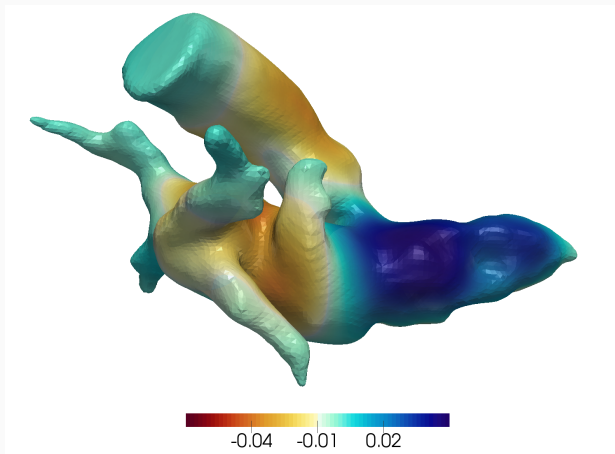
Motivations: shapes analysis using
kernels

LDDMM: generate Diffeomorphic deformations



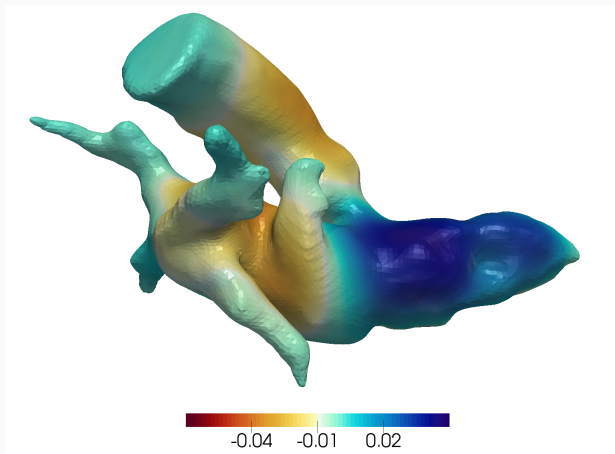
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



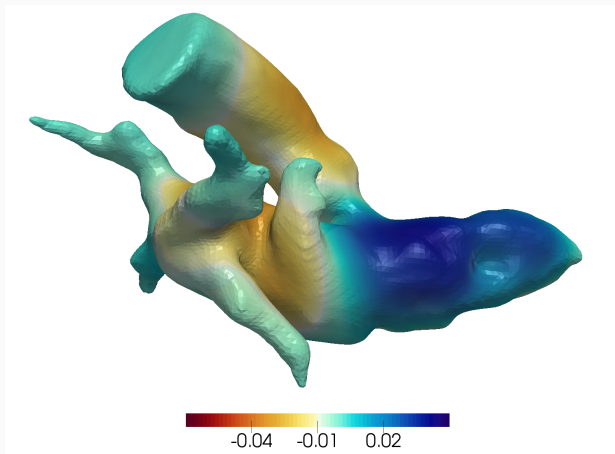
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



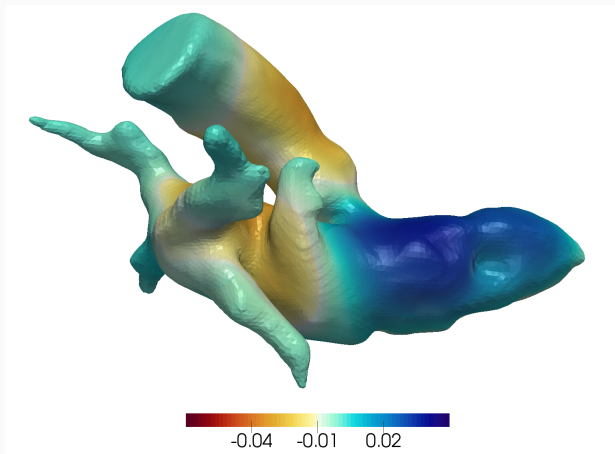
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



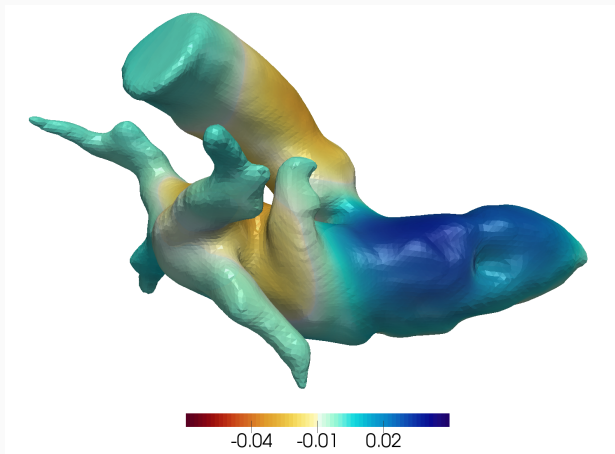
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



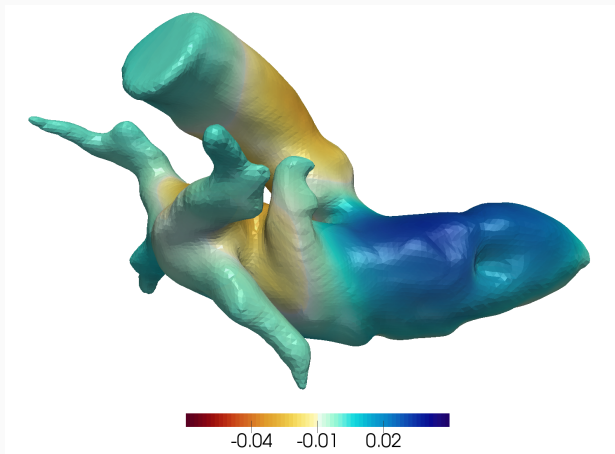
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



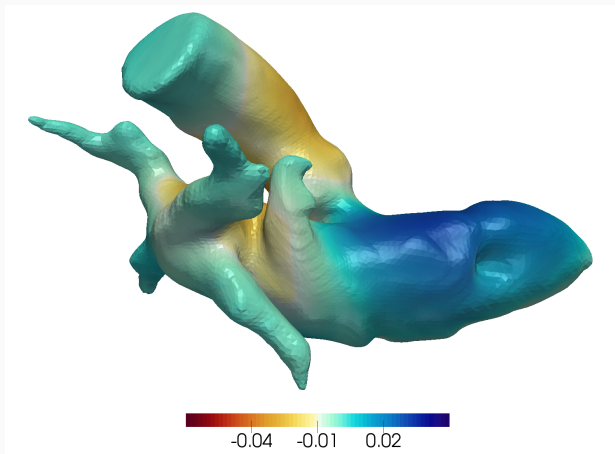
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



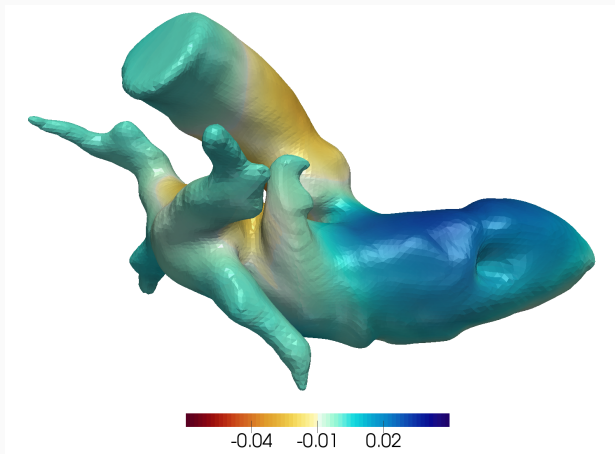
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



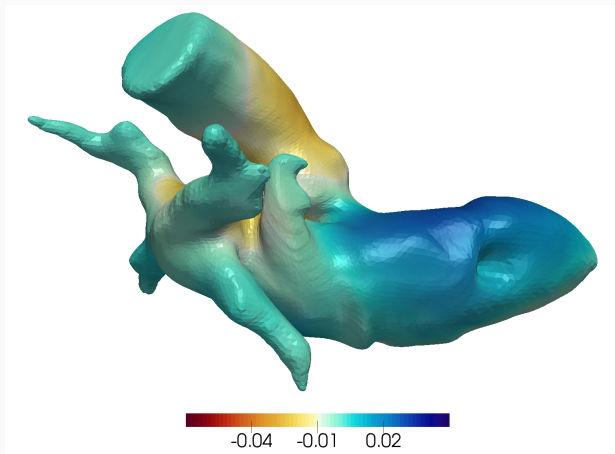
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



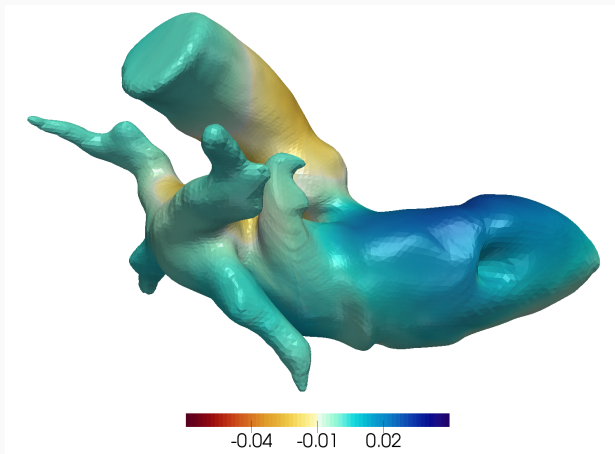
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



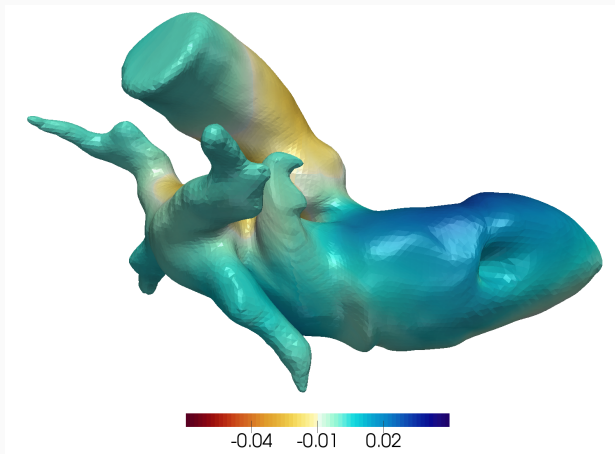
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



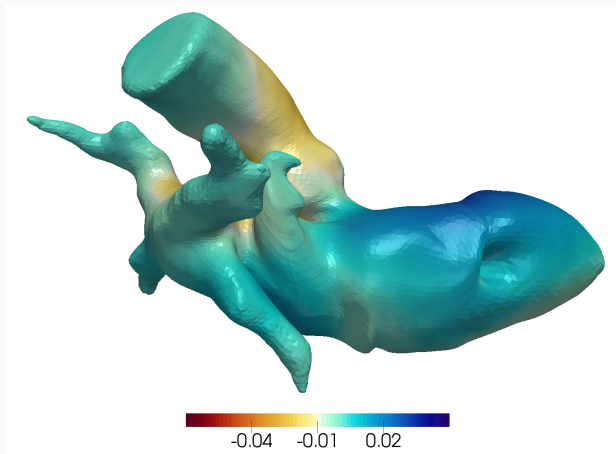
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



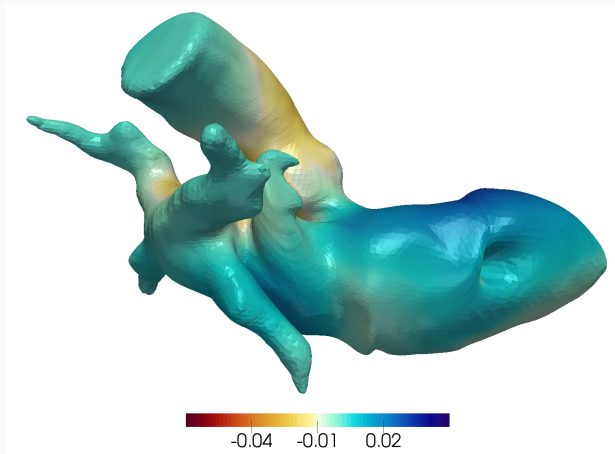
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



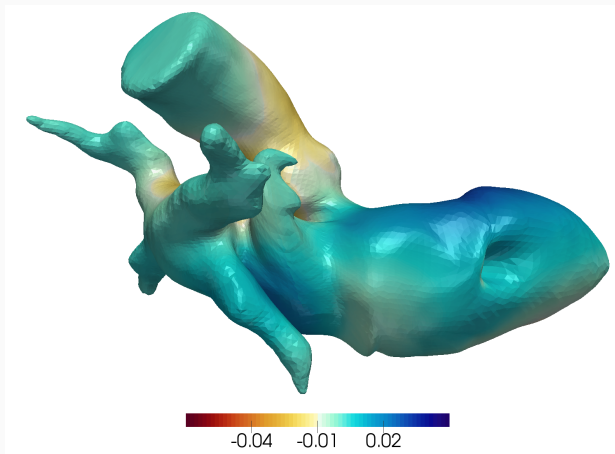
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



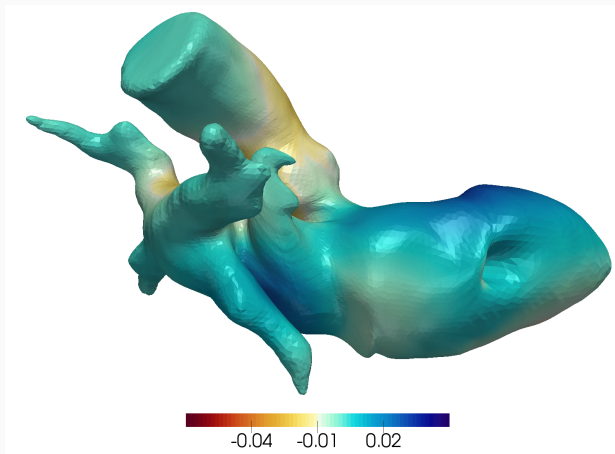
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



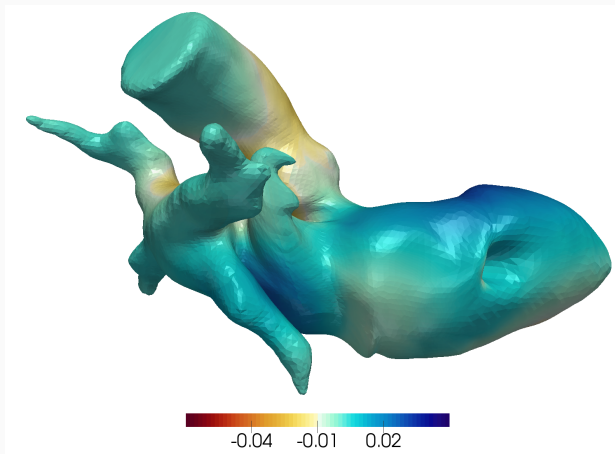
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

LDDMM: generate Diffeomorphic deformations



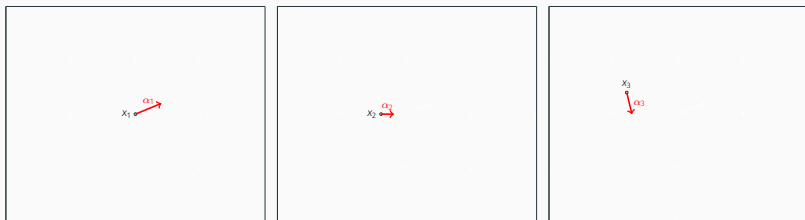
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

- **Space of vectors fields** V : an RKHS of vectors fields (smooth, vanishing at infinity). There exists a kernel $K_s : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}^{D \times D}$ such that

$$\text{Span}\{\delta_x^\alpha = K_s(x, \cdot)\alpha, x \in \mathbb{R}^D, \alpha \in \mathbb{R}^D\}$$

is dense in V . In practice, $D = 2, 3$ and

$$K_s(x, y) = e^{-\frac{\|x-y\|^2}{\sigma^2}} Id_D.$$

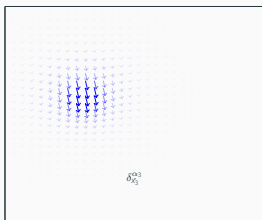
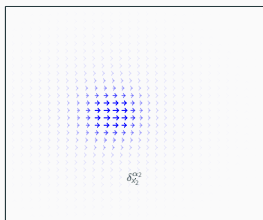
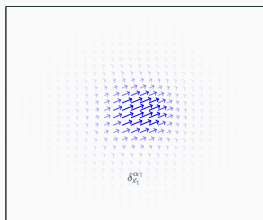


- **Space of vectors fields V** : an RKHS of vectors fields (smooth, vanishing at infinity). There exists a kernel $K_s : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}^{D \times D}$ such that

$$\text{Span}\{\delta_x^\alpha = K_s(x, \cdot)\alpha, x \in \mathbb{R}^D, \alpha \in \mathbb{R}^D\}$$

is dense in V . In practice, $D = 2, 3$ and

$$K_s(x, y) = e^{-\frac{\|x-y\|^2}{\sigma^2}} Id_D.$$

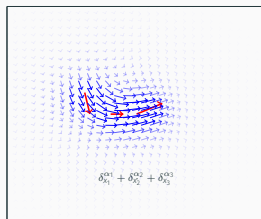
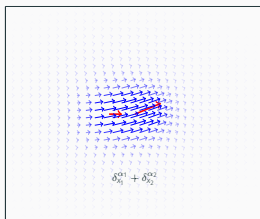
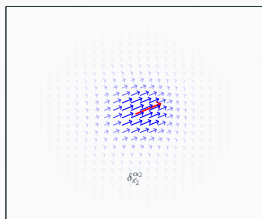


- **Space of vectors fields V** : an RKHS of vectors fields (smooth, vanishing at infinity). There exists a kernel $K_s : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}^{D \times D}$ such that

$$\text{Span}\{\delta_x^\alpha = K_s(x, \cdot)\alpha, x \in \mathbb{R}^D, \alpha \in \mathbb{R}^D\}$$

is dense in V . In practice, $D = 2, 3$ and

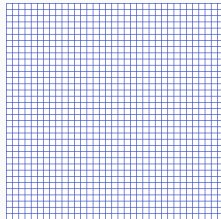
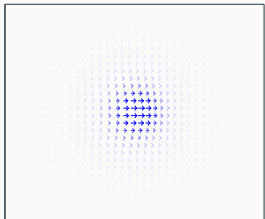
$$K_s(x, y) = e^{-\frac{\|x-y\|^2}{\sigma^2}} Id_D.$$



Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$

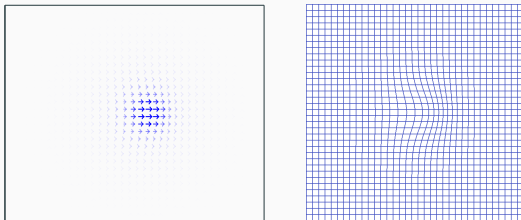


$t = 0$

Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$

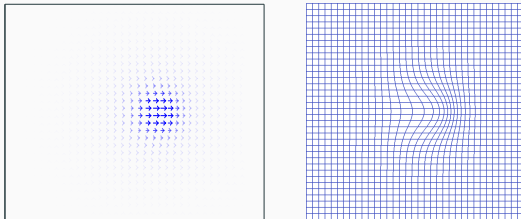


$t = 1/5$

Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$

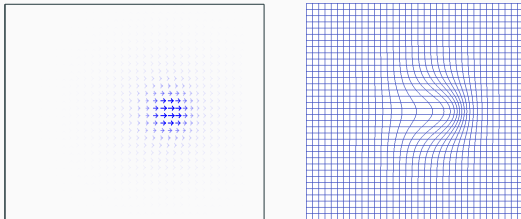


$t = 2/5$

Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$

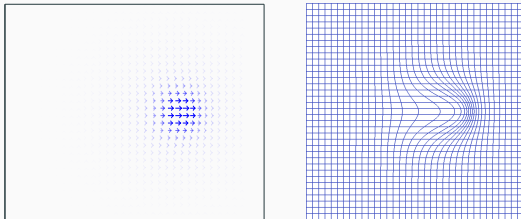


$t = 3/5$

Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$

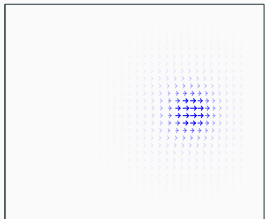


$t = 4/5$

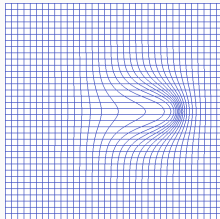
Geometrical deformations: flow of time varying smooth vector field

- **Flow:** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^D$. Let $\varphi : [0, 1] \times \mathbb{R}^D \rightarrow \mathbb{R}^D$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \quad t \in [0, 1] \text{ and } x \in \mathbb{R}^D$$



$t = 1$



Computing an Hamiltonian

Let $\mathbf{q}, \mathbf{p} \in \mathbb{R}^{N \times D}$, we need to compute the Hamiltonian (interpreted as a kinetic energy / RKHS norm).

$$H(\mathbf{q}, \mathbf{p}) = \frac{1}{2} \mathbf{p}^t \underbrace{K_s(\mathbf{q}, \mathbf{q})}_{O(N^2) \text{ terms}} \mathbf{p} = \frac{1}{2} \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2 / s^2)} p_j$$

...and its derivative (up to 2nd order).

Computing an Hamiltonian

Let $\mathbf{q}, \mathbf{p} \in \mathbb{R}^{N \times D}$, we need to compute the Hamiltonian (interpreted as a kinetic energy / RKHS norm).

$$H(\mathbf{q}, \mathbf{p}) = \frac{1}{2} \mathbf{p}^t \underbrace{K_s(\mathbf{q}, \mathbf{q})}_{O(N^2) \text{ terms}} \mathbf{p} = \frac{1}{2} \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2 / s^2)} p_j$$

...and its derivative (up to 2nd order).

```
import torch

torch.set_default_tensor_type(torch.cuda.FloatTensor)

# Clouds of 1000 points in 3D
N, D = 1000, 3

# Generate arbitrary arrays
q = torch.randn(N, D, requires_grad=True)
p = torch.randn(N, D, requires_grad=True)
s2 = torch.tensor([0.5 * 0.5], requires_grad=False)
```

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):  
    x_i = x[:, None, :]          # (M, 1, 1)  
    y_j = y[None, :, :]         # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1)  # (M, N) matrix of squared distances  
    return (-D_ij / sigma2).exp()      # (M, N) matrix of Gaussian kernel
```

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):  
    x_i = x[:, None, :]  
    y_j = y[None, :, :]  
    D_ij = ((x_i - y_j) ** 2).sum(-1)  
    return (-D_ij / sigma2).exp()  
  
K_qq = gaussian_kernel(q, q, s2)
```

(M, 1, 1)
(1, N, 1)
(M, N) matrix of squared distances
(M, N) matrix of Gaussian kernel

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):  
    x_i = x[:, None, :]  
    y_j = y[None, :, :]  
    D_ij = ((x_i - y_j) ** 2).sum(-1)  
    return (-D_ij / sigma2).exp()  
  
K_qq = gaussian_kernel(q, q, s2)  
  
v = K_qq @ p
```

(M, 1, 1)
(1, N, 1)
(M, N) matrix of squared distances
(M, N) matrix of Gaussian kernel

mat. mult. (N,N) @ (N,D) = (N,D)

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):  
    x_i = x[:, None, :]          # (M, 1, 1)  
    y_j = y[None, :, :]         # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1)  # (M, N) matrix of squared distances  
    return (-D_ij / sigma2).exp()      # (M, N) matrix of Gaussian kernel  
  
K_qq = gaussian_kernel(q, q, s2)  
  
v = K_qq @ p                     # mat. mult. (N,N) @ (N,D) = (N,D)  
  
# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>  
H = .5 * torch.dot(p.view(-1), v.view(-1))
```


Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):  
    x_i = x[:, None, :]          # (M, 1, 1)  
    y_j = y[None, :, :]         # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1)  # (M, N) matrix of squared distances  
    return (-D_ij / sigma2).exp()      # (M, N) matrix of Gaussian kernel  
  
K_qq = gaussian_kernel(q, q, s2)  
  
v = K_qq @ p                    # mat. mult. (N,N) @ (N,D) = (N,D)  
  
# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>  
H = .5 * torch.dot(p.view(-1), v.view(-1))  
  
# Automatic differentiation is straightforward... But memory greedy  
[dq,dp] = torch.autograd.grad(H, [q,p])
```

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):
    x_i = x[:, None, :]          # (M, 1, 1)
    y_j = y[None, :, :]         # (1, N, 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)  # (M, N) matrix of squared distances
    return (-D_ij / sigma2).exp()     # (M, N) matrix of Gaussian kernel

K_qq = gaussian_kernel(q, q, s2)

v = K_qq @ p                    # mat. mult. (N,N) @ (N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>
H = .5 * torch.dot(p.view(-1), v.view(-1))

# Automatic differentiation is straightforward... But memory greedy
[dq,dp] = torch.autograd.grad(H, [q,p])

torch.cuda.OutOfMemoryError: CUDA out of memory. [...]
```

Computing the Hamiltonian

```
def gaussian_kernel(x, y, sigma2):
    x_i = x[:, None, :]          # (M, 1, 1)
    y_j = y[None, :, :]         # (1, N, 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)  # (M, N) matrix of squared distances
    return (-D_ij / sigma2).exp()     # (M, N) matrix of Gaussian kernel

K_qq = gaussian_kernel(q, q, s2)

v = K_qq @ p                    # mat. mult. (N,N) @ (N,D) = (N,D)

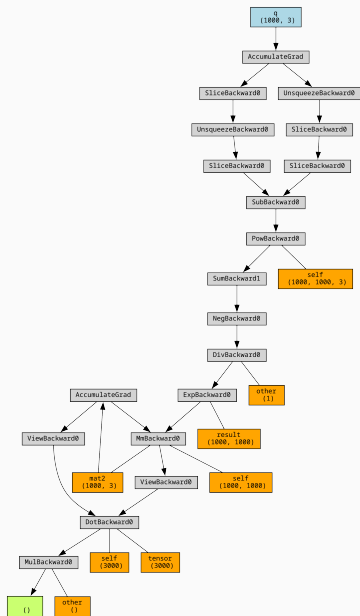
# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>
H = .5 * torch.dot(p.view(-1), v.view(-1))

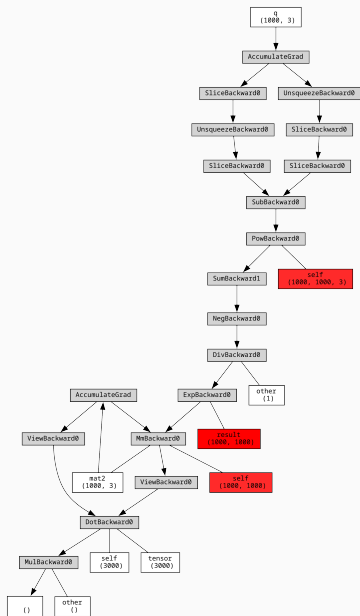
# Automatic differentiation is straightforward... But memory greedy
[dq,dp] = torch.autograd.grad(H, [q,p])

torch.cuda.OutOfMemoryError: CUDA out of memory. [...]
```

```
from torchviz import make_dot

# Display -- see next figure.
make_dot(H, {'q':q, 'p':p, 's':s2}, show_saved=True).render(view=True)
```





KeOps and symbolic matrices

Kernel methods are ubiquitous

Model dataset with **interacting** particles (e.g. through distance or covariance matrix).



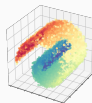
meshes



clustering



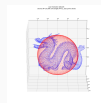
Gaussian processes



Spectral methods



Dimension red (UMAP)



Optimal Transport

Reproducing Kernel Hilbert spaces

- nice mathematical structures (scalar product, norms, completeness, etc...)
- from continuous to discrete problems (evaluation is continuous)

Kernel methods are ubiquitous

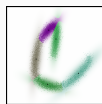
Model dataset with **interacting** particles (e.g. through distance or covariance matrix).



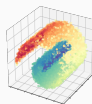
meshes



clustering



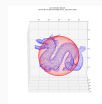
Gaussian processes



Spectral methods



Dimension red (UMAP)



Optimal Transport

Reproducing Kernel Hilbert spaces

- nice mathematical structures (scalar product, norms, completeness, etc...)
- from continuous to discrete problems (evaluation is continuous)

Since 2017, with J. Glaunès, J. Feydy we are developing



KeOps (kernels on

GPU with CUDA)

- developed for Deep Learning framework (NeurIPS 2020)
- autodiff with kernels for optimisation (JMLR 2021)

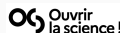
• **Download:** 650k

• **Dependency:** 404

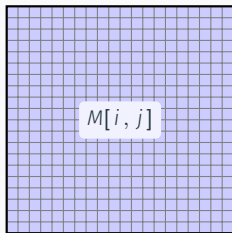
• **Github stars:** 1k

• **Citations:** 170

Prix science ouverte 2023



Scientific computing libraries represent most objects as tensors

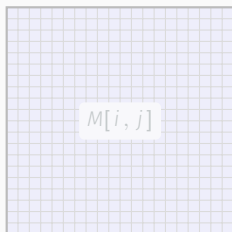


Dense matrix
Coefficients only

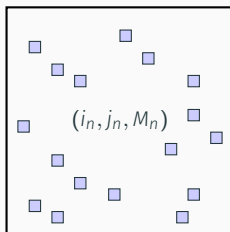
Dense matrices – large, contiguous **arrays** of numbers:

- + **Convenient** and well supported.
- Heavy load on the **memories** of our GPUs, with **time-consuming transfers** that take place between compute units.

Scientific computing libraries represent most objects as tensors



Dense matrix
Coefficients only

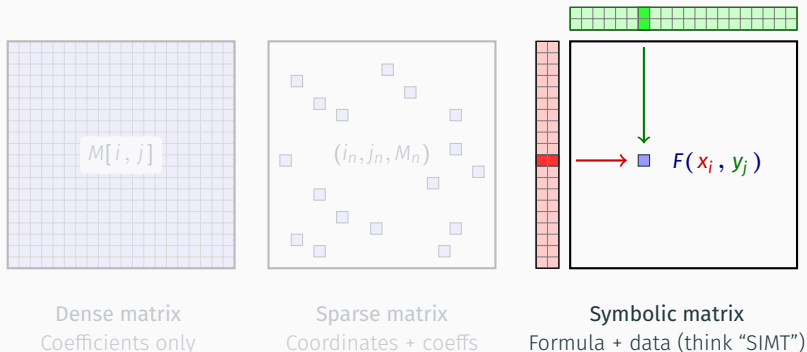


Sparse matrix
Coordinates + coeffs

Sparse matrices – tensors that have **few non-zero entries**:

- + Represent **large tensors** with a small memory footprint.
- Outside of **graph** processing, few objects are **sparse enough** to really benefit from this representation.

Scientific computing libraries represent most objects as tensors



Distance and kernel matrices, point convolutions, attention layers:

- + **Linear** memory usage: no more **memory** overflows.
- + We can optimize the use of registers for a $\times 10 - \times 100$ speed-up vs. a standard PyTorch GPU baseline.

KeOps provide support for this “new abstraction” on the GPU

Our library comes with all the perks of a modern numerical computing library:

- + Transparent **array-like** interface (float16, float32, float64).
- + Full support for automatic **differentiation**.
- + Documented and large collection of **tutorials**, available online.

See *www.kernel-operations.io*

KeOps provide support for this “new abstraction” on the GPU

Our library comes with all the perks of a modern numerical computing library:

- + Transparent **array-like** interface (float16, float32, float64).
- + Full support for automatic **differentiation**.
- + Documented and large collection of **tutorials**, available online.

See *www.kernel-operations.io*

Under the hood:

- + a meta-programming engine **keopscore** written in **Python** and **Cuda**
- + JIT **cuda** code thanks to **nVRTC**
- + binders for **PyTorch**, **NumPy** and **R** (thanks to Ghislain Durif).

We welcome **contributors** for **JAX**, **Julia** and other frameworks!

To get started:

```
pip install pykeops
```

Generic reduction in KeOps

Let $1 \leq i \leq N$ and $1 \leq j \leq M$ where $N, M \approx 10^4$ ou 10^6

- A generic case:

$$\left[\sum_j F(\sigma_1, \dots, \sigma_\ell, X_i^1, \dots, X_i^k, Y_j^1, \dots, Y_j^m) \right]_{i=1, \dots, M} \in \mathbb{R}^M$$

- ...an even more generic case:

$$\left[* F(\sigma_1, \dots, \sigma_\ell, X_i^1, \dots, X_i^k, Y_j^1, \dots, Y_j^m) \right]_{i=1, \dots, M} \in \mathbb{R}^M$$

where $*$ can be any reduction (sum, max, min, logSumExp, etc...) over a dimension

First example: efficient nearest neighbor search in dimension 50

Create large point clouds using **standard PyTorch** syntax:

```
import torch
N, M, D = 10**6, 10**6, 50
x = torch.rand(N, 1, D).cuda() # (1M, 1, 50) array
y = torch.rand(1, M, D).cuda() # (1, 1M, 50) array
```

Turn **dense** arrays into **symbolic** matrices:

```
from pykeops.torch import LazyTensor
x_i, y_j = LazyTensor(x), LazyTensor(y)
```

Create a large **symbolic matrix** of squared distances:

```
D_ij = ((x_i - y_j)**2).sum(dim=2) # (1M, 1M) symbolic
```

Use an **.argmin()** reduction to perform a nearest neighbor query:

```
indices_i = D_ij.argmax(dim=1) # -> standard torch tensor
```

The KeOps library combines performance with flexibility

Script of the previous slide = efficient nearest neighbor query,
on par with the brute-force CUDA scheme of the **FAISS** library...
And can be used with **any metric**!

```
D_ij = ((x_i - x_j) ** 2).sum(dim=2)      # Euclidean
M_ij = (x_i - x_j).abs().sum(dim=2)      # Manhattan
C_ij = 1 - (x_i | x_j)                   # Cosine
H_ij = D_ij / (x_i[...,0] * x_j[...,0]) # Hyperbolic
```

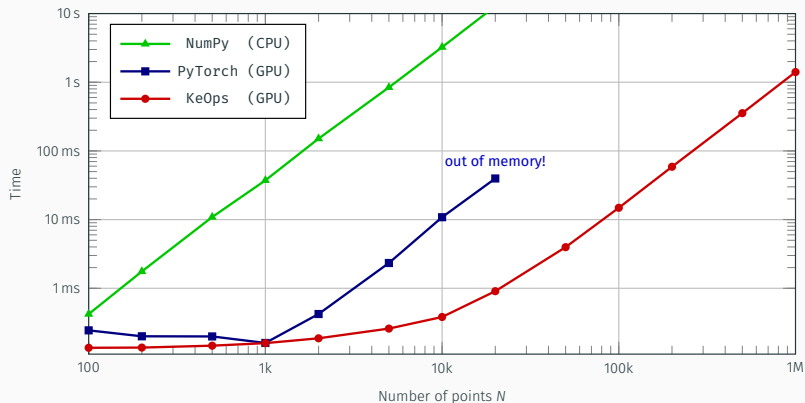
KeOps supports arbitrary **formulas** and **variables** with:

- **Reductions:** sum, log-sum-exp, K-min, matrix-vector product, etc.
- **Operations:** +, ×, sqrt, exp, neural networks, etc.
- **Advanced schemes:** batch processing, block sparsity, etc.
- **Automatic differentiation:** seamless integration with PyTorch.

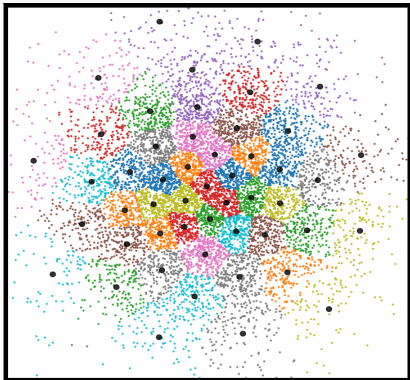
KeOps lets users work with millions of points at a time

Benchmark of a matrix-vector product with a N -by- N Gaussian kernel matrix between 3D point clouds.

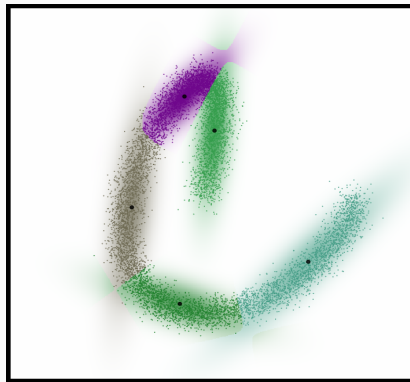
We run NumPy, PyTorch and KeOps on a RTX 2080 Ti GPU.



KeOps is a good fit for various computational fields



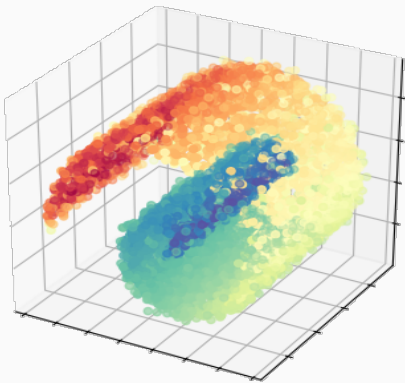
Clustering (K-Means).



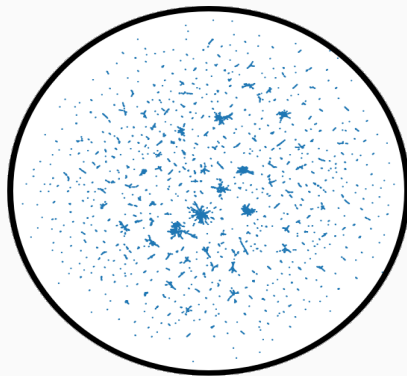
Gaussian Mixture Model.

Use **any** kernel, metric or formula **you** like!

KeOps is a good fit for various computational fields



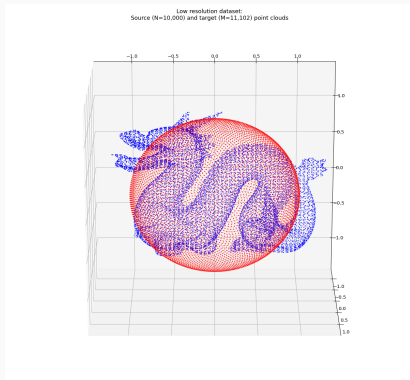
Spectral analysis.



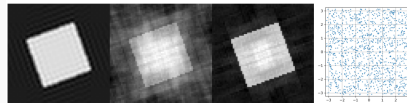
Dimension reduction (UMAP in hyperbolic space).

Use **any** kernel, metric or formula **you** like!

KeOps is a good fit for various computational fields



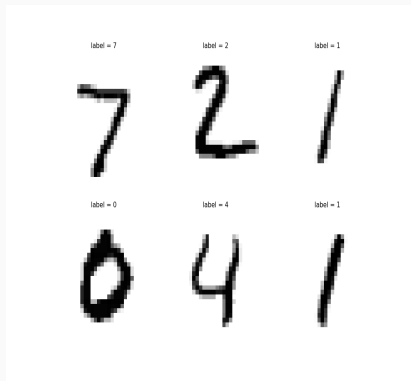
Regularized OT (credit: geomloss).



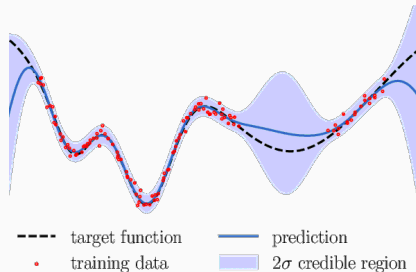
Inverse problems (NUFT) (credit: A. Gossart,
F. de Gourmet, P. Weiss).

Use **any** kernel, metric or formula **you** like!

KeOps is a good fit for various computational fields



Clustering (k -NN).



Gaussian Processes

Use **any** where are PDEs? Boundary Element Method ?

Similar (yet different!) softwares:

- Triton
- Pytorch jit
- Taichi language
- Tensor comprehension (deprecated)

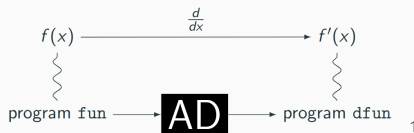
Autodiff engine

Autodiff engine

Introduction

AD as a black box to differentiate

Given a function in a code, how to compute its gradient (when possible, when it makes sens) ?



composition of augmented programs $\leftrightarrow$ composition of differentiable functions

¹Some material from J. Feydy and S. Vaïter (Today at 2pm
<https://orsay.bbb.cnrs.fr/b/gio-4gd-5rm-t4v>)

Let $F : \mathbb{R}^p \rightarrow \mathbb{R}$ be a smooth function. Then:

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

Let $F : \mathbb{R}^p \rightarrow \mathbb{R}$ be a smooth function. Then:

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

$\implies$ costs **(p+1)** evaluations of F , which is poor.

Computing gradient is costly

Let $f : \mathbb{R}^p \rightarrow \mathbb{R}$ a smooth function.

Complexity (number of operations), theoretical bounds are:

- Finite differences: $\text{Cost}(\nabla f) = (p + 1) \text{Cost}(f)$ (linear in p)
- (reverse) AD: $\text{Cost}(\nabla f) \leq Cte \text{Cost}(f)$ [Baur - Strassen '83] (independent of p)

Computing gradient is costly

Let $f : \mathbb{R}^p \rightarrow \mathbb{R}$ a smooth function.

Complexity (number of operations), theoretical bounds are:

- Finite differences: $\text{Cost}(\nabla f) = (p + 1) \text{Cost}(f)$ (linear in p)
- (reverse) AD: $\text{Cost}(\nabla f) \leq Cte \text{Cost}(f)$ [Baur - Strassen '83] (independent of p)

Storage (max buffer size), empirical bounds are:

- Finite differences: $\text{Cost}(\nabla f) \sim \text{Cost}(f)$
- (reverse) AD: $\text{Cost}(\nabla f) \sim 7 \text{Cost}(f)$

Making your (costly) extra Go of GPU RAM critical...

Composition of functions ($f_0 : \mathbb{R}^p = \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_1}, \dots, f_3 : \mathbb{R}^{n_3} \rightarrow \mathbb{R}^{n_4} = \mathbb{R}^n$)

$$f = f_0 \circ f_1 \circ f_2 \circ f_3 : \mathbb{R}^p \rightarrow \mathbb{R}^n$$

Computational graph



Chain rule

$$\frac{\partial y}{\partial x} = \frac{\partial w_1}{\partial w_0} \frac{\partial w_2}{\partial w_1} \frac{\partial w_3}{\partial w_2} \frac{\partial w_4}{\partial w_3} = \text{Jac}_{f_3}(w_3) \text{Jac}_{f_2}(w_2) \text{Jac}_{f_1}(w_1) \text{Jac}_{f_0}(w_0)$$

How to compute these term? Choose your way...



Jacobian-vector products (JVPs)

$$\text{Jac}_f(x) = \begin{pmatrix} \text{Jac}_f(x)\mathbf{e}_1 & \cdots & \text{Jac}_f(x)\mathbf{e}_p \end{pmatrix} \implies \text{need } p \text{ JVPs (column-per-column)}$$

Chain rule for JVPs

$$\text{Jac}_f(x)\mathbf{e}_k = \underbrace{\text{Jac}_{f_3}(w_3) [\text{Jac}_{f_2}(w_2) \{ \text{Jac}_{f_1}(w_1) (\text{Jac}_{f_0}(w_0)\mathbf{e}_k) \}]}_{\text{right-to-left multiplication (forward } w_0 \rightarrow w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow w_4)}$$

Cost of p JVPs: $p \sum_{i=0}^3 n_i n_{i+1}$

$O(p^3)$ if $n_i = p$ for $i \neq 4$

Torch autograd: A first example (part 1)

```
import torch

# function f. Sizes (3, 4) -> (3,)
>>> def func(x):
>>>     return x.exp().sum(dim=1)

>>> inputs = torch.rand(3, 4)  # same size as input of f
>>> v = torch.randn_like(func(inputs))  # same size as output of f
```


Torch autograd: A first example (part 1)

```
import torch

# function f. Sizes (3, 4) -> (3,)
>>> def func(x):
>>>     return x.exp().sum(dim=1)

>>> inputs = torch.rand(3, 4) # same size as input of f
>>> v = torch.randn_like(func(inputs)) # same size as output of f

>>> jac = torch.autograd.functional.jacobian(exp_reducer, inputs) # (3, (3, 4))
# tensor([[[[1.6998, 1.1370, 1.7796, 2.6448],
#          [0.0000, 0.0000, 0.0000, 0.0000],
#          [0.0000, 0.0000, 0.0000, 0.0000]],
#         [[0.0000, 0.0000, 0.0000, 0.0000],
#          [1.5400, 1.3338, 1.1431, 1.0552],
#          [0.0000, 0.0000, 0.0000, 0.0000]],
#         [[0.0000, 0.0000, 0.0000, 0.0000],
#          [0.0000, 0.0000, 0.0000, 0.0000],
#          [2.6941, 2.6420, 1.7159, 2.4825]]]])
```

Torch autograd: A first example (part 1)

```
import torch

# function f. Sizes (3, 4) -> (3,)
>>> def func(x):
>>>     return x.exp().sum(dim=1)

>>> inputs = torch.rand(3, 4) # same size as input of f
>>> v = torch.randn_like(func(inputs)) # same size as output of f

>>> jac = torch.autograd.functional.jacobian(exp_reducer, inputs) # (3, (3, 4))
# tensor([[[[1.6998, 1.1370, 1.7796, 2.6448],
#           [0.0000, 0.0000, 0.0000, 0.0000],
#           [0.0000, 0.0000, 0.0000, 0.0000]],
#          [[0.0000, 0.0000, 0.0000, 0.0000],
#           [1.5400, 1.3338, 1.1431, 1.0552],
#           [0.0000, 0.0000, 0.0000, 0.0000]],
#          [[0.0000, 0.0000, 0.0000, 0.0000],
#           [0.0000, 0.0000, 0.0000, 0.0000],
#           [2.6941, 2.6420, 1.7159, 2.4825]]]])
# ... with same entries as exp(inputs)
>>> inputs.exp()
# tensor([[[1.6998, 1.1370, 1.7796, 2.6448],
#           [1.5400, 1.3338, 1.1431, 1.0552],
#           [2.6941, 2.6420, 1.7159, 2.4825]]])
```

Torch autograd: A first example (part 2)

```
from torch.autograd.functional import vjp, jvp
```

Torch autograd: A first example (part 2)

```
from torch.autograd.functional import vjp, jvp
```

VPJ:

```
# compute f(x), v @ jac_x f: (4,) @ (4, (3, 4)) == (3, 4))
>>> v = torch.randn_like(func(inputs)) # same size as output of f
>>> _, vjp = vjp(exp_reducer, inputs, v)
# tensor([[ -0.3657, -0.2446, -0.3829, -0.5690],
#         [-1.6098, -1.3943, -1.1950, -1.1031],
#         [ 4.4814,  4.3948,  2.8542,  4.1294]])

assert torch.allclose(vjp, v @ jac)
# True
```

Torch autograd: A first example (part 2)

```
from torch.autograd.functional import vjp, jvp
```

VPJ:

```
# compute f(x), v @ jac_x f: (4,) @ (4, (3, 4)) == (3, 4))
>>> v = torch.randn_like(func(inputs)) # same size as output of f
>>> _, vjp = vjp(exp_reducer, inputs, v)
# tensor([[ -0.3657, -0.2446, -0.3829, -0.5690],
#         [-1.6098, -1.3943, -1.1950, -1.1031],
#         [ 4.4814,  4.3948,  2.8542,  4.1294]])

assert torch.allclose(vjp, v @ jac)
# True
```

JVP

```
# tangent vector: same size as x
>>> v2 = torch.randn_like(inputs)
>>> _, jvp = jvp(exp_reducer, inputs, v2)

assert allclose(jvp, (jac.reshape(3, -1) @ v2.reshape(-1,1)).ravel())
# True
```

Backward (or reverse) AD



Vector-Jacobian products (VJPs)

$$\text{Jac}_f(x) = \left(\mathbf{e}_1^\top \text{Jac}_f(x) \quad \cdots \quad \mathbf{e}_n^\top \text{Jac}_f(x) \right)^\top \implies \text{need } n \text{ VJPs (row-by-row)}$$

Chain rule for VJPs

$$E_I \text{Jac}_f(x) = \underbrace{\left[\left(\left(E_I \text{Jac}_{f_3}(w_3) \right) \text{Jac}_{f_2}(w_2) \right) \right] \text{Jac}_{f_1}(w_1)}_{\text{left-to-right multiplication (reverse } w_4 \rightarrow w_3 \rightarrow w_2 \rightarrow w_1 \rightarrow w_0)} \text{Jac}_{f_0}(w_0)$$

Cost of n VJPs: $n \sum_{i=0}^3 n_i n_{i+1}$

$O(p^2)$ if $n_i = p$ for $i \neq 4$

Torch autograd: A second example

```
import torch
>>> a = torch.randn(5,1, requires_grad=True)
# tensor([[ -0.3717],
#         [  0.1786],
#         [  0.5572],
#         [-2.5876],
#         [  0.6250]], requires_grad=True)
```

Torch autograd: A second example

```
import torch
>>> a = torch.randn(5,1, requires_grad=True)
# tensor([[ -0.3717],
#         [  0.1786],
#         [  0.5572],
#         [-2.5876],
#         [  0.6250]], requires_grad=True)

>>> b = .5 * (a ** 2).sum()
# tensor(3.7833, grad_fn=<MulBackward0>)
```


Torch autograd: A second example

```
import torch
>>> a = torch.randn(5,1, requires_grad=True)
# tensor([[ -0.3717],
#         [  0.1786],
#         [  0.5572],
#         [-2.5876],
#         [  0.6250]], requires_grad=True)

>>> b = .5 * (a ** 2).sum()
# tensor(3.7833, grad_fn=<MulBackward0>)

>>> torch.autograd.grad(b, [a], torch.ones(1))
# (tensor([[ -0.3717],
#         [  0.1786],
#         [  0.5572],
#         [-2.5876],
#         [  0.6250]]),)
```

Let $F : (X, \langle \cdot, \cdot \rangle_X) \rightarrow (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between two Hilbert spaces.

Let $F : (X, \langle \cdot, \cdot \rangle_X) \rightarrow (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between two Hilbert spaces.

- the adjoint of the differential is $(d_x F)^*(x_0) : \alpha \in Y^* \rightarrow \beta \in X^*$. Riesz representation theorem gives a map

$$\partial_x F(x_0) : a \in Y \rightarrow b \in X$$

called generalized **gradient**.

Let $F : (X, \langle \cdot, \cdot \rangle_X) \rightarrow (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between two Hilbert spaces.

- the adjoint of the differential is $(d_x F)^*(x_0) : \alpha \in Y^* \rightarrow \beta \in X^*$. Riesz representation theorem gives a map

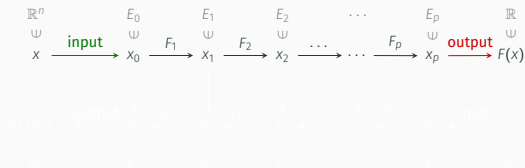
$$\partial_x F(x_0) : a \in Y \rightarrow b \in X$$

called generalized **gradient**.

- If $X = \mathbb{R}^p$, $Y = \mathbb{R}$ endowed with the Euclidean metric,

$$\mathcal{M}_{\partial_x F(x_0)} = \begin{pmatrix} \partial_{x_1} F(x_0) \\ \partial_{x_2} F(x_0) \\ \vdots \\ \partial_{x_p} F(x_0) \end{pmatrix} = \nabla_x F(x_0)$$

Reverse AD = backpropagating

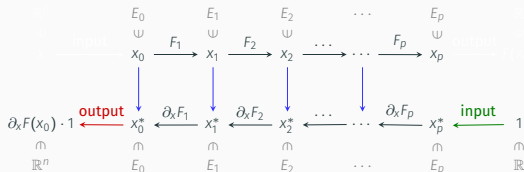


Backpropagating through a computational graph requires:

$$F_i : \begin{array}{ccc} E_{i-1} & \rightarrow & E_i \\ x & \mapsto & F_i(x) \end{array} \quad (1)$$

encoded as **computer programs**.

Reverse AD = backpropagating

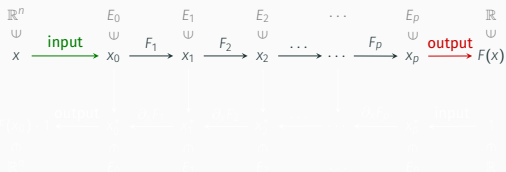


Backpropagating through a computational graph requires:

$$F_i : \begin{array}{ccc} E_{i-1} & \rightarrow & E_i \\ x & \mapsto & F_i(x) \end{array} \quad \partial_x F_i : \begin{array}{ccc} E_{i-1} \times E_i & \rightarrow & E_{i-1} \\ (x_{i-1}, x_i^*) & \mapsto & \partial_x F_i(x_{i-1}) \cdot x_i^* \end{array} \quad (1)$$

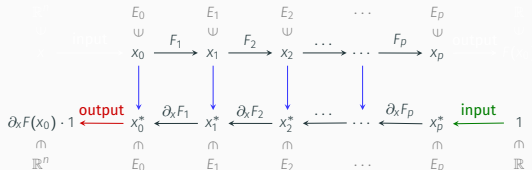
encoded as **computer programs**.

Reverse AD = backpropagating



1. Starting from $x_0 \in \mathbb{R}^n = E_0$, compute and **store in memory** the successive vectors $x_i \in E_i$. The last one, $x_p = F(x_0) \in \mathbb{R}$.

Reverse AD = backpropagating



1. Starting from $x_0 \in \mathbb{R}^n = E_0$, compute and **store in memory** the successive vectors $x_i \in E_i$. The last one, $x_p = F(x_0) \in \mathbb{R}$.
2. Starting from the canonical value of $x_p^* = 1 \in \mathbb{R}$, compute the successive **dual vectors**

$$x_i^* = \partial_x F_{i+1}(x_i) \cdot x_{i+1}^*. \quad (2)$$

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

Autodiff engine

KeOps autograd

Let the formula

$$F(x, y) = \left[\sum_{j=1}^N \exp(x_i + y_j) \right]_{i=1, \dots, M}$$

where

- $x \in \mathbb{R}^M$ is a variable indexed by i ,
- $y \in \mathbb{R}^N$ is a variable indexed by j

Compute $F : \mathbb{R}^M \times \mathbb{R}^N \rightarrow \mathbb{R}^M$ with KeOps:

```
x = torch.rand(4000, 1, requires_grad=True)
y = torch.rand(3000, 1, requires_grad=True)

X = pykeops.torch.LazyTensor(x.reshape(4000, 1, 1))
Y = pykeops.torch.LazyTensor(y.reshape(1, 3000, 1))
Z = (X + Y).exp() # still LazyTensor

F = Z.sum(1)
# [KeOps] Generating code for formula Sum_Reduction(Exp(Var(0,1,0)+Var(1,1,1)),0)
# ... OK

torch.allclose(torch.exp(x + y.t()).sum(1), F.view(-1))
# True
```

Compute the gradient $\mathbb{R}^N \ni y \mapsto F(x, y) \in \mathbb{R}^M$ applied to an arbitrary test vector $e \in \mathbb{R}^M$:

$$[\partial_y F(x, y)](e) = [d_y F^*(x, y)](e) = \left[\sum_{i=1}^M \exp(x_i + y_i) e_i \right]_{j=1}^N$$

Compute dF with KeOps and **Grad**(, ,) operator:

```
e = torch.rand_like(x)

F_grad_y = torch.autograd.grad(F, [y], e)[0]
# [KeOps] Generating code for formula Sum_Reduction(Var(2,1,0)*Exp(Var(0,1,0)+
# Var(1,1,1)),0) ... OK
# [KeOps] Generating code for formula Sum_Reduction(Var(2,1,0)*Exp(Var(0,1,0)+
# Var(1,1,1)),1) ... OK

torch.allclose(torch.exp(x + y.t()).t() @ e, F_grad_y)
# True
```

KeOps effect on the LDDMM example

```
from pykeops.torch import LazyTensor

def gaussian_kernel_keops(x, y, sigma2):
    x_i = LazyTensor(x[:, None, :])      # ([M, 1], 1)
    y_j = LazyTensor(y[None, :, :])      # ([1, N], 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)      # ([M, N]) Symb. mat. of squared distances
    return (-D_ij / sigma2).exp()          # ([M, N]) Symb. mat. of Gaussian kernel
```

KeOps effect on the LDDMM example

```
from pykeops.torch import LazyTensor

def gaussian_kernel_keops(x, y, sigma2):
    x_i = LazyTensor(x[:, None, :])      # ([M, 1], 1)
    y_j = LazyTensor(y[None, :, :])      # ([1, N], 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)      # ([M, N]) Symb. mat. of squared distances
    return (-D_ij / sigma2).exp()          # ([M, N]) Symb. mat. of Gaussian kernel

K_qq = gaussian_kernel_keops(q, q, s2)
```

KeOps effect on the LDDMM example

```
from pykeops.torch import LazyTensor

def gaussian_kernel_keops(x, y, sigma2):
    x_i = LazyTensor(x[:, None, :])      # ([M, 1], 1)
    y_j = LazyTensor(y[None, :, :])      # ([1, N], 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)      # ([M, N]) Symb. mat. of squared distances
    return (-D_ij / sigma2).exp()          # ([M, N]) Symb. mat. of Gaussian kernel

K_qq = gaussian_kernel_keops(q, q, s2)

v = K_qq @ p                             # mat. mult. ([N,N]) @ (N,D) = (N,D)

# [KeOps] Generating code for formula Sum_Reduction(Exp(-Sum((Var(0,3,0)-
#           Var(1,3,1))*2)/Var(2,1,2))*Var(3,3,1),0) ... OK
```

KeOps effect on the LDDMM example

```
from pykeops.torch import LazyTensor

def gaussian_kernel_keops(x, y, sigma2):
    x_i = LazyTensor(x[:, None, :])      # ([M, 1], 1)
    y_j = LazyTensor(y[None, :, :])      # ([1, N], 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)      # ([M, N]) Symb. mat. of squared distances
    return (-D_ij / sigma2).exp()          # ([M, N]) Symb. mat. of Gaussian kernel

K_qq = gaussian_kernel_keops(q, q, s2)

v = K_qq @ p                             # mat. mult. ([N,N]) @ (N,D) = (N,D)

# [KeOps] Generating code for formula Sum_Reduction(Exp(-Sum((Var(0,3,0)-
#           Var(1,3,1))*2)/Var(2,1,2))*Var(3,3,1),0) ... OK

# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>
H = .5 * torch.dot(p.view(-1), v.view(-1))
```

KeOps effect on the LDDMM example

```
from pykeops.torch import LazyTensor

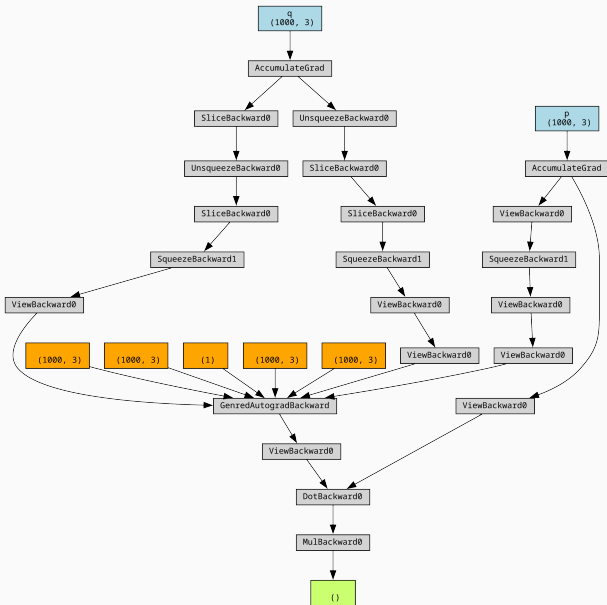
def gaussian_kernel_keops(x, y, sigma2):
    x_i = LazyTensor(x[:, None, :])      # ([M, 1], 1)
    y_j = LazyTensor(y[None, :, :])      # ([1, N], 1)
    D_ij = ((x_i - y_j) ** 2).sum(-1)      # ([M, N]) Symb. mat. of squared distances
    return (-D_ij / sigma2).exp()          # ([M, N]) Symb. mat. of Gaussian kernel

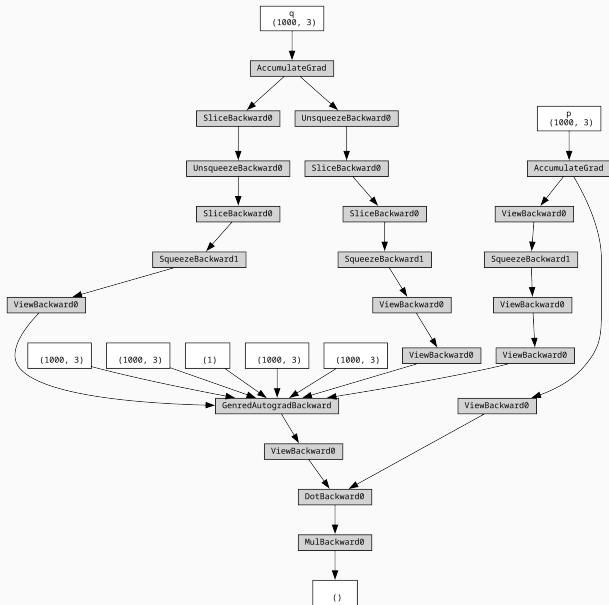
K_qq = gaussian_kernel_keops(q, q, s2)
v = K_qq @ p                             # mat. mult. ([N,N]) @ (N,D) = (N,D)

# [KeOps] Generating code for formula Sum_Reduction(Exp(-Sum((Var(0,3,0)-
#           Var(1,3,1))*2)/Var(2,1,2))*Var(3,3,1),0) ... OK

# Finally, compute the Hamiltonian H(q,p): .5 * <p,v>
H = .5 * torch.dot(p.view(-1), v.view(-1))

# Automatic differentiation is straightforward... and working !
[dq,dp] = torch.autograd.grad(H, [q,p])
```



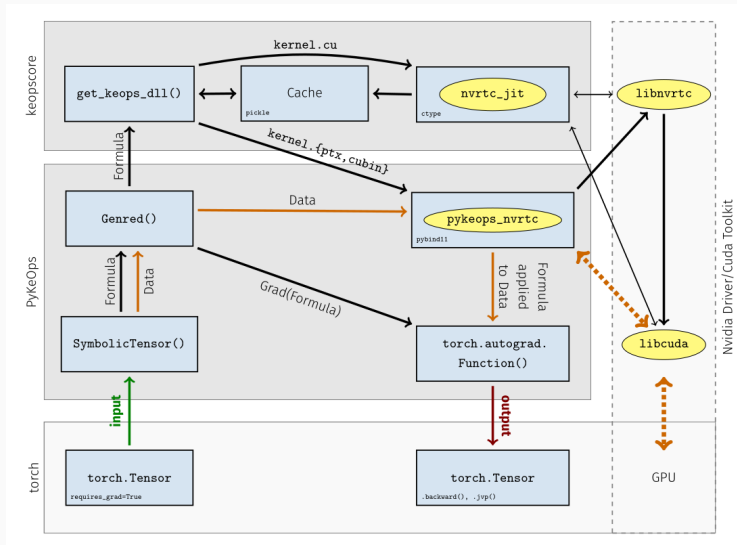
```
class Log(Operation):
    """the logarithm operation"""
    def __init__(self, arg):
        # arg is another instance of Operation
        self.children = arg
        self.string_id = "Log"

    def Op(self, out_var, arg_var) -> str:
        # c++/cuda code to evaluate the log
        loop = f"for(int k = 0; k < {out_var.dim}; k++)"
        code = f"    {out_var.id}[k] = log({arg_var.id}[k]);"
        return "\n".join(loop, log_code)

    def DiffT(self, v, gradin):
        # return an Operation that could be use in the backward pass of the AD engine
        f = self.children
        df = 1 / f
        return f.DiffT(v, gradin) * df

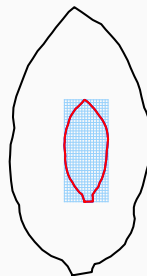
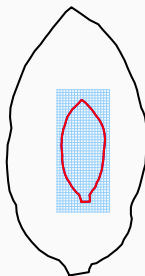
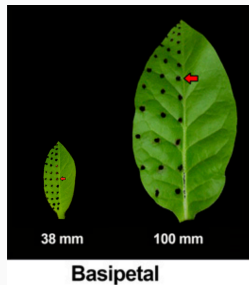
def Grad(formula, v, gradin=None):
    """
    Defines [\partial_V F].gradin function: Symbolic differentiation
    is a straightforward recursive operation,
    """
    return formula.DiffT(v, gradin)
```

KeOps Workflow



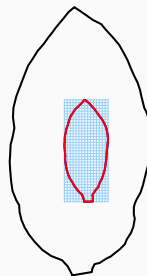
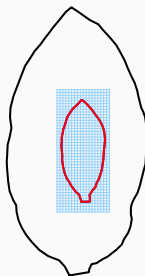
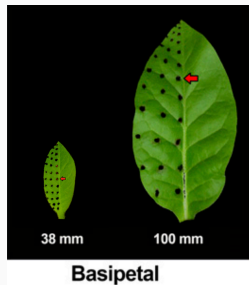
Advanced linear algebra operations

Motivation: deformation with implicit modules



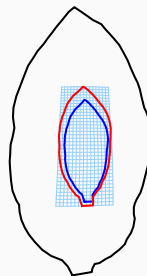
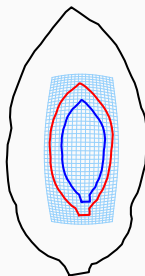
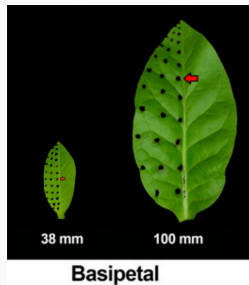
L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules



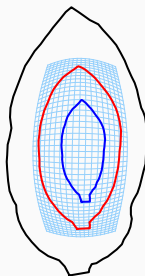
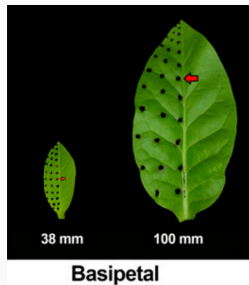
L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules

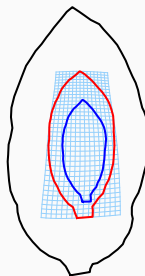


L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules



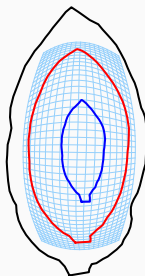
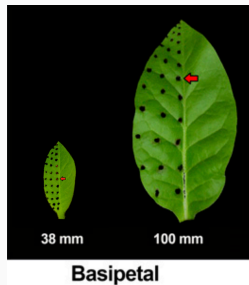
LDDMM



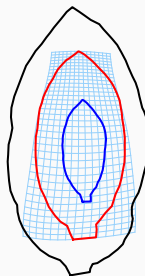
IMODAL

L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules



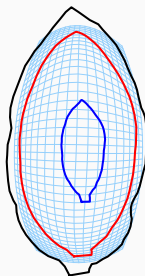
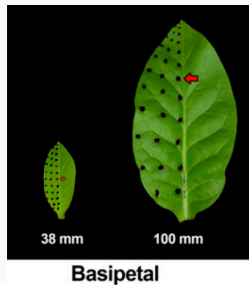
LDDMM



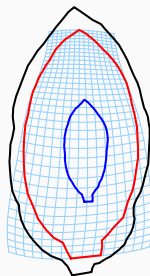
IMODAL

L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules



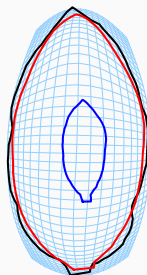
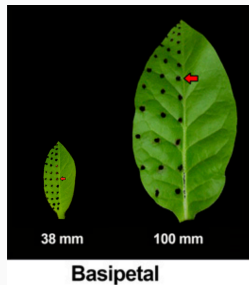
LDDMM



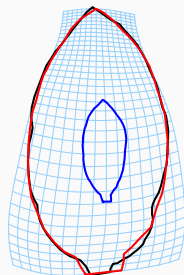
IMODAL

L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

Motivation: deformation with implicit modules



LDDMM



IMODAL

L. Lacroix, B. Charlier, A. Trouvé, B. Gris. *IMODAL: creating learnable user-defined deformation models* – CVPR, 2021

- Given a symmetric positive definite operator $K = [f(|x_i - x_j|/\sigma^2)]_{i,j=1}^n$ and $a \in \mathbb{R}^n$ we may use KeOps to compute $b \in \mathbb{R}^n$ such that

$$Ka = b$$

with a sum reduction.

- Given a symmetric positive definite operator $K = [f(|x_i - x_j|/\sigma^2)]_{i,j=1}^n$ and $a \in \mathbb{R}^n$ we may use KeOps to compute $b \in \mathbb{R}^n$ such that

$$Ka = b$$

with a sum reduction.

- Solve the inverse problem: given $b \in \mathbb{R}^n$ find $a \in \mathbb{R}^n$ such that

$$Ka = b \iff a = K^{-1}b.$$

...which is not a reduction.

- Given a symmetric positive definite operator $K = [f(|x_i - x_j|/\sigma^2)]_{i,j=1}^n$ and $a \in \mathbb{R}^n$ we may use KeOps to compute $b \in \mathbb{R}^n$ such that

$$Ka = b$$

with a sum reduction.

- Solve the inverse problem: given $b \in \mathbb{R}^n$ and $\alpha \geq 0$ find $a \in \mathbb{R}^n$ such that

$$(K + \alpha Id)a = b \Leftrightarrow a = (K + \alpha Id)^{-1}b.$$

...which is not a reduction.

- Given a symmetric positive definite operator $K = [f(|x_i - x_j|/\sigma^2)]_{i,j=1}^n$ and $a \in \mathbb{R}^n$ we may use KeOps to compute $b \in \mathbb{R}^n$ such that

$$Ka = b$$

with a sum reduction.

- Solve the inverse problem: given $b \in \mathbb{R}^n$ and $\alpha \geq 0$ find $a \in \mathbb{R}^n$ such that

$$(K + \alpha Id)a = b \Leftrightarrow a = (K + \alpha Id)^{-1}b.$$

...which is not a reduction.

Possible solutions:

- Gaussian elimination: store n^2 floats; operations: $2n^3/3$ (add, mult, sub, div).
- Conjugate gradient (CG): take advantage of a fast/cheap Matrix-Vector multiplication. Define an iterative procedure to compute an approximate solution x .

Remark: CG may outperform in term of storage and computation load.

Start with an initial guess $x_0 = b$:

- Take a first approximation to the solution as a multiple of b : $x_1 \in \text{Span}\{b\}$
- Compute Kb and search for an approximation as a linear combination of b and Kb :
 $x_2 \in \text{Span}\{b, Kb\}$
- $\vdots$
- Continue so that the solution belongs to the Krylov space

$$x_k \in \text{Span}\{b, Kb, K^2b, \dots, K^{\ell-1}b\} \quad (3)$$

Questions:

1. How can good (optimal) approximation from space (3) be computed with a moderate amount of work and storage ? (for Hermitian def pos K : use CG)
2. How good an approximation solution is contained in the space 3 ? (for small ℓ : use a preconditioners)

Let $K_\alpha = K + \alpha Id$. We may consider the quadratic problem

$$\underset{a}{\operatorname{argmin}} \frac{1}{2} a^t K_\alpha a - a^t b$$

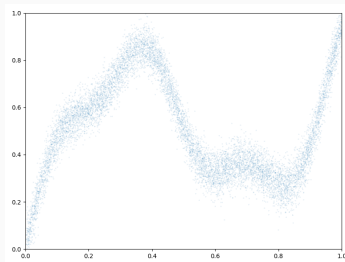
solved for $a = K_\alpha^{-1} b$.

- CG approximates the solution with an iterative procedure involving only terms using “ $K_\alpha a$ ”
- Converge in at most n steps... and if we are lucky a good approximation is given in $\ell \ll n$ steps...but, due to round-off errors **maxiter = n*10** in **scipy**
- Convergence speed depends on the distribution of the eigenvalues of K_α (condition number of K_α is a poor indicator)

Example: Interpolation 1d

```
N = 10000
# Sampling locations:
x = np.random.rand(N, 1)
# Some random-ish 1D signal:
b = x + .5 * np.sin(6 * x) + .1 * np.sin(20 * x) + .05 * np.random.randn(N, 1)

sigma = .1 # Kernel radius
alpha = 1. # Ridge regularization
```



Example: Interpolation 1d

Let V be a RKHS with radial Gaussian Kernel. We want to solve

$$\operatorname{argmin}_{v \in V} \|v\|_V^2 + \frac{1}{\alpha} \|v(x_i) - b_i\|_2^2$$

```
def gaussian_kernel(x, y, sigma=0.1):  
    x_i = LazyTensor(x[:, None, :]) # (M, 1, 1)  
    y_j = LazyTensor(y[None, :, :]) # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1) # (M, N) symbolic matrix of squared distances  
    return (-D_ij / (2 * sigma**2)).exp() # (M, N) symbolic Gaussian kernel matrix
```

Example: Interpolation 1d

Let V be a RKHS with radial Gaussian Kernel. We want to solve

$$\operatorname{argmin}_{v \in V} \|v\|_V^2 + \frac{1}{\alpha} \|v(x_i) - b_i\|_2^2$$

```
def gaussian_kernel(x, y, sigma=0.1):  
    x_i = LazyTensor(x[:, None, :]) # (M, 1, 1)  
    y_j = LazyTensor(y[None, :, :]) # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1) # (M, N) symbolic matrix of squared distances  
    return (-D_ij / (2 * sigma**2)).exp() # (M, N) symbolic Gaussian kernel matrix  
  
# Instantiate KeOps routine  
K_xx = gaussian_kernel(x, x)
```

Example: Interpolation 1d

Let V be a RKHS with radial Gaussian Kernel. We want to solve

$$\operatorname{argmin}_{v \in V} \|v\|_V^2 + \frac{1}{\alpha} \|v(x_i) - b_i\|_2^2$$

```
def gaussian_kernel(x, y, sigma=0.1):  
    x_i = LazyTensor(x[:, None, :]) # (M, 1, 1)  
    y_j = LazyTensor(y[None, :, :]) # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1) # (M, N) symbolic matrix of squared distances  
    return (-D_ij / (2 * sigma**2)).exp() # (M, N) symbolic Gaussian kernel matrix  
  
# Instantiate KeOps routine  
K_xx = gaussian_kernel(x, x)  
  
# Performe the computations  
a = K_xx.solve(b, alpha=alpha)
```

Example: Interpolation 1d

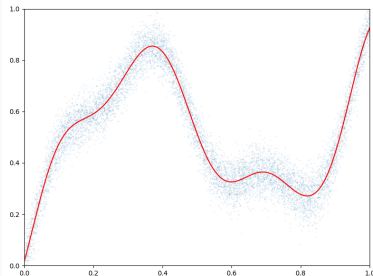
```
# Extrapolate on a uniform sample:  
t = np.reshape(np.linspace(0, 1, 1001), [1001, 1]).astype(dtype)  
  
K_tx = gaussian_kernel(t, x)  
mean_t = K_tx @ a
```

Example: Interpolation 1d

```
# Extrapolate on a uniform sample:
t = np.reshape(np.linspace(0, 1, 1001), [1001, 1]).astype(dtype)

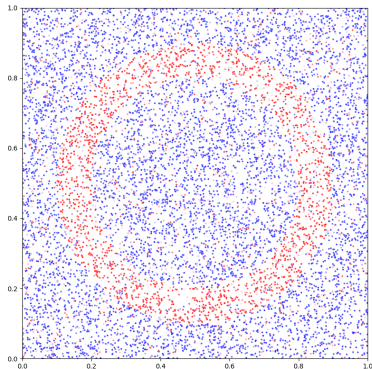
K_tx = gaussian_kernel(t, x)
mean_t = K_tx @ a

plt.scatter(x[:, 0], b[:, 0], s=100 / len(x)) # Noisy samples
plt.plot(t, mean_t, "r")
```



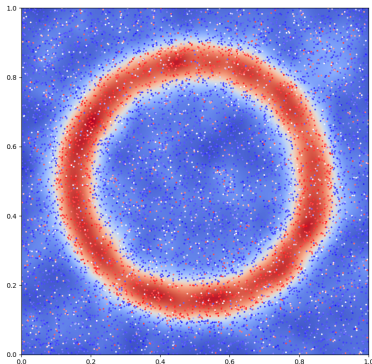
Example: Interpolation 2d

```
def laplacian_kernel(x, y, sigma=0.1):  
    x_i = LazyTensor(x[:, None, :]) # (M, 1, 1)  
    y_j = LazyTensor(y[None, :, :]) # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1) # (M, N) symbolic matrix of squared distances  
    return (-D_ij.sqrt() / sigma).exp() # (M, N) symbolic Laplacian kernel matrix
```



Example: Interpolation 2d

```
def laplacian_kernel(x, y, sigma=0.1):  
    x_i = LazyTensor(x[:, None, :]) # (M, 1, 1)  
    y_j = LazyTensor(y[None, :, :]) # (1, N, 1)  
    D_ij = ((x_i - y_j) ** 2).sum(-1) # (M, N) symbolic matrix of squared distances  
    return (-D_ij.sqrt() / sigma).exp() # (M, N) symbolic Laplacian kernel matrix
```



Create a toy dataset:

```
import numpy as np
N = 10000
x = np.random.randn(N,2)
```

Turn it into a KeOps **LazyTensor**:

```
from pykeops.numpy import LazyTensor
x_i, x_j = LazyTensor(x[:, None, :]), LazyTensor(x[None, :, :])
K_xx = (- ((x_i - x_j) ** 2).sum(2) / 2).exp() # Symbolic (N,N) Gaussian kernel matrix
```

Using a **sum** reduction we may compute a Gaussian convolution...

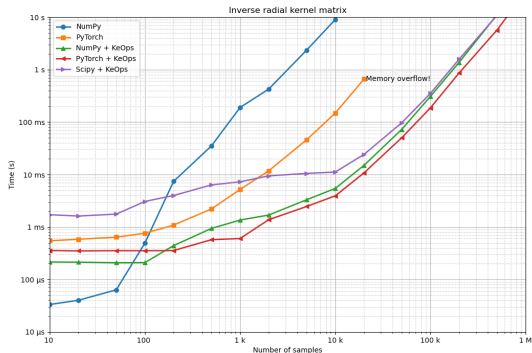
...instead, K_{xx} can be directly understood as a `LinearOperator`:

```
from scipy.sparse.linalg import aslinearoperator
K = aslinearoperator(K_xx)

from scipy.sparse.linalg import eigsh
eigenvalues, eigenvectors = eigsh(K, k=5) # Largest 5 eigenvalues/vectors

# Largest eigenvalues: [ 626.59143  639.14667  663.38983  747.5554  1438.2162 ]
# Eigenvectors of shape: (10000, 5)
```

Benchmark for KernelSolve (Gaussian kernel, dimension 3)



see http://www.kernel-operations.io/keops/_auto_benchmarks/plot_benchmark_invkernel.html