Fast geometric methods with symbolic matrices

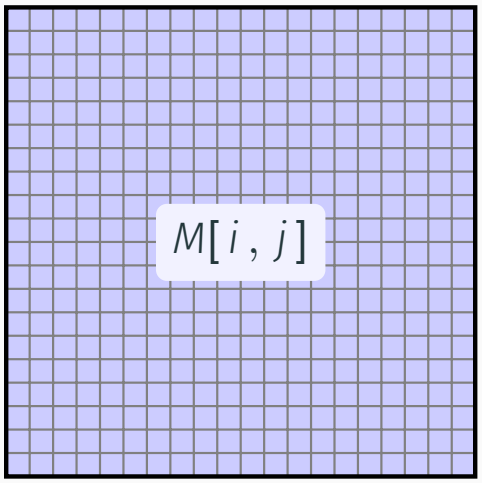
Benjamin Charlier

SMAI2021, La Grande Motte — juin 2021.

Joint work with J. Feydy (Imperial College London), J. Glaunès (Université de Paris)

KeOps and symbolic matrices

Scientific computing libraries represent most objects as tensors

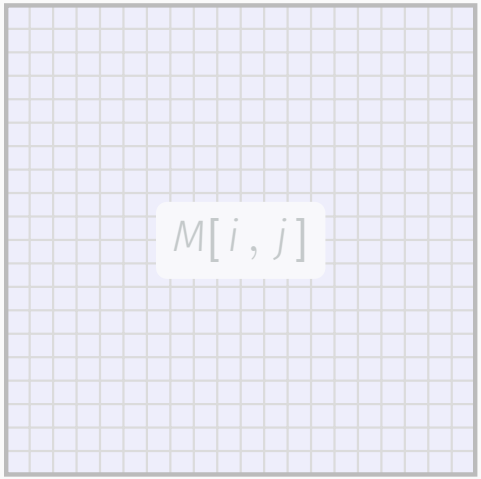


Dense matrix
Coefficients only

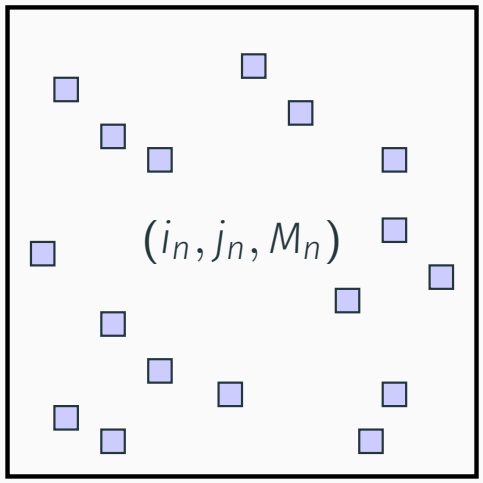
Dense matrices – large, contiguous **arrays** of numbers:

+ **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

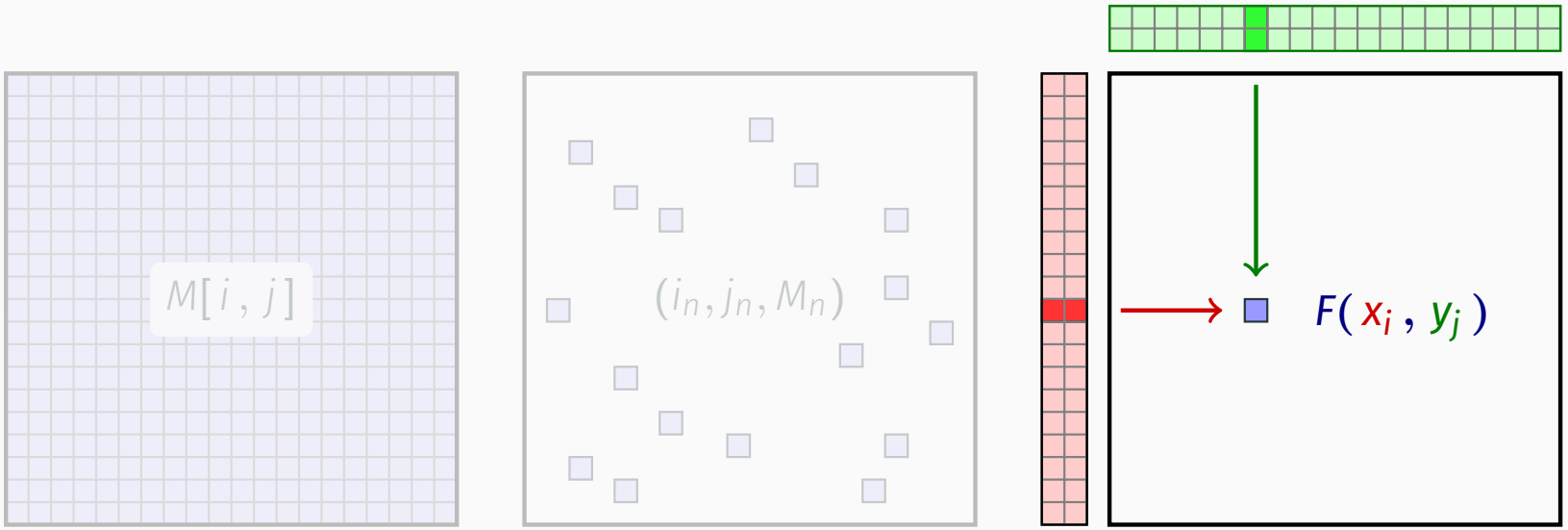


Sparse matrix
Coordinates + coeffs

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



Dense matrix
Coefficients only

Sparse matrix
Coordinates + coeffs

Symbolic matrix
Formula + data

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**.
+ Comprehensive collection of **tutorials**, available online.

Under the hood: combines an optimized **C++** engine with high-level binders for **PyTorch**, **NumPy**, **Matlab** and **R** (thanks to Ghislain Durif).
(We welcome **contributors** for JAX, Julia and other frameworks!)

To get started:

$\Longrightarrow$ `pip install pykeops` $\Longleftarrow$

`www.kernel-operations.io`

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:

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

- ...an even more generic case:

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

where $\bigstar$ 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.argmin(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 bruteforce 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:** $+$, $\times$, sqrt, exp, neural networks, etc.
- **Advanced schemes:** batch processing, block sparsity, etc.
- **Automatic differentiation:** seamless integration with PyTorch.

Some simple examples

- Tutorials: `http://www.kernel-operations.io/keops/_auto_tutorials/a_LazyTensors/plot_lazytensors_a.html`
- Testimony

On 05/10/2020 15:56, Simon M. wrote:
Vers 1000 particules ça commence à vraiment devenir long...

On 06/11/2020 16:07, Ghislain D. wrote:
Wow ! Un facteur 300, rien que ça ! C'est déjà super de pouvoir taquiner les 10^4 particules.

On 08/11/2020 23:49, Benjamin C. wrote:
voici une implementation de velocity_update avec keops.

Benchmark velocity update result

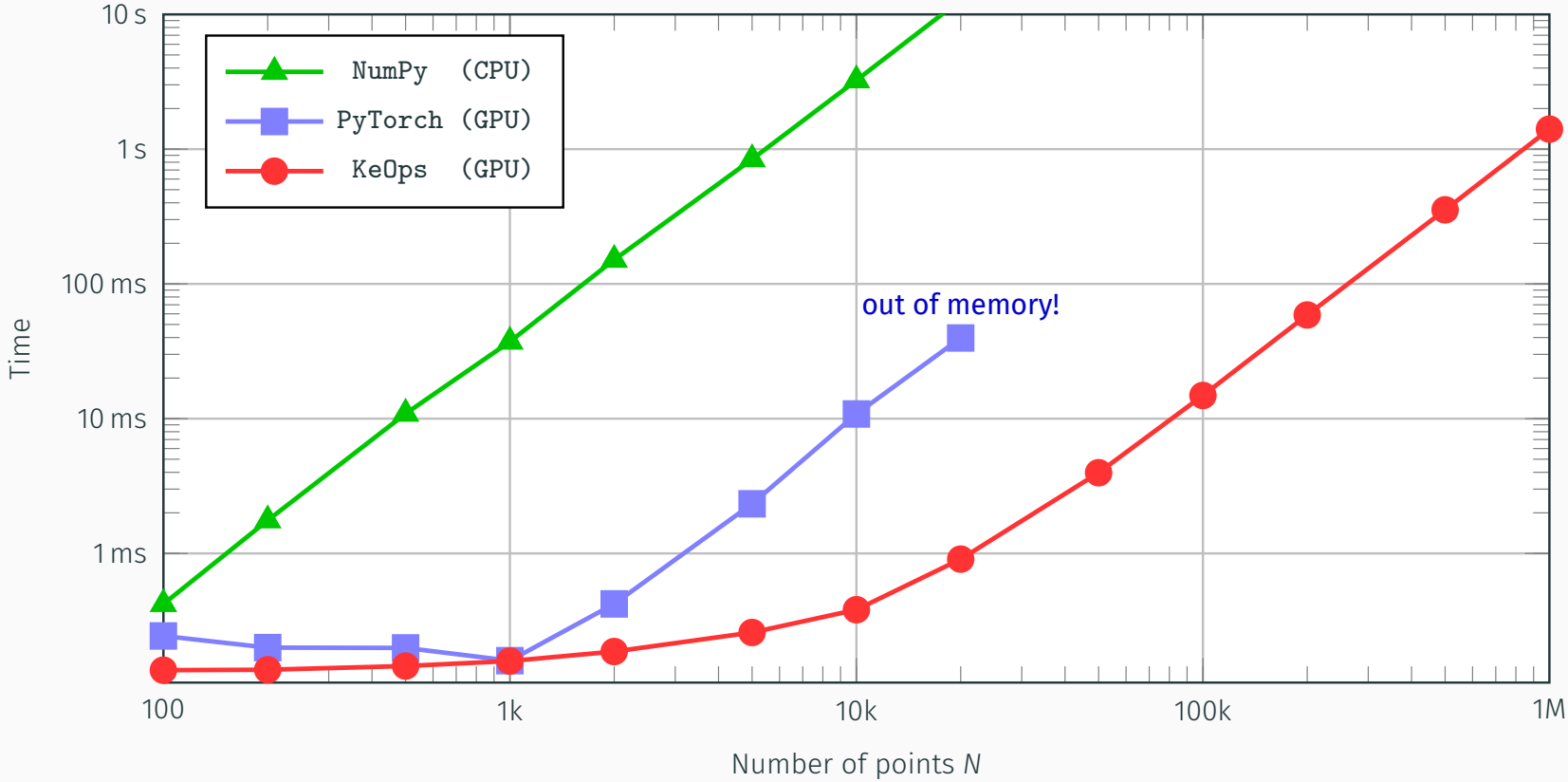
N_part	time_keops32	time_keops	time_numpy
10000	0.00314142	0.0390402	25.2532
50000	0.0273194	0.556156	101.917
100000	0.0650266	2.02865	
200000	0.248779	8.14202	
500000	1.45907	50.1849	

On 10/11/2020 08:57, Simon M. wrote:
wow^2. Ca blow my mind, mais tu m'avais prévenu, Benjamin...

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.



Advanced linear algebra operations

Inverse SPD matrix

- Given a symmetric positive definite operator $K = \left[f(|x_i - x_j|/\sigma^2)\right]_{i,j=1}^{n}$ 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.

Inverse SPD matrix

- Given a symmetric positive definite operator $K = \left[f(|x_i - x_j|/\sigma^2)\right]_{i,j=1}^{n}$ 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.

Inverse SPD matrix

- Given a symmetric positive definite operator $K = \left[f(|x_i - x_j|/\sigma^2)\right]_{i,j=1}^{n}$ 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.

Conjugate gradient solver

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

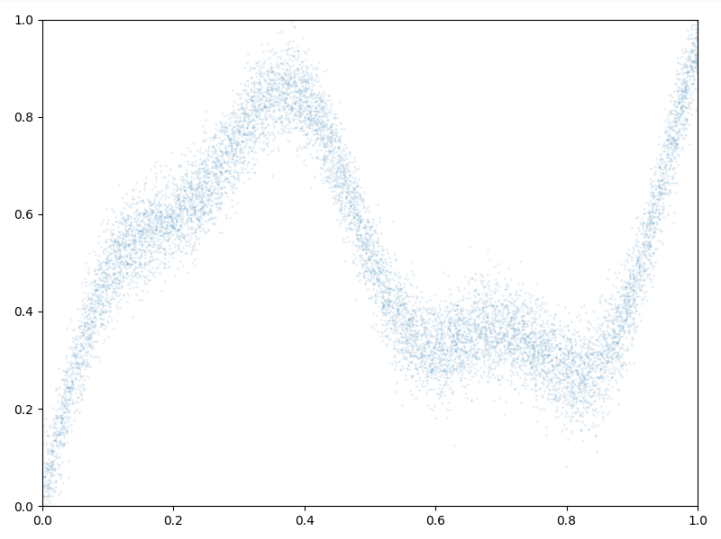
$$\underset{a}{\text{argmin}} \ \frac{1}{2} a^t K_\alpha a - a^t b$$

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

- Approximate 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 $\ll n$ steps.
- Convergence speed given by the condition number of K_α

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
g = np.array([.5 / sigma ** 2])  # RBF bandwidth parameter
alpha = 1.  # Ridge regularization
```



Example: Interpolation 1d

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

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

```
formula = "Exp(- G * SqDist(X,Y) ) * C"  # Gaussian kernel matrix
aliases = ["X = Vi(1)",  # 1st arg: target points, i-variable of size 1
           "Y = Vj(1)",  # 2nd arg: source points, j-variable of size 1
           "C = Vj(1)",  # 3rd arg: source signal, j-variable of size 1
           "G = Pm(1)"]  # 4th arg: scalar parameter, 1/(2*std**2)
```

Example: Interpolation 1d

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

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

```
formula = "Exp(- G * SqDist(X,Y) ) * C"  # Gaussian kernel matrix
aliases = ["X = Vi(1)",  # 1st arg: target points, i-variable of size 1
           "Y = Vj(1)",  # 2nd arg: source points, j-variable of size 1
           "C = Vj(1)",  # 3rd arg: source signal, j-variable of size 1
           "G = Pm(1)"]  # 4th arg: scalar parameter, 1/(2*std**2)

# Instantiate KeOps routine
Kinv = KernelSolve(formula, aliases, "C", axis=1)
```

Example: Interpolation 1d

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

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

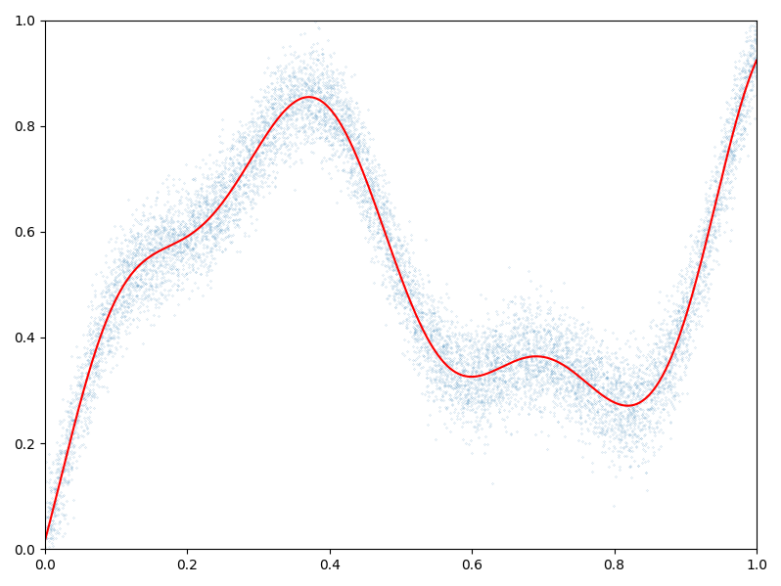
```
formula = "Exp(- G * SqDist(X,Y) ) * C"  # Gaussian kernel matrix
aliases = ["X = Vi(1)",  # 1st arg: target points, i-variable of size 1
           "Y = Vj(1)",  # 2nd arg: source points, j-variable of size 1
           "C = Vj(1)",  # 3rd arg: source signal, j-variable of size 1
           "G = Pm(1)"]  # 4th arg: scalar parameter, 1/(2*std**2)

# Instantiate KeOps routine
Kinv = KernelSolve(formula, aliases, "C", axis=1)

# Performe the computations
a = Kinv(x, x, b, g, alpha=alpha)
```

Example: Interpolation 1d

```
t = np.linspace(0, 1, 1001)
K = Genred(formula, aliases, reduction='Sum', axis=1)
xt = K(t, x, a, g)
```

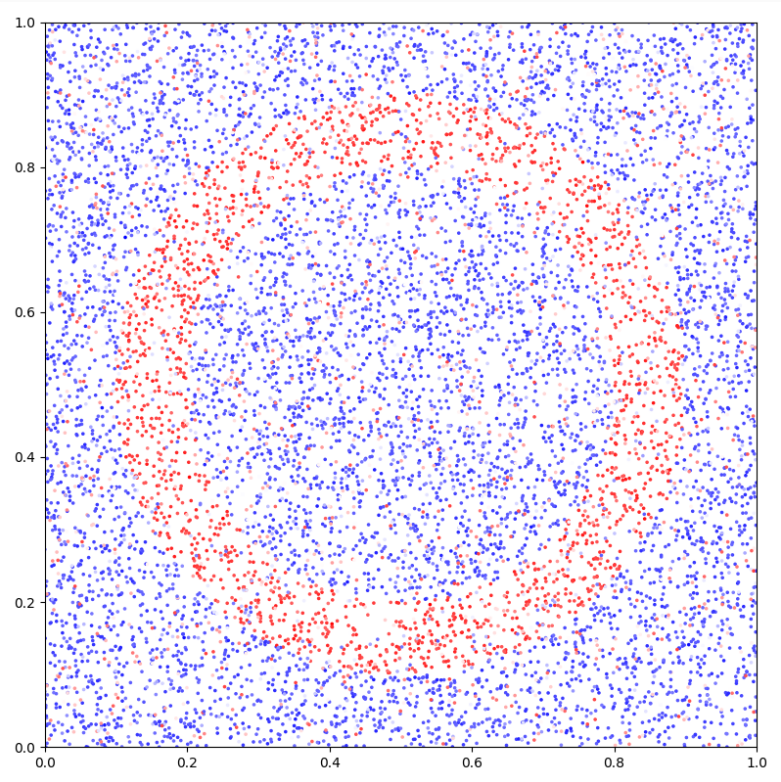
Example: Interpolation 1d

```
t = np.linspace(0, 1, 1001)
K = Genred(formula, aliases, reduction='Sum', axis=1)
xt = K(t, x, a, g)
```



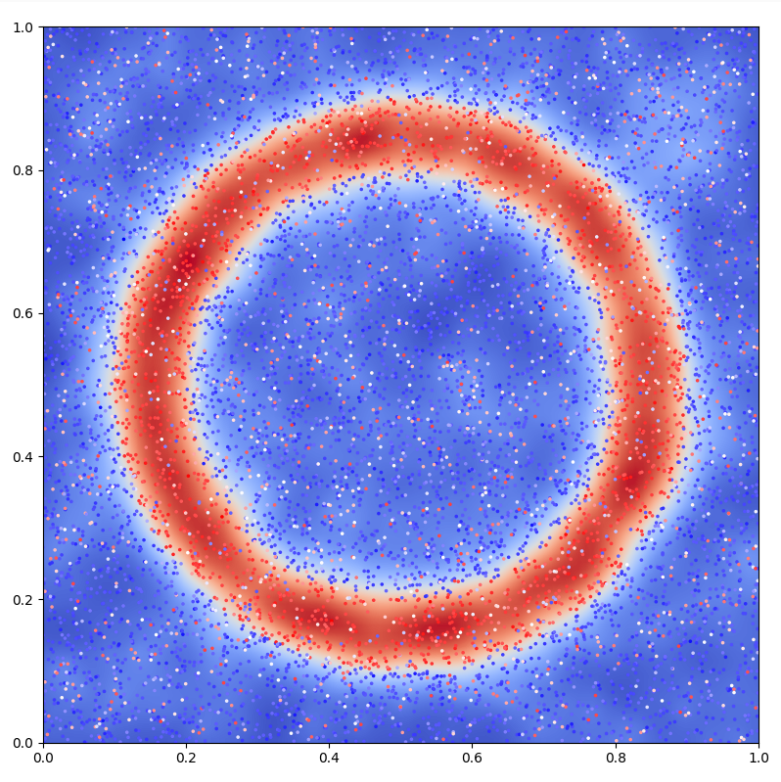
Example: Interpolation 2d

```
formula = "Exp(- G * Norm2(X-Y) ) * C"  # Laplacian kernel matrix
aliases = ["X = Vi(2)",   # 1st arg: target points, i-variable of size 2
           "Y = Vj(2)",   # 2nd arg: source points, j-variable of size 2
           "C = Vj(1)",   # 3rd arg: source signal, j-variable of size 1
           "G = Pm(1)"]   # 4th arg: scalar parameter, 1/std
```



Example: Interpolation 2d

```
formula = "Exp(- G * Norm2(X-Y) ) * C"  # Laplacian kernel matrix
aliases = ["X = Vi(2)",   # 1st arg: target points, i-variable of size 2
           "Y = Vj(2)",   # 2nd arg: source points, j-variable of size 2
           "C = Vj(1)",   # 3rd arg: source signal, j-variable of size 1
           "G = Pm(1)"]   # 4th arg: scalar parameter, 1/std
```



Using Scipy LinAlg routines

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
```

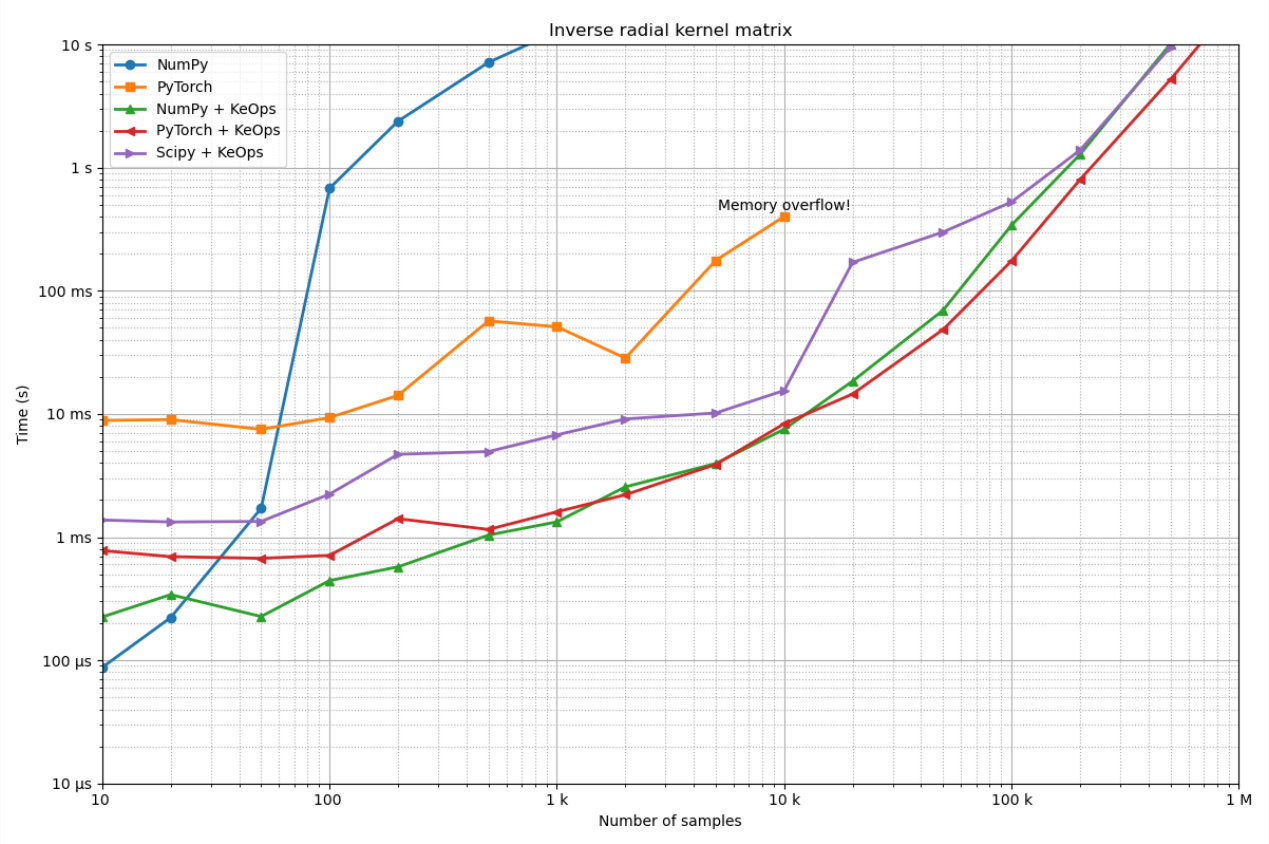
Using a **sum** reduction we may compute a Gaussian convolution...

Using Scipy LinAlg routines

...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`

Autodiff engine

Autodiff engine

Introduction

Finite differences?

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

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

Finite differences?

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

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

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

Reverse AD

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

Reverse AD

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

- the adjoint of the differential is $(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$. Riesz representation theorem gives a map

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

  called generalized **gradient**.

Reverse AD

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

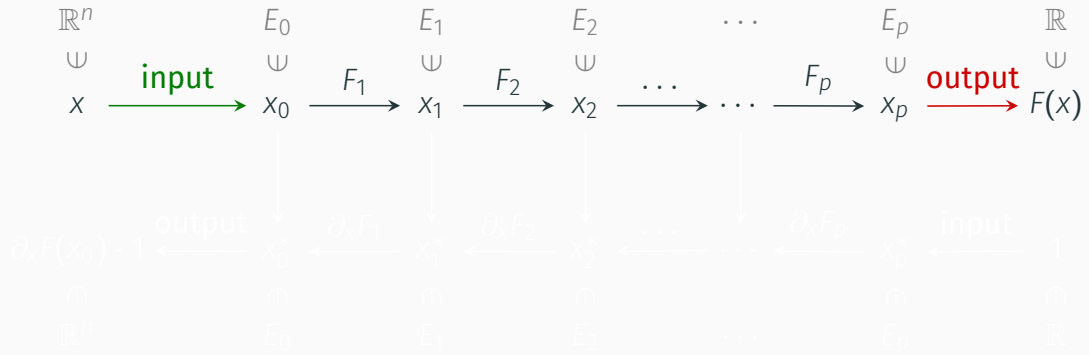
- the adjoint of the differential is $(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$. Riesz representation theorem gives a map

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

  called generalized **gradient**.
- If $X = \mathbb{R}^n$, $Y = \mathbb{R}$ endowed with the Euclidean metric,

$$\mathcal{M}_{\partial_x F(x_0)} = \begin{pmatrix} \partial_{x1} F(x_0) \\ \partial_{x2} F(x_0) \\ \vdots \\ \partial_{xn} F(x_0) \end{pmatrix} = \nabla_x F(x_0)$$

Reverse AD = backpropagating = chain rules

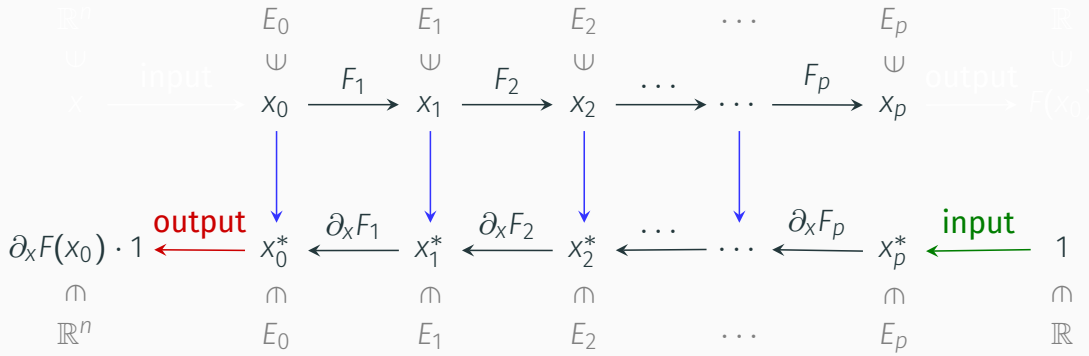


Backpropagating through a computational graph requires:

$$\begin{array}{cccc} F_i & : & E_{i-1} & \to & E_i \\ & & x & \mapsto & F_i(x) \end{array} \tag{1}$$

encoded as **computer programs**.

Reverse AD = backpropagating = chain rules

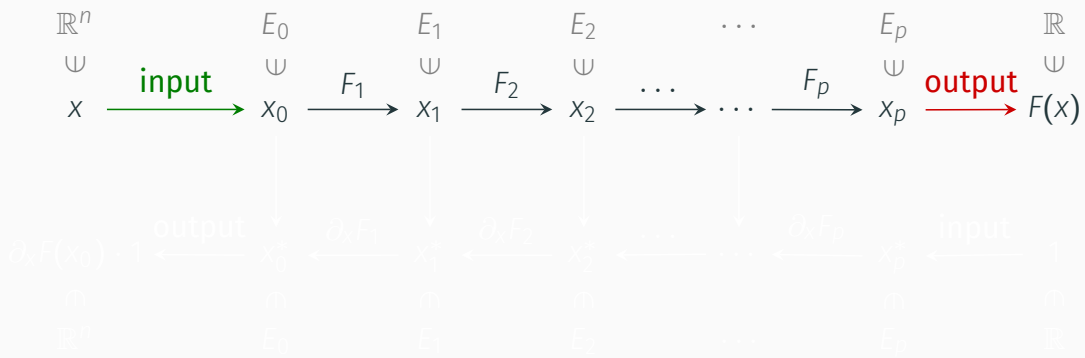


Backpropagating through a computational graph requires:

$$\begin{array}{cccccc} F_i & : & E_{i-1} & \to & E_i \\ & & x & \mapsto & F_i(x) \end{array} \qquad \begin{array}{cccccc} \partial_x F_i & : & E_{i-1} \times E_i & \to & E_{i-1} \\ & & (x_{i-1}, x_i^*) & \mapsto & \partial_x F_i(x_{i-1}) \cdot x_i^* \end{array} \tag{1}$$

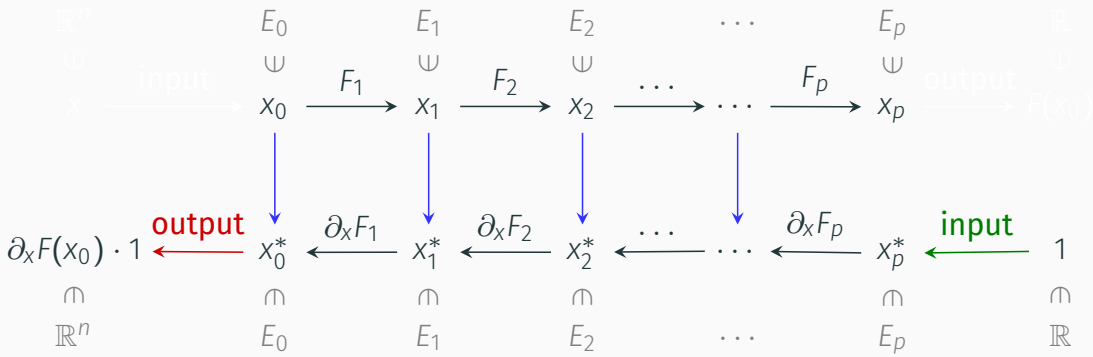
encoded as **computer programs**.

Reverse AD = backpropagating = chain rules



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 = chain rules



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

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

   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

Pytorch bindings

Torch autograd: A first 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 first 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 first 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]]),)
```

Autodiff engine

KeOps autograd

KeOps autograd

Let the formula

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

where

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

Compute $F : \mathbb{R}^M \times \mathbb{R}^N \to \mathbb{R}^M$ 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)
# [pyKeOps] Compiling libKeOpstorch39addd09d9 in /home/.../pykeops/build:
#   formula: Sum_Reduction(Exp((Var(0,1,0) + Var(1,1,1))),0)
#   aliases: Var(0,1,0); Var(1,1,1);
#   dtype  : float32
# ... Done.

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

KeOps autograd

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

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

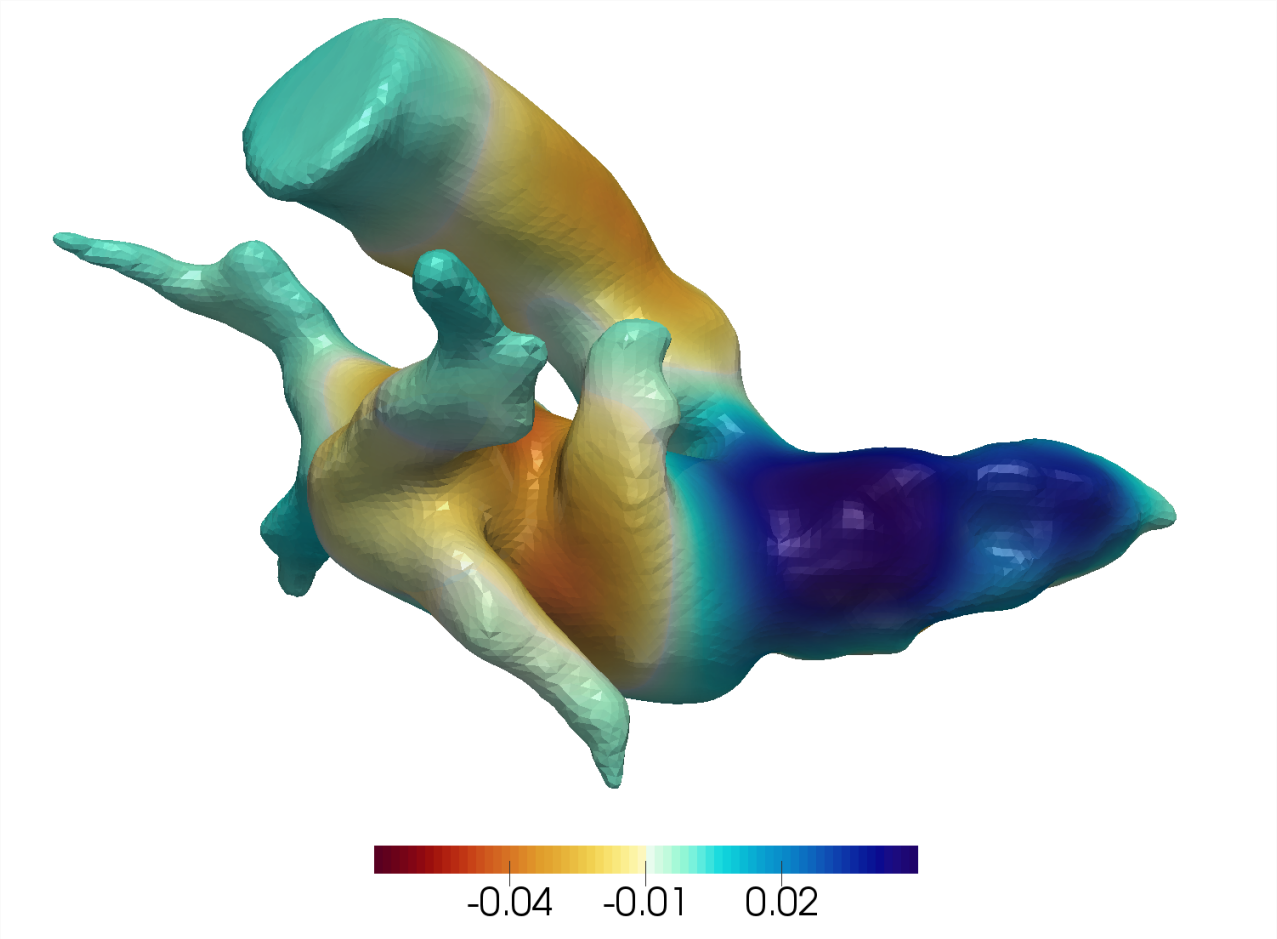
Compute dF with KeOps and `Grad( , , )` operator:

```
e = torch.rand_like(x)

F_grad_y = torch.autograd.grad(F, [y], e)[0]
# [pyKeOps] Compiling libKeOpstorchfe441a4e76 in /home/.../pykeops/build:
#   formula: Grad(Sum_Reduction(Exp((X + Y)),0), X, E)
#   aliases: X = Var(0,1,0); Y = Var(1,1,1); E = Var(2,1,0);
#   dtype  : float32
# ...  Done.
# [pyKeOps] Compiling libKeOpstorch7036350482 in /home/.../pykeops/build:
#   formula: Grad(Sum_Reduction(Exp((X + Y)),0), Y, E)
#   aliases: X = Var(0,1,0); Y = Var(1,1,1); E = Var(2,1,0);
#   dtype  : float32
# ... Done.

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

Autodiff engine

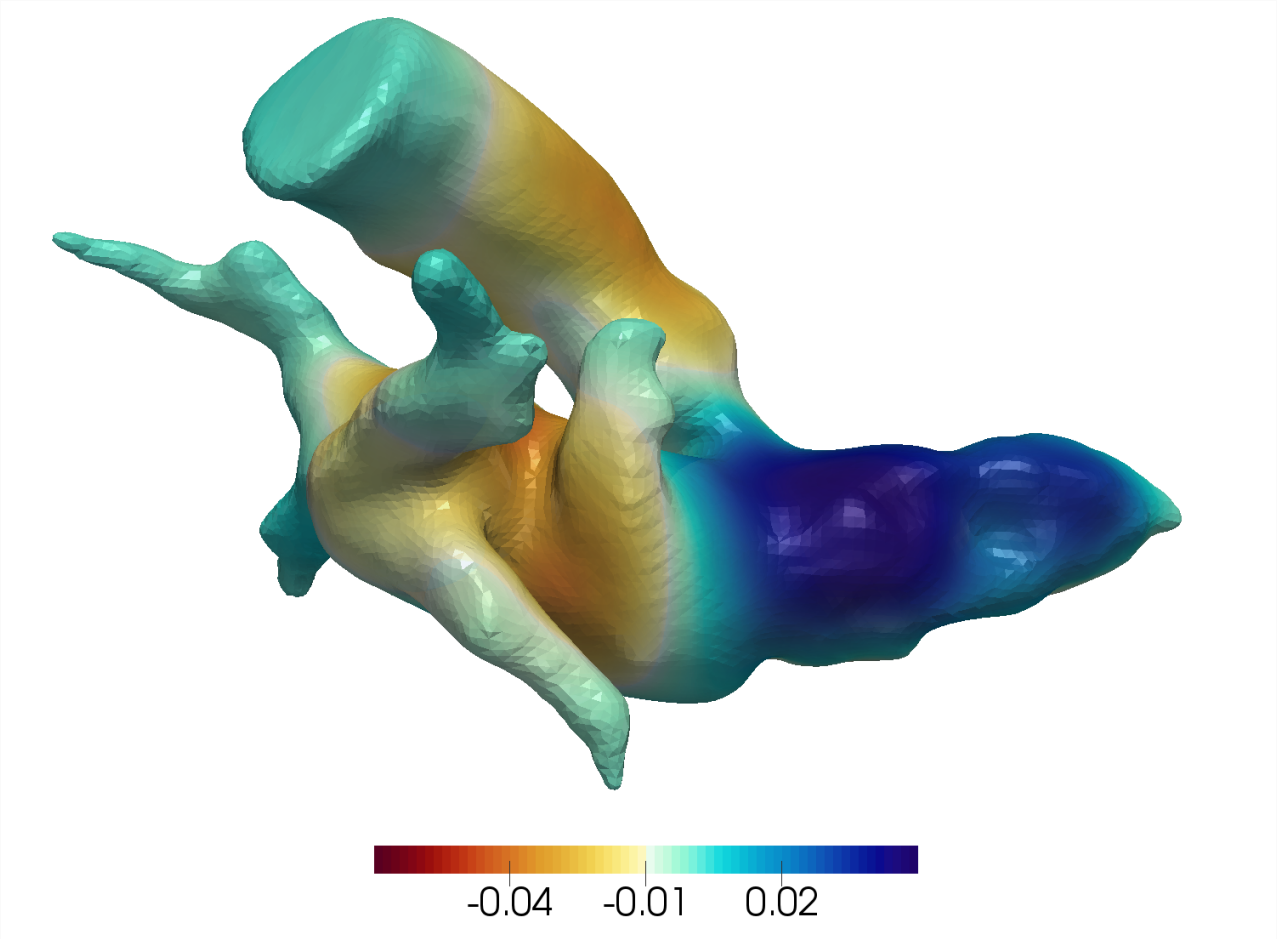
An example: LDDMM

Motivations: LDDMM



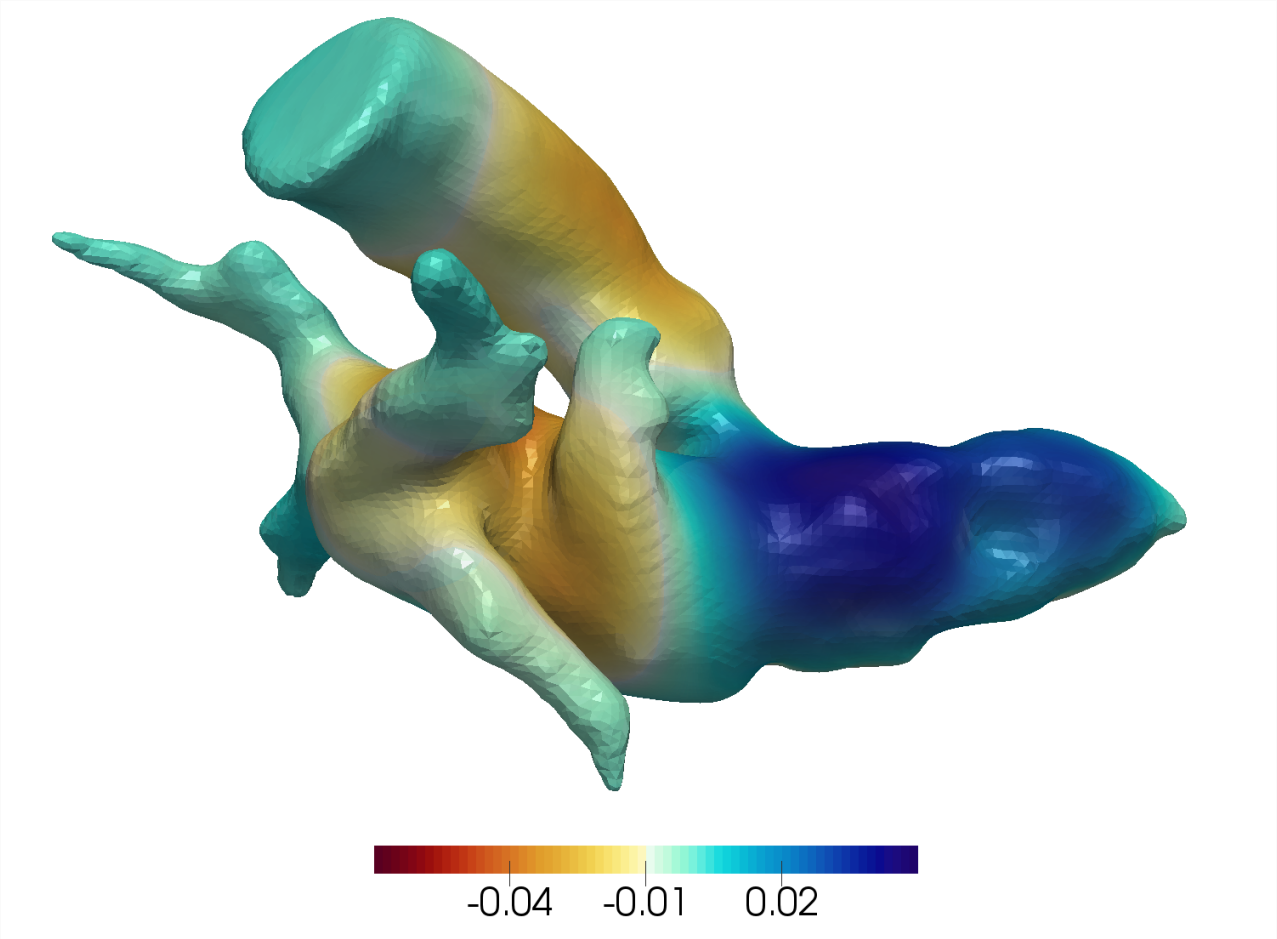
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



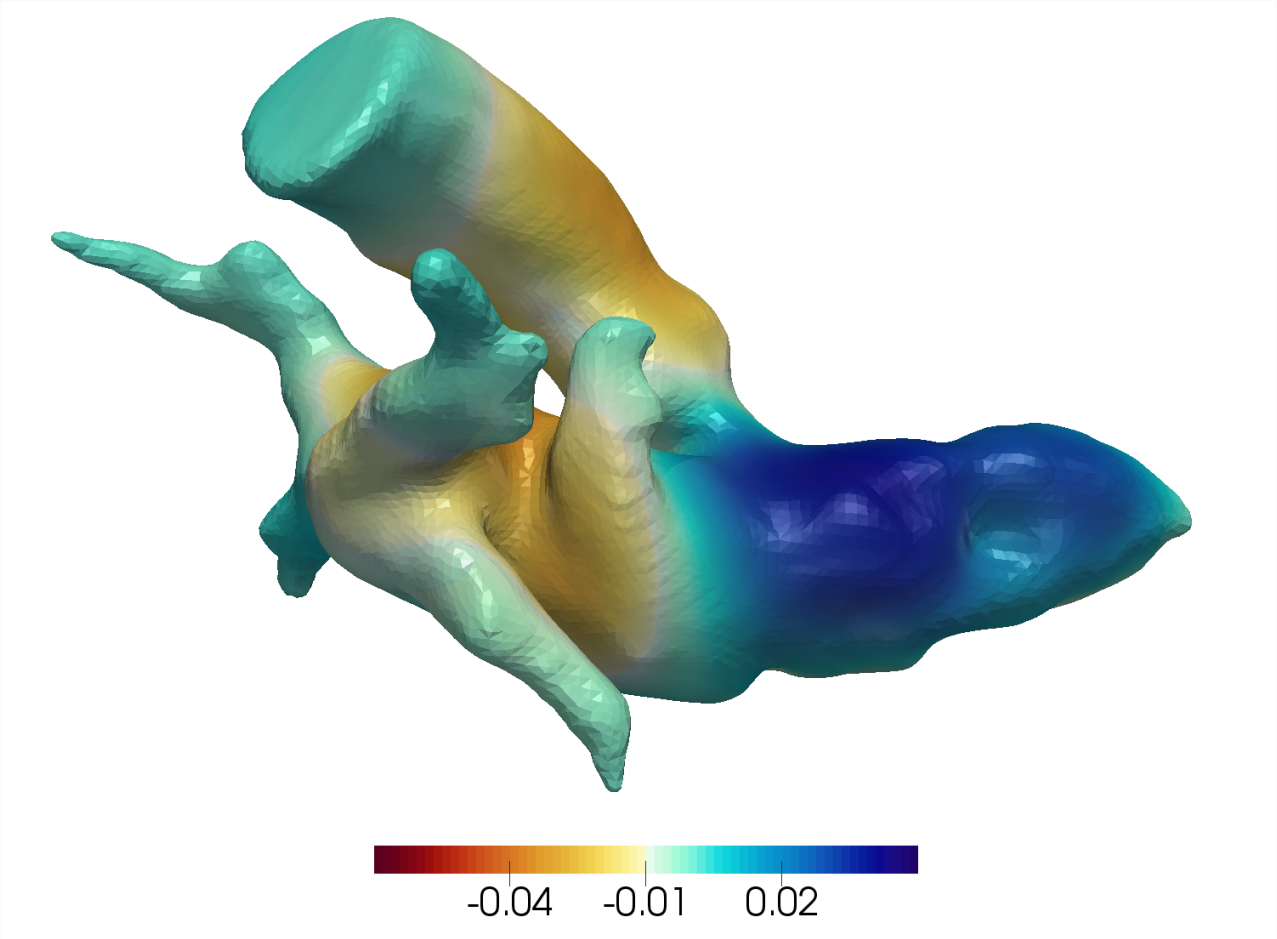
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



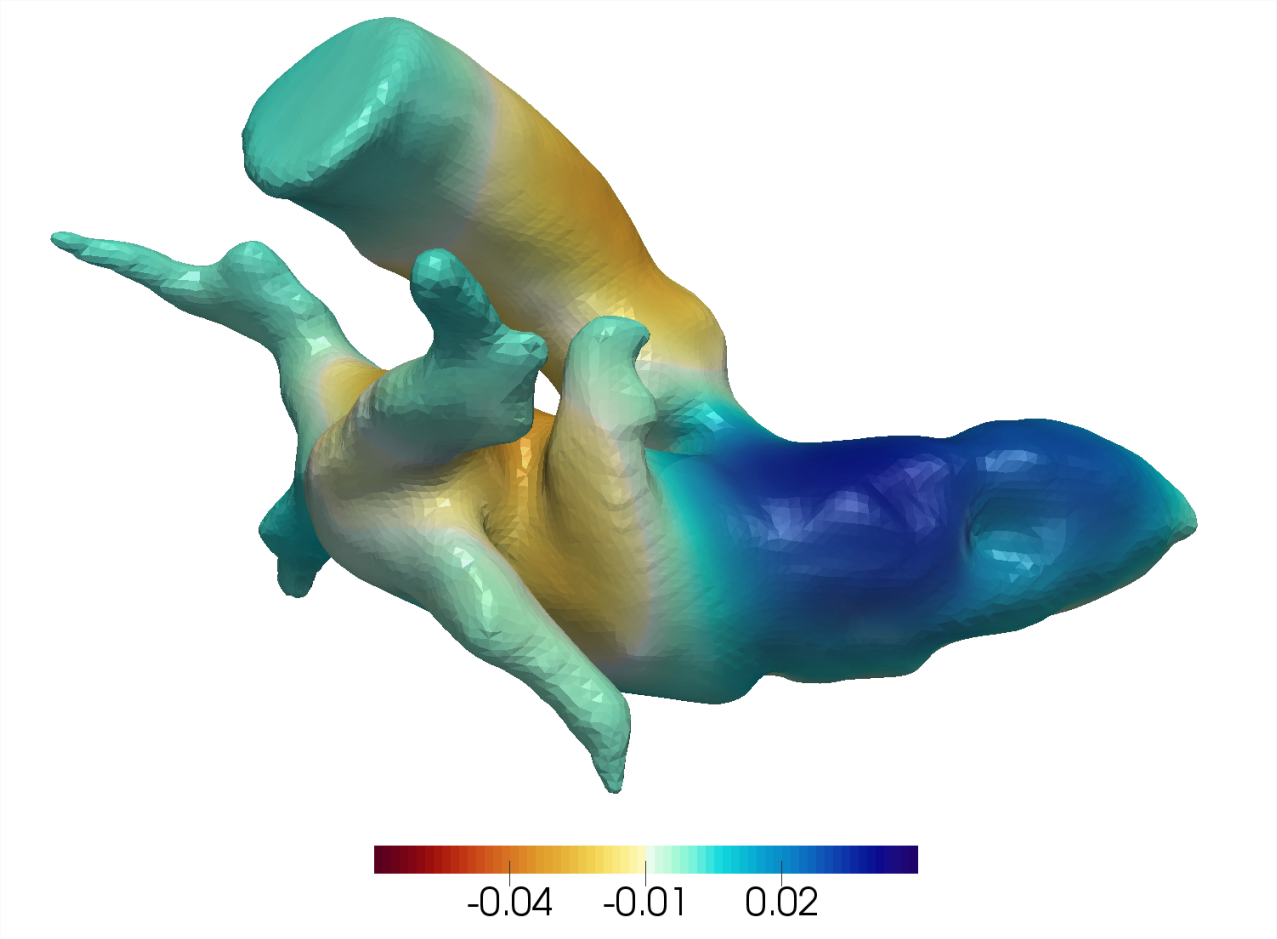
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



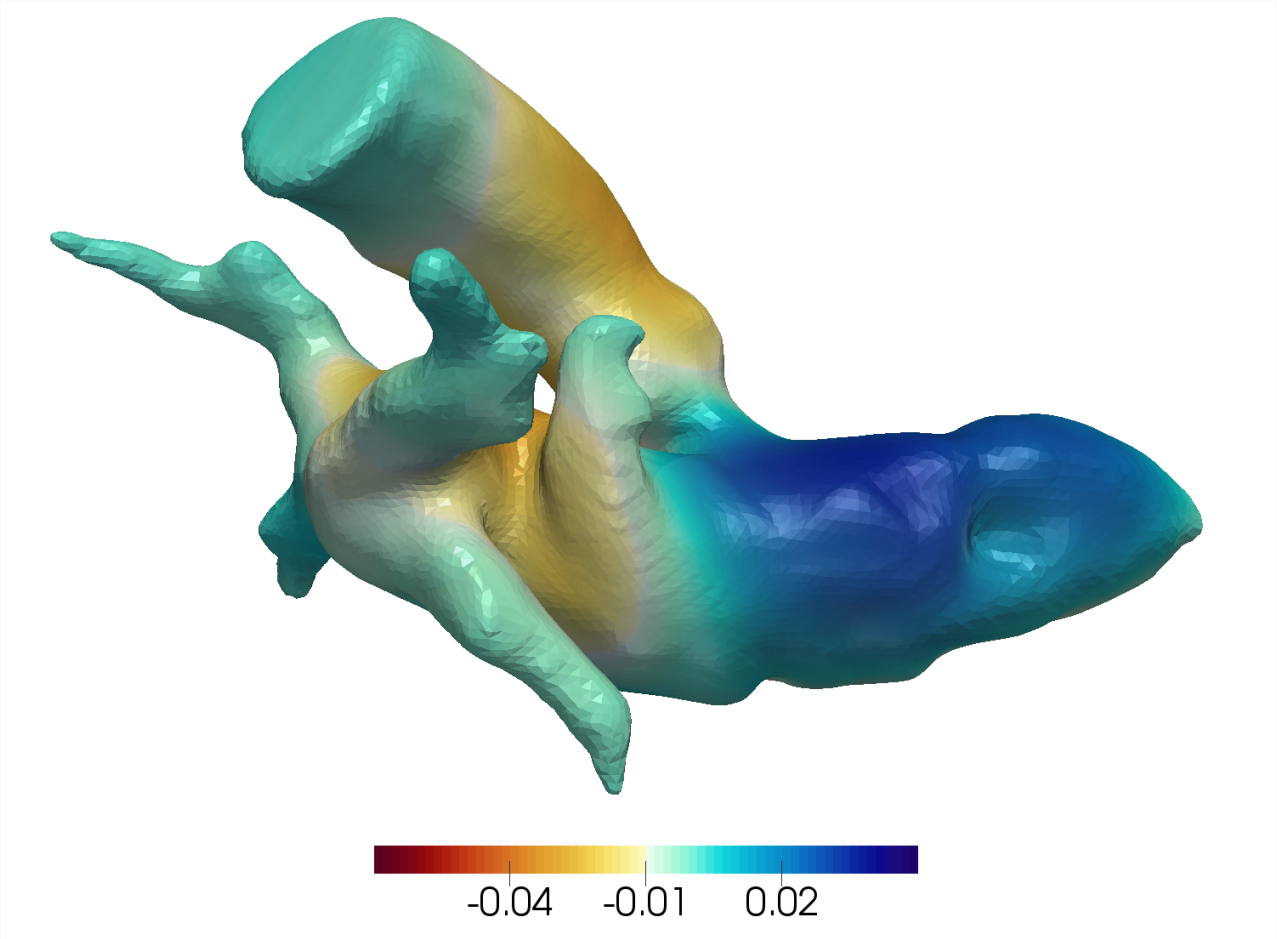
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



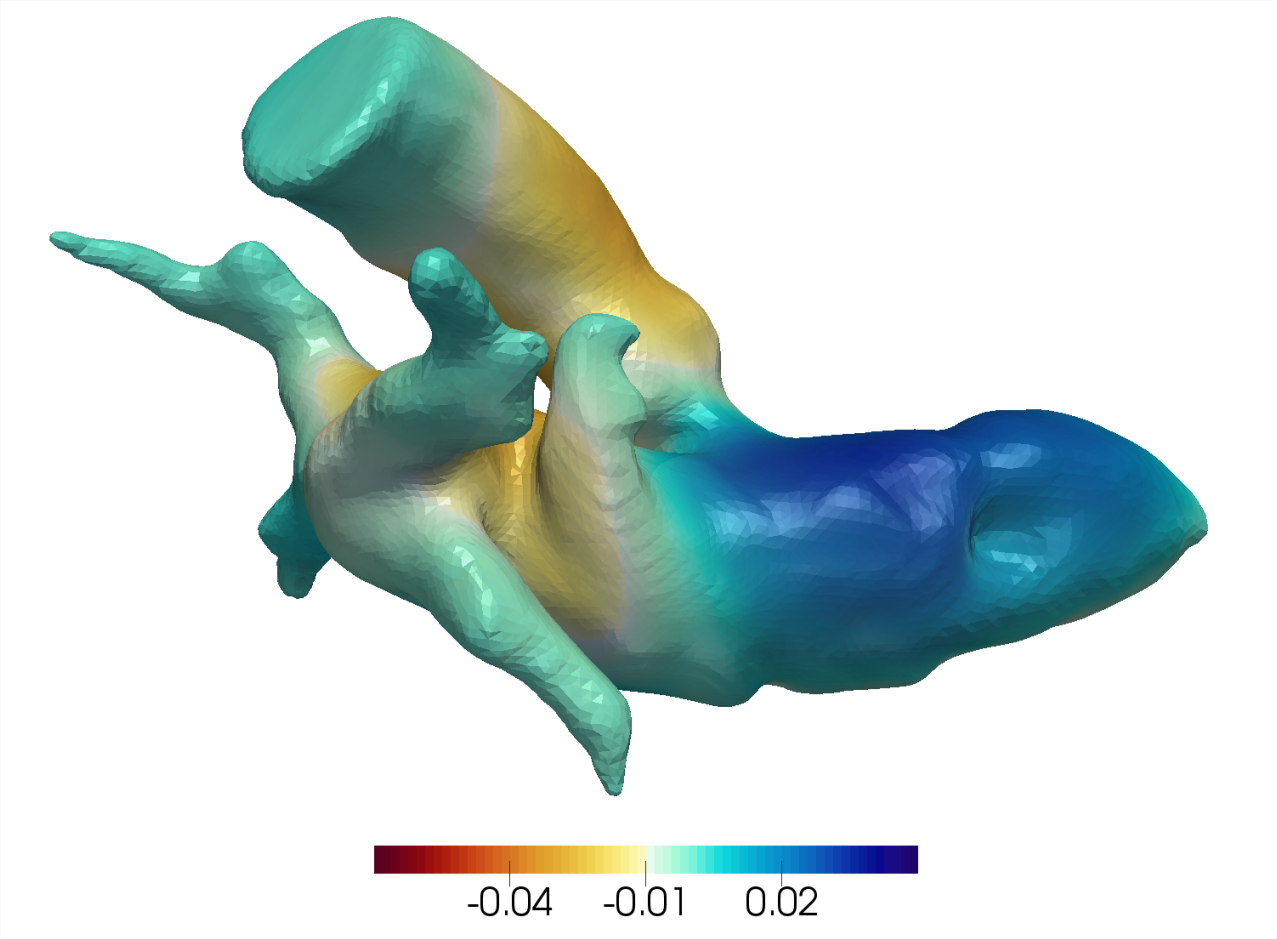
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



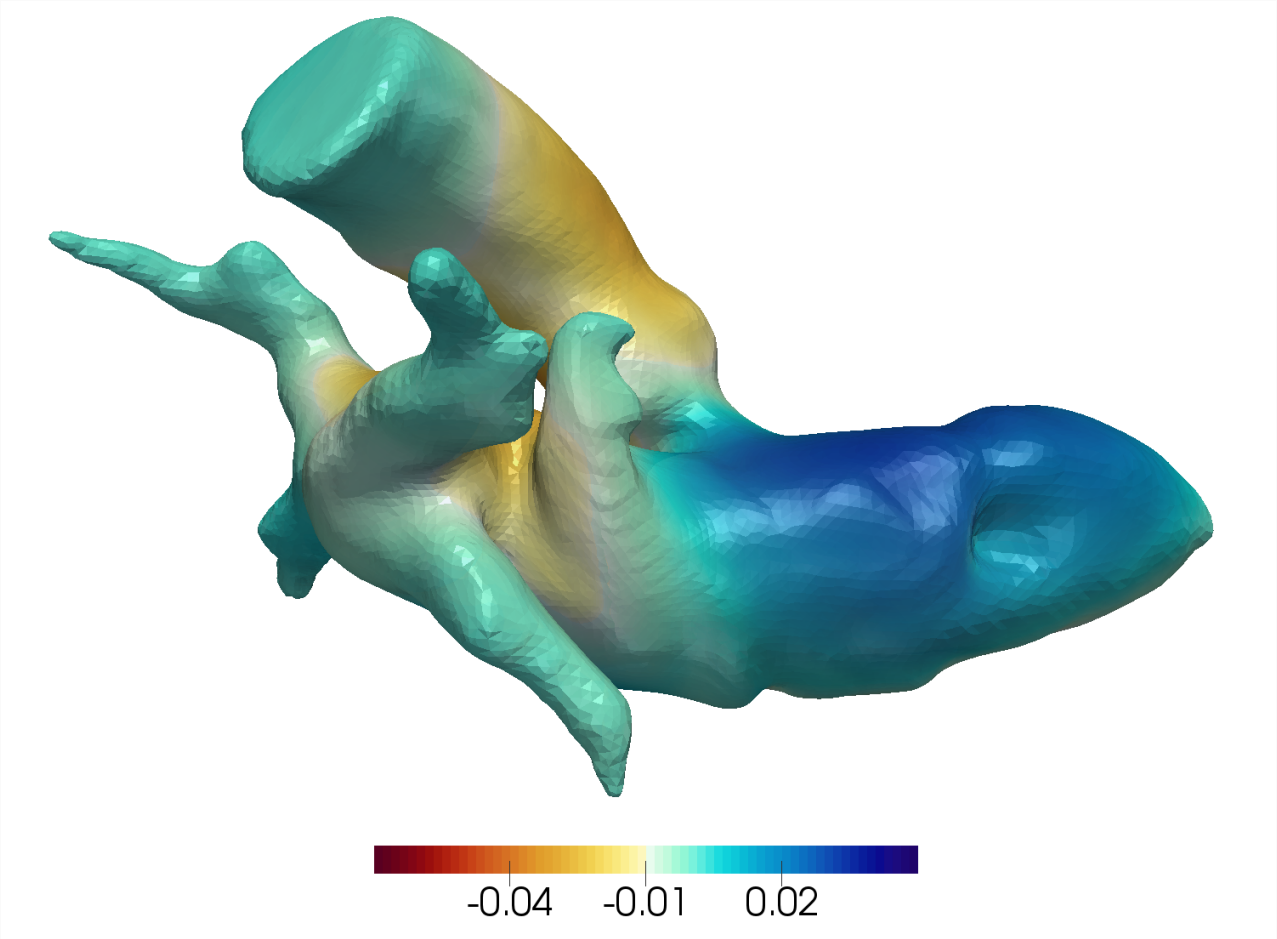
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



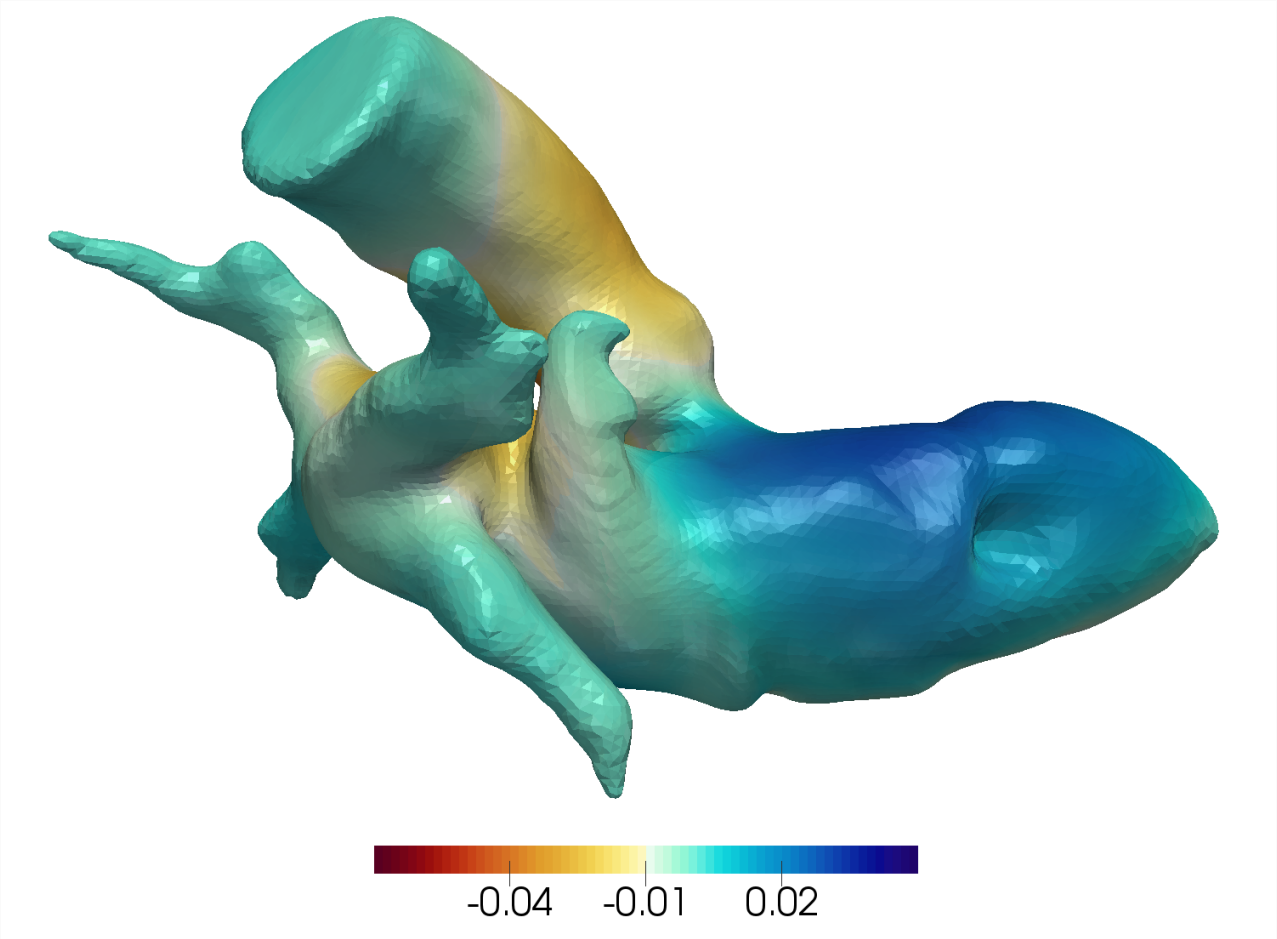
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



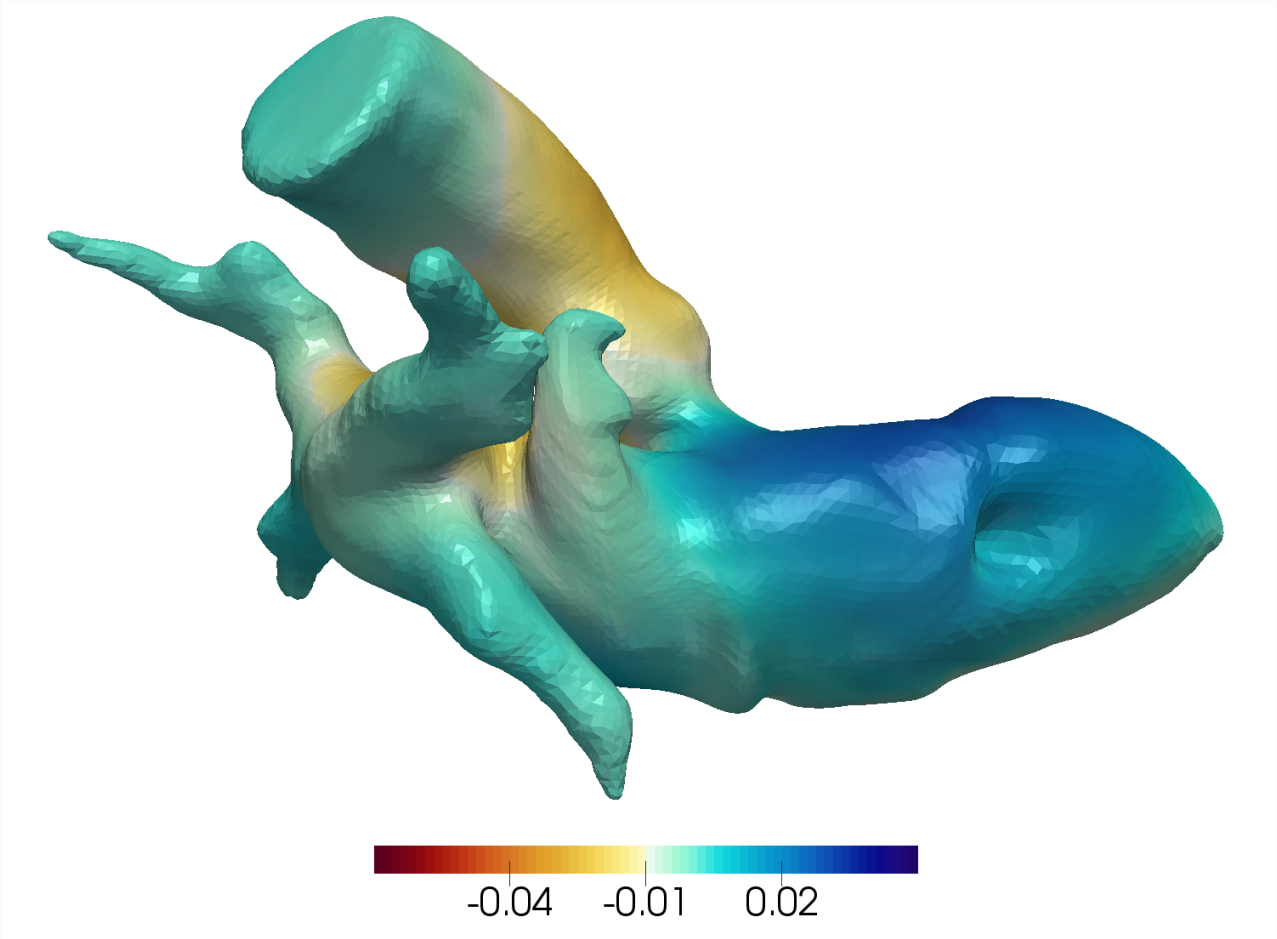
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



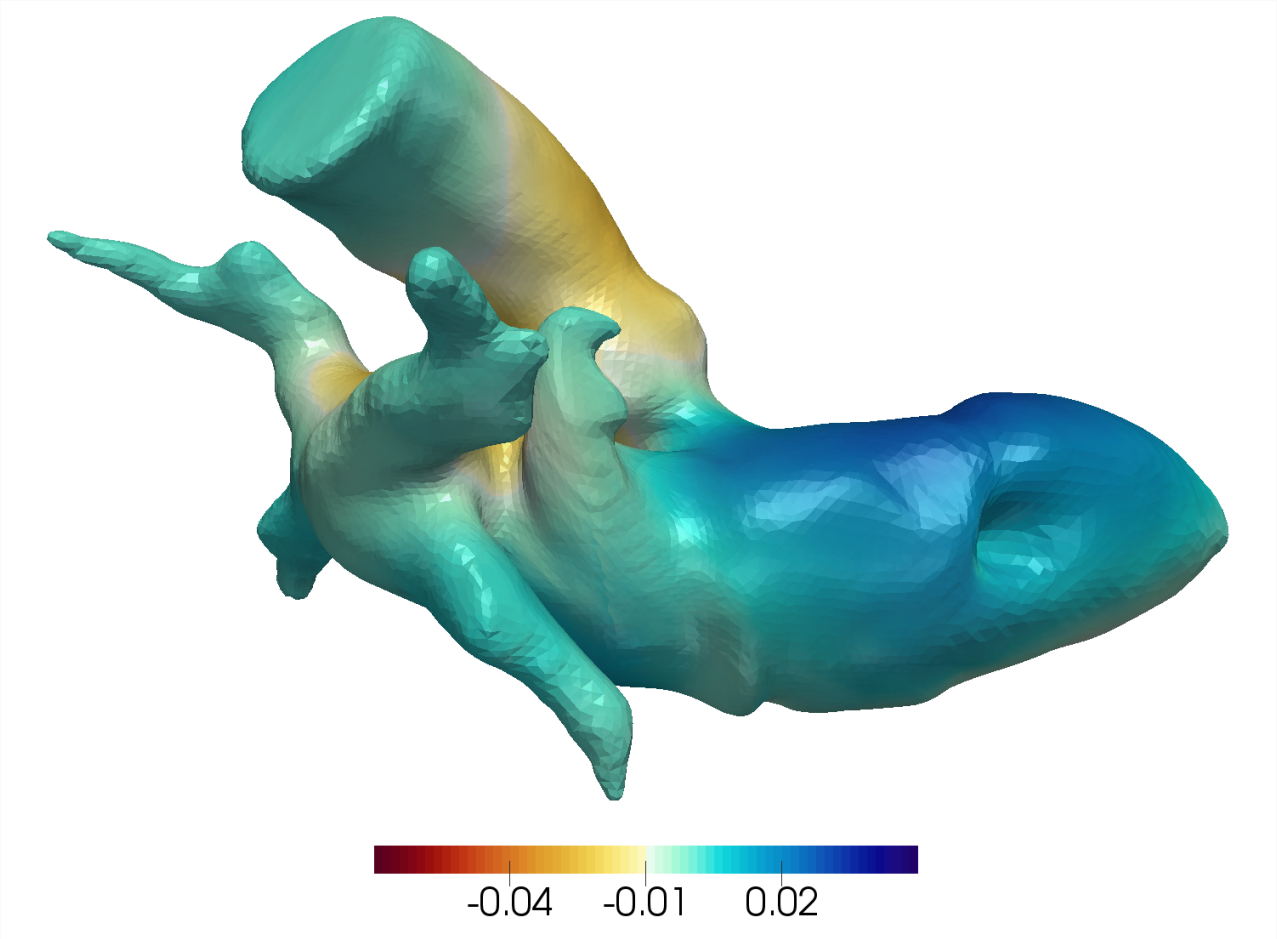
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



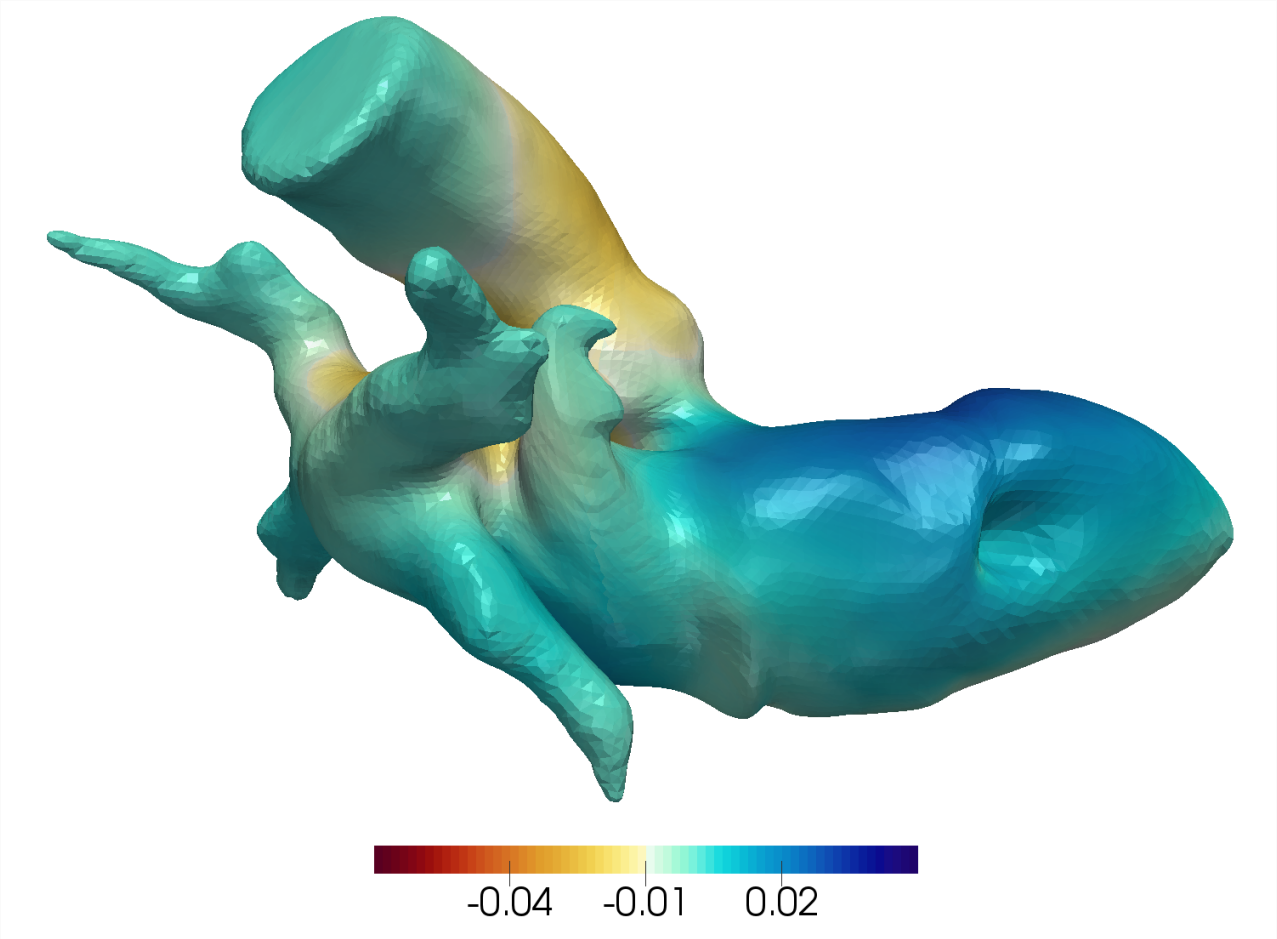
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



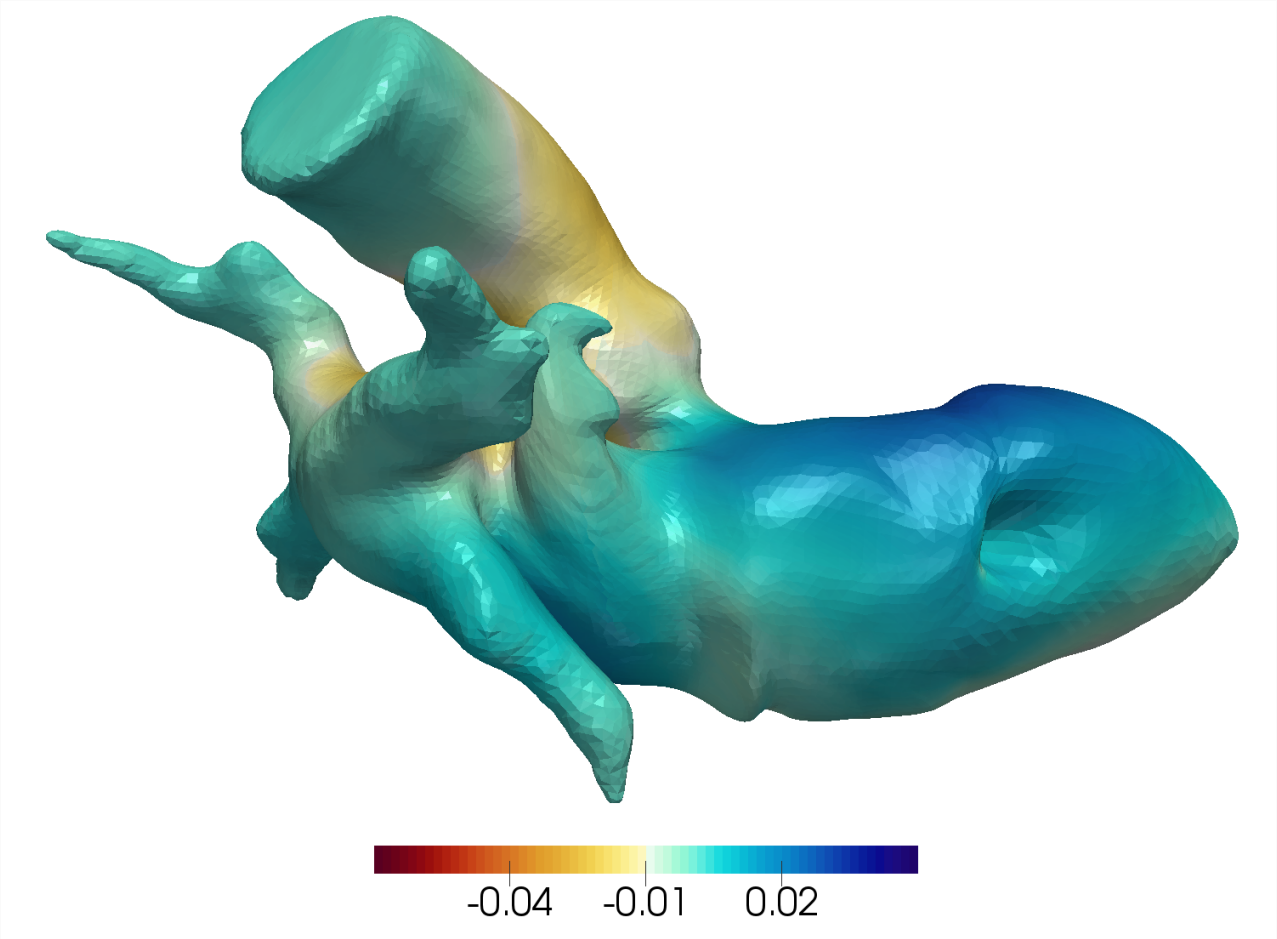
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



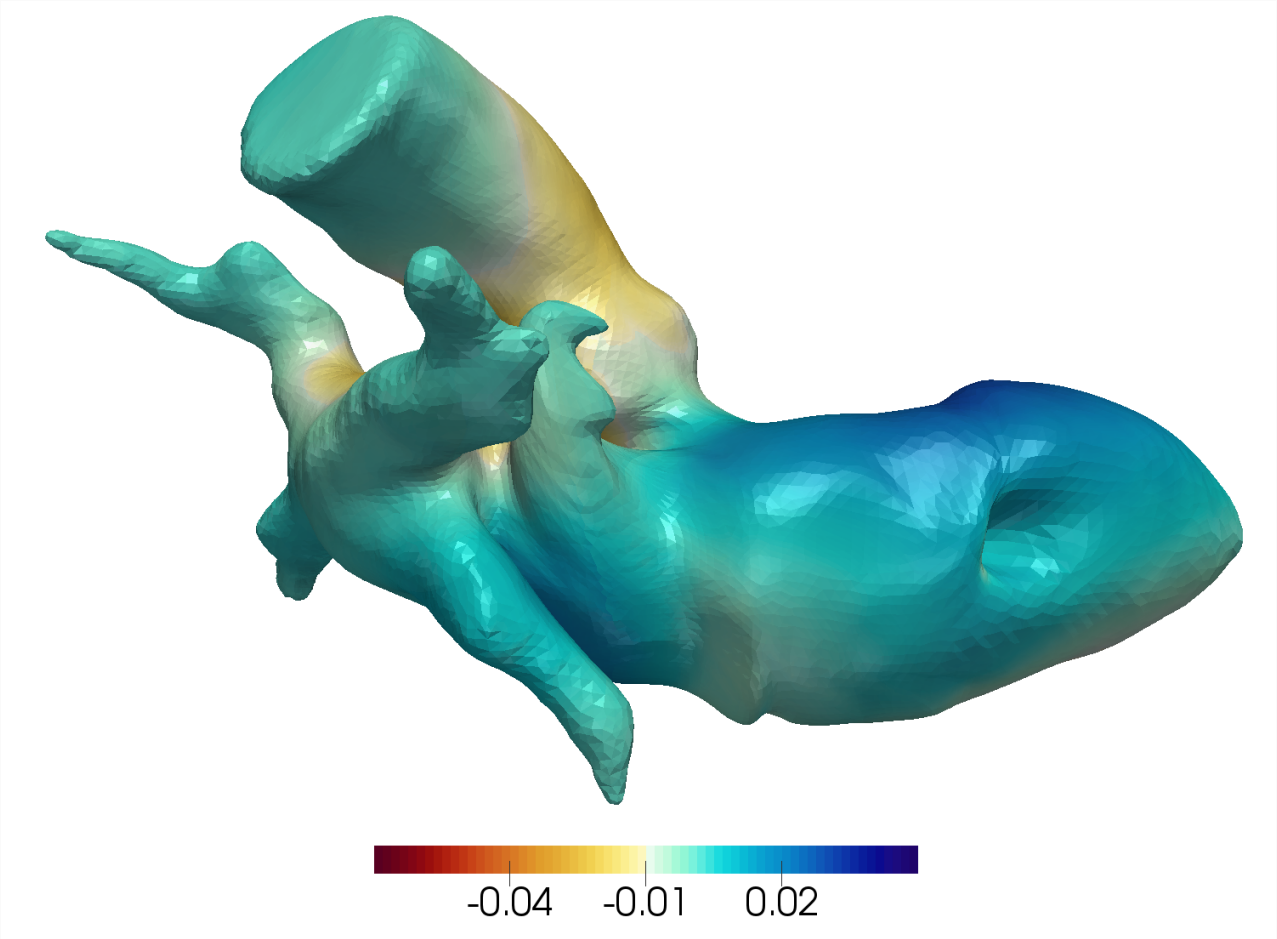
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



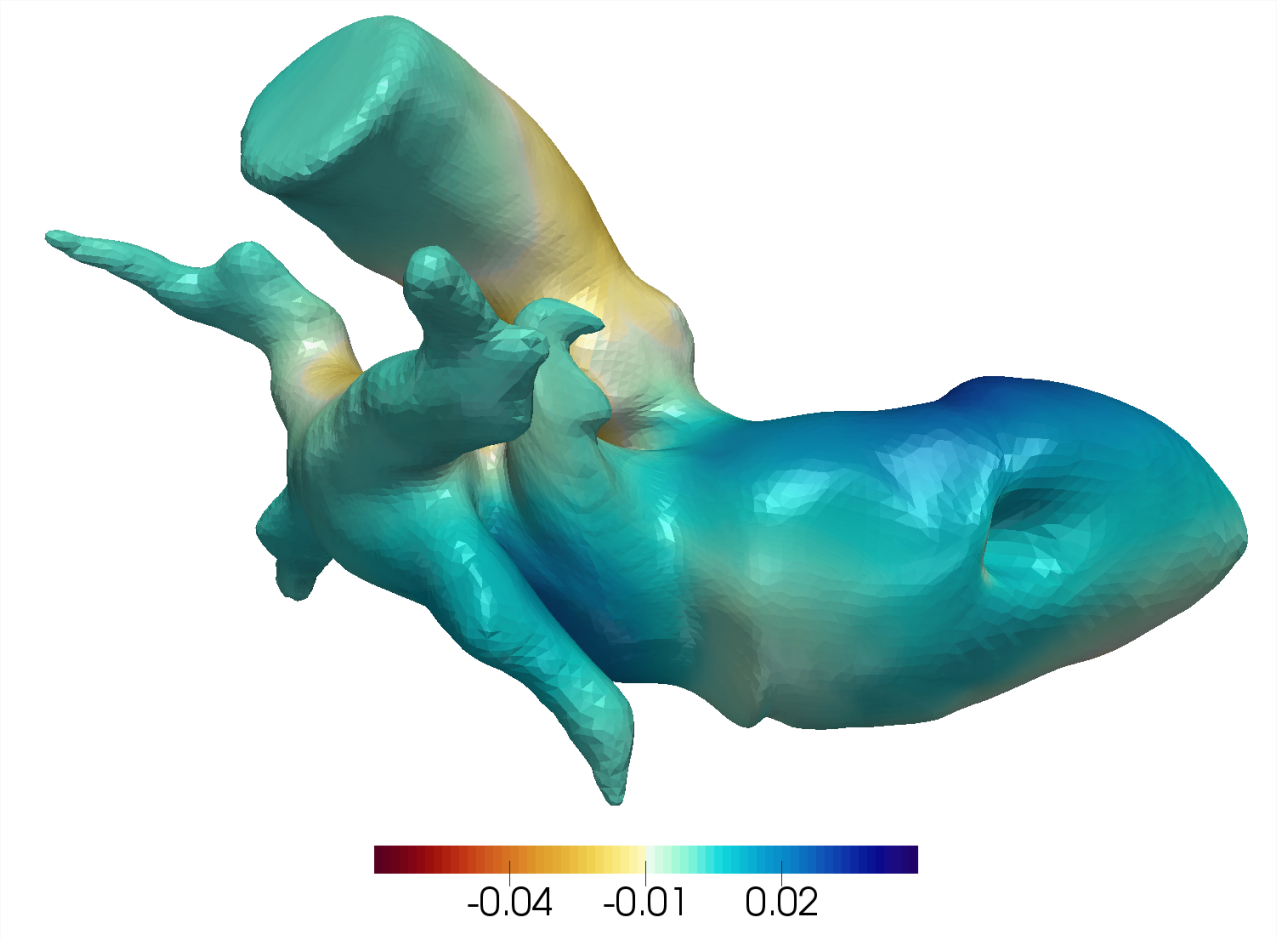
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



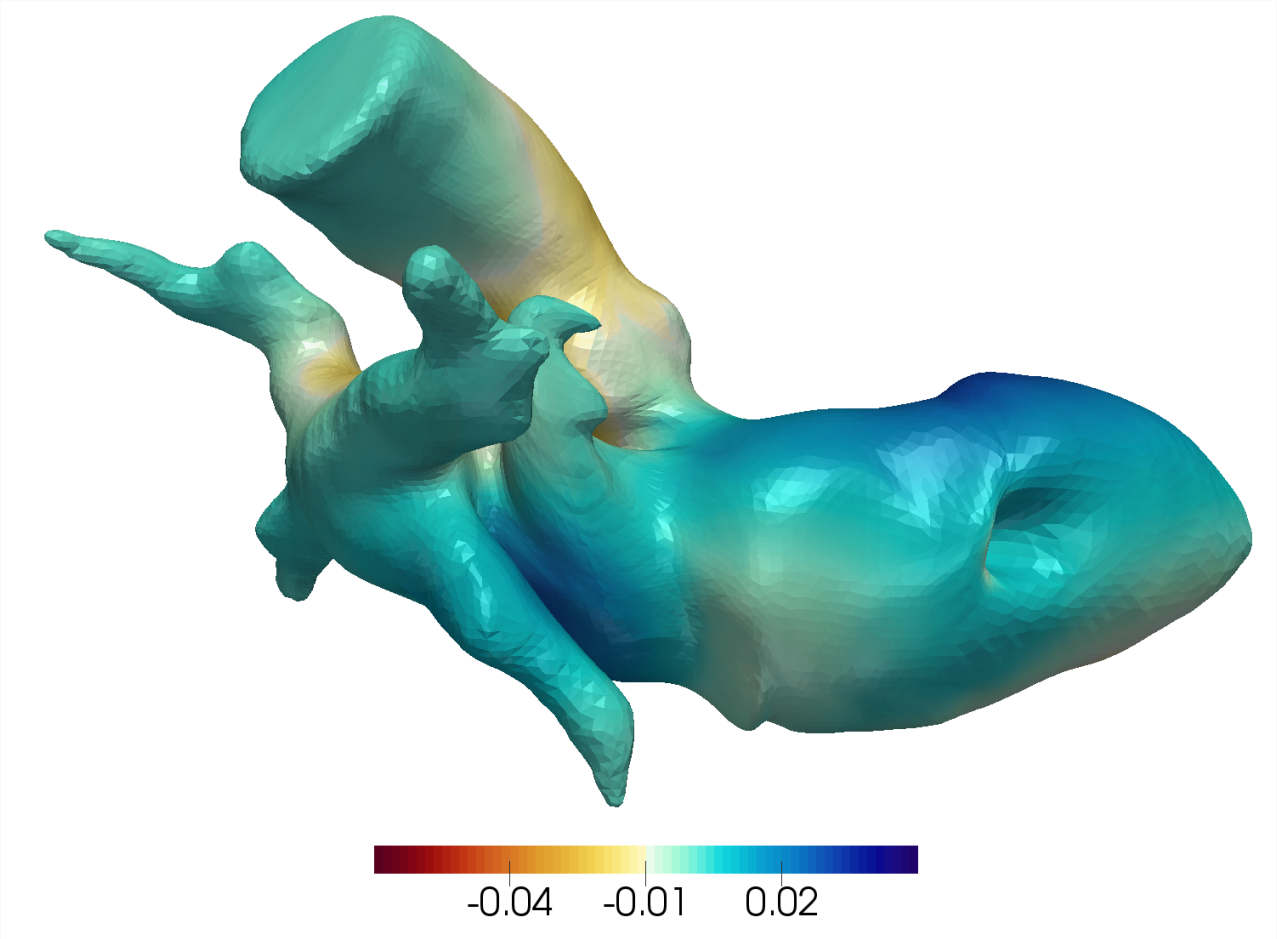
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



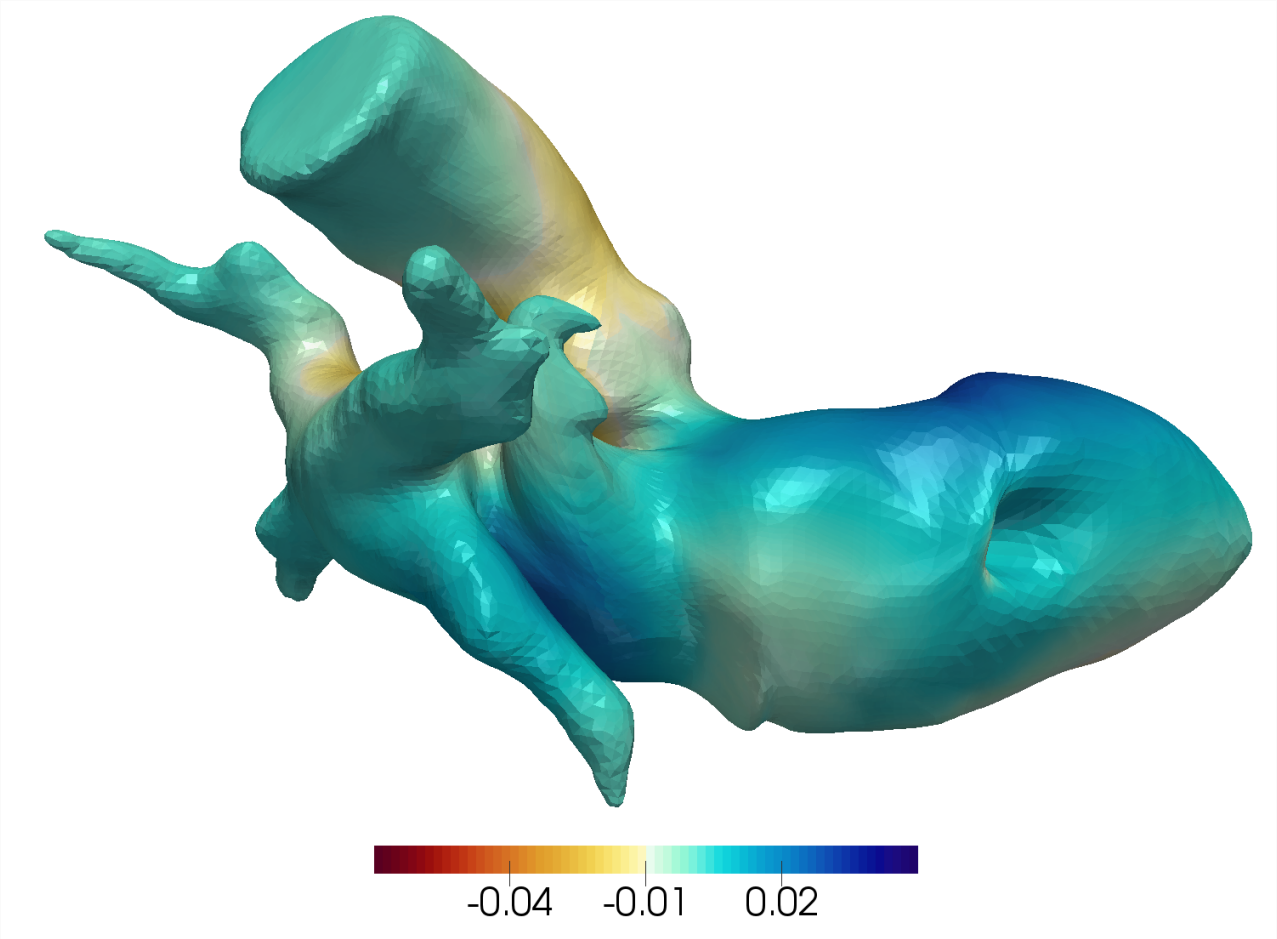
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



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

Motivations: LDDMM



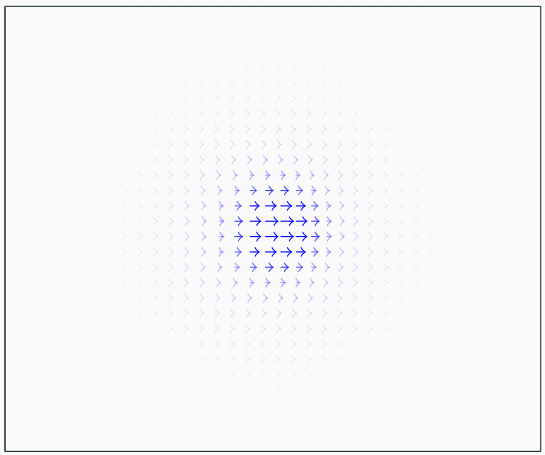
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

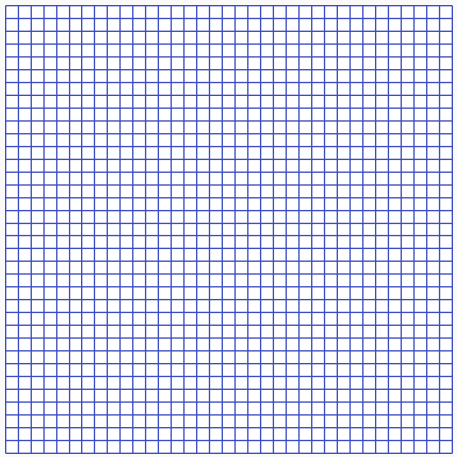
Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$





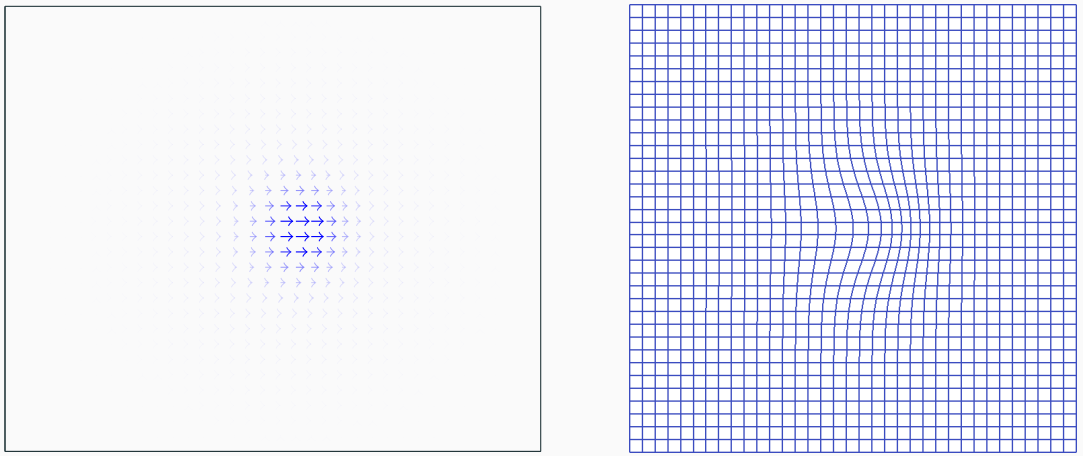
$t = 0$

Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



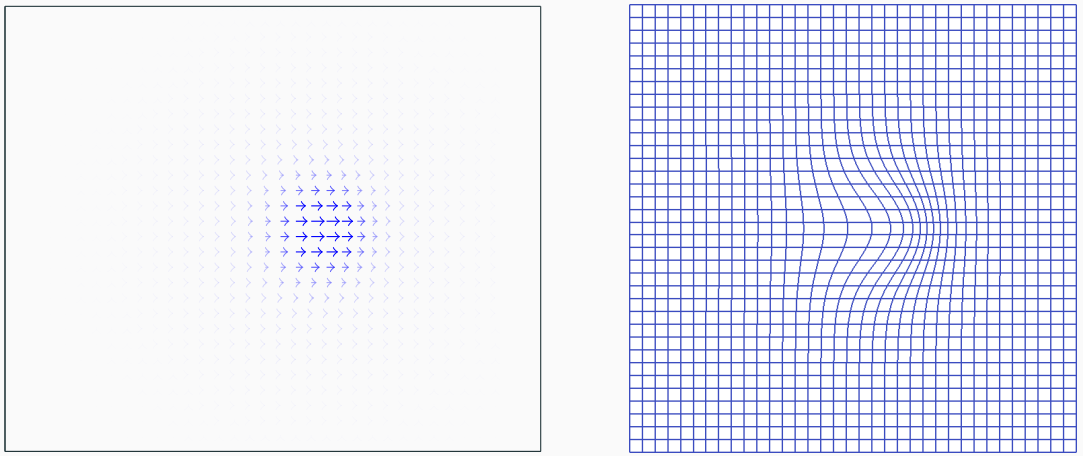
$t = 1/5$

Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



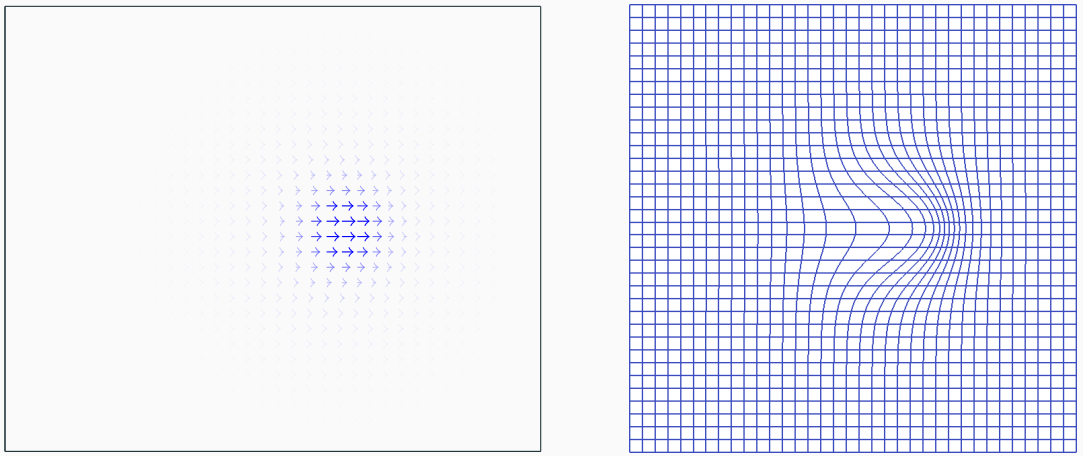
$t = 2/5$

Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



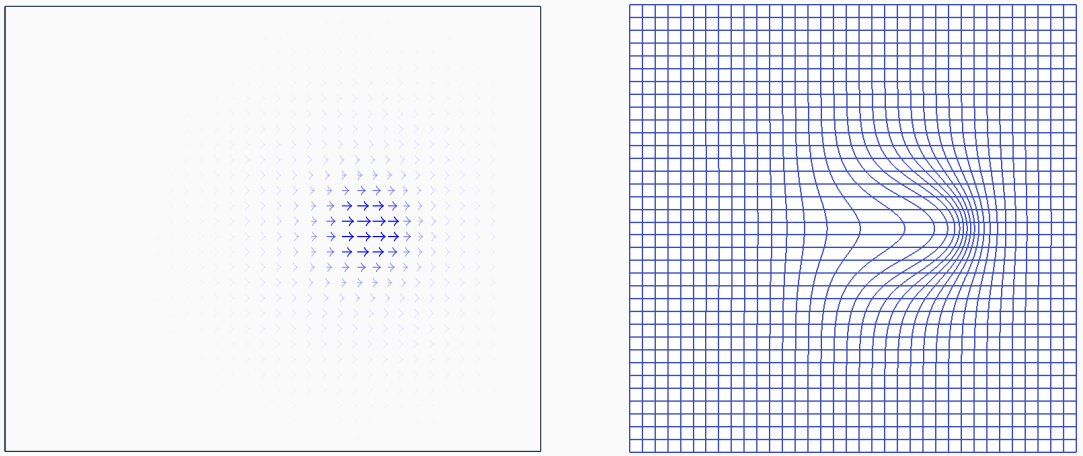
$t = 3/5$

Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



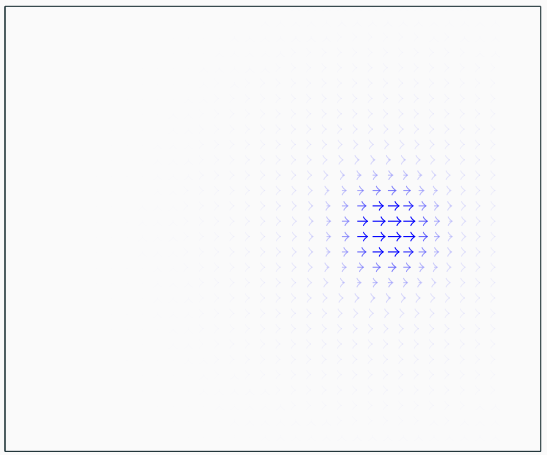
$t = 4/5$

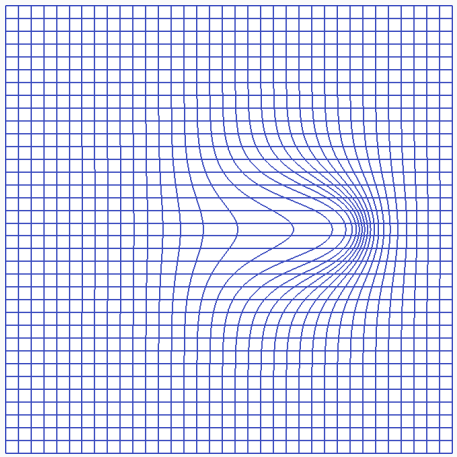
Motivation: LDDMM

Deformation = 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 $\mathbb{R}^3$. Let $\varphi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\varphi}_t(x) = v_t(\varphi_t(x)) \\ \varphi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$





$t = 1$

Computing an Hamiltonian

Soit $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^{N \times D}$, on cherche à calculer

$$H(\boldsymbol{q}, \boldsymbol{p}) = \boldsymbol{p}^t K_{\boldsymbol{q},\boldsymbol{q}} \boldsymbol{p} = \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2/s)} p_j$$

peut être interprété comme une energie cinétique ou une norme d'un RKHS.

Computing an Hamiltonian

Soit $p, q \in \mathbb{R}^{N \times D}$, on cherche à calculer

$$H(q,p) = p^t K_{q,q} p = \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2/s)} p_j$$

peut être interprété comme une energie cinétique ou une norme d'un RKHS.

```
import torch

# Load dataset
N = 1000; D = 3 ; # Clouds of 1 000 points in 3D

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

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1)                    # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0)                    # shape (N,D) -> (1,N,D)
```

Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
```

Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
```

Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
    v    = K_qq @ p                    # mat. mult. (N,N)@(N,D) = (N,D)
```

Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
   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

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
   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 straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )       # Gaussian kernel
    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 straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

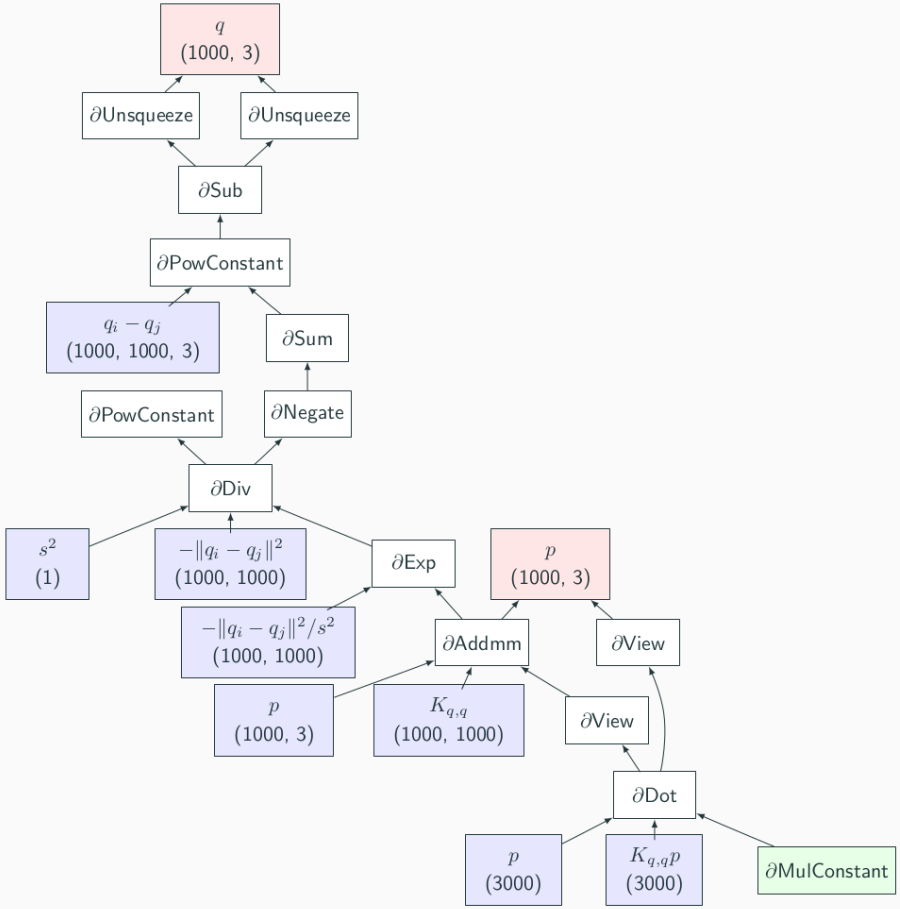
```
RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66
```

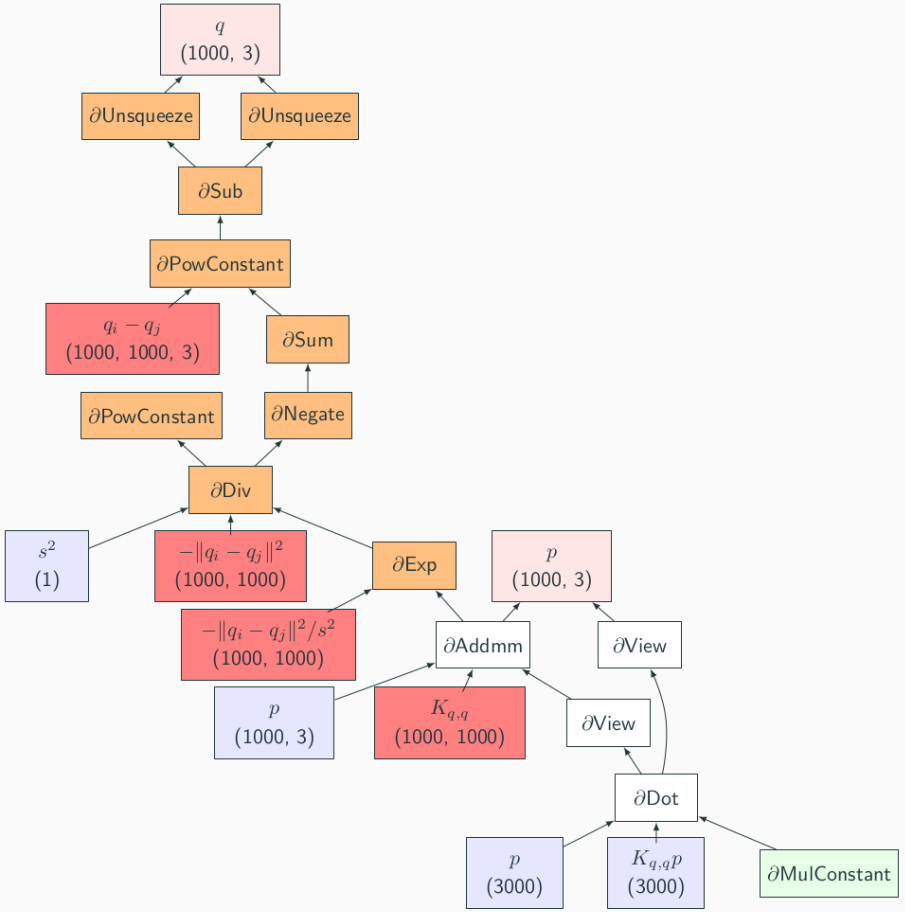
Computing the Hamiltonian

```
# Actual computations.
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 ) # matrix |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )       # Gaussian kernel
    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 straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

```
RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66
```

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


q
(1000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(1000, 1000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(1000, 1000)
∂Exp
p
(1000, 3)
$-\|q_i - q_j\|^2/s^2$
(1000, 1000)
∂Addmm
∂View
p
(1000, 3)
$K_{q,q}$
(1000, 1000)
∂View
∂Dot
p
(3000)
$K_{q,q}p$
(3000)
∂MulConstant

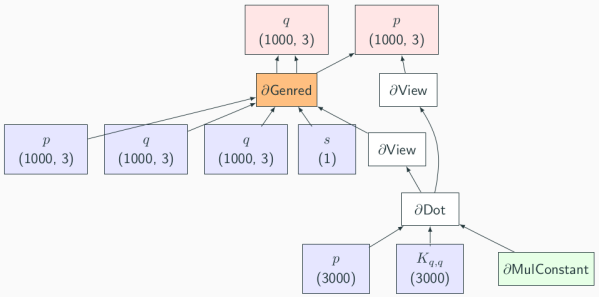

q
(1000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(1000, 1000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(1000, 1000)
∂Exp
p
(1000, 3)
$-\|q_i - q_j\|^2/s^2$
(1000, 1000)
∂Addmm
∂View
p
(1000, 3)
$K_{q,q}$
(1000, 1000)
∂View
∂Dot
p
(3000)
$K_{q,q}p$
(3000)
∂MulConstant

KeOps effect!

```
from pykeops.torch import kernelproduct
# Compute the kernel convolution with keops
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p): .5*<p,v>
H = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation works
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

KeOps effect!

```python
from pykeops.torch import kernelproduct
# Compute the kernel convolution with keops
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p): .5*<p,v>
H = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation works
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```



Conclusion

Take-home message

KeOps:

Seamless Kernel Operations...

→ write formulas with simple tensor operations (Python, Matlab, R)

...on GPU...

→ fast computations

...with auto-differentiation...

→ automatic gradient computation

...and without memory overflows

→ implementation with tiling for efficient memory usage on GPU

Thank you for your attention

Questions?

`http://www.kernel-operations.io/`

`https://github.com/getkeops/keops`